An Algorithmic View on Sensor Networks
–
Surveillance, Localization, and Communication

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Dennis Schieferdecker

aus Aalen

Tag der mündlichen Prüfung: 17.07.2014

Erster Gutachter: Herr Prof. Dr. Peter Sanders

Zweiter Gutachter: Herr Prof. Dr. Stefan Funke

To my parents.

Acknowledgements

At this point, I wish to thank all of the people that have accompanied me on my journey over the last couple of years. This includes but is not limited to my current and former colleagues in the research group of Peter Sanders, members of the research group of Dorothea Wagner as well as my dear friends and family.

In particular, I wish to thank my doctoral advisor Peter Sanders for giving me the opportunity to join his group and experience many facets of research in Karlsruhe and all over the world. I further wish to thank my long-term office mate and namesake Dennis Luxen for the great atmosphere and interesting discussions in "Casa Dennis". Additional thanks go to Stefan Funke for immediately agreeing and taking the time to review my PhD thesis.

I also wish to thank my co-authors Reinhard Bauer, Daniel Delling, Daniel Funke, Robert Geisberger, Thomas Hauth, Marco Huber, Vincenzo Innocente, Moritz Kobitzsch, Dennis Luxen, Günter Quast, Marcel Radermacher, Samitha Samaranayake, Peter Sanders, Dominik Schultes, Markus Völker, and Dorothea Wagner. It was a pleasure working with you. Thank you for the fruitful collaboration. In addition, I wish to thank Dennis Luxen and Markus Völker for proofreading parts of my thesis and the members of our group whom I bothered with countless questions about formulations, structuring, and layout over the last months, particularly Veit Batz, Timo Bingmann, and Moritz Kobitzsch. Special thanks go to Daniel Delling who encouraged me to accept the position in the research group of Peter Sanders.

I am grateful to the German Research Foundation (DFG) for supporting the first years of my research within the Research Training Group GRK 1194 "Self-Organizing Sensor Actuator Networks", and I wish to thank the people involved in organizing this research training group for all the effort they put into this project.

Last but not least, I wish to thank my parents for their continued support in countless ways during my studies of physics and informatics as well as during my doctorate. This work is dedicated to them. Thank you!

Deutsche Zusammenfassung

Die systematische Umweltbeobachtung hat in den letzten Jahren immer mehr an Bedeutung gewonnen. Kontrolle von Schadstoffbelastungen, Beobachtung von Wildtieren sowie Schutz vor Waldbränden und Wilderern sind nur ein kleiner Auszug aus ihrem Aufgabenbereich. Um diesen Aufgaben möglichst unauffällig nachzukommen, können wir viele kleinste Sensoren im zu beobachtenden Gebiet ausbringen. Das beständige Ablaufen und Auslesen dieser Sensoren würde allerdings schnell zu mühsam werden. Daher bietet es sich an, die Sensoren selbstorganisiert über Funk zu einem *(drahtlosen) Sensornetzwerk* zusammenzuschließen. Dies löst das Problem der Datennahme elegant, da wir nun einfach vom Rand des Gebietes eine Anfrage an das Netzwerk stellen können, um alle gesammelten Daten zu erhalten. Des Weiteren kann es nützlich sein, den Sensoren ein gewisses Maß an Eigenintelligenz zu geben, so dass Messdaten schon im Netzwerk vorverarbeitet werden können und es auf bestimmte Ereignisse selbständig reagieren kann. Wenn sich die Sensoren zum Beispiel der Grenzen des Netzwerkes bewusst sind, können sie eine Warnung senden, falls sich diese plötzlich verschieben oder neue Löcher im Netzwerk entstehen, wie es zum Beispiel beim Ausbruch eines Feuers möglich wäre. Solche Warnungen und andere Nachrichten müssen möglichst effizient im Netzwerk weitergeleitet werden. Es kann aber auch nötig werden, sinnvolle Umwege zu wählen, falls einzelne Sensoren ausfallen oder ansonsten überlastet würden. Die Hauptaufgabe der Sensoren bleibt aber weiterhin die Beobachtung ihrer Umgebung. Da wir sie nicht mit beliebig großen Batterien ausstatten können, müssen wir ihren Energieverbrauch allerdings irgendwie beschränken. Falls eine große Menge an Sensoren ausgebracht wurde, genügt es zum Beispiel, immer nur einen Teil von ihnen einzuschalten, um die volle Funktionalität des Netzwerkes zu gewährleisten.

Für die oben genannten Funktionen, Randerkennung, Nachrichtenübermittlung sowie Sensoreinsatzplanung, existieren bereits Lösungen. Da Sensornetzwerke jedoch immer größer und dichter werden, wird es immer schwieriger, dies effizient umzusetzen. Unsere Arbeit betrachtet daher skalierbare Algorithmen für eben diese Problemstellungen.

Ganz allgemein besteht ein Sensornetzwerk aus kleinen, autonomen Einheiten, den *Sensorknoten*. Diese wiederum bestehen zumindest aus einer Recheneinheit, einer Kommunikationseinheit und irgendeiner Art der Energieversorgung. Sensoren selbst sind nicht unbedingt erforderlich, z.B. bei Knoten, die nur zur Nachrichtenübermittlung dienen. Obwohl der technische Fortschritt der letzten Jahre zu immer kleineren und leistungsfähigeren Sensorknoten geführt hat, sind sie immer noch weit eingeschränkter als selbst einfache eingebettete Systeme. Insbesondere der Energieverbrauch ist ein beständiges Problem, da man die Knoten nicht mit großen Batterien ausstatten kann und Solarzellen oder andere Systeme zur Energiegewinnung bestenfalls zur Unterstützung dienen können. Die Kommunikation zwischen den Knoten benötigt hierbei für gewöhnlich die meiste Energie.

Aufgrund ihrer besonderen Struktur weisen Sensornetzwerke viele technische aber auch algorithmische Herausforderungen auf. Beim Algorithmenentwurf müssen unter anderem die *beschränkten Ressourcen* der Hardwareplattform wie Rechenleistung, Speicherplatz und Batteriekapazität berücksichtigt werden. Ebenso muss die *Skalierbarkeit* der Algorithmen gewährleistet werden. Was auf ein paar Knoten noch problemlos funktioniert, kann mit wachsender Knotenanzahl schnell unmöglich werden, insbesondere wenn Kommunikation über das gesamte Netzwerk oder lokaler Speicherplatz proportional zur Netzwerkgröße benötigt wird. Andererseits bieten Sensornetzwerke auch viele interessante Möglichkeiten, die bisher nur wenig Beachtung erhielten oder schlicht nicht möglich waren. Zum Beispiel stellt jeder einzelne Sensorknoten einen vollständigen Rechner dar. Daher bietet es sich an, massiv *verteile Algorithmen* zu entwickeln, um den Beschränkungen der einzelnen Knoten entgegenzuwirken. Berechnung und Datenhaltung können über viele Knoten verteilt werden, so dass im Idealfall jeder Knoten nur einen Bruchteil des kompletten Problems bearbeitet. Ein weiterer wichtiger Punkt in diesem Zusammenhang ist die Beschränkung auf *lokale Informationen*. Ein Sensorknoten kann normalerweise weder eine globale Sicht auf das gesamte Netzwerk halten, noch sie effizient beziehen. Algorithmen müssen dies berücksichtigen und so ausgelegt werden, dass ein Knoten mit seinem lokalen Wissen oder Daten, die mit wenig Kommunikation bezogen werden können, auskommt. Schließlich weisen Sensornetzwerke für gewöhnlich ein hohes Maß an *Redundanz* auf, das man ausnutzen kann, um z.B. Fehlertoleranz zu gewährleisten oder durch intelligente Sensoreinsatzplanung die Laufzeit des Netzwerkes zu verlängern.

Die Forschung an Sensornetzwerken wird durch spezielle Anwendungen wie Simulationsumgebungen und Analysewerkzeuge unterstützt. Dies sind wichtige Hilfsmittel, um die Funktionalität sowie die Möglichkeiten von Netzwerken und Algorithmen, die auf ihnen ausgeführt werden, zu analysieren. Sie werden auf klassischen Systemen eingesetzt und müssen eine große Anzahl an Sensorknoten berücksichtigen, ohne auf die massive Parallelität eines echten Sensornetzwerkes zurückgreifen zu können. Dadurch ergeben sich weitere algorithmische Herausforderungen. Die hierfür benötigten Algorithmen und Datenstrukturen müssen effizient und skalierbar sein, damit diese Anwendungen die Forschung sinnvoll unterstützen können.

Die vorliegende Arbeit befasst sich mit der Skalierbarkeit von Sensornetzwerken in verschiedenen Anwendungsgebieten. Dabei gibt es zwei grundlegende Aspekte zu berücksichtigen. Zum einen gibt es Anwendungen, die direkt auf Sensornetzwerken ausgeführt werden. Hier können wir erhöhten Rechenanforderungen durch verteilte Algorithmen begegnen. Allerdings müssen wir immer noch das Kommunikationsvolumen und den von jedem Knoten benötigten Speicherplatz im Auge behalten. Des Weiteren gibt es Anwendungen wie Simulationsumgebungen oder Analysewerkzeuge, die auf klassischen Systemen ausgeführt werden. In diesem Fall können wir den verteilten Parallelismus der Sensornetzwerke nicht ausnutzen. Allerdings sind Speicherplatz und Kommunikationsvolumen seltener ein Problem, da die Algorithmen eine globale Sicht auf das Netzwerk haben können. Generell gilt jedoch, dass Probleme, die für kleine Netzwerke noch einfach zu lösen sind, mit zunehmender Knotenanzahl oder -dichte schnell sehr komplex wenn nicht sogar unlösbar werden können.

Diese Arbeit behandelt exemplarisch drei verschiedene Problemstellungen, für die wir mit Hilfe von *algorithm engineering* versuchen, skalierbare Lösungen zu finden. Falls optimale Lösungen nicht effizient berechenbar sind, geben wir skalierbare Approximationsalgorithmen mit einer beweisbar guten Approximationsgüte an.

Sensoreinsatzplanung. Sensornetzwerke bieten die Möglichkeit kontinuierliche Messungen von einem gesamten Gebiet zu erstellen. Dies bildet die Grundlage für viele Anwendungen. Es ist daher wichtig, diese Funktionalität so lange und mit einem so geringen Energieverbrauch wie möglich zu erbringen. Wir können die inhärente Redundanz der Netzwerke ausnutzen und einen Einsatzplan für jeden Knoten erstellen. Dieser erlaubt es einem Knoten, in einen energiesparenden Zustand zu wechseln, falls andere Knoten die gleichen Messungen durchführen können.

Unsere Arbeit zeigt, dass die Bestimmung eines optimalen Einsatzplanes für alle Knoten ein $\mathcal{NP}$-vollständiges Problem ist. Um dennoch ein skalierbares Lösungsverfahren zu erhalten, beschreiben wir ein effizientes Approximationsschema (EPTAS), das in Linearzeit läuft. Unser Verfahren nützt die Diskretisierung von Knotenpositionen sowie die Aufteilung des zu beobachtenden Gebietes in kleinere Bereiche aus, um diese Ergebnisse zu erzielen. Im Vergleich zu früheren Ansätzen bietet unsere Lösung eine bessere Zeitkomplexität als auch eine bessere Approximationsgüte. Des Weiteren beschreiben wir ein Verfahren zur nachträglichen Optimierung der Reihenfolge, in der Knoten ihren Zustand zwischen schlafend und beobachtend wechseln, falls sich dieser Wechsel als energieintensiv erweist. Neben diesen theoretischen Ergebnissen zeigen wir außerdem wie man kleinere bis mittlere Instanzen optimal löst. Dieses Verfahren basiert auf Linearer Programmierung und bedient sich der verzögerten Spaltenerzeugung (*delayed column generation* [DW60]) sowie des Garg-Könemann Algorithmus [GK07]. Wir zeigen in ausführlichen Simulationen, dass unsere Methode frühere Verfahren bei weitem schlägt. Außerdem untersuchen wir den Einfluss von unterschiedlichen Netzwerkinstanzen auf die Laufzeit unseres Algorithmus.

Die beschriebenen Verfahren sind primär für den Einsatz in Analysewerkzeugen geeignet, um damit die Eigenschaften von Sensornetzwerken zu untersuchen. Wir können u.a. obere Schranken für die Dauer, die ein Sensornetzwerk Messungen vornehmen kann, bestimmen und damit die Qualität von verteilten Algorithmen zur Sensoreinsatzplanung bewerten. Falls sich die Netzwerkstruktur nicht ändert und Knotenausfälle unwahrscheinlich sind, können die berechneten Einsatzpläne sogar auf echten Sensornetzwerken verwendet werden. Des Weiteren kann unser Verfahren als Ansatzpunkt zur Entwicklung neuer verteilter Verfahren dienen.

Randerkennung. Viele Anwendungen benötigen ein gewisses Wissen über die Netzwerktopologie, insbesondere über die Löcher und Ränder des Sensornetzwerkes. Exakte Systeme zur Positionsbestimmung, wie z.B. GPS, sind oft nicht vorhanden, um diese Aufgabe zu erleichtern, da sie zu viel Energie benötigen. Andererseits sollten Algorithmen auch keine generellen Annahmen über die Netzwerkstruktur treffen.

Daher beschreiben wir ein neues Verfahren zur Bestimmung dieser topologischen Strukturen, das keine Knotenpositionen benötigt. Wir setzen dabei auf eine verteilte, dezentrale Ausführung, damit das Verfahren mit der Netzwerkgröße skaliert. Jeder Knoten entscheidet selbständig, nur mit Hilfe von lokalen Verbindungsdaten, ob er im Inneren des Netzwerkes oder an dessen Rand liegt. Dazu berechnet er mittels multidimensionaler Skalierung [Tor52] eine Einbettung aus den Verbindungsdaten seiner 2-Hop Nachbarschaft und prüft diverse Winkeleigenschaften. Im Gegensatz zu früheren Verfahren benötigt unser Algorithmus sehr wenige Ressourcen, sowohl was den Rechenaufwand als auch das Kommunikationsvolumen betrifft. Wir vergleichen unser Verfahren in ausführlichen Simulationen mit früheren Ansätzen. Es stellt sich als außerordentlich robust gegenüber verschiedensten Netzwerkmodellen heraus und erzeugt trotz seiner Einfachheit extrem wenige Fehlklassifikationen.

Durch seine verteilte Arbeitsweise ist unser Verfahren klar für den direkten Einsatz auf Sensornetzwerken konzipiert. Hier kann es außerdem dabei helfen, Gebiete mit schlechter Überdeckung zu identifizieren—dort wo sich nur vereinzelte Knoten als Randknoten klassifizieren. Des Weiteren kann es breitere Bänder entlang der Ränder des Netzwerkes markieren und andere Anwendungen wie unser Verfahren zur Sensoreinsatzplanung darüber informieren, wenn sich diese verschieben, z.B. durch Knotenausfall aufgrund von Energiemangel oder durch externe Einflüsse wie den Ausbruch eines Feuer.

Effizientes Routing. Die Weiterleitung von Informationen zwischen Knoten ist eine der grundlegendsten aber auch eine der wichtigsten Aufgaben, die ein Sensornetzwerk zu leisten hat. Es existieren viele effiziente Heuristiken für den verteilten Einsatz. Die Berechnung von (fast) optimalen Routen oder guten Alternativen zu diesen kann in einer Simulationsumgebung aber leicht zu einem Flaschenhals werden. Bereits bewährte Beschleunigungstechniken zur Berechnung von kürzesten Wegen können nicht verwendet werden, da sie für die besondere Struktur von Sensornetzwerken nicht geeignet sind.

Unsere Arbeit stellt ein echt polynomielles Approximationsschema (FPTAS) zur Berechnung von kürzesten Wegen vor. Es basiert auf *Contraction Hierarchies* [BGSV13], einer für Straßennetzwerke sehr effizienten Technik. Wir belegen durch ausführliche Simulationen, dass unser Ansatz auch auf Sensornetzwerken weit effizienter ist als frühere exakte und approximative Techniken. Zudem verwenden wir diesen Algorithmus als Baustein, um gute Alternativen zu einer optimalen Route zu bestimmen. Dieser Ansatz beruht ebenfalls auf einer effizienten Technik für Straßennetzwerke [ADGW13], die wir unter Ausnutzung der Beobachtung, dass gute Alternativen zwischen zwei Bereichen des Netzwerkes nur über wenige Zwischenknoten verlaufen, erweitern. Durch ihre kurzen Antwortzeiten bieten unsere Algorithmen skalierbare Lösungen in Bezug auf die Netzwerkgröße als auch auf die Anzahl zu bearbeitender Anfragen.

Unsere Verfahren sind für klassische Systeme optimiert, da sie primär für den Einsatz in Simulationsumgebungen vorgesehen sind. Sie können aber auch für weitere Analysewerkzeuge von Nutzen sein, z.B. um Engstellen in der Kommunikationsinfrastruktur aufzuzeigen oder um Gleichgewichte wie in [LS11] für Straßennetzwerke zu berechnen. Zudem lassen sie sich auch in statischen Sensornetzwerken, wie z.B. großflächigen infrastrukturellen Netzwerken entlang von Straßen und Autobahnen, einsetzen.

Unsere Resultate zeigen exemplarisch, wie man skalierbare Algorithmen für Sensornetzwerke entwirft, sowohl für Anwendungen, die auf den Sensornetzwerken selbst ausgeführt werden, als auch für Hilfsanwendungen, die zur Analyse der Netzwerke und Algorithmen dienen. In jedem der betrachteten Bereiche haben wir große Fortschritte im Vergleich zu bestehenden Verfahren erzielen können.

Contents

Author's Information

Chapter 1

1 Introduction

Beginning. The first part or earliest stage of something.
— Oxford Dictionary of English

In recent years, environmental monitoring has become more important than ever before. Protection from wildfires and poachers, assistance in the studies of wild animals, or measuring the concentration of pollutants are all highly relevant. To do so unobtrusively, we can deploy many diverse sensors throughout the considered area. However, visiting each of them to read out the measured data would soon become strenuous. Having the sensors link with each other wirelessly and autonomously in a *(wireless) sensor network* solves this issue conveniently as it allows us to initiate queries from the fringes of the monitored area to gather all data. It may further be beneficial to endow the sensors with some intelligence so that data can be processed in the network, or that they can react to certain events. For example, if sensors are aware of network boundaries, they can send a warning when boundaries suddenly shift or holes emerge inside the network, e.g. due to an outbreak of fire. Such warnings and other messages should naturally be passed to the respective recipients in an efficient way with as little overhead as possible. Though, taking alternative routes may become necessary if sensors fail or cannot relay a message for some reason. Still, monitoring the area they were deployed in, remains the main task of the sensors. As we cannot equip them with large batteries, we have to somehow limit their energy consumption. For example, if we have deployed a large amount of sensors, we can activate just a subset of them at each moment to conserve energy while still providing the full functionality of the network.

All of the aforementioned tasks, boundary recognition, routing of messages, and scheduling of sensors are feasible in principal and have already been considered in various contexts. However, with sensor network sizes becoming ever larger in both density and expansion, it gets more and more difficult to solve these problems efficiently. Thus, we consider and provide scalable algorithms for all of these tasks in this thesis.

1.1 A Brief History of Sensor Networks

Technology has come a long way since the effects of electricity have first been studied. The experimental proof of electromagnetic waves by Hertz in the late 19th century, a theory previously established by Maxwell and Faraday, marked an important stepping stone towards the ubiquitous wireless communication of today. The invention of the bipolar transistor by a research team at Bell Laboratories in the middle of the last century was another important event that eventually led to a myriad of tiny technological gadgets. We now have smartphones, tablets, and soon intelligent glasses, all communicating with each other, gathering information, and processing data. But this is only one of the more visual aspects of the technological advances that have been made in the last century. Many ideas that have formerly been placed into the realm of science fiction came to fruition thanks to this technological progress, whereas other figments like transforming robots still belong there.

One of these developments are *(wireless) sensor networks*. As many other new technologies they have first been studied for militaristic reasons. The oldest ancestor of sensor networks is probably SOSUS, the Sound Surveillance System, a chain of underwater listening posts, devised in the early days of the Cold War by the US military. After their investments in the ARPANET, the predecessor of the Internet, the Defense Advanced Research Projects Agency (DARPA) took an interest in sensor networks as early as the 1970s. They organized the Distributed Sensor Nets Workshop in 1978 that focused on sensor network research challenges such as networking technologies, signal processing techniques, and distributed algorithms. This led to the Distributed Sensor Networks (DSN) program and later to the Sensor Information Technology (SensIT) program. Civil and more prevalent research of sensor networks only took off at the end of the last century around the time when the University of California, Berkley, started their Smart Dust project [KKP99], an initiative to pack the whole functionality of a single network component into one cubic millimeter. This development was fostered by rapid technological advances that allowed for ever smaller and less power-consuming devices. Novel sensor network applications became feasible for the first time and kindled the interest of diverse research communities in this topic. This ultimately led to widespread academic research projects in multiple fields.

A more elaborate history of sensor networks is given by Chong and Kumar in [CK03].

1.2 Principal Components and Challenges

A sensor network is composed of small, autonomous elements. At the bare minimum, each of these *sensor nodes* is equipped with a processing unit, a communication unit, and a means of power supply. A sensing component, though their namesake and usually present, is not strictly required—one can imagine a relay node that only receives and transmits messages. The nodes in a sensor network can be heterogeneous, specialized

for various tasks like the aforementioned relay node. They are often tiny and pretty cheap so that lots of them can be deployed easily. Each individual sensor node is similarly structured as a classical embedded system with a dedicated microcontroller and typically both volatile and non-volatile memory, running an embedded operating system like TinyOS[1]. However, the node itself is much more restricted due to cost and size constraints. Its power supply is usually handled by a battery that gets depleted over time. Thus, conserving energy becomes a major concern in this context. There are means to harvest energy, though, e.g. through solar panels or piezoelectric elements, but they can only support a battery and do not sustain the whole node. These energy constraints also restrict the selection of applicable processors and memory modules to low-energy and thus less performant or, respectively, less capacious, models. Communication, likely the most energy-intensive task, is generally performed wirelessly, though there exist tethered sensor networks, such as infrastructural networks along roads and highways, in which data transmission and even power supply are handled by wire. The underlying communication network is organized autonomously by the sensor nodes themselves with a flexible structure that allows nodes to join and leave the network easily. The nodes can be equipped with any type of sensor imaginable, but here as well, energy consumption and size of the component have to be taken into account. Common types of sensors include localization devices such as GPS (Global Positioning System) receivers and environmental sensors for measuring temperature, light, humidity, or the concentration of some pollutant.

The unique structure of sensor networks, consisting of many dispersed nodes, all linked in a flexible communication network and capable of acting autonomously or cooperating with each other, offers new and exciting challenges for both technology and algorithmics. When developing algorithms for these kinds of systems, we have to take into account the *limited resources* available to each sensor node. Processing power, storage space, and especially battery capacities are all much more restricted than in a classical system. We further need to consider the *scalability* of our algorithms. What is feasible for a couple of nodes can quickly become impractical in large networks, especially if communication over the whole network or storage space proportional to the network size at each node are required.

On the other hand, sensor networks also offer new prospects that have not received a lot of attention previously or that simply have not been feasible. As each sensor node is a self-contained computing system, massively *distributed algorithms* are an obvious choice. Computation and data storage can be spread over the network to counter the processing and memory limitations of the individual nodes. In an extreme case, every node in the network computes a tiny fraction of the solution with the locally available information. This gives rise to another important paradigm, *local information*. A node typically cannot hold a global view of the entire network nor obtain it efficiently. Thus, algorithms have to be adapted accordingly so that nodes can work with the data

[1] `http://www.tinyos.net/`. Accessed: 2014-08-06.

available to them or that can be acquired with little communication. Finally, sensor networks usually offer a lot of *redundancy* due to the large number of sensor nodes. We can exploit these additional resources e.g. by introducing an activation schedule for saving energy and extending the lifetime of the entire network without compromising its functionality, or by guaranteeing some degree of reliability against node failures or other disturbances of the network.

Research in sensor networks is always accompanied by auxiliary applications such as simulation frameworks or theoretical analysis tools. These are important utilities in studying the functionality and capabilities of sensor networks and the algorithms running on them. As these applications are executed on classical systems but have to emulate a large number of sensor nodes without the benefit of the massive parallelism an actual sensor network provides, new algorithmic challenges arise. The respective underlying data structures and algorithms have to be efficient and scalable for them to be of any practical value.

There exists a large body of work covering the various aspects of sensor networks. As we can only give a brief introduction here, we refer to [YMG08, DP10] for a general overview on their capabilities and to [WW07, NS10] for more algorithmic aspects.

1.3 Contributions and Thesis Outline

This thesis focuses on the scalability of sensor networks in diverse applications. Problems that are easy to solve on the small scale with at most a few hundred nodes to consider quickly become intractable with growing network sizes. It is up to algorithm engineering to devise scalable solutions before such large sensor networks emerge. If exact solutions cannot be given efficiently, at least scalable approximation algorithms that provide good guarantees on the solution quality have to be found.

There are two main aspects to scalability in a sensor network context that we need to consider. First, there are applications that run on the sensor network. Here, we can counter increased computational requirements by distributed algorithms. However, we still have to keep the communication overhead and the data needed at each node in check. Second, there are offline applications such as simulation frameworks or analysis toolsets. We cannot exploit the distributed parallelism of sensor networks in this case. However, data storage and communication are usually of less concern as we have a global view of the network.

The main body of this thesis is structured as follows:

Chapter 2: Foundations. We introduce fundamental concepts and notations that are used throughout this work. We cover topics in complexity theory and graph theory insofar as they concern our thesis before detailing two important mathematical tools, (integer) linear programming and multidimensional scaling. We conclude with an overview of our simulational environment and measuring methods.

Chapter 3: Lifetime Maximization of Monitoring Sensor Networks. The ability of sensor networks to offer continuous measurements of their surroundings is the basis of many applications. It is therefore crucial to provide these monitoring capabilities for as long as possible and with little impact on the battery reserves of the sensor nodes. Luckily, we can exploit the redundancy inherent to these networks to determine an activation schedule for each node that allows them to stay in a sleeping state if another node can perform their measurements.

We show that finding an optimal activation schedule is an $\mathcal{NP}$-complete problem. To retain scalability to large networks, we introduce the first efficient polynomial-time approximation scheme for this problem. Previous approaches offer worse time complexities and approximation guarantees. We further describe a method to optimize the order in which nodes are activated that minimizes the total number of state changes between sleeping and monitoring. Following these theoretical results, we consider solving small to medium-sized instances to optimality. In extensive simulations, we show that our method outperforms previous approaches by a considerable margin and study the impact of different network settings on the performance of our algorithm.

Chapter 4: Location-free Detection of Network Boundaries. Multiple applications on sensor networks require some knowledge of the underlying network topology, especially of the holes and boundaries of the network. Exact positioning systems are often not available, though, to facilitate this task. Moreover, algorithms should not make any general assumptions on the structure of the networks.

We therefore introduce a novel approach for the location-free detection of these topological structures. To keep our solution scalable in the advent of vast networks, the computation is performed distributed. Each node decides independently based on local connectivity information alone whether it is in the interior of the network or on its fringes. In contrast to previous solutions, our algorithm requires very little resources in terms of both computation and communication, while still offering reliably good classification results. In extensive simulations, we compare the performance of our approach to several previous ones. We find that it is very robust to any considered network setting and yields very few misclassifications despite its simplicity.

Chapter 5: Determining Efficient Paths in Large-Scale Sensor Networks. Relaying information between nodes is arguably one of the most basic and important tasks a sensor network has to perform. While there exist efficient heuristics for distributed usage, determining (near) optimal routes or good alternatives to them can become a bottleneck in simulation frameworks. Common speed-up techniques for shortest path computation do not offer a viable option as they are not well suited for the special structure of sensor networks.

We propose a fully polynomial-time approximation scheme for finding shortest paths based on a successful technique for road networks. In extensive simulations, we show

that our approach can handle sensor networks much more efficiently than previous exact or approximate techniques. We further apply this algorithm as a building block for determining good alternatives to the optimal route. This approach is again based on and extends a successful technique for road networks as we find and exploit that good alternatives between two regions pass over one of few intermediate nodes. Both of our algorithms offer scalable solutions in terms of both the size of the sensor network and the number of queries to process due to their short practical runtimes.

Chapter 6: Discussion. Our thesis concludes with this chapter. We summarize and discuss our findings of the previous chapters and propose possible future directions for research in the field of sensor networks.

The main body of this thesis is followed by our bibliographical references and three appendices to complement Chapters 3–5. We present additional results that support our findings in the respective chapters and explain how to generate the problem instances that we use in our simulations.

This work was partly supported by the German Research Foundation (DFG) within the Research Training Group GRK 1194 “Self-Organizing Sensor Actuator Networks”.

Chapter 2
2 Foundations

Basics. The essential facts or principles of a subject or skill.
— Oxford Dictionary of English

This chapter introduces notations and fundamentals on complexity theory and graph theory as well as mathematical tools that are used in this thesis. Furthermore, we give an overview of our simulational environment and methods. We start by introducing some general conventions and notations.

The symbols $\mathbb{Z}$ and $\mathbb{R}$ are used according to common convention as the set of integers and the set of real numbers, respectively. A subscript plus symbol indicates that the set of numbers is restricted to non-negative values. We also write $\mathbb{N}_0 = \mathbb{Z}_+$. The residue class ring of k is denoted by $\mathbb{Z}_k$, i.e. $\mathbb{Z}_k = \{0, \dots, k-1\}$. Value ranges indicate real numbers unless otherwise stated. $\mathbb{R}^d$ denotes the d-dimensional Euclidean space. We may write $||\mathbf{v}||$ for the Euclidean norm of vector $\mathbf{v} \in \mathbb{R}^d$ and $d_{\mathbf{u},\mathbf{v}} = ||\mathbf{u} - \mathbf{v}||$ as Euclidean distance between two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$. Vectors and matrices are written in bold, vectors in lower case, matrices in upper case. The same notation is used for functions returning them. Vectors of zeros and ones are denoted by $\mathbf{0}$ and $\mathbf{1}$, respectively.

2.1 Complexity Theory

When considering computational problems, we often want to assess the difficulty of a problem, i.e. whether it takes long to determine a solution. There is a whole field of study dedicated to this subject. Here, we only briefly discuss the theory of computational complexity and, as an extension, the approximability of problems. We refer to text books [GJ79, Weg03] for an introduction to complexity theory and to text books [Vaz02, Weg03] for further details on approximation algorithms. Our following notations and explanations are loosely based on these works.

2.1.1 Computational Complexity

A *computational problem* Π describes a general question with some degrees of freedom in a formal system. A *problem instance* $I \in \mathcal{I}_\Pi$ is an input to the problem that fully specifies the question. $\mathcal{I}_\Pi$ denotes the set of all possible inputs to Π. *Algorithm* $\mathcal{A}_\Pi$ describes a precise procedure to solve any instance I of problem Π, and $\mathcal{A}_\Pi(I)$ denotes a solution if one exists. A problem instance may permit multiple correct solutions. We call any correct solution *feasible*. If a problem instance permits a feasible solution, we also call it feasible.

An *optimization problem* Π asks for a solution with some corresponding maximum (or minimum) value. Each instance $I \in \mathcal{I}_\Pi$ is associated with a subset $S_\Pi(I)$ of all feasible solutions $\mathcal{S}_\Pi$ of Π. An *objective function* $w : \mathcal{S}_\Pi \mapsto \mathbb{R}$ maps each feasible solution to an objective value. An algorithm $\mathcal{A}_\Pi$ that solves the problem computes an optimal solution $\mathcal{A}_\Pi(I) = \arg\max_{x \in S_\Pi(I)} w(x)$. If a maximum does not exist, we call the problem instance *infeasible* if $S_\Pi(I)$ is empty, i.e. there exists no feasible solution for this instance, or *unbounded* otherwise. We denote the *optimal objective value* by $\mathrm{opt}_\Pi(I) = \max_{x \in S_\Pi(I)} w(x)$. Minimization problems are defined accordingly.

A *decision problem* Π asks whether a question in some formal system is true or false. Each instance $I \in \mathcal{I}_\Pi$ either permits a *yes* or a *no* answer. We can transform an instance of an optimization problem $I \in \Pi_O$ into an instance of a decision problem $(I, p) \in \Pi_D, p \in \mathbb{R}$, by asking whether the optimal objective value of the optimization problem is $\mathrm{opt}_{\Pi_O}(I) \geq p$ when maximizing ($\mathrm{opt}_{\Pi_O}(I) \leq p$ when minimizing).

Landau Notation. We classify algorithms by the amounts of resources, usually *running time*, they require. Running time is measured by the number of operations a random access machine (RAM) takes to solve a problem instance $I \in \mathcal{I}_\Pi$. To generalize from specific instances, we consider running time as a function $f : \mathbb{N}_0 \mapsto \mathbb{R}$ of the size of the problem instance $n = |I|$. We refer to f as *time complexity function*. We classify time complexity functions by their upper or lower bounded asymptotic behavior. Classes are denoted by the common *Landau symbols* $\mathcal{O}$, Ω, and Θ. We specify

$$
\begin{aligned}
\mathcal{O}(g) &= \{f : \mathbb{N}_0 \mapsto \mathbb{R} \mid \exists C \quad > 0, n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : \qquad\qquad f(n) \quad \leq C \cdot g(n)\}\,, \\
\Omega(g) &= \{f : \mathbb{N}_0 \mapsto \mathbb{R} \mid \exists c \quad > 0, n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : \quad c \cdot g(n) \leq f(n) \qquad\qquad \}\,, \\
\Theta(g) &= \{f : \mathbb{N}_0 \mapsto \mathbb{R} \mid \exists c, C > 0, n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : \quad c \cdot g(n) \leq f(n) \quad \leq C \cdot g(n)\}\,.
\end{aligned}
$$

We may refer to the asymptotic running time of an algorithm as time complexity for short. To further differentiate between the theoretical running time of an algorithm and actual time measurements on real hardware, we denote the latter by *runtime*.

We say a problem Π is *polynomially solvable*, if there exists an algorithm $\mathcal{A}_\Pi$ that computes a solution in polynomial time, i.e. in $\mathcal{O}(n^k), k \in \mathbb{N}_0$, time. On the other hand, problem Π is *exponentially growing*, if any algorithm $\mathcal{A}_\Pi$ requires $\Omega(k^n), k > 0$, time.

Complexity Classes. We further generalize from the running time of algorithms to the complexity of problems. All problems that are solvable in polynomial time belong to the class $\mathcal{P}$ (polynomially solvable). We consider these problems as *tractable*. Another important class $\mathcal{NP}$ (non-deterministically polynomially solvable) consists of all problems for which we can verify in polynomial time whether a candidate solution is correct. Obviously, $\mathcal{P} \subseteq \mathcal{NP}$. It is still an open question, though, whether both problem classes are equal.

Given two decision problems Π_1, Π_2, we say Π_1 is *polynomially reducible* to Π_2 if we can transform any instance of Π_1 into an instance of Π_2 with the same solution and in polynomial time. We write $\Pi_1 \leq_p \Pi_2$. A problem Π^* is $\mathcal{NP}$*-hard* if $\Pi \leq_p \Pi^*$ for all problems $\Pi \in \mathcal{NP}$. If the problem itself is also in $\mathcal{NP}$, we say that it is $\mathcal{NP}$*-complete*. If any problem in $\mathcal{P}$ is also $\mathcal{NP}$-complete, we have $\mathcal{P} = \mathcal{NP}$.

2.1.2 Approximation Algorithms

Many interesting problems are not tractable. We are able to obtain useful results in practice for some of them, though. If the size of the problem instance is small, for example, we can resort to exponentially growing algorithms. A restricted set of problem instances might be solvable by a polynomial-time algorithm, or when relaxing the constraint on optimal solutions, there might exist an efficient algorithm that finds feasible, if not optimal, solutions for any instance of the problem.

Approximation algorithms belong to the latter category. They solve optimization problems, but instead of providing an optimal—or exact—solution, an approximation algorithm only computes a feasible one. Even though they are mainly used for solving intractable problems, approximation algorithms also pose a viable option for problems in $\mathcal{P}$ with long practical runtimes.

Consider maximization (minimization) problem Π with objective function $w : \mathcal{S}_\Pi \mapsto \mathbb{R}$. We say that an approximation algorithm $\mathcal{A}_\Pi$ has an *approximation ratio* $\rho \in [1, \infty)$ iff it finds a solution $\mathcal{A}_\Pi(I)$ for any feasible problem instance $I \in \mathcal{I}_\Pi$ such that

$$\frac{w(\mathrm{OPT}_\Pi(I))}{w(\mathcal{A}_\Pi(I))} \leq \rho \qquad \left(\frac{w(\mathcal{A}_\Pi(I))}{w(\mathrm{OPT}_\Pi(I))} \leq \rho\right)$$

holds. OPT_Π denotes a theoretical algorithm that computes an optimal solution for any instance $I \in \mathcal{I}_\Pi$. Note that there exists another common definition in the literature with $\rho \in (0, 1]$ for maximization problems that requires slightly different formula.

Complexity Classes. We classify problems with respect to the running time $T(n, \epsilon)$ and the approximation ratio $\rho(n, \epsilon)$ of the approximation algorithms they permit, with n the size of the problem instance and $\epsilon > 0$ some approximation factor. A problem Π belongs to class $\mathcal{APX}$ if there exists an algorithm that solves Π with a running time polynomial in n and with a constant approximation ratio. We say an algorithm is an

approximation scheme for an optimization problem if it has an approximation ratio $\rho = (1 + \epsilon)$. If it has a polynomial time complexity in n, we call it a *polynomial-time approximation scheme* (PTAS). If we can write the time complexity as $f(1/\epsilon) \cdot p(n)$, with f an arbitrary function and p a polynom, we speak of an *efficient polynomial-time approximation scheme* (EPTAS). If the running time is polynomial in n as well as in $1/\epsilon$, the algorithm is a *fully polynomial-time approximation scheme* (FPTAS). This classification is due to [GJ78], with [CT97] later introducing EPTAS.

Problems that permit these approximation schemes belong to the classes $\mathcal{PTAS}$, $\mathcal{EPTAS}$, or $\mathcal{FPTAS}$. With $\mathcal{PO}$ and $\mathcal{NPO}$ denoting the optimization problems in $\mathcal{P}$ and $\mathcal{NP}$, respectively, we have $\mathcal{PO} \subseteq \mathcal{FPTAS} \subseteq \mathcal{EPTAS} \subseteq \mathcal{PTAS} \subseteq \mathcal{APX} \subseteq \mathcal{NPO}$. The subset relations are strict unless $\mathcal{P} = \mathcal{NP}$. "In a very technical sense, an FPTAS is the best one can hope for an $\mathcal{NP}$-hard optimization problem" [Vaz02], though in practice an EPTAS already works well.

2.2 Graph Theory

A graph is an abstract entity that models objects and the relations between them. It is a common tool in mathematics and computer sciences to describe and to formally define a problem. In the following, we give a formal definition of graphs and introduce several basic graph algorithms. For more details on graph theory and graph algorithms, we refer to text books [MS08, CLRS09], as well as for a general overview on algorithms and data structures.

2.2.1 Definitions

Formally, we define a *graph* $G = (V, E)$ as a set of *nodes* V and a relation $E \subseteq V \times V$ on them. We denote the ordered set $(s, t) \in E$ as *edge* with *source* s and *target* t. Accordingly, E is the set of edges of G. We call this graph *directed*. An edge does not have to be unique. In this case, we speak of *parallel* edges. If there exists a *reverse* edge $\overline{(s, t)} = (t, s) \in E$ for each edge $(s, t) \in E$, we say the graph is *bidirected*. If the edge direction is irrelevant in a bidirected graph, we denote an edge by an unordered set $\{s, t\}$ and omit the reverse edge. The respective graph is called *undirected*. In our context, V and E are always finite. We speak of a *finite* graph. The *reverse graph* $\overline{G} = (V, \overline{E})$ to a directed graph G consists of the same set of nodes and of the set of reverse edges $\overline{E} = \{\overline{e} \mid e \in E\}$.

Nodes and edges can be augmented by further attributes. For example, we often associate an edge with some non-negative *cost* $c \in \mathbb{N}_0$. In our context, this could indicate the distance between the two sensor nodes or the required energy for communication between them. For easier reading, attribute symbols also denote the respective mapping functions, e.g. $c : E \mapsto \mathbb{N}_0$ maps edges to their cost values. We call it *edge cost function*—or *metric*—of G. We may further abbreviate $c(u, v)$ as costs of edge $(u, v) \in E$.

A *path* $p_{s,t} = \langle (s = v_0, v_1), \ldots, (v_{k-1}, t = v_k) \rangle = \langle e_1, \ldots, e_k \rangle$ with $v_i \in V$ and $e_i \in E$ is a sequence of edges such that the target of each edge and the source of the following edge are the same. We may abbreviate by only writing nodes, i.e. $\langle s, \ldots, t \rangle$. The path describes a connection or transitive relation between nodes s and t. We say that t is *reachable* from s. The cardinality of the sequence is called *hop count* $h(p_{s,t}) = |p_{s,t}|$. We may write $h(s,t)$ for short. If t is not reachable from s, we set $h(p_{s,t}) = \infty$. The *length*—or *cost*—of path $p_{s,t}$ is given by the sum of all edge costs $c(p_{s,t}) = \sum_{i=1}^{k} c(e_i)$. Again, we may abbreviate by $c(s,t)$. A path of minimum cost between two nodes $s, t \in V$ is called *shortest path* $P_{s,t}$. It is not unique in general. The associated cost is called *shortest path distance* $d(s,t) = c(P_{s,t})$. A concatenation of two shortest paths $P_{s,v}$, $P_{v,t}$ is denoted by $P_{s,v,t}$. We similarly write $p_{s,v,t}$ for general paths.

The *neighborhood* N of a node v encompasses all nodes that are "close" to v in some sense. The k-hop neighborhood $N_k(v) \subseteq V$ is a subset of V that contains all nodes that are reachable from V by a path of hop length at most k. This includes node v. We write $N_{k \setminus l}(v) = N_k(v) \setminus N_l(v)$ as an abbreviation.

An *embedding* $\mathbf{p} : V \mapsto \mathbb{R}^k$ assigns a position $\mathbf{p}(v)$ in a k-dimensional space to each node $v \in V$. If there exists a two-dimensional embedding $\mathbf{p}$ with $||\mathbf{p}(u) - \mathbf{p}(v)|| \leq 1 \Leftrightarrow (u,v) \in E$ for all nodes $u, v \in V$, we call $G = (V, E)$ a *unit disk graph* (UDG).

2.2.2 Graph Algorithms

The most common techniques for graph exploration are *breath first search* (BFS) and *depth first search* (DFS). In each technique, the search starts at some node and recursively considers all of its neighbors that have not been seen so far. A BFS considers all neighbors of a node first before continuing with their neighbors. In a DFS, the algorithm directly descends to a neighbor of a node before considering the remaining neighbors of this node. Both algorithms can be modelled with a queue data structure, a BFS inserts and removes nodes at opposite ends of the queue, a DFS inserts and removes at the same side. The time complexity of both algorithms is in $\mathcal{O}(|V| + |E|)$.

The *Floyd-Warshall algorithm* [Flo62] offers a simple means to find shortest path distances between all pairs of nodes in a graph. A distance matrix $\mathbf{D}$ is initialized with $\mathbf{D}_{v,v} = 0$ for all nodes $v \in V$, $\mathbf{D}_{u,v} = c(u,v)$ for all edges $(u,v) \in E$, and $\mathbf{D}_{u,v} = \infty$ otherwise. The algorithm runs in $|V|$ iterations. In iteration i, $\mathbf{D}_{u,v}$ is replaced with $\mathbf{D}_{u,i} + \mathbf{D}_{i,v}$ if the path over node i is shorter than the current best one. The basic approach has a cubic time complexity, i.e. it runs in $\mathcal{O}(|V|^3)$ time.

2.3 Mathematical Tools

Next, we consider two mathematical tools that we apply in the following chapters. One is a very general method for solving optimization problems. The other one is a more specialized technique, originally used for visualizing the dissimilarities in a data set.

2.3.1 Mathematical Programming

When speaking of *mathematical programming* in general, we refer to a whole class of mathematical optimization problems: Given n variables written as n-dimensional vector $\mathbf{x} \in P$ over some domain P and an *objective function* $f : P \mapsto \mathbb{R}$, we search for a value of $\mathbf{x}$ that maximizes (or minimizes) the value of $f(\mathbf{x})$. For additional details on this subject, we refer to texts books [Sch89, BT97]. The following brief introduction uses elements of both works, historical details are taken from [Dan63, Sch89].

Linear Programming. Linear programming goes back to Fourier in the early 18th century and was later revisited in the 1940s by several authors. The mathematical foundations were laid by von Neumann [vN47].

A *linear program* (LP) is an optimization problem. It can be written in the form

$$\begin{array}{lll} \max & \mathbf{c}^\intercal \mathbf{x} & \mathbf{c}, \mathbf{x} \in \mathbb{R}^n \\ \text{s.t.} & \mathbf{M}\mathbf{x} \leq \mathbf{b} & \mathbf{M} \in \mathbb{R}^{m \times n}, \mathbf{b} \in \mathbb{R}^m \\ & \mathbf{x} \geq \mathbf{0}\,. & \end{array} \tag{2.1}$$

The goal is to maximize a linear function $f(\mathbf{x}) = \mathbf{c}^\intercal \mathbf{x}$ in n variables that is subject to m linear constraints. Each row of $\mathbf{M}\mathbf{x} \leq \mathbf{b}$ is interpreted as one constraint. All feasible solutions are described by polytope $P = \{\mathbf{x} \in \mathbb{R}^n \mid M\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$, and the desired optimum by $\max_{\mathbf{x} \in P} \mathbf{x}$. If P is empty, the problem is *infeasible*. LPs are solvable in polynomial time in the number of their variables as first shown by Khachiyan's *ellipsoid method* [Kha79]. By now, there exist theoretically more efficient techniques such as the *interior point method* by Karmarkar [Kar84]. Anstreicher [Ans99] gives the currently best time complexity for solving LPs with $\mathcal{O}(\frac{n^3}{\ln n})$. The worst-case exponential time *simplex method* [Dan51] is still of practical use, though, as it offers expected linear running times.

Linear programming theory states that each linear program has a dual formulation [Dan63]. More precisely, the strong duality theorem states that if a linear problem has an optimal solution $\mathbf{x}^*$ then its dual also has an optimal solution $\mathbf{y}^*$ with $\mathbf{c}^\intercal \mathbf{x}^* = \mathbf{b}^\intercal \mathbf{y}^*$. The *dual problem* to the *primal problem* in Equation (2.1) is given by

$$\begin{array}{lll} \min & \mathbf{b}^\intercal \mathbf{y} & \mathbf{b}, \mathbf{y} \in \mathbb{R}^m \\ \text{s.t.} & \mathbf{M}^\intercal \mathbf{y} \geq \mathbf{c} & \mathbf{M} \in \mathbb{R}^{m \times n}, \mathbf{c} \in \mathbb{R}^n \\ & \mathbf{y} \geq \mathbf{0}\,, & \end{array} \tag{2.2}$$

with the cardinality of constraints and variables switched compared to the primal problem. [vN47] and [GKT51] were the first to show that if a problem is feasible, so is its dual, and the solution values of the respective optimal solutions $\mathbf{x}^*, \mathbf{y}^*$ are the same, i.e. $\mathbf{c}^\intercal \mathbf{x}^* = \mathbf{b}^\intercal \mathbf{y}^*$. Transformation between both problems is possible in polynomial time. This implies that both problems are equally hard to solve. The dual problem is used as part of several solving strategies such as the *primal-dual method* [DFF56].

Integer Programming. Integer programming is a generalization of linear programming. When we consider an LP with all (some) variables restricted to integer values, we speak of a (Mixed) Integer Linear Program ((M)ILP). In contrast to linear programming, this class of problems is $\mathcal{NP}$-hard to solve in general [GJ79].

By removing the restriction to integral values of an (M)ILP, we obtain the *LP relaxation* of the problem. The solution to the relaxed problem gives a bound on the optimal objective value of the integral problem. This can be exploited e.g. by exact solvers. Techniques for solving (M)ILPs include the *cutting plane* and the *branch-and-bound* methods, introduced by Gomory [Gom58] and Land and Doig [LD60], respectively. The *branch-and-cut* algorithm [PR91] uses aspects of both methods.

2.3.2 Multidimensional Scaling

Multidimensional scaling (MDS) refers to a collection of techniques aimed at visualizing the dissimilarities between objects in a data set by transforming them to distances that can be represented in a lower-dimensional space. It has its origins in psychophysics. By now, MDS is frequently applied in other fields such as behavioral sciences, statistical analysis, or as a means to estimate coordinates from a set of distance measurements. There exist numerous variants to cope with diverse demands such as metric MDS for dissimilarities that can be represented by some metric, or non-metric MDS if only the ranking of the dissimilarities is important but not their actual value. In this thesis, we focus on classical scaling by Torgerson [Tor52], a special case of metric MDS which assumes dissimilarities to be Euclidean distances and the lower-dimensional space to be Euclidean as well. The following description of classical scaling is based on [BG97]. For a broader overview on the topic, we refer to text books [CC94, BG97].

Classical Scaling. We are given a set of n objects V and pairwise dissimilarities—or distances—$\delta_{i,j} \in \mathbb{R}$ between all objects $i, j \in V$. Objects are assumed to be enumerated from 1 to n. Our goal is to find an embedding $\mathbf{p} : V \mapsto \mathbb{R}^k$ of all objects into an Euclidean space, usually into $\mathbb{R}^2$, that minimizes the quadratic differences

$$\sum_{i,j=1}^{n} \left(\delta_{i,j} - ||\mathbf{p}(i) - \mathbf{p}(j)||\right)^2 \tag{2.3}$$

between dissimilarities $\delta_{i,j}$ and induced distances $||\mathbf{p}(i) - \mathbf{p}(j)||$. The problem can be transformed into finding the k dominant eigenpairs of a matrix. Now, consider a perfect embedding into $\mathbb{R}^k$. This implies $\delta_{i,j} = ||\mathbf{p}(i) - \mathbf{p}(j)||$, $i, j \in V$, which we we can rewrite as $\mathbf{B}_\Delta = \mathbf{E}\mathbf{E}^\intercal$, with $\mathbf{E} = [\mathbf{p}(1), \ldots, \mathbf{p}(n)]^\intercal$ the embedding of V. To determine $\mathbf{B}_\Delta$, we first compute the matrix of squared dissimilarities

$$(\mathbf{\Delta}_{i,j}) = \delta_{i,j}^2, \qquad i, j \in \{1, \ldots, n\} = \mathbf{E}\mathbf{E}^\intercal$$

between all objects and apply *double centering*, i.e. setting the sum of each column and row to zero. This operation removes the indeterminacy of the solution by translating the origin of the embedding to the centroid of all objects. We obtain

$$\mathbf{B}_\Delta = -\tfrac{1}{2}\mathbf{J}\boldsymbol{\Delta}\mathbf{J} = \mathbf{E}\mathbf{E}^\intercal\,,$$

with $\mathbf{J} = \mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\intercal$ the centering matrix and $\mathbf{I} = \mathrm{diag}(1,\ldots,1) \in \mathbb{R}^{n\times n}$ the identity matrix. A matrix decomposition of $\mathbf{B}_\Delta$ yields the desired coordinates. As $\mathbf{B}_\Delta$ is an orthogonal matrix, we can compute its eigendecomposition as

$$\mathbf{B}_\Delta = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^\intercal\,,$$

with $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1,\ldots,\lambda_n)$ the diagonal matrix of eigenvalues and $\mathbf{V} = [\mathbf{v}_1,\ldots,\mathbf{v}_n]$ the matrix of corresponding normalized eigenvectors of $\mathbf{B}_\Delta$. The n-dimensional embedding follows as $\mathbf{E} = \mathbf{V}\boldsymbol{\Lambda}^{\frac{1}{2}}$. To obtain the k-dimensional embedding, we restrict the solution to the k dominant eigenvalues. If we assume eigenvalues to be sorted by descending magnitude, we can write

$$\mathbf{E}_k = \mathbf{V}_k\boldsymbol{\Lambda}_k^{\frac{1}{2}}\,,$$

with embedding $\mathbf{E}_k \in \mathbb{R}^{n\times k}$ and $\mathbf{V}_k = [\mathbf{v}_1,\ldots,\mathbf{v}_k]$, $\boldsymbol{\Lambda}_k = \mathrm{diag}(\lambda_1,\ldots,\lambda_k)$ the matrices of eigenvectors and eigenvalues of $\mathbf{B}_\Delta$, restricted to the k dominant components.

We assess the time complexity of this approach as follows: Problem transformation takes $\mathcal{O}(n^2)$ time. Eigenpairs can be determined one by one with the *power method* [vMPG29] in $\mathcal{O}(n^2)$ time per iteration, with the number of iterations determining the accuracy of the result. Alternatively, [GB08] show how to compute multiple dominant eigenpairs at once. To conclude this section, we give a sample embedding in Figure 2.1.

$\delta_{i,j}$	a	b	c	d	e
a	0	2	3	4	3
b	2	0	4	6	4
c	3	4	0	5	6
d	4	6	5	0	5
e	3	4	6	5	0

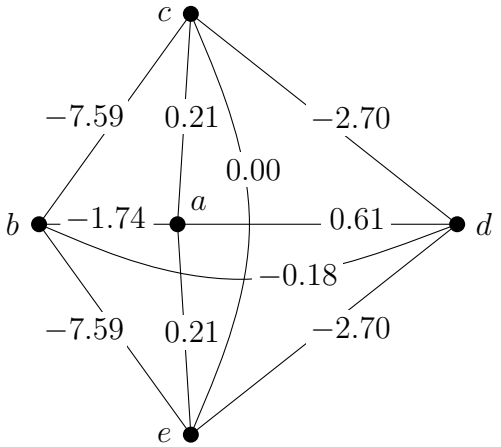


Figure 2.1: *The left side lists pairwise distances $\delta_{i,j}$ between 5 vertices. The right side shows an embedding that minimizes equation (2.3). Edge costs denote the error $(||\mathbf{p}(i) - \mathbf{p}(j)||/\delta_{i,j} - 1)$ between given and embedded distance in percent.*

2.4 Simulational Environment

The simulational studies presented in this thesis are conducted on one of two machines, both running a Linux distribution. Machine A is a consumer-grade machine equipped with a quad-core Intel Core i7-920 CPU and 12 GB of RAM. Machine B comes with four eight-core Intel Xeon E5-4640 CPUs and a total of 512 GB of RAM. Table 2.1 details the respective hardware platforms and software installations.

name	processor	sockets × cores	clock [Ghz]	L1 [kB]	L2 [kB]	L3 [MB]	RAM [GB]
Machine A	Intel Core i7-920	1 × 4	2.67 Ghz	4 × 32	4 × 256	1 × 8	12
Machine B	Intel Xeon E5-4640	4 × 8	2.40 Ghz	32 × 32	32 × 256	4 × 20	512

name	architecture	memory channels	interconnect	distribution / kernel
Machine A	Bloomfield	3 × DDR3-800	1 × 4.8 GT/s QPI	openSUSE 11.3 / 2.6.34
Machine B	Sandy Bridge	4 × DDR3-1600	2 × 8.0 GT/s QPI	Ubuntu 12.04 / 3.2.0-38

Table 2.1: *Hardware platforms and software installations used for our experiments.*

Implementation. Our algorithms are implemented in C++ and compiled with the C++ compiler of the GNU Compiler Collection[1] (GCC), version 4.8.1, with tuning parameters `-std=c++11 -O3 -mtune=native` as well as `-fopenmp -msse4.2` where applicable. We apply the Boost libraries[2], version 1.54.0, for multiple basic algorithms and data structures, the Intel Math Kernel Library[3], version 11.1.0.080, for computing eigenpairs of matrices in MDS, and the Gurobi Optimizer[4], version 5.6.0, for solving LP and ILP problems.

Runtime Measurements. We perform runtime measurements by reading the clock cycle counter available in 64-bit x86 CPUs. Hyperthreading and energy-saving measures are switched off on both machines. In addition, turbo-mode is disabled on Machine B.

[1] `http://gcc.gnu.org/`. Accessed: 2014-08-06.
[2] `http://www.boost.org/`. Accessed: 2014-08-06.
[3] `http://software.intel.com/intel-mkl/`. Accessed: 2014-08-06.
[4] `http://www.gurobi.com/products/gurobi-optimizer/`. Accessed: 2014-08-06.

Chapter 3

3 Lifetime Maximization of Monitoring Sensor Networks

Surveillance. Close observation, especially of a suspected spy or criminal.
— Oxford Dictionary of English

Surveillance in the context of sensor networks usually comprises more civil tasks than described in the dictionary definition above. We speak of a surveilling—or monitoring—sensor network whenever there is some property we want to observe over a longer span of time. This could be the concentration of some pollutant in the air [TYIM05], the seismic movements of the pillars of a bridge [XRC+04], or simply the temperature distribution in an expansive area [MCP+02]. Besides continuous measurements, this also comprises the detection of dynamic events like the outbreak of wildfires [YWM05] or tsunamis [APM05] as well as tracking tasks such as the movement of animal flocks [JOW+02] or as found in traffic guidance systems [KSC06]. A more detailed overview on monitoring applications can be found in [YMG08]. One of the key features of sensor networks is their ability to perform monitoring tasks autonomously. They are able to collect data in inhospitable or even hostile environments without requiring constant human supervision on site. Moreover, with their very small environmental footprint, sensor networks are ideal to monitor and preserve fragile ecosystems.

This chapter focuses on allowing energy-constraint sensor networks to provide their monitoring capabilities for as long as possible. We exploit the redundancy present in the deployed sensor nodes and compute an activation schedule that permits nodes to sleep and conserve energy without impeding the functionality of the network while prolonging its lifetime at the same time. This approach effectively minimizes the energy wasted by the sensor network.

References. The contents of this chapter are partially based on joint work with Peter Sanders [SS10] and discussions with David Steurer who provided an outline of our approximation algorithm in Section 3.4. Wordings of the above publication are used in this thesis.

3.1 Introduction

The ability of sensor networks to provide continuous monitoring is the basis of many applications. Examples include the tasks listed above as well as surveillance tasks, like fencing or intrusion detection, in which we want to be informed when someone or something crosses a designated area. All of these applications require the monitoring to be uninterrupted and without any blind spots to be most effective. However, the unique structure and limitations of sensor networks present us with new and challenging problems. Single nodes are usually only equipped with non-rechargeable batteries. Therefore, energy is a highly limited resource and energy consumption becomes a critical factor in this context. On the other hand, the sensor nodes themselves are usually cheap and available in abundance. This fact can be exploited to counter their inherent limitations. A lot of research concentrates on maximizing the lifetime of monitoring sensor networks under these constraints while guaranteeing complete coverage. The main idea is to activate only a subset of sensor nodes at each moment while the remaining nodes can be in an energy conserving sleep mode.

There exist many variations of the basic problem, though. In the *target monitoring problem*, there is a set of locations that has to be monitored, i.e. during the entire lifetime of the network and for every target t, there has to be an active node with t in its sensing range. In the *area monitoring problem*, every spot in a designated area has to be monitored. Some applications may introduce further degrees of freedom, such as variable sensing ranges, or impose additional constraints, like fault tolerance or requiring the active nodes to form a connected communication network. Moreover, we have to differentiate between distributed algorithms and centralized approaches. While the former can be directly deployed on sensor networks, the latter are usually not applicable in real-life scenarios as costly broadcasts over the entire network would be required. However, we can still use centralized approaches for theoretical studies of the problem or to determine bounds on various network properties, like the maximum lifetime. Moreover, with values like the (approximate) maximum lifetime readily available, we can evaluate the quality of distributed algorithms.

3.1.1 Related Work

We consider the problem of maximizing the lifetime of a sensor network deployed to monitor a given area with a focus on approximation guarantees and exact solvers. There exists a large body of work in this field, each with its own take on the problem. We therefore limit ourselves to introducing the most prominent as well as recent contributions to the field.

Our problem can be divided into two subproblems: The auxiliary *coverage problem* of determining sets of sensor nodes that fulfill certain coverage (and possible other) constraints, and the primary *lifetime problem* of selecting a subset of these node sets

with appropriate durations to maximize the total lifetime of the sensor network while respecting the energy capacities of all nodes. Both problems are usually considered in combination. The coverage problem proves to be more difficult, though, and there exists a lot work only concerned with this subproblem. In the following, we present some general results before discussing centralized and distributed algorithms dealing with the full problem. We conclude our overview by detailing several approaches that focus solely on the coverage problem. Surveys on the general problem are offered by Cardei and Wu [CW06] as well as Wang [Wan11].

General Results. Cardei and Wu [CW06] classify coverage problems into three general groups—area coverage, target coverage, and barrier coverage. In [BCSZ05], Berman et al. argue that area coverage can be reduced to target coverage with at most n^2 targets, given a network of n sensor nodes. However, guaranteeing complete coverage of an area might not be sufficient in a real-life scenario. An underlying application could, for example, require a connected network to ensure communication between all nodes. Zhang and Hou [ZH05] show that if a convex area is covered by a set of sensor nodes, the nodes are connected when their communication ranges r_c are at least twice their sensing ranges r_s. Tian and Georganas extend this result in [TG05]. They prove that if a sensor network is connected, so is any subset of its nodes that covers the same area as the whole network when $r_c \geq 2 \cdot r_s$. Unlike simple area coverage, connected coverage cannot be reduced to covering a set of targets, though, as previously shown by Lu et al. in [LWCL09].

In one dimension, the general problem is reduced to monitoring a border, i.e. barrier coverage. A special case of this problem—all sensor nodes having the same capacity—can be reduced to finding node-disjoint paths in a network. This can be modelled as a maximum flow problem as shown by Frisch [Fri67]. The general case can be considered as minimum cost flow problem according to Steiglitz and Bruno [SB71]. The problem can be augmented to "1.5" dimensions, meaning that the nodes are distributed in an area but we do not want to cover the whole area, just an arbitrary path from one side to the other. This problem can be solved by the same means as in one dimension.

Centralized Setting. We identify two main approaches to the lifetime maximization problem in the centralized setting. One approach focuses on the combinatorial aspects of the problem, while the other one considers the problem as a linear program. The latter approach can be further classified by the strategy used to deal with the exponential number of possible covers.

The first group of algorithms focuses on the combinatorial nature of the problem and considers it as a variant of the set cover problem. The sensor nodes are partitioned into (possibly) disjoint sets, i.e. covers, and activated in sequence for a fixed duration. Slijepcevic and Potkonjak are among the first to study the problem in [SP01], focusing on area monitoring. They model the problem as a set cover variant by imposing the

following constraints: Each sensor node has the same initial energy and is active in at most one cover. The authors provide a heuristic to compute disjoint sets of nodes that each cover the designated area. They start by organizing the area into smaller regions so that each region is coverable by a different set of sensor nodes. Constructing a cover is done iteratively by selecting an uncovered region that is coverable by the smallest amount of remaining nodes and adding one of them to the cover. If multiple nodes are eligible, nodes are favored that cover a larger amount of yet uncovered regions while introducing less redundant coverage. Covers remain active until the batteries of the constituting nodes are depleted. The authors state that their heuristic has a worst-case time complexity of $\mathcal{O}(n^2)$, with n the number of sensor nodes. Approximation guarantees are not given. In [CD05], Cardei and Du consider the same constrained problem but focus on target monitoring. They show the problem to be $\mathcal{NP}$-complete and approximable within a factor of 2 by polynomial-time algorithms. The authors present a heuristic solution by transforming the problem into a maximum flow problem. Time complexities and approximation ratios are not given. Simulations show that their approach yields more disjoint covers than [SP01], though at a higher computational cost. Later in [CTLW05], Cardei et al. drop the restriction on disjoint sets. By allowing sensor nodes to participate in multiple sets, network lifetimes are further improved. The authors provide a proof of $\mathcal{NP}$-completeness, reducing from 3-SAT. This proof is subsequently cited by most of the literature on this subject, even though it does not take into account the geometric structure nor the battery capacities of the problem. We rectify this shortcoming in Section 3.2.3 with our own proof of $\mathcal{NP}$-completeness. Cardei et al. further propose two heuristics to solve the problem. The first one is based on a linear programming relaxation of an ILP formulation of the problem. It has a time complexity of $\mathcal{O}(n^3p^3)$, with n the number of sensor nodes and p an upper bound on the number of set covers. The second one is a greedy heuristic. Single covers are constructed iteratively, similar to [SP01]. In each step, a critical target is selected and an available node with the greatest contribution is chosen to cover it. Once a cover is complete, a fixed amount of energy is subtracted from the contributing nodes and the next cover is computed. The running time of the heuristic is in $\mathcal{O}(dm^2n)$, with n the number of sensor nodes, m the number of targets, and d the minimum number of nodes covering a single target. Approximation guarantees are not given for either of the two approaches. A more recent publication by Deshpande et al. [DKMT11] revisits the problem. In contrast to previous work, the authors do not aim at maximizing the lifetime while guaranteeing coverage. Instead, they maximize the coverage for a given lifetime, i.e. for a given number of node sets. The authors still assume uniform battery capacities, though. Variants with each node being active in multiple sets as well as having each target covered by more than one node are considered. They further study a variant in which each node can only cover a limited number of targets at once. The authors introduce (randomized) approximation algorithms and provide approximation guarantees whenever the considered problem does not permit a polynomial-time solution.

Algorithms belonging to the second group describe the problem as a linear program and apply the Garg-Könemann approach [GK07] to compute a $(1+\epsilon)\cdot f$-approximation in $\mathcal{O}(\frac{1}{\epsilon^2} n \log n \cdot T)$ time. The subproblem of computing a single cover is handled by an f-approximate algorithm in time T. This approach was originally devised by Garg and Könemann to provide faster and simpler algorithms for multi-commodity flow and other fractional packing problems.

Berman et al. discuss area monitoring in [BCSZ05] which expands upon [BCSZ04]. They do not enforce any constraints on node sets or battery capacities as seen for the previous group of algorithms. The authors formalize the problem as *sensor network lifetime problem* and identify the coverage subproblem as a weighted geometric set cover problem. They outline an efficient data structure and algorithm to transform the area coverage task to a target coverage task with at most n^2 targets, n being the number of sensor nodes. This reduces the subproblem to weighted set cover. The authors propose to use a basic greedy approach to solve it and state an approximation ratio of $f = H(n^2)$, $H(k)$ denoting the k-th harmonic number. Using a tighter approximation ratio, we can reduce this to $f = (1 + \log M)$, with M the maximum number of nodes covering a single target. The time complexity is not given. Berman et al. consider further extensions to the basic problem, either only requiring partial coverage, or taking into account communication costs. Finally, they suggest optimizing the duration of each cover with an LP solver. A variation of the problem is considered by Dhawan et al. in [DVZ+06]. They generalize the sensor model by introducing variable sensing ranges that affect the energy consumption. The authors describe a greedy solution for the modified subproblem and obtain the same approximation ratio as given for the original problem. Again, the time complexity is not stated. Zhao and Gurusamy [ZG08] extend the model of Berman et al. [BCSZ05] by requiring all active nodes to be directly or indirectly connected to a dedicated sink with unlimited energy reserves. They describe an algorithm that is a variation of the basic Garg-Könemann approach and achieves the same approximation ratio as the approach by Berman et al. [BCSZ05]. Solving its subproblem requires the computation of shortest path trees. The time complexity of the algorithm is polynomial in the number of nodes and targets. The authors further propose a simpler greedy heuristic based on their approximation algorithm. Erlenbach et al. [EGK11] consider a fault-tolerant variant in which each target has to be covered by at least two nodes. They give a polynomial-time $(6 + \epsilon)$-approximation for their coverage subproblem. The problem is decomposed into smaller blocks, and enumeration techniques are used in combination with dynamic programming to find approximate solutions. The authors apply a geometric shifting strategy by Hochbaum and Maass [HM85] to eliminate boundary effects due to the partitioning. Erlenbach et al. further show how to adapt their results for connected coverage. In [DWW+12], Ding et al. revisit the basic sensor network lifetime problem. They give a polynomial-time $(4 + \epsilon)$-approximation for the subproblem using a similar technique as [EGK11]. The primary problem is solved by a modified Garg-Könemann approach with a slightly different scaling method. The authors do not give credit to [GK07], though.

The final group of algorithms applies delayed column generation, introduced by Dantzig and Wolfe [DW60] in the early days of linear programming, to iteratively add new covers to the considered linear program. Two factors impact the speed of convergence to an optimum solution, the set of covers that is used for initialization, and the algorithm that computes a new cover in each iteration. Depending on whether this algorithm finds optimal solutions, the final result is either exact or approximate. Moreover, the column generation process can be terminated early with a near optimal solution as the convergence often stretches over many iterations. Time complexities are not stated as the considered problem is $\mathcal{NP}$-hard.

The sensor network lifetime problem of Berman et al. [BCSZ05] is picked up by Alfieri et al. in [ABBC07]. They extend the model by considering energy costs for transmitting data to a sink. The column generation approach is initialized with a set of covers computed by the approach in [SP01]. The auxiliary problem of finding appropriate minimum weight covers is formulated as a flow problem and solved optimally with an ILP solver. The authors further describe a simple greedy heuristic. To form a cover, a random subset of non-depleted nodes is chosen and all constraints are checked. This is repeated until a feasible cover is found. Not surprisingly, the simulations find the greedy approach to be much faster but offering worse results. Gu et al. [GLZ07] consider a similar extension to the basic problem, demanding the existence of a data gathering tree and factoring in communication costs. In addition, they require each target to be covered by multiple sensor nodes. Initialization is done by constructing a set of random but feasible covers. The subproblem is solved with integer linear programming. The authors further describe a heuristic to stop the algorithm early when the improvement in solution quality stays below a given threshold over several iterations. Their simulations show that the solution converges the more quickly to an optimum, the more initial covers are used. Later in [GJLZ09], Gu et al. revisit the problem and provide a more thorough analysis while dropping the requirement on data gathering trees. The authors further study the effect of their early termination heuristic, but the presented results remain unconvincing. The following publication [GZJL11] by Gu et al. focuses on the basic problem. They describe how to compute an upper bound on the maximum network lifetime by considering a relaxed problem in which each target is only required to be covered by one node on average. The algorithm terminates when the current solution is within a $(1-\epsilon)$ fraction of this bound. The authors further describe a linear programming relaxation of the ILP subproblem and how to derive a feasible integer solution from the solution of the relaxed problem. Average runtimes improve by one third while results remain optimal up to a factor of $(1-\epsilon)$. Luo et al. consider the basic problem as one example to demonstrate their technique for solving large LPs in [LGR09]. Their main contribution is an efficient method for generating new columns via enumeration that yields a considerable speed-up. The approach works best if the considered problem is not too regular as this allows to significantly restrict the complete enumeration process. In [Des11], Deschinkel also studies the basic sensor network lifetime problem. She proposes to solve the auxiliary problem either exactly

using an ILP formulation or with a simple greedy heuristic that iteratively covers each target with the most beneficial of the still available nodes. Her simulations show that a combination of both approaches works best, with the greedy heuristic used first until no more feasible covers are found and then switching to the ILP solver. Initialization is simply done by ten random but minimal covers. In three publications, Raiconi and a group of co-authors extend the basic problem in various directions. In [RG11], they require the covers to be connected. They develop a greedy heuristic based on [CTLW05] and embed it into a GRASP scheme (greedy random adaptive search procedure [FR89]). Both approaches can be used either stand-alone or to initialize a column generation approach. Simulations show that the greedy heuristic is faster, while GRASP yields better results. The authors further suggest solving the ILP subproblem repeatedly with random weights to generate initial covers. The following work [CdDR12] considers variable sensing ranges and applies a similar combination of greedy heuristic and local search scheme. In [GR13], the authors study the effects of allowing each cover to ignore a fraction $(1 - \alpha)$ of the targets. The basic tools to solve the problem remain the same. Their results show that we can greatly improve network lifetime and average target coverage time even for small values of α. The work of Rossi et al. [RSS12] considers adjustable sensing ranges. They are the first to apply a genetic algorithm to solve the subproblem. This decreases average runtimes by a considerable amount. However, they potentially lose a lot of performance by initializing the column generation with only a single cover in which all nodes are active at their maximum sensing range. The subsequent publication by Castaño et al. [CRSV13] considers connected coverage and partial coverage. The authors propose a GRASP and VNS heuristic (variable neighborhood search [MH97]) to solve the auxiliary problems. Their simulations show that GRASP quickly provides good covers, while VNS yields better overall solutions. The authors further discuss a multi-phase approach that applies GRASP first, followed by VNS. This combination offers a faster convergence than VNS alone while usually yielding even better results. If optimal solutions are required, an exact ILP solver can be applied after VNS finds no more feasible covers. In comparisons to [GR13], Castaño et al. dominate their results in both, runtimes and solution quality.

Distributed Setting. As our approach is not distributed, we only provide a brief overview on distributed algorithms, focusing mainly on the distributed implementations found in the previously cited publications. Following [Dha12], we classify distributed algorithms into greedy, randomized, and other approaches.

The distributed algorithms presented in [SP01, BCSZ05, LWCL09] all construct covers in a greedy fashion by selecting nodes according to some priority rule. Among them, only Slijepcevic and Potkonjak [SP01] consider area coverage. Their distributed approach works similar to the previously discussed centralized method. However, before each node can perform its computation independently, coverage information has to be distributed over the network. The procedure suggested by Berman et al. [BCSZ05]

tries to balance the load evenly between all nodes. Nodes broadcast their remaining energy and the targets they cover to their direct neighbors. A node decides to go offline if there is another node with more remaining energy that also covers its targets. This process is repeated as needed. Lu et al. [LWCL09] describe two algorithms for connected coverage. One starts by constructing a virtual (communication) backbone before adding nodes to ensure coverage. The other approach considers coverage first. Sensing ranges are adjusted as needed to maximize lifetime. Both approaches proceed in rounds of equal duration, computing a new cover at the start of each round.

All of the following approaches introduce some form of randomization. Tian and Georganas describe an algorithm for area coverage in [TG02]. Nodes decide periodically whether to switch themselves off according to a coverage-based eligibility rule. Coverage is preserved by a randomized delay before nodes switch themselves off. No two nodes leave the cover at the same time, preventing the emergence of coverage holes. Zhang and Hou [ZH05] derive a set of optimality conditions under which a subset of sensor nodes can be chosen to completely cover an area. Based on these conditions, the authors describe a randomized algorithm that maintains connectivity. In each round a new cover is computed and activated for a fixed duration. One node is randomly chosen and two more nodes are selected in relation to the first one. This is repeated until a full cover is found. The distributed algorithm by Alfieri et al. [ABBC07] works in rounds of variable duration. In each round, nodes activate themselves at random. They verify whether the desired target coverage is achieved and whether all of them can reach the designated sink. If this is not possible, the node set is discarded. Otherwise, the nodes remain active until the verified conditions no longer hold.

The remaining approaches belong to the "other" category. Gupta et al. [GZDG06] describe two algorithms that guarantee connected coverage. One is a distributed variant of their approximation algorithm and considers paths of nodes. The other is based on node priorities. It offers a lower communication overhead but no guarantees on the size of the constructed cover. The authors only consider computing single covers. To obtain an activation schedule one has to repeat the algorithm when nodes become depleted. In [ZG08], Zhao and Gurusamy show how to implement their greedy heuristic for connected coverage in a distributed manner. Each node determines its profit in terms of uncovered targets it can cover and communication costs to the sink. Locally maximal nodes declare themselves to be part of the current cover and transmit this information to the sink. This is repeated until all targets are covered. The sink then broadcasts the duration until a new cover is selected. Special care has to be taken if nodes on the transmission path to the sink do not have enough energy reserves. In [Dha12], Dhawan introduces the notion of lifetime dependency graphs to model overlapping covers and upper bounds on their combined lifetime. In each round of his proposed heuristic, every node constructs a lifetime dependency graph based on local neighborhood information and deduces a best order of local covers to activate. To deal with the exponential number of covers and high runtimes, he suggests considering the linear-sized graph of equivalent covers and to sample a subset of all covers.

Coverage Problem. We conclude our overview with several results focusing on the coverage problem. In [GZDG06], Gupta et al. describe an $\mathcal{O}(\log n)$-approximation algorithm for connected coverage of minimum cardinality, with n the number of sensor nodes. Their greedy approach iteratively constructs paths to nodes whose sensing areas overlap the currently covered area and adds the most beneficial path to the current cover. The algorithm runs in polynomial time. Funke et al. [FKK+07] improve upon these results with various polynomial-time, constant-factor approximation schemes for minimum connected area coverage and a PTAS when covering targets. Their algorithms are based on grid placement strategies. The authors further show how to guarantee connectivity when sensing ranges are equal to communication ranges.

As the weighted dominating set problem can be reduced to weighted set cover at the same approximation ratio, see e.g. [Vaz02], we briefly discuss it here. The most recent results obtain a $(4+\epsilon)$-approximation in polynomial time. This is shown independently in [EM09, ZWX+11] using partitionings of the area in combination with geometric shifting [HM85]. The respective subproblems are solved by dynamic programming techniques. Erlenbach and Mihalák [EM09] introduce an approach mimicking a sweep-line algorithm with multiple sweep lines, while Zou et al. [ZWX+11] compute chromatic covers. Both results can be augmented to yield connected node sets using node-weighted Steiner trees. However, the respective approximation ratios become worse.

3.1.2 Contribution

This chapter studies the problem of maximizing the lifetime of a sensor network deployed to monitor a given area under the assumption that the monitoring task consumes the most amount of energy. At first, we show the problem to be $\mathcal{NP}$-complete and provide a proof that takes into account its geometric structure, unlike the previous work by Cardei et al. [CTLW05]. We present an efficient polynomial-time approximation scheme (EPTAS) to solve large problem instances and discuss its adaption to monitoring single targets. The algorithm offers better approximation guarantees along with a better asymptotic time complexity than any previous approach.

As our approximation algorithm needs to solve smaller instances of the same problem as a subtask, we study how to optimally solve small to medium-sized problem instances. We propose an approach based on delayed column generation in combination with an efficient initialization step. In simulations, we compare ourselves to other techniques and study the runtime behavior on various network instances. We find that our approach converges much faster to an optimum solution than when using any of the previously suggested initialization methods. Our simulations further show that the problem becomes more difficult to solve the more regular it gets.

We further discuss a post-optimization based on the traveling salesperson problem that can minimize the number of node state changes if the energy consumption of node activations and deactivations becomes significant. Our solution is generic and can be applied to any lifetime maximization algorithm that generates a set of node covers.

3.2 Model and Problem Definition

Before going into the details of our algorithms, we need to introduce our sensor network model and the notations used throughout this chapter. We formally define our problem and provide alternate problem formulations as well as a proof of $\mathcal{NP}$-completeness.

3.2.1 Network Model

We consider a sensor network consisting of n nodes $v_i \in V$, with $i \in \{1, \ldots, n\}$. Each node $v = (\mathbf{v}, b, r)$ is described by three attributes, its location in the plane $\mathbf{v} \in \mathbb{R}^2$, its battery capacity $b \in \mathbb{R}_+$, and its maximum sensing range $r \in \mathbb{R}_+$. We assume sensor nodes to monitor circular areas of radius r around their position. Communication between nodes and thus network connectivity is not considered.

Node Distribution. Our principal strategy, *random placement*, distributes sensor nodes uniformly at random in the designated area. We further consider a regular node distribution with the *grid placement* strategy. Here, nodes are located at the intersections of a quadratic grid. A variation thereof, *perturbed grid placement*, allows nodes to randomly deviate from these positions by a small amount.

Energy Consumption. When active, the energy consumption of a sensor node is constant over time and independent of small changes in the sensing range. We assume the energy consumption to be negligible when the node is not actively monitoring its surroundings. We further assume that the costs for communication, processing, and state changes, i.e. from active to sleeping and back, are proportional to the monitoring costs. This allows us to incorporate these costs implicitly by using *effective* monitoring costs that are a linear combination of all costs. We lift our assumption on the cost of node state changes in Section 3.5.

For convenience, we normalize all quantities, i.e. a sensor node consumes one unit of energy per unit of time, and battery capacities are specified in units of energy. This implies that a node with battery capacity b can be active for b units of time.

Coverage. We say that a sensor node $v = (\mathbf{v}, b, r) \in V$ *covers* a region R if a disk centered at position $\mathbf{v}$ with radius r contains region R completely. If the disk only intersects said region, we say v *intersects*—or overlaps—region R.

Now, let an *area* be a connected or unconnected region with a description complexity linear in n. Given an area A, a set of sensor nodes $\mathbf{c} \subseteq V$ is called a *cover* of A, if the area is contained in the union of disks centered at each sensor node with radii corresponding to the sensing ranges of the respective nodes. In particular, the set of all sensor nodes V has to be a cover of A if any cover exists at all. We denote the set of all possible covers by $\mathcal{C}$. Its size is exponential in the number of nodes, i.e. $|\mathcal{C}| = \mathcal{O}(2^n)$.

3.2.2 Problem Definition

Given a set of sensor nodes and a designated area, we want to determine an activation schedule for each node so that we can monitor the area for as long as possible without gaps before this becomes impossible due to node failures because of empty batteries. Definition 3.1 formalizes this problem.

Definition 3.1 (Sensor Network Lifetime Problem [BCSZ05]—SNLP)**.** *Given a sensor network V and an area A as specified above, find an activation schedule $(\mathbf{C}, \mathbf{t})$ for area A with maximum duration $T_{\text{opt}}\langle V, A\rangle$ so that the active time of any node $v_i \in V$ does not exceed its battery capacity b_i.*

We denote a problem instance of SNLP by the tuple (V, A), with V the considered sensor network and A the area to be monitored. In the special case of a uniform, fixed maximum sensing range R for all nodes, i.e. $r_i = R, i \in \{1, \ldots, n\}$, we write (V, A, R). A solution to (V, A) is of the form $(\mathbf{C}, \mathbf{t})$. Such a *schedule* comprises a set of m covers $\mathbf{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_m\} \subseteq \mathcal{C}$ of area A and a corresponding set of durations $\mathbf{t} = \{t_1, \ldots, t_m\}$. Applying a schedule implies activating each cover iteratively for its corresponding duration. If a cover is active, all of its sensor nodes are active and all other nodes are sleeping. We call a schedule, i.e. a solution, *feasible* if it respects the limited battery capacity b_i of each node $v_i \in V$, i.e.

$$\sum_{j:v_i \in \mathbf{c}_j} t_j \leq b_i, \qquad \forall v_i \in V\,. \tag{3.1}$$

We abbreviate the duration $\sum_{j=1}^{m} t_j$ of a schedule $(\mathbf{C}, \mathbf{t})$ for problem instance (V, A) by $T\langle V, A\rangle$. We refer to this duration as *lifetime* of the sensor network. The lifetime of an optimum solution is denoted by $T_{\text{opt}}\langle V, A\rangle$. Similarly, we write $T\langle V, A, R\rangle$ and $T_{\text{opt}}\langle V, A, R\rangle$ for instances with uniform sensing ranges. For easier reading, we may omit writing out area A and normalize distances to one maximum sensing range.

Linear Programming Formulation. To complement the combinatorial, set-based view on the sensor network lifetime problem given above, we introduce an equivalent formulation based on linear programming. Here, sets become vectors, and sets of sets become matrices. To avoid confusion, we apply the same or similar notations for both representations of the problem.

We reformulate our problem as a packing linear program in which we want to

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{m} \mathbf{t}_j\,, && && \text{(lifetime)}\\
\text{subject to} \quad & \sum_{j=1}^{m} \mathcal{C}_{ij}\mathbf{t}_j && \leq \mathbf{b}_i,\ i \in \{1, \ldots, n\}\,, && \text{(capacity constraints)}\\
& \mathbf{t} && \geq \mathbf{0}\,, && \text{(sanity check)}
\end{aligned}$$

with $\mathbf{t} \in \mathbb{R}_+^m$, $\mathbf{b} \in \mathbb{R}_+^n$, and $\mathcal{C} \in \mathbb{Z}_2^{m \times n}$. Vector $\mathbf{b}$ and matrix $\mathcal{C}$ are (implicitly) given by problem instance (V, A) of SNLP, while vector $\mathbf{t}$ is an open variable of the problem. Our linear program can also be written in a more convenient, shorthand form as

$$\max \{\mathbf{1}^\intercal \mathbf{t} \mid \mathcal{C}\mathbf{t} \leq \mathbf{b},\ \mathbf{t} \geq \mathbf{0}\}, \qquad \mathbf{t} \in \mathbb{R}_+^m, \mathbf{b} \in \mathbb{R}_+^n, \mathcal{C} \in \mathbb{Z}_2^{m \times n}. \tag{3.2}$$

We model a cover $\mathbf{c}$ as binary column vector $\mathbf{c} \in \mathbb{Z}_2^n$, with entry $\mathbf{c}_i$ denoting whether node v_i is active ($\mathbf{c}_i = 1$) or not ($\mathbf{c}_i = 0$). In other words, $v_i \in / \notin \mathbf{c}$ in the set formulation translates to a binary value in the vector notation. The set of all covers $\mathcal{C}$ corresponds to binary matrix $\mathcal{C} = [\mathbf{c}_1, \ldots, \mathbf{c}_m] \in \mathbb{Z}_2^{m \times n}$. Every column of the matrix conforms to a single cover. The duration of each cover is described by column vector $\mathbf{t} \in \mathbb{R}_+^m$. An entry $\mathbf{t}_j$ corresponds to the duration t_j of cover $\mathbf{c}_j$, $j \in \{1, \ldots, m\}$. The battery capacities of all sensor nodes are compiled in column vector $\mathbf{b} \in \mathbb{R}_+^n$, with each entry $\mathbf{b}_i$ denoting the battery capacity b_i of a node v_i, $i \in \{1, \ldots, n\}$. Note that the geometric structure of the problem instance is implicitly represented by matrix $\mathcal{C}$.

The result of the maximization problem conforms to the optimum lifetime $T_{\text{opt}}\langle V, A\rangle$ of a problem instance (V, A) of SNLP. The corresponding schedule is given by $(\mathcal{C}, \mathbf{t})$, with $\mathbf{t}$ the argument of the maximum. As $\mathbf{t}$ usually contains many zero valued entries, we may want to condense the schedule to the actually contributing covers by removing all entries $\mathbf{c}_j \in \mathcal{C}$ and $\mathbf{t}_j \in \mathbf{t}$ for which $\mathbf{t}_j = 0$.

Given this linear programming formulation, we can now make use of the full range of techniques available for linear programs to solve the sensor network lifetime problem. This is discussed in more detail in Section 3.4.

Alternative Problem Interpretation. Assuming uniform fixed sensing ranges is a common practice in theoretical publications on the subject at hand as it simplifies problem descriptions and analyses. We have to admit, though, that this is not a reasonable assumption for real-life scenarios. Sensor nodes rarely offer exact circular monitoring ranges of uniform radii. Therefore, we introduce a reinterpretation of the sensor network lifetime problem that does not require sensing ranges at all.

Consider the task of continuously taking measurements of a designated area A at a given minimum *resolution* R. This implies that each point of the considered area has to be within distance R of a position where measurements are taken, i.e. from a sensor node. The problem corresponds to SNLP with nodes covering circular regions of uniform radii. If we introduce variable measurement resolutions, this formulation further covers the case of SNLP with non-uniform sensing ranges.

Overall, we are convinced that our reinterpretation of the problem offers a much more realistic view on the sensor network lifetime problem and shows that even a very simplistic theoretical model can be of actual relevance in real-life scenarios. However, for the sake of consistency to the previous work, we follow the original problem interpretation for the remainder of this chapter.

3.2.3 Proof of $\mathcal{NP}$-Completeness

We now take a look at the hardness of the sensor network lifetime problem. It has been shown to be $\mathcal{NP}$-complete before by Cardei et al. in [CTLW05]. However, we are convinced that a novel proof is necessary as the previous one by Cardei et al. does not take into account the geometric structure of the problem. Geometric variants of $\mathcal{NP}$-hard problems can have a vastly different structure than the general problem and are often easier to solve. Thus, by ignoring this additional information, the authors of the previous proof essentially considered a different, potentially much more difficult problem. Moreover, Cardei et al. require battery capacities to be uniform, a restriction we do not make in our proof.

A short outline of our proof follows: We introduce a geometric problem and prove it to be $\mathcal{NP}$-complete. Using linear programming, we show that SNLP is equally hard to solve as a derived problem. We then prove the $\mathcal{NP}$-completeness of SNLP by showing that the geometric problem and the derived problem are equivalent.

Geometric Problem. We begin by recapitulating a well-known problem, weighted minimum dominating set, and some of its properties. Thereafter, we show a connection between dominating sets and area covers to aid us in the subsequent proofs.

Definition 3.2 (Weighted Minimum Dominating Set—wMDS)**.** *A dominating set in a graph $G = (V, E)$ is a subset $D \subseteq V$ so that every node $v \in V$ is either in D or adjacent to a node in D. In the weighted case, a weight w_v is associated with each node $v \in V$ and a dominating set D of minimum total weight $\sum_{d \in D} w_d$ is requested. The decision variant asks whether a dominating set D with $\sum_{d \in D} w_d < W$ exists.*

Theorem 3.1. *The Minimum Dominating Set (MDS) problem is $\mathcal{NP}$-complete for unit disk graphs [MIH81]. The same holds true for wMDS and unit disk graphs.*

Lemma 3.1. *Given a unit disk graph $G = (V, E)$ and an embedding of G in $\mathbb{R}^2$, a set $D \subseteq V$ is a dominating set iff the set of unit disks centered at the nodes in D covers V.*

Proof. Let the set of unit disks centered at D be a cover of V. Thus, there is at least one node $d \in D$ for each node $v \in V$ of distance 1 or less. By definition of unit disk graphs, an edge (d, v) exists unless $d = v$. Thus, D is a dominating set of G.

Conversely, let D be a dominating set of G. Every node $v \in V$ is either in D or neighboring to a node $d \in D$. By definition of unit disk graphs, the distance between v and d is at most 1. Therefore, the set of unit disks centered at D covers V. □

We are now ready to introduce a geometric problem, weighted minimum geometric disk cover, and show its $\mathcal{NP}$-completeness. It allows us to incorporate the geometric structure of SNLP into our proof of $\mathcal{NP}$-completeness. The problem definition appears to be equivalent to wMDS on unit disk graphs at first glance, but a closer look reveals it to be a more general problem.

Definition 3.3 (Weighted Minimum Geometric Disk Cover—wMGDC)**.** *Given a set of points $P \subset \mathbb{R}^2$ and a set of disks U with arbitrary radii and associated weights w_u, $u \in U$, find a cover $D \subseteq U$ of point set P with minimum total weight $\sum_{d \in D} w_d$. The decision variant asks whether a cover D with total weight $\sum_{d \in D} w_d < W$ exists.*

Theorem 3.2. *The decision variant of wMGDC is $\mathcal{NP}$-complete.*

Proof. $\mathcal{NP}$-hardness is shown by reduction from the decision variant of wMDS. The nodes V in input graph $G = (V, E)$ of wMDS become centers of disks U with radius 1 in wMGDC. The set of nodes doubles as set of points P that has to be covered. Any geometric cover of U computed by wMGDC is a dominating set of V, as shown by Lemma 3.1. As weights in wMGDC correspond to weights in wMDS, a solution of the decision variant of wMGDC is also a solution of the decision variant of wMDS. $\mathcal{NP}$-completeness follows trivially as we can verify in time polynomial in $|P|$ and $|U|$ whether a candidate solution of wMGDC is a cover with total weight W or less. □

Separation Problem. We now consider the linear programming formulation of SNLP and introduce two associated problems to aid us in our proof of $\mathcal{NP}$-completeness. We first give a short overview of the definition and the interpretation of the dual linear program to SNLP.

Definition 3.4 (Dual to SNLP)**.** *Following Section 2.3.1, the dual to the linear program for SNLP* (3.2) *is given by* $\min\{\mathbf{b}^\intercal \mathbf{w} \mid \mathcal{C}^\intercal \mathbf{w} \geq \mathbf{1},\ \mathbf{w} \geq \mathbf{0}\}$*. We read the problem as finding (optimal) weights for all sensor nodes in V so that the weight $\mathbf{c}^\intercal \mathbf{w}$ of each cover $\mathbf{c} \in \mathcal{C}$ is at least* 1 *and the sum of the weighted battery capacities $\mathbf{b}^\intercal \mathbf{w}$ is minimal. The dual variables $\mathbf{w}$ are considered to be node weights, i.e. $\mathbf{w}_i$ is the weight of node $v_i \in V$.*

The *separation problem* to a linear program verifies whether a candidate solution meets all of the constraints of this LP and otherwise provides a counter-example. We specify the separation problem associated with the dual LP to SNLP below and introduce a general property of separation problems that can be used to assess the time complexities of linear programs.

Definition 3.5 (Separation Problem of the Dual to SNLP)**.** *Given a candidate solution consisting of a set of sensor nodes V with associated weights $\mathbf{w}$, decide whether there exists a cover of area A using nodes with total weight less than* 1*. If true, the candidate solution does not meet all constraints. The cover serves as counter-example.*

Theorem 3.3. *The separation problem associated with an LP is polynomially solvable if and only if the corresponding LP is polynomially solvable [GLS81].*

SNLP. Having introduced two auxiliary problems, wMGDC and the separation problem, we can now focus on our main problem, the sensor network lifetime problem. We first show our auxiliary problems to be equivalent before providing a proof of $\mathcal{NP}$-completeness for SNLP with the introduced problems as main ingredients.

Lemma 3.2. *The separation problem of the dual to SNLP is equivalent to wMGDC.*

Proof. As per [BCSZ05], covering an area is equivalent to covering a set of points. Thus, considering Definition 3.3 and Definition 3.5, the separation problem is equivalent to wMGDC with $W = 1$ and sensor nodes substituted by disks at the same positions, with the same weights, and with their radii corresponding to the sensing ranges. □

Theorem 3.4. *The sensor network lifetime problem (SNLP) is $\mathcal{NP}$-complete.*

Proof. The separation problem of the dual to SNLP is equivalent to wMGDC according to Lemma 3.2. Theorem 3.2 shows wMGDC to be $\mathcal{NP}$-complete. Therefore, so is the separation problem. It follows as per Theorem 3.3 that the dual to SNLP is also $\mathcal{NP}$-complete. A solution of the dual to an LP can be transformed into a solution of the LP in polynomial time [Dan63]. Thus, SNLP is $\mathcal{NP}$-complete as claimed. □

This completes our proof. In contrast to the previous proof given by Cardei et al. in [CTLW05], we allow sensor nodes to have varying battery capacities, and we take into account the geometric structure of the problem. By ignoring this structure, Cardei et al. showed the $\mathcal{NP}$-completeness of a different, potentially much more difficult problem. Thus, we claim to have given the first complete proof of $\mathcal{NP}$-completeness for SNLP.

3.3 Approximation Algorithm

As seen in the last section, the sensor network lifetime problem is $\mathcal{NP}$-complete. Thus, to obtain an algorithm that handles arbitrarily large problem instances efficiently, we need to introduce some relaxations to the original problem. We consider increasing sensing ranges by a small amount when applying a discretization technique. Secondly, we allow slightly suboptimal results when building a global solution from the solutions of smaller regions. Finally, we assume uniform maximum sensing ranges. We combine all of these ingredients into an EPTAS and prove its approximation ratio and time complexity. Constant factors are not optimized in our proofs for easier reading. Tighter bounds are discussed at the end of Section 3.3.3.

3.3.1 Discretizing Positions

Rounding continuous values to few discrete values is a common technique of many approximation schemes to obtain simpler problems that can be solved more efficiently. We show how to apply this approach to SNLP. Consider a modified problem with sensor nodes restricted to positions on a grid, i.e. a set of points $\{(\gamma \cdot x, \gamma \cdot y) \mid x, y \in \mathbb{Z}\} \subset \mathbb{R}^2$, with $\gamma \in \mathbb{R}$ the width of the grid. If we allow the maximum sensing range R to increase by a small amount and given an algorithm $\mathcal{A}$ that computes an f-approximate solution for this altered problem, we can find an f-approximate solution for the original problem with only a small computational overhead compared to algorithm $\mathcal{A}$.

We first consider a single sensor node and the area it covers. Corollary 3.1 shows that when moving a sensor node by a small amount and increasing its maximum sensing by the same amount, the node still covers at least the same area as before.

Corollary 3.1. *Consider disk D_1 with center $\mathbf{x} = (x, y)$ and radius r, covering area A, and disk D_2 with center $\mathbf{x} + \mathbf{dx}$, $\mathbf{dx} = (dx, dy)$ and radius $r + dr$, $dr \geq ||\mathbf{dx}||$. Then, disk D_2 also covers area A.*

Proof. We have $||\mathbf{p} - \mathbf{x}|| \leq r$ for each point $\mathbf{p} = (x_p, y_p)$ in A as D_1 covers A. According to the triangle inequality $||\mathbf{p} - (\mathbf{x} + \mathbf{dx})|| \leq ||\mathbf{p} - \mathbf{x}|| + ||\mathbf{dx}|| \leq r + ||\mathbf{dx}||$ holds. Thus, no point $\mathbf{p}$ in A is further away from $\mathbf{x} + \mathbf{dx}$ than $r + ||\mathbf{dx}||$. With $dr \geq ||\mathbf{dx}||$, it follows that D_2 covers A. □

With this corollary shown, we can now formalize our previous claims in Lemma 3.3. The basic approach is summarized by Algorithm 3.1. If multiple sensor nodes are shifted to the same grid position, we replace them by one node with the combined battery capacity. The approach further assumes that small changes in the maximum sensing range R have only a negligible impact on the energy consumption of the sensor nodes. Figure 3.1 depicts the general idea for a single node.

Algorithm 3.1 Approximation algorithm for SNLP

Input: Parameter $\delta \in (0, 1]$, set of sensor nodes V, algorithm $\mathcal{A}$
Output: Set of feasible covers $\mathbf{C}$, set of corresponding durations $\mathbf{t}$

1: $\tilde{V} \leftarrow snapToGrid(V, \delta/2)$ ▷ move each node to nearest point on grid
2: $(\mathbf{C}, \mathbf{t}) \leftarrow \mathcal{A}(\tilde{V}, 1 + \delta/2)$ ▷ solve relaxed problem
3: **return** $(\mathbf{C}, \mathbf{t})$

Lemma 3.3. *Let $\delta \in (0, 1]$. Algorithm 3.1 yields a feasible solution to instance $(V, 1+\delta)$ of SNLP with lifetime $T\langle V, 1 + \delta\rangle \geq f \cdot T_{\text{opt}}\langle V, 1\rangle$. The asymptotic time complexity is $\mathcal{O}(n + t_{\mathcal{A}})$, with $n = |V|$ and $t_{\mathcal{A}}$ the running time of algorithm $\mathcal{A}$.*

Proof. Approximation Guarantee. Consider the problem instance $(V, 1)$. Moving all nodes in V to the nearest position on a grid of width $\gamma = \delta/2$ yields $\tilde{V}$. Each node is shifted by at most $\frac{\sqrt{2}}{2} \cdot \delta/2 < \delta/2$. If we increase the maximum sensing range R by a factor $(1 + \delta/2)$, a cover with respect to $(V, 1)$ is a cover with respect to $(\tilde{V}, 1 + \delta/2)$ as shown by Corollary 3.1. Thus, $T_{\text{opt}}\langle\tilde{V}, 1 + \delta/2\rangle \geq T_{\text{opt}}\langle V, 1\rangle$. Algorithm $\mathcal{A}$ computes a solution to $(\tilde{V}, 1+\delta/2)$ with lifetime $T\langle\tilde{V}, 1+\delta/2\rangle \geq f \cdot T_{\text{opt}}\langle\tilde{V}, 1+\delta/2\rangle \geq f \cdot T_{\text{opt}}\langle V, 1\rangle$. A solution to $(\tilde{V}, 1 + \delta/2)$ is a solution to $(V, 1 + \delta)$ by the same argument as above for problems $(V, 1)$ and $(\tilde{V}, 1 + \delta/2)$. Thus, the solution provided by algorithm $\mathcal{A}$ is a feasible solution for $(V, 1 + \delta)$ with lifetime $T\langle V, 1 + \delta\rangle \geq f \cdot T_{\text{opt}}\langle V, 1\rangle$.

Time Complexity. Relocation of a sensor node to a grid position is done by basic arithmetic operations and requires time $\mathcal{O}(1)$. Thus, the time complexity of shifting all n nodes and performing algorithm $\mathcal{A}$ amounts to $\mathcal{O}(n + t_{\mathcal{A}})$. □

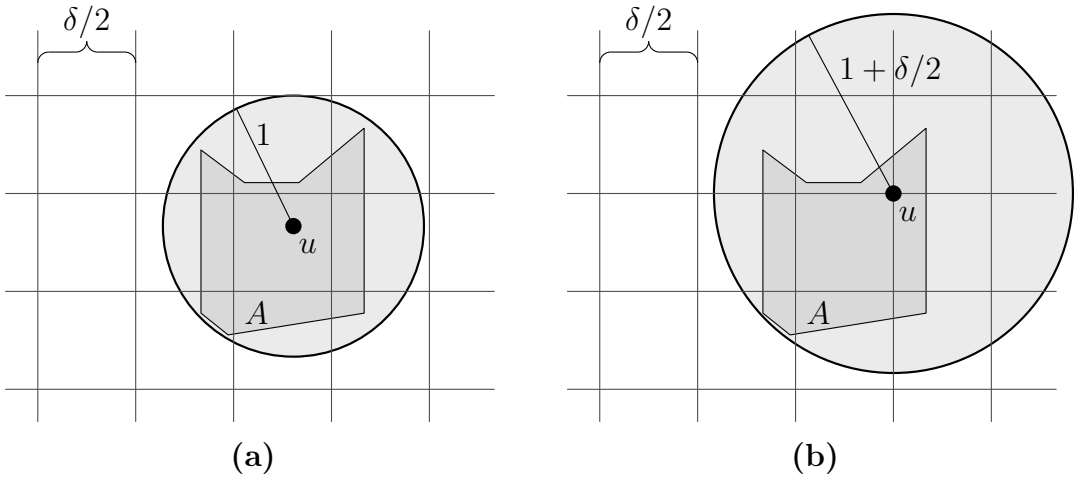


Figure 3.1: *A grid of width $\delta/2$ is shown. Intersections symbolize grid positions. (a) Sensor node u with sensing range $R = 1$ covers area A. (b) Node u is moved to a grid position and its sensing range is increased by $\delta/2$ to still cover area A.*

This completes the first ingredient of our approximation algorithm. In Section 3.3.3, we show how this discretization technique helps us to simplify a subproblem of SNLP so that only a constant number of nodes have to be considered.

3.3.2 Area Partitioning

Geometric shifting is another common technique when dealing with hard problems. Hochbaum and Maas introduced it in [HM85] to formulate polynomial approximation schemes for numerous $\mathcal{NP}$-complete geometric covering and packing problems. The basic idea is to partition the problem into smaller problems that are easier to solve. Their respective solutions are then combined to a solution for the global problem. This is repeated for multiple partitionings to eliminate boundary effects and achieves an almost optimal result. We show how to adapt this technique for our purposes.

Consider an instance $(V, A, 1)$ of SNLP and a partitioning $\mathcal{T}$ of the plane into axis-aligned squares of width k. If we confine our problem to a single square—or tile—T of this partitioning, we only have to deal with covering area $(A \cap T)$ by the subset of sensor nodes in V that are located within T or up to one maximum sensing range away. We write $(V \cap T^+)$ for this subset, with T^+ denoting tile T extended by one maximum sensing range in all directions. Given an algorithm $\mathcal{A}$ that computes an f-approximate solution for a problem restricted to a small (quadratic) area, we can find a solution for each tile of partitioning $\mathcal{T}$ and merge them to a solution of the whole problem. Here, a node has to be active if the solution for any tile requires it to be. This implies that the schedule for each tile can be executed concurrently and independently. Unfortunately, such a solution does not have to be feasible. Consider a sensor node that is required

for the coverage of more than one tile. As the solutions for all tiles are independent, this node might be assigned to be active for longer than its battery capacity allows.

After having considered solving SNLP for a single partitioning, we now describe how to use the solutions for multiple partitionings to avoid infeasible solutions. Consider a set of k partitionings $\boldsymbol{\mathcal{T}} = \{\mathcal{T}^l\}$, $l \in \mathbb{Z}_k$. Each partitioning consists of axis-aligned tiles of width k. Partitioning $\mathcal{T}^{l+1}$ is formed from partitioning $\mathcal{T}^l$ by translation to the top and to the right by one, see Figure 3.2(b). Note that $\mathcal{T}^k = \mathcal{T}^0$. In each partitioning, a sensor node has to be considered for the coverage of at most four tiles. The case of more than one tile only occurs for at most two partitionings if we assume $k > 2$ as depicted in Figure 3.2 and shown in the following corollary.

Corollary 3.2. *Let $\boldsymbol{\mathcal{T}}$ be a set of k partitionings as described above. If $k > 2$, any node is considered for the coverage of at most four tiles of any $\mathcal{T} \in \boldsymbol{\mathcal{T}}$ and for the coverage of more than one tile in at most two partitionings.*

Proof. Consider disk D of radius 1 at position $\mathbf{p} = (x_p, y_p)$. It intersects multiple tiles iff the corner of a tile is within its bounding box $B = [x_p - 1, x_p + 1] \times [y_p - 1, y_p + 1]$. There is at most one tile corner inside the bounding box unless $k \leq 2$, the width of B. Note that k also denotes the width of a tile. Thus, disk D intersects at most four tiles of any partitioning. By construction, each partitioning is shifted to the top and to the right by one compared to the previous one. Therefore, a tile corner remains within B for at most two consecutive partitionings. As at most one tile corner lies within B, disk D intersects multiple tiles for only two partitionings. □

Consider the (potentially infeasible) solution $(\mathbf{C}^l, \mathbf{t}^l)$ to instance $(V, 1)$ of SNLP, constructed from the solutions of each tile of partitioning $\mathcal{T}^l$ as shown above. We have

$$T\langle V, 1\rangle^l = \sum_{j:\mathbf{c}_j^l \in \mathbf{C}^l} t_j^l \geq f \cdot T_{\text{opt}}\langle V, 1\rangle \text{ , and} \tag{3.3}$$

$$\forall v_i \in V : \sum_{j:v_i \in \mathbf{c}_j^l} t_j^l \leq \begin{cases} 4 \cdot b_i & \text{node } v_i \text{ needed by more than one tile,} \\ 1 \cdot b_i & \text{otherwise.} \end{cases} \tag{3.4}$$

Lifetime $T\langle V, 1\rangle^l$ is the sum of the durations $t_j^l \in \mathbf{t}^l$ of its covers $\mathbf{c}_j^l \in \mathbf{C}^l$. It is optimal up to a factor f as algorithm $\mathcal{A}$ provides f-approximate solutions for each tile. Similarly, the active time of each sensor node $v_i \in V$ is the sum of the durations $t_j^l \in \mathbf{t}^l$ of each cover $\mathbf{c}_j^l \in \mathbf{C}^l$ that contains v_i. It exceeds the battery capacity of the node by at most four times as a node is in at most four covers according to Corollary 3.2. We obtain a feasible, almost optimal solution to the full problem by applying the solutions for each partitioning $\mathcal{T}^l \in \boldsymbol{\mathcal{T}}$, $l \in \mathbb{Z}_k$, in succession and scaling the duration of each cover by a constant a, i.e.

$$(\mathbf{C}, \mathbf{t}) = (\bigcup_{l \in \mathbb{Z}_k} \mathbf{C}^l, \bigcup_{l \in \mathbb{Z}_k} a \cdot \mathbf{t}^l) \,. \tag{3.5}$$

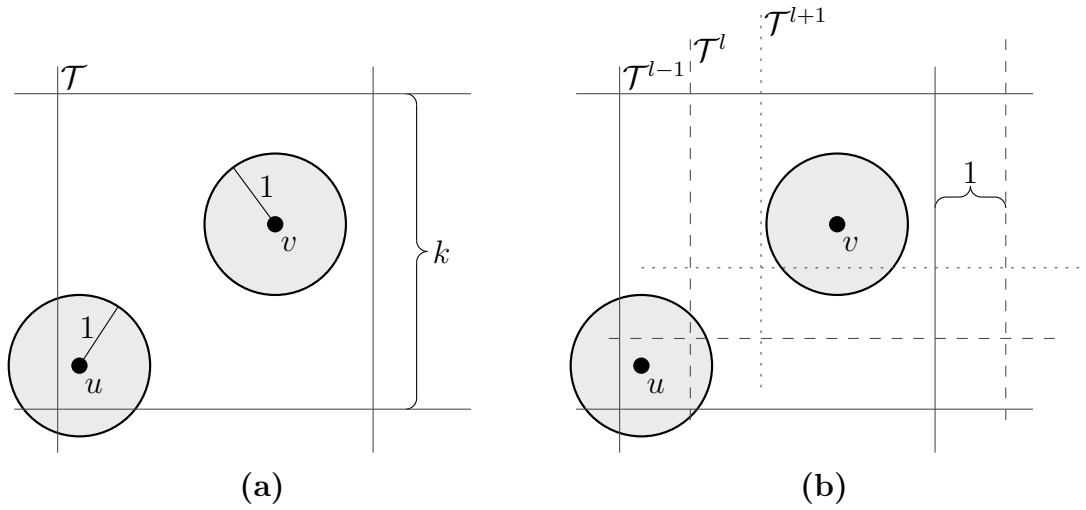


Figure 3.2: *(a) A partitioning $\mathcal{T}$ is depicted. Sensor node u has to be considered for the coverage of four tiles, i.e. the disk centered at u with radius equal to one maximum sensing range intersects four tiles. (b) Three consecutive partitionings $\mathcal{T}^{l-1}$, $\mathcal{T}^{l}$ and $\mathcal{T}^{l+1}$ are shown. In partitionings $\mathcal{T}^{l-1}$ and $\mathcal{T}^{l}$, node u has to be considered for the coverage of four tiles and node v for only one. In partitioning $\mathcal{T}^{l+1}$, the disk at u overlaps just one tile while the one at v intersects two tiles.*

These claims are summarized by Lemma 3.4, which completes the description of the second ingredient for our approximation algorithm.

Lemma 3.4. *Let $k = \lceil 10/\epsilon \rceil$ with $\epsilon \in (0, 1]$. The union $(\mathbf{C}, \mathbf{t})$ of the solutions for each partitioning in Equation (3.5) is a feasible solution to instance $(V, 1)$ of SNLP with lifetime $T\langle V, 1\rangle \geq f \cdot (1 - \epsilon) \cdot T_{\text{opt}}\langle V, 1\rangle$, if $a = (1 - \epsilon)/k$.*

Proof. Approximation Guarantee. The total lifetime $T\langle V, 1\rangle$ is the sum of the lifetimes $T\langle V, 1\rangle^l$ of each partitioning $\mathcal{T}^l$ scaled by $a = (1 - \epsilon)/k$. This sum is bounded below by

$$T\langle V, 1\rangle = \frac{1-\epsilon}{k} \sum_{l \in \mathbb{Z}_k} T\langle V, 1\rangle^l \geq \frac{1-\epsilon}{k} \sum_{l \in \mathbb{Z}_k} f \cdot T_{\text{opt}}\langle V, 1\rangle = f \cdot (1 - \epsilon) \cdot T_{\text{opt}}\langle V, 1\rangle \,,$$

as claimed. The inequality follows by Equation (3.3).

Feasibility. Each sensor node $v_i \in V$ is active for the sum of the durations t_j^l of all covers $\mathbf{c}_j^l \in \mathbf{C}^l$ with $v_i \in \mathbf{c}_j^l$ over all partitionings $\mathcal{T}^l$, with $l \in \mathbb{Z}_k$. This sum is bounded above by

$$\frac{1-\epsilon}{k} \sum_{l \in \mathbb{Z}_k} \sum_{j : v_i \in \mathbf{c}_j^l} t_j^l \leq \frac{1-\epsilon}{k} \Big((k-2) \cdot 1 + 2 \cdot 4 \Big) \cdot b_i \leq b_i \,. \tag{3.6}$$

The first inequality follows by Equation (3.4) and Corollary 3.2. The second inequality follows due to our choice of k. The active time of each node v_i is bounded by its battery capacity b_i. Therefore, the solution is feasible. □

3.3.3 Full Method

With our two main ingredients described in the previous sections, we can now show how to combine these relaxation techniques into an efficient linear-time approximation scheme for the sensor network lifetime problem. We assume the availability of an f-approximate algorithm $\mathcal{A}$ that can compute solutions for instances of SNLP on small (quadratic) areas with sensor nodes restricted to positions on a grid and a running time $t_{\mathcal{A}}$ dependent on the number of nodes. This algorithm combines the restrictions of both algorithms assumed in the previous sections. Now, consider a general instance $(V, 1)$ of SNLP. We construct a feasible solution $(\mathbf{C}, \mathbf{t})$ as in Section 3.3.2 by computing solutions for tiles of partitionings of the plane with algorithm $\mathcal{A}$ and merging them. As algorithm $\mathcal{A}$ assumes the sensor nodes to be in grid positions, we have to use the same approach as in Section 3.3.1 when discretizing node positions and allow the sensing ranges to increase by a small amount. The complete approach is depicted by Algorithm 3.2. It runs in pseudo-linear time and yields a solution that comes arbitrarily close to the maximum network lifetime if we allow sensing ranges to grow by a small amount. Theorem 3.5 summarizes these claims.

Algorithm 3.2 EPTAS for SNLP

Input: Parameters $\delta \in (0, 1]$, $\epsilon \in (0, 1]$, set of sensor nodes V, algorithm $\mathcal{A}$, area A
Output: Set of feasible covers $\mathbf{C}$, set of corresponding durations $\mathbf{t}$

1: $\tilde{V} \leftarrow snapToGrid(V, \delta/2)$ ▷ move each node to nearest point on grid
2: **for all** partitionings $\mathcal{T}^l \in \mathcal{T}$ **do** ▷ loop over all k partitionings
3: **for all** tiles $T \in \mathcal{T}^l$ **do** ▷ and each tile in the partitioning
4: $(\mathbf{C}_T^l, \mathbf{t}_T^l) \leftarrow \mathcal{A}(\tilde{V} \cap T^+, A \cap T, 1 + \delta/2)$ ▷ solve relaxed problem confined to tile T of partitioning $\mathcal{T}^l$
5: **end for**
6: $(\mathbf{C}^l, \mathbf{t}^l) \leftarrow merge(\{(\mathbf{C}_T^l, \mathbf{t}_T^l) \mid T \in \mathcal{T}^l\})$ ▷ combine partial solutions
7: **end for**
8: $(\mathbf{C}, \mathbf{t}) \leftarrow (\cup_{l \in \mathbb{Z}_k} \mathbf{C}^l, (1 - \epsilon)/k \cdot \cup_{l \in \mathbb{Z}_k} \mathbf{t}^l)$ ▷ unite all k solutions
9: **return** $(\mathbf{C}, \mathbf{t})$

Theorem 3.5. *Let $\delta \in (0, 1]$ and $k = \lceil 10/\epsilon \rceil$ with $\epsilon \in (0, 1]$. Algorithm 3.2 computes a feasible solution $(\mathbf{C}, \mathbf{t})$ to instance $(V, 1 + \delta)$ of SNLP with lifetime*

$$T\langle V, 1 + \delta \rangle \geq (1 - \epsilon) \cdot f \cdot T_{\text{opt}}\langle V, 1 \rangle. \tag{3.7}$$

The time complexity of Algorithm 3.2 is pseudo-linearly bounded in $n = |V|$ by

$$\mathcal{O}\Big(n + \tfrac{1}{\epsilon} n \cdot t_{\mathcal{A}}(\mathcal{O}(\tfrac{1}{\delta^2 \epsilon^2}))\Big) = \mathcal{O}(n)\,. \tag{3.8}$$

It is an EPTAS for the sensor network lifetime problem if $f = (1 + \xi), \xi \in \mathbb{R}_+$.

Proof. Feasibility. The feasibility of solution $(\mathbf{C}, \mathbf{t})$ follows directly from the proofs of feasibility in Lemma 3.3 and Lemma 3.4. The latter lemma states that a combination of feasible solutions for small (quadratic) tiles results in a feasible solution. According to the former, a solution for each tile is feasible with sensor nodes relocated to grid positions and their radii slightly increased. Therefore, $(\mathbf{C}, \mathbf{t})$ is feasible.

Approximation Guarantee. By discretizing node positions, we have an approximation guarantee $T\langle V, 1+\delta\rangle \geq f \cdot T_{\text{opt}}\langle V, 1\rangle$ for the solution of each tile according to Lemma 3.3. As stated by Equation (3.3), the same approximation guarantee holds for each solution $(\mathbf{C}^l, \mathbf{t}^l)$ of $(V, 1+\delta)$, with $(\mathbf{C}^l, \mathbf{t}^l)$ the merged solutions of all tiles of partitioning $\mathcal{T}^l$. Combining the solutions of all k partitionings to $(\mathbf{C}, \mathbf{t})$ as in Equation (3.5), introduces an additional factor $(1-\epsilon)$ to the approximation guarantee according to Lemma 3.4. The claimed lifetime follows.

Time Complexity. According to Lemma 3.3, there is an additive computational overhead of $\mathcal{O}(n)$ when discretizing node positions. The solution for each tile can be found in time $t_{\mathcal{A}}(\mathcal{O}(\frac{1}{\delta^2\epsilon^2}))$ as each tile only contains $\lfloor k^2/\delta^2 \rfloor = \mathcal{O}(\frac{1}{\delta^2\epsilon^2})$ distinct grid positions and thus at most as many sensor nodes. The number of nodes to be considered for covering each tile is higher by at most a constant factor as only nodes closer to the tile than one maximum sensing range have to be considered additionally. There are k partitionings and in each partitioning there are at most $4n$ tiles to be considered since, according to Corollary 3.2, each sensor node is required for the coverage of at most four tiles in each partitioning. Thus, solutions for $k \cdot \mathcal{O}(4n) = \mathcal{O}(\frac{1}{\epsilon}n)$ tiles have to be found. Altogether, we obtain the claimed asymptotic running time.

EPTAS. Following Section 2.1.2, an EPTAS requires a $(1+\zeta)$-approximation ratio, $\zeta \in \mathbb{R}_+$, and a time complexity polynomial in the problem size. This polynomial may further depend only multiplicatively on a function of the approximation parameters. For f of the form $(1+\xi), \xi \in \mathbb{R}_+$, we can choose ζ appropriately to satisfy the requirement on the approximation ratio. The requirement on the time complexity holds too, as seen in Equation (3.8). Thus, Algorithm 3.2 is an efficient polynomial-time approximation scheme. For f not of the above form, we only obtain a linear-time approximation algorithm without the required approximation ratio. □

We do not have to be concerned about the time complexity of algorithm $\mathcal{A}$ as it only contributes a constant factor to the running time of Algorithm 3.2—we could apply any algorithm that solves the problem. In an actual implementation, though, we would opt for a preferably efficient one since constant factors matter in practice. However, we still need to take into account the solution quality. Here, the exact solver in Section 3.4 is an interesting choice. It is very fast for an optimal algorithm and as it covers the general problem, it can handle any instance, algorithm $\mathcal{A}$ is expected to solve.

As sensor nodes usually cannot extend their sensing ranges in an actual application, we can take $(1+\delta)$ as the maximum feasible sensing range. Then, our solution would correspond to an approximate solution for an instance with slightly reduced sensing ranges, i.e. $(V, 1-\delta)$. Similar reasonings hold for our alternate problem formulation.

Refinement. The asymptotic time complexity of Algorithm 3.2 can be further refined compared to our results in Theorem 3.5 if we have additional knowledge of the problem structure. We show how to improve upon the number of tiles, $\mathcal{O}(\frac{1}{\epsilon}n)$, that have to be considered for each partitioning before tightening constant factors.

If area A is connected, the disks of radius $R = (1 + \delta)$ representing the sensing area of each node have to be connected as well. Unconnected disks do not contribute to the coverage of area A and can be ignored. Assume there are n disks. They span at most $2R \cdot n/k = \mathcal{O}\big((1 + \delta)\,\epsilon n\big)$ tiles of width k if we arrange them on a horizontal or vertical line touching each other. Thus, the number of tiles to be considered in each partitioning is restricted accordingly. This extreme arrangement requires A to be an axis-aligned rectangular area of infinitesimal width. For extensive areas, the actual number of tiles is much smaller, though.

We can use packing arguments to assess the number of relevant tiles if area A is good-natured as specified below. As per [Wil79], at least $\frac{2}{\sqrt{27}R^2}$ disks of radius R are required to cover an area of size 1. Therefore, n disks of radius $(1 + \delta)$ cover at most $\mathcal{O}\big((1 + \delta)^2\,\epsilon^2 n\big)$ tiles of size k^2. The number of tiles to be considered in each partitioning is restricted accordingly. Area A does not have to be connected for this assessment. However, most disks have to be fully inside of area A, i.e. the greatest extent of area A times $(1 + \delta)$ has to be much smaller than the area of A.

If we can assess the lifetime of the sensor network, the time complexity can be further refined. With T_{lower} a lower bound on the lifetime of the sensor network and b_{max} the maximum capacity over all sensor nodes, $\lceil T_{lower}/b_{max} \rceil$ denotes the minimum number of sensor nodes required to cover any position for the complete lifetime of the network. This bounds the number of nodes that can be active in each partitioning to at most $n/\lceil b_{max}/T_{lower} \rceil$ if a cover exists. We can similarly bound this number from below by $n/\lceil b_{min}/T_{upper} \rceil$, given an upper bound on the network lifetime T_{upper} and the minimum battery capacity b_{min}. Since the number of relevant tiles in each partitioning depends on the number of nodes, it is bounded accordingly.

We did not optimize the constant factors in our proofs for easier reading. However, much tighter bounds are possible without having to change any of our other statements. We can increase the grid width introduced in Section 3.3.1 to $\gamma = \delta/\sqrt{2}$. As discretizing shifts a node by at most $\frac{\sqrt{2}}{2}\gamma$ to the nearest grid position and since we want to retain the same increase $(1 + \delta)$ in the sensing ranges at before, the stated value follows by $\delta \geq 2 \cdot \frac{\sqrt{2}}{2}\gamma$. The minimum width of a (quadratic) tile and thus the number of partitionings as given in Section 3.3.2 is bounded below by $k_{quad} = \lceil 6\frac{1-\epsilon}{\epsilon} \rceil$. This follows from Equation (3.6), i.e. $\frac{1-\epsilon}{k}((k-2) \cdot 1 + 2 \cdot 4) \leq 1$.

In addition, we can consider further partitioning patterns to improve constant factors. For example, using a hexagonal partitioning scheme results in only $k_{hex} = \lceil 4\frac{1-\epsilon}{\epsilon} \rceil$ partitionings that have to be computed. However, each hexagonal tile covers about $\sqrt{3}$ times more space than a quadratic tile. A rough estimation $k_{hex}/k_{quad} \cdot \sqrt{3} > 1$ suggests that actual runtimes are likely to increase with this pattern.

The range of parameter δ is strict only if we consider the maximum sensing range to be normalized to $(1+\delta)$ as described above and compute solutions of problem instances $(V, 1-\delta)$ with sensing ranges reduced by δ. The range of parameter ϵ is always strict.

3.3.4 Target Monitoring

Our approximation algorithm is designed for the problem of monitoring a complete, arbitrary area. In [BCSZ05], Berman et al. describe how to reduce area coverage to target coverage, and many recent studies simply refer to this work and only consider target monitoring. We therefore show in this section how to easily adapt our general approach to monitoring a set of discrete targets.

In principle, Algorithm 3.2 is already capable of handling this task. We only need to switch algorithm $\mathcal{A}$ for another one that solves the target monitoring problem on a small (quadratic) area with sensor nodes restricted to grid positions. However, the time complexity of this algorithm may also depend on the number of targets. We can resolve this problem by discretizing target positions similarly to node positions. Multiple targets occupying the same position can be regarded as one target for the purpose of coverage. If we discretize target positions to the same grid positions as we did for the sensor nodes, only a constant number of $\mathcal{O}(\frac{1}{\delta^2\epsilon^2})$ targets remains to be considered in each tile. Thus, the time complexity of our approximation algorithm no longer depends on the total number of targets.

For all of our statements to remain correct, we have to double the increase in the sensing ranges compared to the area monitoring problem, i.e. $T_{\text{opt}}\langle V, 1+\delta\rangle$ is replaced by $T_{\text{opt}}\langle V, 1+2\delta\rangle$ and solving $(\tilde{V}, 1+\delta/2)$ becomes solving $(\tilde{V}, 1+\delta)$. This is required as now both, sensor nodes and targets, are relocated. Figure 3.3 depicts the general idea in analogy to Figure 3.1 and Corollary 3.1 for area monitoring.

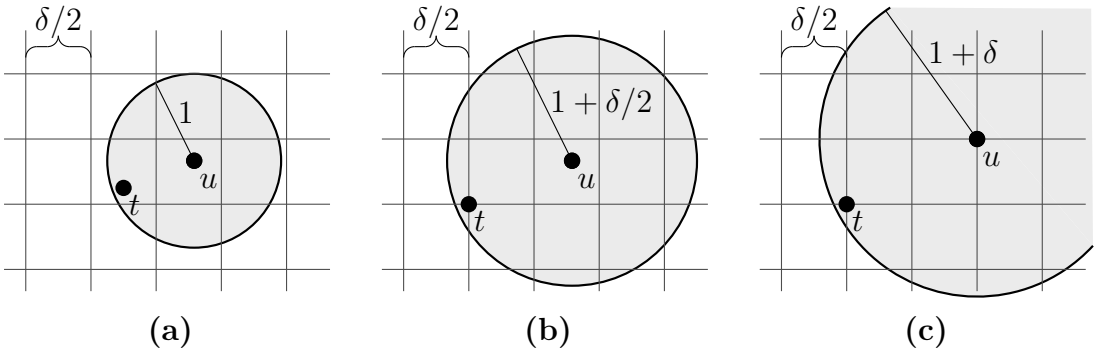


Figure 3.3: *A grid of width $\delta/2$ is shown. (a) Sensor node u with sensing range $R = 1$ covers target t. Both are located in arbitrary positions. (b) Target t is moved to a grid position. The sensing range of sensor node u has to be increased by $\delta/2$ to still cover t. (c) Sensor node u is also moved to a grid position and its sensing range is increased by an additional $\delta/2$ to $(1+\delta)$. It still covers target t.*

3.4 Exact Algorithm

The last section demonstrated how to construct an efficient linear-time approximation scheme for SNLP using dual relaxations. This approximation algorithm required solving a restricted variant of the same problem as a subproblem. The time complexity of the solver for this subproblem is virtually irrelevant as it only contributes a constant factor to the total running time, see Equation (3.8). However, the factor f introduced into the approximation ratio in Equation (3.7) is a possible issue. We therefore study how to solve the subproblem to optimality.

Our focus is on the general problem, though, as being able to solve this problem also allows us to solve the restricted subproblem. Section 3.2.2 showed that the sensor network lifetime problem can be modelled as linear program

$$\max\left\{\mathbf{1}^\intercal\mathbf{t} \mid \mathcal{C}\mathbf{t} \leq \mathbf{b},\ \mathbf{t} \geq \mathbf{0}\right\}, \qquad \mathbf{t} \in \mathbb{R}_+^m, \mathbf{b} \in \mathbb{R}_+^n, \mathcal{C} \in \mathbb{Z}_2^{m\times n}. \tag{3.9}$$

We refer to this linear program as our *master problem*. This formulation implies that the problem is solvable in polynomial time in the number of its variables m. However, this number, i.e. the number of columns in matrix $\mathcal{C}$ and thus the number of possible covers, is exponential in the number of nodes $n = |V|$. We have $m = \mathcal{O}(2^n)$. Therefore, constructing and handling the complete matrix of covers $\mathcal{C}$ is not an option but for the smallest of problem instances.

We have argued before that the time complexity of our subproblem is of no concern. However, this is only true for theoretical results. In an efficient implementation, one wants to minimize all constant factors. Moreover, constructing the full cover matrix $\mathcal{C}$ can easily become infeasible due to the required memory. To cope with these issues, we make use of delayed column generation to iteratively construct the linear program column by column as needed during the solution process. This concept was introduced by Dantzig and Wolfe in [DW60]. By now, it has become a prominent method in handling linear programs with an exponential number of variables. The approach offers several degrees of freedom that directly impact its runtimes. We will discuss several existing techniques for each of them in order to achieve best results.

3.4.1 Delayed Column Generation

First, we describe how to apply the delayed column generation approach in general to solve our master problem (3.9) to optimality. The procedure is summarized by Algorithm 3.3. For a theoretical discussion on column generation, we refer to the original work [DW60] and a survey article by Lübbecke and Desrosiers [LD05].

We start by computing an initial set of covers that form matrix $\mathbf{C} = [\mathbf{c}_1, \ldots, \mathbf{c}_k]$, with $\mathbf{c}_i \in \mathcal{C},\ 1 \leq k \ll m$ (line 1). How they are found is detailed later. We solve the *restricted master problem*

$$\max\left\{\mathbf{1}^\intercal\mathbf{t} \mid \mathbf{C}\mathbf{t} \leq \mathbf{b},\ \mathbf{t} \geq \mathbf{0}\right\}, \qquad \mathbf{t} \in \mathbb{R}_+^k, \mathbf{b} \in \mathbb{R}_+^n, \mathbf{C} \in \mathbb{Z}_2^{k\times n}, \tag{3.10}$$

to obtain a tentative solution $\mathbf{t}$ (line 3). The problem corresponds to our master problem (3.9) but with much less variables, i.e. covers, to consider. After determining the solution $\mathbf{w}$ of the dual problem to (3.10) in line 4, compare Section 2.3.1, we can solve the so-called *oracle problem* (line 5)

$$\max\{1 - \mathbf{c}^\intercal\mathbf{w} \mid \mathbf{c} \in \mathcal{C}\}, \qquad \mathbf{w} \in \mathbb{R}_+^n, \mathbf{c} \in \mathbb{Z}_2^n. \tag{3.11}$$

The problem is an integer linear program. If its solution value $(1 - \mathbf{c}^\intercal\mathbf{w})$ is non-positive, the tentative solution $\mathbf{t}$ corresponds to an optimal solution of the master problem (3.9), and we are done (line 10). Otherwise, we add $\mathbf{c}$ to the restricted master problem as a new column in cover matrix $\mathbf{C}$ (line 7), our tentative set of "known" covers, and repeat the process by re-optimizing the modified problem (3.10) in line 3.

The final solution is described by a tuple, the set of durations $\mathbf{t}$ and the set of related covers $\mathbf{C}$. The solution value $\mathbf{1}^\intercal\mathbf{t}$ denotes the lifetime $T_{\text{opt}}\langle V, R\rangle$ of the sensor network.

Algorithm 3.3 Column Generation Approach

Input: Battery capacities $\mathbf{b}$, set of all covers $\mathcal{C}$ (implicitly)
Output: Set of feasible covers $\mathbf{C}$, set of corresponding durations $\mathbf{t}$

1: $\mathbf{C} = [\mathbf{c}_1, \ldots, \mathbf{c}_k] \leftarrow init\_covers(\mathcal{C}, \mathbf{b})$ ▷ determine initial covers
2: **repeat**
3: $\quad\mathbf{t} \leftarrow \arg\max\{\mathbf{1}^\intercal\mathbf{t} \mid \mathbf{C}\mathbf{t} \leq \mathbf{b},\ \mathbf{t} \geq 0\}$ ▷ solve restricted master LP
4: $\quad\mathbf{w} \leftarrow find\_dual(\mathbf{C}, \mathbf{t}, \mathbf{b})$ ▷ compute dual solution
5: $\quad\mathbf{c} \leftarrow \arg\max\{1 - \mathbf{c}^\intercal\mathbf{w} \mid \mathbf{c} \in \mathcal{C}\}$ ▷ solve oracle ILP
6: $\quad$**if** $(1 - \mathbf{c}^\intercal\mathbf{w}) > 0$ **then**
7: $\quad\quad\mathbf{C} \leftarrow [\mathbf{C}, \mathbf{c}]$ ▷ add cover $\mathbf{c}$ to set $\mathbf{C}$
8: $\quad$**end if**
9: **until** $(1 - \mathbf{c}^\intercal\mathbf{w}) \leq 0$
10: **return** $(\mathbf{C}, \mathbf{t})$ ▷ optimum solution found

Note that the set of all covers is given implicitly, i.e. by stating the problem geometry.

Discussion. The solution of the master problem (3.9) is uniquely defined by the argument of the maximum $\mathbf{t}$. However, when using delayed column generation, we do not have a complete enumeration of all covers but still need to know which duration corresponds to which cover. Therefore, a solution of the column generation approach consists of the computed durations $\mathbf{t}$ as well as of the associated covers $\mathbf{C}$. The set of covers $\mathbf{C}$ comprises all covers that have been found by the initialization step and that have been computed during column generation. Not all of them have to contribute to an optimum solution, though. In this case, their respective durations are set to zero. The solution of the column generation approach, tuple $(\mathbf{C}, \mathbf{t})$, directly corresponds to an optimal schedule for the sensor network.

The column generation approach has several degrees of freedom that have to be assigned carefully to obtain competitive runtimes. The initialization step in line 1 is crucial for a quick convergence to an optimum solution. We discuss possible options in the next section. Solving the LP in line 3 is not critical, though. The problem size is small enough so that any reasonably fast LP solver can be applied. The oracle problem in line 5, however, requires more attention. It is easier to handle than the master problem (3.9) as we only need an implicit description of $\mathcal{C}$, which is polynomial in the number of nodes n—recall that we can describe area coverage by at most n^2 targets [BCSZ05], with each of them coverable by at most n nodes. However, the problem still remains an ILP. Section 3.4.3 takes a closer look at this problem. As a final degree of freedom, we can choose when to terminate the algorithm. We can stop early after any iteration. The tentative solution $\mathbf{t}$ at that point is a feasible solution for the master problem (3.9) as it adheres to all constraints. However, the solution value, i.e. the network lifetime $T\langle V, R\rangle = \mathbf{1}^\intercal\mathbf{t}$, is less than the optimum. We consider this degree of freedom in Section 3.4.4.

Finally, note that unlike our approximate algorithm in the last section, the principal approach described here supports arbitrarily shaped sensing areas for each node. The underlying geometry is hidden in cover matrix $\mathbf{C}$. It is only required when generating covers in the initialization step and when solving the oracle problem (3.11).

3.4.2 Initialization Step

The delayed column generation approach requires an initial set of covers $\mathbf{C} \subseteq \mathcal{C}$. There are many strategies to choose such a set, but for quick convergence, it should obviously be close to the set of covers of an optimum solution. Another important aspect is that every cover $\mathbf{c} \in \mathbf{C}$ should be minimal in the number of nodes. By removing any node $v \in \mathbf{c}$, set $\mathbf{c}$ should no longer be a cover, i.e. $\mathbf{c} \setminus \{v\} \notin \mathcal{C}$. If this is not guaranteed, we may end up with many similar covers or, even worse, with covers that are supersets of other covers. In algebraic terms, the columns of matrix $\mathbf{C}$ should be linearly independent. This is not strictly required, but a cover that is a superset of another cover only increases the problem size without contributing to the solution, so this should be avoided. The same is true for covers that are a superset of the same cover, e.g. $\mathbf{c}_1 \cap \mathbf{c}_2 = \mathbf{c} \in \mathcal{C}$, even if $\mathbf{c}$ is not part of the current set of covers $\mathbf{C}$.

We therefore minimize each cover after its computation, and before adding it to the current set of covers $\mathbf{C}$, we check for duplicates. Minimization is done with a very simple heuristic that iterates once over all nodes in the cover and checks for each node whether the remaining nodes still form a cover. If true, we can safely remove the node. This process does not yield a minimum set of nodes, but a minimal one is good enough to prevent linearly dependent columns in our cover matrix $\mathbf{C}$.

Below we introduce several strategies that we study as possible initialization step during our simulations in Section 3.6. We classify these strategies into three categories.

Basic Strategies. The most basic strategy offers the full set of nodes as a single cover to the column generation approach. This node set is obviously a cover, for otherwise, there would not exist any cover. Publications [LGR09, RSS12, CRSV13] apply this approach. We call it the *basic* strategy.

Another simple strategy, labelled *random*, is used e.g. in [GLZ07, GZJL11, Des11]. A fixed but adjustable number of covers is computed. For each cover, we iteratively select nodes at random and add them to a node set until it forms a cover. After minimizing the cover, we verify that it is not a duplicate before accepting it. Note that this last step is not documented in the referenced previous work.

Greedy Strategies. Using the same principal approach as for the random strategy in the last paragraph, but applying a more sophisticated greedy algorithm to find covers, is another obvious strategy to obtain an initial set of covers. Our basic greedy strategy, called *greedy*, uses the heuristic for the wMGDC problem that we introduce in the next section. We repeatedly run this algorithm with random weights to generate covers as suggested in [RG11]. As before, we minimize the covers and verify that we do not add duplicates to our set of initial covers.

The greedy heuristics by Slijepcevic and Potkonjak [SP01] and Cardei et al. [CTLW05] solve slightly different problems than SNLP with nodes restricted to one or few covers and uniform battery capacities. However, we can still use these approaches to find an initial set of covers for our column generation approach. In the following, we refer to them by their first author. The *Slijepcevic* approach computes a maximal set of disjoint covers. For each cover, we start by determining a critical region of the area that we want to cover, i.e. an uncovered region that is difficult to cover by the remaining available nodes. We cover it by greedily choosing an appropriate node that covers the most yet uncovered regions of the whole area while introducing the least redundant coverage. This is repeated until a cover for the whole area is found. The process is iterated until no more covers can be computed. Alfieri et al. [ABBC07] make use of this approach. The *Cardei* approach works similarly, but it is more sophisticated as each node is allowed to contribute to a (small) fixed number of covers. Raiconi and a group of co-authors use this method in [RG11, CdDR12, GR13].

Further Strategies. Advanced approximation schemes for SNLP can also serve as initialization step to the column generation approach. We have to carefully balance solution quality and runtimes, though. Otherwise, the improved speed of convergence could be nullified by the time spent during initialization.

In this category, we focus on the approach by Garg and Könemann [GK07]. It is a particular interesting choice. Even though a very effective heuristic, it has not yet been applied in our context of delayed column generation. We give a full description of the *Garg-Könemann* approach in Section 3.4.5 and compare its performance to the other previously introduced initialization methods in Section 3.6.

3.4.3 Oracle Problem

The oracle problem (3.11) tries to determine a new cover $\mathbf{c} \in \mathcal{C} \setminus \mathbf{C}$ whose addition would improve the solution value compared to only considering the set of covers $\mathbf{C}$ of the current tentative solution. We can interpret the term $(1 - \mathbf{c}^\intercal \mathbf{w})$ as improvement in the solution value if cover $\mathbf{c}$ is added with unit duration, minus the compensation required in the durations of the other covers to keep the solution feasible [BT97]. If the term is non-positive for all covers $\mathbf{c} \in \mathcal{C}$, no further improvement can be achieved.

The oracle problem (3.11) can be rewritten as

$$\min \{\mathbf{c}^\intercal \mathbf{w} \mid \mathbf{c} \in \mathcal{C}\}, \qquad \mathbf{w} \in \mathbb{R}_+^n, \mathbf{c} \in \mathbb{Z}_2^n .$$

In this representation, we see that the ILP models a weighted set cover problem for some yet to be specified sets and computes a cover $\mathbf{c}$ of minimal weight. The dual variables $\mathbf{w}$ of the restricted master problem (3.10) serve as node weights. The problem corresponds to the $\mathcal{NP}$-complete weighted minimum geometric disk cover problem (wMGDC) introduced in Section 3.2.3 as we can translate area coverage to target coverage with a polynomial number of targets [BCSZ05]. Targets represent the sets in the set cover problem. Appendix A shows how to derive the targets from an area coverage problem. Overall, the oracle problem is hard to solve.

Fortunately, there exist many highly optimized algorithms for this kind of problem. We have the choice between various approximate and exact solvers. If we apply a heuristic to compute the covers, we are not guaranteed that the solution of the column generation approach is optimal once $(1 - \mathbf{c}^\intercal \mathbf{w}) \leq 1$ holds, though. To deal with this problem, we can switch to an exact solver as soon as the heuristic does not find any more covers as in [Des11]. The process until we switch to the exact solver can be regarded as a sophisticated initialization step. Using an exact ILP solver from the start obviously does not lead to this issue, however, average runtimes might be longer. In either case, we have to guarantee that the computed cover is minimal in the number of nodes and no duplicate of a previous cover as reasoned in the last section. Non-minimal covers are, for example, caused by nodes with zero weight.

In the following, we describe a simple heuristic for wMGDC that we apply to solve the oracle problem (along with an exact ILP solver) as well as a subroutine in the basic greedy strategy and in the Garg-Könemann approach, introduced in the last section as possible initialization steps for the column generation approach.

Solving wMGDC. Given a set of n sensor nodes, the wMGDC problem asks for a cover of a set of points—or targets—with minimal weight with respect to some node weights $\mathbf{w}$. The problem corresponds to a weighted set cover problem as reasoned above. Thus, we follow common solution strategies for this type of problem. Our approach is a greedy heuristic that iteratively adds sensor nodes to a set until the nodes in this set form a cover of the considered area, i.e. until all targets are covered.

In each step, a node is chosen for which the ratio of newly covered targets to node weight is maximized. Ties are resolved arbitrarily.

The algorithm achieves an approximation ratio of $H(M)$, with $H(x)$ the x-th harmonic number and M the maximum number of nodes covering a single target. This follows from the basic greedy solution for the weighted set cover problem [Chv79]. With at most n nodes covering a single target and the estimate $H(x) \leq 1 + \log x$, we obtain an $(1 + \log n)$-approximation in the number of sensor nodes n. Or, more generally speaking, the wMGDC heuristic has an approximation ratio of $\mathcal{O}(\log n)$.

The time complexity of the algorithm is in $\mathcal{O}(n^3)$. This follows from the asymptotic running time of the basic greedy set cover algorithm [CLRS09], which grows with the sum of the nodes that cover each target. With at most n nodes covering each of the $\mathcal{O}(n^2)$ targets—recall that area coverage can be reduced to target coverage with at most n^2 targets [BCSZ05]—we observe the claimed running time.

3.4.4 Termination Condition

Each iteration of the column generation approach yields a feasible solution $\mathbf{t}$ for the master problem (3.9). The solution value $T\langle V, R\rangle = \mathbf{1}^\intercal\mathbf{t}$ converges to the optimum $T_{\text{opt}}\langle V, R\rangle$. However, this process can be very slow as the improvements tend to become smaller with each subsequent iteration. If we are content with a good but suboptimal solution, we can terminate the process early.

A common termination condition, compare [GJLZ09], is to stop the algorithm as soon as the relative improvement in the solution value stays below a certain threshold for a given number of iterations. We usually do not want this number to be too small, though, as the improvement does not monotonically decrease, and we would risk stopping the algorithm too soon. The whole approach is purely heuristic in nature and so is the choice of the threshold value and the number of iterations.

Unfortunately, the former method does not give any guarantees on the expected solution quality. This shortcoming can be remedied if we can assess an upper bound $T_{\text{upper}}\langle V, R\rangle$ on the optimal solution value $T_{\text{opt}}\langle V, R\rangle$. As described in [GZJL11], we iteratively generate new columns until $\mathbf{1}^\intercal\mathbf{t} \geq (1 - \epsilon) \cdot T_{\text{upper}}\langle V, R\rangle$, $\epsilon \in [0, 1]$. Once this threshold is reached, we terminate the algorithm. This ensures the solution value to be within a factor of $(1 - \epsilon)$ of the optimum.

If we are more interested in guaranteed maximal runtimes instead of optimal solutions, though, we can terminate the column generation after a fixed amount of time has passed or a fixed number of iterations have been performed. In the final iteration, we only solve the restricted master problem to obtain a new value for $\mathbf{t}$ that incorporates the latest cover. We do not compute another cover with the ILP as it would not contribute to the solution anymore. As an extreme variant, we can terminate the algorithm after the first iteration. This simply takes the set of initial covers $\mathbf{C}$ and optimizes their durations $\mathbf{t}$ according to the restricted master problem. Berman et al. [BCSZ05] suggest this approach as a post-optimization step for their SNLP heuristic.

3.4.5 Garg-Könemann Approach

The Garg-Könemann approach was introduced by its namesakes in [GK07]. It is an approximation algorithm for packing linear programs like our master problem (3.9). The approach computes an $(1+\epsilon)\cdot f$-approximation given an f-approximate solver for some subproblem. Its asymptotic running time is in $\mathcal{O}(\frac{1}{\epsilon^2} n \log n \cdot T)$, with T the time complexity for solving the subproblem and n the number of nodes.

Below we describe the Garg-Könemann approach adapted to our requirements. Algorithm 3.4 summarizes this procedure. For a theoretical discussion on the general mechanics and properties of this approach, we refer to the original work [GK07] as well as to [DWW+12] who adapt it to a similar problem.

Algorithm 3.4 Garg-Könemann Algorithm

Input: Parameter $\epsilon \in (0,1]$, battery capacities $\mathbf{b}$, set of all covers $\mathcal{C}$ (implicitly)
Output: Set of feasible covers $\mathbf{C}$, set of corresponding durations $\mathbf{t}$

1: $\mathbf{C} \leftarrow []$, $\mathbf{t} \leftarrow ()$ ▷ initialization
2: $\delta \leftarrow (1+\epsilon)/\sqrt[\epsilon]{(1+\epsilon)m}$, $\mathbf{w}_i \leftarrow \delta/\mathbf{b}_i,\ i \in \{1,\dots,n\}$
3: **while** $\mathbf{b}^\intercal\mathbf{w} < 1$ **do**
4: $\quad\mathbf{c} \leftarrow \arg\min\{\mathbf{c}^\intercal\mathbf{w} \mid \mathbf{c} \in \mathcal{C}\}$ ▷ find cover of minimal weight
5: $\quad\mathbf{b}_{min} \leftarrow \min\{\mathbf{b}_i \mid v_i \in \mathbf{c},\ i \in \{1,\dots,n\}\}$ ▷ find minimum capacity
6: $\quad\mathbf{C} \leftarrow [\mathbf{C}, \mathbf{c}]$, $\mathbf{t} \leftarrow (\mathbf{t}, \mathbf{b}_{min})$ ▷ add cover $\mathbf{c}$ with duration $\mathbf{b}_{min}$
7: $\quad\mathbf{w}_i \leftarrow \mathbf{w}_i \cdot (1+\epsilon\cdot\mathbf{b}_{min}/\mathbf{b}_i),\ v_i \in \mathbf{c},\ i \in \{1,\dots,n\}$ ▷ update node weights
8: **end while**
9: **return** $(\mathbf{C}, \mathbf{t}/\log_{1+\epsilon}\frac{1+\epsilon}{\delta})$ ▷ scale durations

Note again that the set of all covers is given implicitly as before in Algorithm 3.3.

The algorithm starts with empty sets of covers $\mathbf{C}$ and associated durations $\mathbf{t}$ (line 1). We initialize node weights $\mathbf{w}_i = \delta/\mathbf{b}_i, i \in \{1,\dots,n\}$ in line 2 before beginning to iteratively compute covers. At the start of each iteration, we compute a minimal cover $\mathbf{c}$ with respect to our node weights $\mathbf{w}$ (line 4). This is done by an algorithm $\mathcal{A}$ that we discuss below. We determine the minimum battery capacity b_{min} of all nodes contributing to cover $\mathbf{c}$ (line 5). It is added to the set of covers $\mathbf{C}$ with a duration corresponding to the minimal battery capacity $\mathbf{b}_{min}$ (line 6), i.e. we activate cover $\mathbf{c}$ for as long as possible. At the end of each iteration, we increase the weights of all nodes in $\mathbf{c}$ by a fraction of ϵ relative to their usage (line 7). We repeat the process until $\mathbf{b}^\intercal\mathbf{w} \geq 1$ holds (line 3). To obtain a feasible solution, we have to scale the durations $\mathbf{t}$ by a factor of $\log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ to satisfy all constraints (line 9), i.e. nodes can be used in multiple covers and thus possibly longer than their battery capacities allow.

The solution is described by a tuple, the set of covers $\mathbf{C}$ and the set of corresponding durations $\mathbf{t}$. It provides an $(1+\epsilon)\cdot f$-approximation for linear program (3.9), i.e. for the sensor network lifetime problem.

Discussion. The approach by Garg and Könemann is a primal-dual method as introduced in Section 2.3.1 that exploits the duality of linear programs to more efficiently compute (approximate) solutions for several types of problems. The dual to our master problem (3.9) is of the form

$$\min \{\mathbf{b}^\intercal \mathbf{w} \mid \mathcal{C}^\intercal \mathbf{w} \geq \mathbf{1},\ \mathbf{w} \geq \mathbf{0}\}, \qquad \mathbf{w} \in \mathbb{R}_+^n, \mathbf{b} \in \mathbb{R}_+^n, \mathcal{C} \in \mathbb{Z}_2^{m \times n},$$

with the dual variables corresponding to our node weights $\mathbf{w}$. Algorithm 3.4 iteratively constructs primal and dual solutions $\mathbf{t}$, $\mathbf{w}$ at the same time.

The Garg-Könemann approach yields a feasible but suboptimal schedule $(\mathbf{C}, \mathbf{t})$ for SNLP. When using the algorithm as part of our column generation approach, we only require the set of covers $\mathbf{C}$ and can skip computing the corresponding durations $\mathbf{t}$. However, we have to take them into account if we want to compare the solution qualities of the various initialization steps to each other and to the exact approach.

The problem solved by algorithm $\mathcal{A}$ corresponds to the oracle problem (3.11) in the column generation approach. Therefore, we can apply the same principal approaches as discussed in Section 3.4.3. We propose to use the previously introduced greedy heuristic for wGDMC with its $\mathcal{O}(\log n)$-approximation guarantee and a worst-case cubic running time. For the oracle problem, we further suggested applying an exact ILP solver. In this context, however, the ILP would be too expensive as we only want to generate some initial covers and do not require optimality.

Similar to what we described in Section 3.4.4 for Algorithm 3.3, we can terminate the Garg-Könemann approach early and use the tentative set of covers $\mathbf{C}$ to initialize our column generation approach. We did not consider this direction more closely, though, as the time spent in the initialization step is not particularly critical.

3.4.6 Full Method

After having discussed the column generation approach and its various degrees of freedom in the previous sections, we can now combine everything into one algorithm. In the following, we describe our proposed approach for solving the sensor network lifetime problem to optimality.

Our method applies delayed column generation as described in Algorithm 3.3 as its foundation. We use the Garg-Könemann approach of the last section to compute initial covers. Its subproblem is solved by the greedy wMGDC heuristic introduced in Section 3.4.3. Both, the restricted master problem (3.10) and the oracle problem (3.11), are solved with exact solvers. We do not apply any early termination condition as we are looking for exact solutions.

This combination of techniques has not been studied before in the related literature. The results of our simulations in Section 3.6 show that, in particular, the union of a delayed column generation with the Garg-Könemann approach for initialization is a very powerful technique for solving our considered problem.

3.5 Optimizing State Changes

In the previous sections, we assumed that the costs for node state changes are proportional to the monitoring costs and that we can incorporate them implicitly through effective monitoring costs. We now drop this assumption and consider changes in the state of a node, i.e. switching from an active state to sleeping, or vice versa, to consume a substantial amount of energy. There exists a large body of work that does not take into account this source of battery drain explicitly, including our own algorithms in the previous sections. We therefore opt for a post-processing strategy that can be applied after any of these algorithms have computed their results. Our strategy only requires a set of independent covers that can be activated in an arbitrary order.

We present an optimization algorithm that orders a set of covers of an area so that the number of state changes and thus the induced energy drain is minimized. As our approach reduces the problem to an instance of the traveling salesperson problem, we first give a short recapitulation of this well-known optimization problem before going into the details of our algorithm.

3.5.1 Traveling Salesperson Problem

The traveling salesperson problem (also traveling salesman problem) is a classic problem in optimization theory. It goes back to the 1800s and was first mathematically formalized by Menger in [Men31]. We summarize the general problem in Definition 3.6.

Definition 3.6 (Traveling Salesperson Problem—TSP)**.** *Given a complete graph $G = (V, E)$ with an edge cost function $c : E \mapsto \mathbb{R}_+$, compute a path of minimal length that visits all nodes exactly once and then returns to the starting node. Such a full cycle over all nodes is called a* tour*.*

The problem is $\mathcal{NP}$-complete [Kar72] and in general hard to approximate within a constant factor of the optimum [SG76]. However, when we consider the special case of *metric* TSP, we can find constant factor approximations, e.g. Christofides gives a 1.5-approximation in [Chr76]. A metric TSP assumes edge costs to be symmetric and adhere to the triangle inequality, i.e. $c(u, v) + c(v, w) \geq c(u, w), \forall u, v, w \in V$. Today, heuristics without approximation guarantees outperform most approximation algorithms in practice. For example, the local search heuristic by Lin and Kerningham in [LK73] is still one of the best methods for approximating metric TSP. An efficient implementation is provided by Helsgaun[1], free for academic use.

An extensive overview on the history and the theory of the traveling salesperson problem can be found in the textbook by Applegate et al. [ABCC07].

[1] `http://www.akira.ruc.dk/~keld/research/LKH/`. Accessed: 2014-08-06.

3.5.2 Minimizing Node State Changes

We formalize our optimization problem, minimum node state changes (MNSC), in Definition 3.7. In order to solve this problem, we show how to translate an instance of MNSC to an instance of metric TSP. Given this transformation, we can apply any (approximate) solver for the traveling salesperson problem to find a solution of our MNSC instance. The resulting tour corresponds to an (almost) optimal order of our covers with respect to the battery drain due to node state changes.

Note that this technique is not tailored towards our SNLP problem. It can be used to optimize the order of any set of covers that can be activated in an arbitrary order.

Definition 3.7 (Minimum Node State Changes—MNSC)**.** *Given a set of covers* $\mathbf{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_m\}$*, find an order that requires a minimal amount of state changes between covers, i.e. switches from an active to a sleeping state, and vice versa. All nodes start and end in an inactive state.*

The number of state changes is proportional to the amount of energy required by these operations. It is therefore sufficient to consider only this number when minimizing the energy consumption due to state changes. One could argue that activating a node requires more energy than switching it off. However, as we start and end with all nodes in an inactive state, each node performs an equal amount of state changes in either direction. Thus, we can amortize energy costs over both types of operations.

Problem Translation. We now describe how to translate an instance of the MNSC problem to an instance of metric TSP. Consider the set of covers $\mathbf{C}$, i.e. an instance of MNSC. Each cover $\mathbf{c} \in \mathbf{C}$ corresponds to a (graph) node $v_{\mathbf{c}}$ in node set V. In addition, the empty set $\mathbf{0} = \{\}$, i.e. all sensor nodes being inactive, is added to V as $v_{\mathbf{0}}$. It serves as the starting and ending node of our TSP tour. Node set V induces the complete graph $G = (V, E)$. Edge costs are defined as $c(v_{\mathbf{c}}, v_{\mathbf{c}'}) = |\mathbf{c} \setminus \mathbf{c}'| + |\mathbf{c}' \setminus \mathbf{c}|$, with $(v_{\mathbf{c}}, v_{\mathbf{c}'}) \in E$. This corresponds to the number of state changes required to switch from cover $\mathbf{c}$ to $\mathbf{c}'$, i.e. the number of nodes only active in cover $\mathbf{c}$ that have to be switched off going to $\mathbf{c}'$, plus the number of nodes only active in cover $\mathbf{c}'$ that have to be switched on coming from $\mathbf{c}$. The edge costs are obviously symmetric. They also satisfy the triangle inequality since the relative complement of sets with respect to set cardinality does too, i.e. $|A \setminus B| + |B \setminus C| \geq |A \setminus C|$ for some sets A, B, C. This can be easily reasoned, for instance by using Venn diagrams. Thus, we have constructed a complete graph $G = (V, E)$ with metric edge costs c from our instance $\mathbf{C}$ of MNSC that can be used as an instance of metric TSP.

The length of a path $\langle v_{\mathbf{c}_1}, \ldots, v_{\mathbf{c}_k} \rangle$ corresponds to the number of state changes that have to performed, switching between covers $\mathbf{c}_1, \ldots, \mathbf{c}_k$ in this order. Thus, a shortest path describes an order with a minimal number of state changes, and a TSP tour starting at $v_{\mathbf{0}}$, i.e. with all sensor nodes inactive, gives an optimal order of all covers.

3.6 Simulations

We conclude with extensive simulations of our exact algorithm for solving SNLP. The performance of this approach is evaluated with respect to the previously discussed degrees of freedom. All simulations were performed on one core of Machine A.

3.6.1 Simulational Setup

Network Setting. We generate the topologies of our simulated sensor networks by iteratively placing nodes on a squared area following one of the distribution strategies in Section 3.2.1, random placement, grid placement, or perturbed grid placement. The size of the area is chosen to match the desired number n and average density d_{avg} of nodes, i.e. we set its edge length to $\sqrt{n/d_{avg}}$. Grid positions are spaced 0.5 apart, and perturbation allows for a deviation of 0.25 from these positions. Nodes have a maximum sensing range of $(1+\rho)$, with ρ taken uniformly at random from $[0, R]$. R is called variation in the sensing range. Battery capacities are uniformly set to 1. Nodes in the same location are merged and their battery capacities are added up.

We consider multiple distinct network settings in our simulations. Each setting is defined by the node distribution strategy, the number of nodes, the node density, and the variation in the sensing range. Our default setting uses random placement, $n = 300$ nodes with an average density $d_{avg} = 2.5$, and a fixed sensing range, i.e. $R = 0$.

Measurement Procedure. We evaluate each of our network settings 100 times with different random seeds and a hard runtime limit of three hours. Our quantitative analysis states mean values for all measurements. For each setting, we present the required runtime, the relative error $(1 - T/T_{\text{opt}})$ of the solution T to the maximum lifetime T_{opt} where applicable, the number of covers in the solution, and the number of iterations required by the column generation approach.

Before computing an activation schedule, though, we have to define the area A that has to be monitored by the considered sensor network instance. We cannot simply take the whole area that is covered by the sensor nodes as there are sparsely covered regions on the fringes of this area. If we were to include them in area A, the maximum lifetime of the network would be (severely) limited by the minimum number of nodes covering any of them. To make a reasonable choice for area A, we partition the area covered by the union of all nodes into smaller, not necessarily connected regions, called *entities*. Each entity is covered by a unique set of sensor nodes. We define area A to consist of all entities covered by at least $\lceil cov_{avg}/2 \rceil$ nodes, with cov_{avg} the average coverage over all entities. The entities also correspond to targets when converting from area coverage to target coverage [BCSZ05]. Appendix A details how to compute these entities and lists the number of considered entities, the average coverage over all entities, and the connectivity of the considered areas for each network setting in Table A.1.

Considered Approaches. We take the general column generation approach introduced in Section 3.4.1 as a basis and consider the different techniques for its various degrees of freedom that we introduced in the previous sections.

For the initialization step, we consider the six strategies described in Section 3.4.2. We provide our own implementations for all strategies according to the respective publication where applicable. As there are no recommended parameter settings, we use the following ones: Both, the random and the greedy strategy, generate 1 000 distinct covers. The Slijepcevic approach uses four minor scaling variables. We introduce them here to make our results reproducible. Using the same notation as in [SP01], we set $K = 1$, $L = 1$, $M = |\{V_j \in V \mid e_i \in V_j\}|$, and $N = |\{V_j \in C \setminus C_i \mid e_i \in V_j\}|$. The Cardei approach does not define how to choose a critical entity or how to select a node to cover it. We specify that an entity is critical if it is covered by the least amount of sensor nodes with respect to their remaining energy. The chosen node has to cover the most yet uncovered entities. The remaining energy of the node serves as a tie-breaker. Moreover, each use of a sensor node is set to consume 10% of its battery capacity. The Garg-Könemann approach is applied with $\epsilon = 0.1$ and the wMGC heuristic for solving its subproblem. In a separate study, we consider different values for the error parameter ϵ as well as using an exact solver for the subproblem.

For the oracle problem (3.11), we compare three different solution strategies: Only using a greedy heuristic, using the greedy heuristic followed by an exact solver once the greedy approach does not yield any more covers, and only applying the exact solver. We use our greedy wMGDC heuristic in the first two cases.

For the termination condition, we consider three strategies: Only performing the initialization step, performing exactly one iteration of the column generation approach, and generating columns until the approach finishes normally. We may abbreviate column generation by CG in our tabular results. In a separate study, we consider terminating the algorithm once the improvements in the solution quality become too small. We apply the parameter settings of [GJLZ09], requiring 10 successive iterations with a relative improvement below 1% and 10%, respectively.

3.6.2 Comparison to Previous Work

First, we evaluate our method for solving SNLP with respect to the previous work before studying its performance for different network settings in the next section. We cannot compare ourselves directly to most of the previous work, though, as they often study (slightly) varied problem settings, e.g. allowing variable sensing ranges or taking into account communication costs. However, we can consider the basic techniques used in these publications to solve their respective LPs. They are general techniques for the various degrees of freedom of the column generation approach as detailed in the last sections. We compare them to each other and to our choices for these degrees of freedom under the default network setting of Section 3.6.1. Tabulated results list our choices in the last row with best runtimes and error ratios marked in bold.

Initialization Step. In a first series of simulations, we consider multiple options for the initialization step of the column generation approach. Table 3.1 summarizes our findings. We list the results of the initialization step and show how they impact a subsequent column generation approach. Note that performing only one iteration of column generation corresponds to computing optimal durations for the covers found by the initialization step. The runtimes given for the column generation approach do not include the time spent in the initialization step.

Table 3.1: *Results of the column generation approach for solving SNLP with different methods for computing initial covers. The basic (b), random (rnd), and greedy (gr) strategies as well as the Slijepcevic (sp), Cardei (c), and Garg-Könemann (gk) approaches are considered.*

	Initialization			**CG (1 iter.)**			**Full CG**		
	time	error	covers	time	error	covers	time	covers	iter.
strategy	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[#]	[#]
b	**0.0**	70.9	1	**0.4**	70.9	1	90.4	170	793
rnd	0.2	–	1 000	0.5	38.4	59	124.0	172	461
gr	0.6	–	1 000	0.8	13.1	115	95.5	169	307
sp	0.0	44.8	2	0.4	44.8	2	89.9	171	768
c	0.4	34.0	23	0.4	32.9	20	79.4	169	640
gk	1.5	**12.1**	1 839	2.0	**0.6**	181	**20.2**	181	25

Focusing on the results of the initialization step for the moment, we see that our proposed method, applying the Garg-Könemann approach, requires the largest amount of time and generates the most covers. However, its approximate solutions come much closer to the optimum solution values at an average error of about 12% than any of the other approaches. The random and greedy strategies only generate covers and do not provide an initial solution. We therefore do not give an error value for them. The same is true for the basic strategy, but as it consists of only one cover, we can simply activate this cover for as long as possible to obtain a solution value that we can compare to the optimum. The Slijepcevic approach only generates two covers as it is looking for disjoint covers. The low number of covers in the Cardei approach is due to it allowing each node to be part of at most 10 covers by setting the duration of a cover to 10% of the maximum battery capacity.

When considering the middle columns of Table 3.1, we can assess the potential of the various initialization methods, i.e. how many of the covers required by an optimal solution they are able to find. Most notably, the covers computed by the greedy strategy allow for solutions that are just 13% worse than optimal on average. Only the Garg-Könemann approach that is used by our solver fares better. In 57% of all cases, the set of covers provided by this approach already contains all covers of an optimum solution. Thus, the column generation can stop after only one iteration.

These observations can only be transferred partially to a full run of the column generation approach, though. We see clearly that our runtimes with the Garg-Könemann approach are the lowest by a wide margin. On average, we only need 25 iterations to find all additional covers required for an optimal solution. We have 5 outliers with more than 100 iterations, though. On the other hand, when using any other initialization method, we need several hundred iterations on average! This implies that the initial set of covers provided by them was not very good—a conclusion we already drew from the results after one iteration. The other initialization methods perform about equally poor, with the Cardei approach being slightly better and the random strategy being slightly worse. In particular the greedy strategy that gave quite good results after the first iteration could not confirm its performance in the long run. Finally, we observe an interesting fact when considering the number of covers in the optimal solutions. When using the Garg-Könemann approach to compute an initial set of covers, the optimal solution has slightly more covers than for the other initialization methods. This implies that our approach found a different optimal solution than the other approaches.

Overall, it is evident that our choice of using the Garg-Könemann approach provides the best results. Simply generating a lot of covers as in the random and greedy strategies is not enough, they have to be diverse, too. It is surprising that the combination of the Garg-Könemann approach and column generation has not been considered before, even more so as the former is frequently used for monitoring problems on sensor networks.

Oracle Problem. Next, we consider using a heuristic instead of an exact algorithm for solving the oracle problem (3.11) during column generation as often suggested in the previous work. Since this does not yield optimal solutions, one may switch to an exact algorithm once the heuristic does not provide any more covers. Table 3.2 gives results for both variants as well as for using an exact solver from the start. We consider all previously studied initialization methods once more since one of them might fare better in combination with the heuristic.

Table 3.2: *Results of the column generation approach with different methods for solving the oracle problem. Using a greedy heuristic alone and in combination with an exact solver is considered as well as only using an exact solver.*

	heuristic				**heur. + exact**			**exact**		
	time	error	covers	iter.	time	covers	iter.	time	covers	iter.
strategy	[s]	[%]	[#]	[#]	[s]	[#]	[#]	[s]	[#]	[#]
b	29.2	5.5	157	696	89.3	170	907	90.4	170	793
rnd	42.2	5.4	158	331	126.3	171	564	124.0	172	461
gr	20.6	6.0	154	161	103.7	171	398	95.5	169	307
sp	29.2	5.4	157	687	90.8	169	898	89.9	171	768
c	23.9	5.4	155	585	82.2	169	797	79.4	169	640
gk	**1.6**	**0.6**	181	1	**20.1**	180	26	**20.2**	181	25

We observe that only applying the wMGDC heuristic for solving the oracle problem already yields good results with a small error. The results improve by a large amount compared to only optimizing the durations of the initial covers (see Table 3.1). This is true for all initialization methods but for the Garg-Könemann approach. The sets of covers it provides are often close to optimal so that the greedy heuristic cannot find any more covers during column generation. Among the other initialization methods, the greedy strategy fares best in terms of runtimes and required iterations, though, the error of its solutions is marginally higher than for the other methods. The random strategy fares worst, requiring even more time than the basic strategy, even though its average number of iterations is very high. This is due to the column generation approach starting with many more covers (1 000) than for any of the other initialization methods except for the greedy one.

Comparing the results when applying the heuristic followed by the exact solver to only applying the exact solver, we find them to be almost identical. Sometimes the former is better, sometimes the latter. This suggests that most of the work is done by the exact solver for the oracle problem. The heuristic discovers the "easy" covers, while the exact solver has to generate the remaining covers that are required for an optimal solution. Considering the number of iterations, we see that applying the exact solver after the heuristic adds roughly 200 iterations. Using the exact solver from the start results in less overall iterations, but each one requires more time on average yielding roughly the same total runtimes.

Concluding, we can state that applying a heuristic before switching to an exact solver does not pay off. This observation holds true for other network settings as the results in Table A.2 and Table A.3 show for the Garg-Könemann approach. Therefore, we opt not to add unnecessary complexity to our method and only use an exact solver for the oracle problem during column generation.

Termination Condition. As we are only interested in optimal results or at least in results with an approximation guarantee, most of the termination conditions proposed in the previous work are not suited for us. Moreover, the only one giving an approximation guarantee requires us to compute good upper bounds on the optimal lifetime. For the sake of completeness, though, we consider terminating the column generation early if the relative improvement in the solution quality stays below a threshold value Δ for 10 iterations. The results are listed in Table 3.3 with $\Delta = 0$ indicating early termination. We further state results for the initialization step and after one iteration for reference. Note that they do not change with Δ as there is always at least one iteration.

We see directly that the number of iterations is much lower when using this early termination condition and, consequently, so is the required amount of time. The difference in the computed network lifetime to the optimum is very small, which is good news. However, we cannot guarantee these near optimal results. Moreover, the presented numbers have to be taken with a grain of salt as only 23% of the studied

problem instances require more than 10 iterations. Thus, the termination condition only affects a fraction of all instances. In the other cases, it is simply not necessary. We give further results in Table A.5 and Table A.6, though, to support our findings.

Table 3.3: *Results of the column generation approach when terminating early. The algorithm is stopped if the relative improvement stays below Δ for 10 iterations.*

	Initialization			**CG (1 iter.)**			**Full CG**			
Δ	time	error	covers	time	error	covers	time	error	covers	iter.
[%]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[%]	[#]	[#]
10.0	**1.5**	**12.1**	1 839	**2.0**	**0.6**	181	**5.6**	0.2	182	4
1.0	**1.5**	**12.1**	1 839	**2.0**	**0.6**	181	6.9	0.1	182	5
0.1	**1.5**	**12.1**	1 839	**2.0**	**0.6**	181	6.9	0.1	182	5
0.0	**1.5**	**12.1**	1 839	**2.0**	**0.6**	181	20.2	**0.0**	181	25

In summary, we can say that this termination condition is effective in reducing the runtime of our method while retaining good results. We have to keep in mind, though, that there is no guarantee for near optimal results. Moreover, we cannot get arbitrarily close to an optimum solution by simply decreasing Δ since the relative improvement does not monotonically decrease in each iteration, compare the results for $\Delta = 1\%$ and 0.1%. Increasing the required number of iterations with small improvements should have a greater impact. Our choice of 10 iterations is arbitrary, but as the total number of iterations varies strongly, there is little potential for a general optimization.

Garg-Könemann Approach. As we have seen, the Garg-Könemann approach plays an important part in the quick convergence to an optimum solution. It offers several degrees of freedom of its own, which we study next. We consider switching to an exact solver for its subproblem and varying its error parameter ϵ. Table 3.4 lists our findings.

Table 3.4: *Results of the column generation approach with different settings for the Garg-Könemann approach. Different values for error parameter ϵ are studied as well as using an exact solver for the subproblem of this approach.*

		Initialization			**CG (1 iter.)**			**Full CG**		
	ϵ	time	error	covers	time	error	covers	time	covers	iter.
solver	[1]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[#]	[#]
exact	0.10	63.2	22.7	649	0.5	10.6	85	46.3	161	194
heuristic	0.02	66.9	**6.4**	46 536	16.7	**0.1**	216	35.2	215	6
	0.50	0.1	48.4	54	0.4	9.3	49	68.3	167	482
	0.90	**0.0**	88.4	4	**0.4**	38.3	3	87.5	169	726
	0.10	1.5	12.1	1 839	2.0	0.6	181	**20.2**	181	25

First, we consider varying the error parameter ϵ. When decreasing its value from the default value, $\epsilon = 0.1$, we see that the solutions of the Garg-Könemann are getting closer to being optimal but at the cost of a significantly increased runtime. The number of computed covers increases, too. Both changes impact the subsequent column generation approach. The number of required iterations decreases, but as each iteration requires more time due to the larger amount of covers, the total runtime increases, too. In the other direction, increasing ϵ, we see a quicker termination of the Garg-Könemann approach that yields a smaller set of covers and, as expected, a larger error. The time saved during the initialization step does not pay off during the column generation approach, though, as runtimes and, in particular, the numbers of required iterations increase by a large amount. However, the time spent in each iteration decreases as there are less covers to consider initially. Results for additional values of ϵ are listed in Table A.4. Note that when using $\epsilon = 1.0$, the Garg-Könemann approach terminates immediately without computing any cover.

Next, we study the effects of replacing the heuristic solver for the subproblem of the Garg-Könemann approach by an exact one. It is evident that the runtimes increase by a significant amount. However, the solution quality of the Garg-Könemann approach does not improve by using the exact solver. We are provided with less covers on average, and the error ratio even increases by a fair amount. This is a side effect of the interplay between the termination condition of the Garg-Könemann approach and the solutions provided by the exact solver that we do not fully understand. Focusing on the rightmost columns of Table 3.4, we see the poor solution quality after the initialization step reflected in the results of the column generation approach. It requires much more iterations and thus longer to find an optimal solution. However, whereas the number of iterations increases by a factor of 8, the runtime only doubles. This is again due to the smaller number of covers the column generation approach has to deal with initially. We only briefly considered different values of ϵ as even for the default setting $\epsilon = 0.1$ both, the runtime and the solution quality, are worse than when applying the heuristic. We did not expect better results as changing ϵ should only improve one of these values while deteriorating the other. Our trial studies support this assumption to be correct.

In a final study, we consider to restrict the number of iterations in the Garg-Könemann approach, i.e. setting a limit on the number of covers it computes. This could prove to be beneficial as the column generation approach would have to process less covers. As shown e.g. in Table 3.1, the number of covers in an optimal solution is much less than the amount the Garg-Könemann approach computes. However, we are not guaranteed that the covers needed for an optimal solution are found early on. Table 3.5 shows the results of our studies. We see that the time spent in the initialization step scales almost linearly with the number of iterations apart from a small overhead. However, even by limiting their number only slightly, the errors grow drastically. More importantly, the column generation approach does not profit from the reduced number of input covers. In fact, its runtimes increase by a large amount, much more than the time saved during the initialization step. We conclude that the covers that become useful during the

column generation process are generated over the whole course of the Garg-Könemann approach. They are, in particular, not all already found in the earlier iterations.

Table 3.5: *Results of the column generation approach with the number of iterations of the Garg-Könemann approach and thus the number of returned covers restricted.*

	Initialization			**CG (1 iter.)**			**Full CG**		
gk iter. [#]	time [s]	error [%]	covers [#]	time [s]	error [%]	covers [#]	time [s]	covers [#]	iter. [#]
10	**0.0**	99.5	10	**0.4**	38.6	9	82.7	168	672
100	0.1	95.2	100	0.4	7.6	74	56.4	164	357
1 000	0.8	52.2	1 000	1.2	1.0	166	23.4	169	40
∞	1.5	**12.1**	1 839	2.0	**0.6**	181	**20.2**	181	25

In summary, we have learned two things. The better the solution quality after the initialization step becomes, the less iterations of column generation are required, and the more covers are handed to the column generation approach, the longer each iteration takes. However, these two findings are linked as the better the solution quality of the Garg-Könemann approach becomes, the more covers it generates. Thus, we have to balance both aspects for a short runtime of our complete algorithm. Overall, we can say that our default choices for the Garg-Könemann approach, using a heuristic to solve its inner subproblem, setting the error ratio to $\epsilon = 0.1$, and not stopping early, yield the best results for our setting.

Minimizing Covers. The minimization of covers before adding them to our set of known covers seems to be a minor optimization. However, as we have already reasoned in Section 3.4.2, it is actually very important for the performance of our approach, and the results in Table 3.6 clearly support this claim. We see that by not minimizing covers during the Garg-Könemann approach, the initialization step terminates earlier with a smaller number of computed covers. However, the gap of its solution quality to the optimal lifetime becomes larger, and thus the quality of the initial set of covers handed to the column generation approach gets worse. In turn, the column generation approach requires almost four times as long to finish. Judging from the much higher number of iterations, we spend a lot of time searching for new covers required by an optimal solution. If we do not minimize the covers found by the oracle problem (3.11) when generating columns, the average runtime increases by an even larger amount, by a factor of 11. The number of iterations also increases but not as much as before. The initialization step provides a suitable subset of the covers needed for an optimal solution. However, the total number of covers in the solution increases. This implies that the solution consists of several covers that are a superset of other covers. They could be removed while retaining the smaller covers and increasing their durations

appropriately. When not minimizing covers in either case, both effects multiply and our runtimes become more than 40 times higher. Similar effects are seen for the number of iterations and the number of covers in the optimal solutions.

Table 3.6: *Results for not minimizing covers during the Garg-Könemann approach (gk), during the column generation approach (cg), and during both (both) steps are shown next to the normal (norm) results when always minimizing covers.*

	Initialization			**CG (1 iter.)**			**Full CG**		
covers not minimal in	time [s]	error [%]	covers [#]	time [s]	error [%]	covers [#]	time [s]	covers [#]	iter. [#]
gk	**1.0**	16.2	1 753	3.6	4.3	241	76.9	182	120
cg	1.5	**12.1**	1 839	2.0	**0.6**	181	235.8	203	85
both	1.1	16.2	1 753	3.6	4.3	241	835.7	260	509
norm	1.5	**12.1**	1 839	**2.0**	**0.6**	181	**20.2**	**181**	25

The reason for the observed huge increase in runtimes has already been described in Section 3.4.2. In short, our LP and ILP solvers have to deal with a large number of very similar covers that essentially provide the same benefit in an optimal solution. We conclude that it is vital to minimize covers to obtain reasonable runtimes.

Further Techniques. The attentive reader might wonder about the techniques in the related work section that we did not cover in our comparison, e.g. the genetic approach in [RSS12] or the complete enumeration strategy in [LGR09]. We cannot reimplement and evaluate all of them, but we can give some general reasonings to argue that our method fares better than them.

First, we are able to consider much more entities, i.e. targets, than any of the previous work. Whereas their test instances usually stay well below 1 000 targets, we handle one to two orders of magnitude more entities (see Table A.1). This is significant since the running time of our approach grows linearly with the number of entities—compare the time complexity of the wMGDC heuristic in Section 3.5 and consider the time required to check whether a set of nodes forms a cover. The same holds true for the considered number of nodes, though to a lesser extent. However, their impact on the time complexity is more significant. Second, one of our main results is that the Garg-Könemann approach is a potent source of an initial set of covers. Another one is that all covers should be minimized. These results can be combined with the other techniques that we did not study in detail. For example, the genetic approach in [RSS12] requires an initial population of covers. This can be provided by the Garg-Könemann approach, and newly generated covers can be minimized before adding them to the population. Finally, we have to see whether the proposed techniques are even suited for general problem instances. For example, the complete enumeration approach [LGR09] is only

competitive if the problem instance is not too regular and the enumeration process can prune a large fraction of the considered covers. However, the uniform sensing ranges already render our problem too regular so that this technique cannot be successfully applied to our network setting.

We conclude this section by stating that our combination of well-known techniques is better suited for solving the sensor network lifetime problem to optimality than any previously proposed technique. The combination of a column generation approach with the Garg-Könemann approach for initialization and the minimization of covers lets us compute optimum monitoring schedules faster than ever before.

3.6.3 Network Settings

In the last section, we studied multiple techniques for the various degrees of freedom of the column generation approach for solving SNLP. We found that our novel combination of well-known techniques clearly outperformed previous approaches. This section now only considers our method, i.e. delayed column generation with the Garg-Könemann approach during initialization, and studies its performance for various other network settings than the default one specified in Section 3.6.1. We briefly considered the other strategies for the initialization step to see whether their relative performance compared to using the Garg-Könemann approach changes for different network settings. A small trial study suggests that this is not the case. We therefore do not include them in our subsequent studies. As before, we list results for the initialization step, after one round of column generation, and after the full algorithm.

Network Size. In a first study, we vary the network size between 100 and 1 000 nodes to assess the scalability of our method. Table 3.7 provides the respective results. The values for 1 000 nodes have to be considered separately, though, as we reached our hard time limit of three hours for this network setting. In 38% of the respective problem instances, we had to stop before finding an optimal solution. The starred numbers in Table 3.7 represent results that depend on values for which this occurred.

Our simulations show a steep increase in the runtimes for larger sensor networks. The values for the initialization step and after performing the column generation once suggest a polynomial growth in the number of nodes. This is actually expected behavior. Recall that performing only one iteration of the column generation approach corresponds to solving an LP that can be solved in polynomial time. Likewise, the Garg-Könemann approach and the wMGDC heuristic used during initialization have a polynomial time complexity. The runtimes of the full algorithm grow exponentially, though. This is also expected behavior as our main problem is $\mathcal{NP}$-hard. Interestingly, the maximum lifetime remains roughly constant over all network sizes. This seems obvious at first glance as the average coverage remains about the same over all network sizes. However, with a growing number of entities to cover (see Table A.1 for both properties), one would expect that it becomes more likely for one of these entities to

be less conveniently covered, leading to a shorter maximum lifetime. This effect is eclipsed by other factors, though, as the maximum lifetimes are always around 70% of what is theoretically possible based on the (required) minimal coverage for each entity. Even with the maximum lifetimes remaining roughly constant, we see that the number of covers required by the optimal solutions increases with the network size. This is most likely due to our approach not enforcing a minimal number of covers in an optimal solution. As larger problem instances offer a larger amount of possible covers, it is more likely that there exist multiple optimal solutions with different numbers of required covers. Now, considering only the initialization step, we see that its solution quality decreases for larger network sizes. This is not surprising as the approximation ratio of our wMGDC solver depends on the number of sensor nodes. The number of required covers also increases with the network size due to the same reasons as for the full algorithm.

Overall, we observe an expected strong dependence of the runtime of our algorithm on the network size. We conclude that our exact approach is capable of handling medium-sized network instances well, but for larger networks one would have to switch to other methods, e.g. to our approximation algorithm. The number of entities to cover is very high in our area coverage setting, though, much higher than the number of nodes (see Table A.1). We expect much shorter runtimes if our method is applied to target coverage settings as there are usually far less targets than sensor nodes.

Table 3.7: *Results of our approach when varying the number of nodes between* 100 *and* 1 000 *are listed. Starred numbers have to be considered separately as they depend on values for which we reached our hard time limit of three hours.*

	Initialization			**CG (1 iter.)**			**Full CG**		
nodes	time	error	covers	time	error	covers	time	covers	iter.
[#]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[#]	[#]
100	0.2	11.6	1 397	0.1	0.4	65	1.7	61	14
300	1.5	12.1	1 839	2.0	0.6	181	20.2	181	25
500	3.7	12.5	1 999	7.1	1.6	265	232.4	284	91
1 000	13.6	12.7*	2 204	28.2	2.8*	412	6 553.0*	527*	561*

Network Density. Next, we keep the number of nodes fixed at 300 and change the node density between 1.0 and 10.0 nodes per unit square. The first thing to notice from our results in Table 3.8 is that the runtimes increase with the node density. Thus, our problem instances become more difficult to solve. Table A.1 tells us that the number of entities we have to consider grows similarly as when varying the number of nodes. The resulting problem instances remain more manageable, though, as the maximal number of covers stays constant. This is reflected in the shorter runtimes compared to the results given in the last paragraph. We further see the number of iterations of the column generation approach as well as the number of covers in the optimal solutions

increasing with higher node densities. This is due to an increase in the total lifetime of the sensor network as every point in the monitored area A is covered by more sensor nodes on average. Considering the initialization step, we see that the Garg-Könemann approach yields much more covers with each increase in node density while the error ratios also become much larger. This is probably due to the approach not being able to remove previously computed covers nor to alter their duration. When the algorithm computes a new cover and assigns it with a shorter than the optimal duration, it will most likely compute another similar cover later on to fill this gap. This may happen recursively and leads to a large amount of similar covers with decreasing durations. It could have been prevented if the initial cover would have been assigned the optimal duration. Our assumption is supported by the results after the first iteration of the column generation approach. Recall that we only optimize the durations of the covers provided by the Garg-Könemann approach in this case. The number of covers used in the solution decreases drastically, and the error ratio is reduced back to normal levels as encountered e.g. in the last paragraph.

Overall, we see that an increasing node density makes the sensor network lifetime problem more difficult to be solved. It remains more manageable as when increasing the number of sensor nodes n, though, as this number directly impacts on an exponentially growing quantity, the maximal number of covers (2^n). The increasing node density leads to a more complex geometry and thus to a larger number of entities. However, this quantity only grows polynomially as it is bounded by $\mathcal{O}(n^2)$, see [BCSZ05].

Table 3.8: *Results of our approach when varying the node density between* 1.0 *and* 10.0 *nodes per unit square are listed.*

	Initialization			**CG (1 iter.)**			**Full CG**		
density [#/1]	time [s]	error [%]	covers [#]	time [s]	error [%]	covers [#]	time [s]	covers [#]	iter. [#]
1.0	0.7	9.1	1 015	0.5	0.0	137	6.3	136	40
2.5	1.5	12.1	1 839	2.0	0.6	181	20.2	181	25
5.0	4.5	16.2	3 285	5.4	2.0	216	86.7	217	73
10.0	18.2	22.6	6 110	12.5	5.4	236	490.5	243	236

Node Distribution. So far, we have only considered random node placement. We now take a look at how (perturbed) grid placement impacts on our algorithm. The respective results are listed in Table 3.9. It is evident that the more regular network settings are much more difficult to solve with our method. Though, they do not become more complicated with respect to the number of nodes or the number of entities to cover (compare Table A.1). Considering the results for performing one iteration of the column generation approach, i.e. for solving the LP once, we also see that the runtimes do not deviate much between each network setting. However, the total number of iterations increases steeply when we switch to more regular node distributions. A more

regular problem instance exhibits more inherent symmetry and thus more redundant covers that have to be considered. The column generation approach has to work through a lot of redundant covers before finding the few distinct covers that are needed for an optimal solution. When using perturbed grid placement, the problem appears to be more difficult to solve as when using the normal grid placement strategy, even though the latter yields a more regular problem structure. This is probably due to the problem instances that use this placement strategy requiring far less covers for an optimal solution. Thus, our method has to search less. It seems that the issue of many redundant covers is alleviated for completely regular problems due to the optimal solutions requiring less covers. Interestingly, our initialization step is much less impacted by the node distribution strategy as the subsequent column generation process. This might be due to the Garg-Könemann approach not looking for optimal solutions and thus not searching for the last few missing distinct covers.

We conclude that for less random network settings, it may be advantageous to apply a specialized solver that can exploit the inherent problem structure. Our approach, however, is aimed at solving the general problem and thus may perform weaker for certain degenerate network settings or problem instances.

Table 3.9: *Results of our approach with different node distribution strategies. We consider grid placement (g), perturbed grid placement (pg), and random node placement (rnd).*

	Initialization			**CG (1 iter.)**			**Full CG**		
node distr.	time [s]	error [%]	covers [#]	time [s]	error [%]	covers [#]	time [s]	covers [#]	iter. [#]
g	1.5	10.3	1 806	1.2	0.5	140	52	137	66
pg	1.6	11.1	1 952	1.7	0.7	174	129	172	90
rnd	1.5	12.1	1 839	2.0	0.6	181	20	181	25

Sensing Range. In the previous paragraph, we have seen that our problem becomes more difficult to solve, the more regular it becomes. Now, we are considering the reverse case. By allowing a variation in the sensing ranges of the nodes, we make the problem less regular. In the results listed in Table 3.10, we immediately see a very pronounced effect. The runtimes of the column generation approach decrease by a large amount as we allow a larger variation in the sensing ranges. The number of required iterations decreases likewise. This is due to two effects. First, the problem becomes easier to solve, the less regular it becomes. There is less symmetry and therefore less equivalent solutions. Thus, we do not generate as many similar covers before finding the ones contributing to an optimal solution. For the second effect, we need to take a look at the initialization step. As the problem becomes less regular, the runtimes and the numbers of computed covers increase slightly. However, it also yields approximations

that are closer to the optimal solutions. Thus, the Garg-Könemann approach provides better initial sets of covers for the column generation approach, which, in turn, has to perform less work. This can be seen especially well for $R = 1$, the maximum variation in the sensing range we consider. When performing the column generation once, we almost always obtain an optimal solution. The full algorithm rarely needs to perform any additional iterations. Results for other network settings are given in Table A.7 and Table A.8. Our general findings hold for them as well.

Overall, we obtain similar results as in the last paragraph. The less symmetry our problem instances exhibit, the easier they become to solve by our algorithms. This is actually good news for real-world applications as they are unlikely to offer completely regular conditions. For example, sensing ranges always vary by a small amount. And as we have seen, even a small variation by 1% makes the problem easier to solve.

Table 3.10: *Results of our approach with different sensing ranges. We consider varied sensing ranges for each node taken uniformly at random from* $[1, 1 + R]$*. Note that* $R = 0$ *corresponds to our default setting.*

	Initialization			**CG (1 iter.)**			**Full CG**		
R	time	error	covers	time	error	covers	time	covers	iter.
[1]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[#]	[#]
0.00	1.5	12.1	1 839	2.0	0.6	181	20.2	181	25
0.01	1.5	12.0	1 849	1.9	0.6	182	18.0	185	21
0.10	1.6	11.8	1 921	1.7	0.4	181	10.5	182	12
1.00	2.1	10.0	2 510	1.2	0.0	134	1.4	134	1

Concluding this section, we can say that our method is capable of solving a wide range of network settings efficiently. Only for very large networks of 1 000 nodes and with over 25 000 entities to cover, we sometimes hit our hard time limit of three hours. An important insight we gained is the fact that the problem becomes significantly more difficult to solve optimally with our general purpose method, the more regular it becomes. As soon as there is some variation, e.g. randomized node positions or sensing ranges, symmetries are broken and we need to consider far less similar covers. The next chapter introduces a different kind of problem for which we observe the reverse effect. It becomes significantly more difficult to solve, the more random the structure of the considered sensor network becomes.

3.7 Concluding Remarks

We presented the sensor network lifetime problem (SNLP) and introduced a more realistic interpretation for this basic surveillance problem, i.e. guaranteeing a minimum measurement resolution for a given area over a maximum timespan. The combination

of a simple model and a realistic interpretation sets this problem apart from many other theoretical problems and was one of the main motivations for us to start looking into this particular problem.

We developed an efficient linear-time approximation scheme for the basic problem. The approach takes advantage of two relaxation techniques to achieve both, a better approximation ratio and a lower time complexity than previous results. We allow sensing ranges to increase by a small amount when exploiting discretized node positions, and we are content with a slightly shorter lifetime when decomposing the problem into smaller subproblems and utilizing a geometric shifting strategy. In combination, this yields an efficient dual approximation scheme that offers better approximation ratios at a lower time complexity than previous approaches.

As a subproblem requires us to solve smaller instances of SNLP, we also considered exact solvers based on the linear programming formulation of our problem. We proposed a novel combination of existing techniques, applying the Garg-Könemann approach to compute an initial set of covers that is subsequently used by a delayed column generation approach to find an optimal node schedule. Minimizing covers before adding them to the tentative set of known covers was another crucial insight. Our proposed method runs faster and on larger problem instances than previous attempts at solving SNLP or similar problems. We further showed how to optimize the order of a set of covers to conserve energy if node state changes become an important factor in the total energy consumption of the sensor nodes.

Our approaches are centralized in nature and thus intended for a supporting role. They are well suited for application in simulational environments. For example, as our approximation algorithm scales linearly with the number of sensor nodes, it is ideal for studying very large sensor networks efficiently. Another possible field of application is in assessing the quality of distributed (and localized) algorithms. They scale to large network sizes by virtue of their very nature but usually do not come with any performance guarantees. Thus, if one wants to rate their results, (centralized) algorithms are required that can handle equally large networks while giving tight upper bounds. Our algorithms provide just that.

Finally, we hope that our novel proof of $\mathcal{NP}$-completeness will be recognized by the community and, henceforth, be referenced as (correct) proof for the sensor network lifetime problem (SNLP), i.e. the problem of continuously monitoring an area with battery-constraint sensor nodes of arbitrary capacity.

Outlook. Even though our theoretical results are very strong, they only mark an intermediate step. There is still a lot of potential to enhance the underlying model, e.g. by considering non-uniform or variable sensing ranges. We are confident that our approximation algorithm can be based more firmly in the field of computational geometry. By adapting our proofs for general low-dimensional metrics, the inclusion of obstacles, and variable sensing areas should become easy. Furthermore, a generalization

to higher dimensions and angular dependent sensing ranges would be possible as well as removing the dependence on squared partitions, which would give us more flexibility in setting up the required subproblems.

Similarly, our exact algorithm offers very strong results. However, an extension to more involved settings has to come next. We mainly focused on analyzing different techniques for solving the basic problem. These techniques should be easy to adapt to other settings, though, simply by exchanging the underlying linear program and modifying the required heuristics. Integrating our post-processing step for optimizing the order of covers into the linear program can be done at the same time by requiring a minimum number of covers next to a maximum lifetime. Finally, looking for other methods than the common column generation approach is an interesting direction as it could lead to further insights into the general problem structure.

Since we are considering sensor networks, a distributed implementation is a natural extension to our efforts. The structure of our approximation algorithm is already well-suited for a parallel implementation as each tile of each partitioning can be considered independently. This could be taken one step further, e.g. by allowing the sensor nodes to autonomously organize themselves into these partitionings. One node in each tile could be elected to compute an (exact) schedule for the tile, or one could apply a distributed heuristic for each tile. Our approximation scheme would still guarantee a solution for the whole network that is optimal up to a small factor. A completely distributed implementation, especially with little communication overhead, would require new approaches, though.

Chapter 4

4 Location-free Detection of Network Boundaries

Localization. Assign (something) to a particular place.

— Oxford Dictionary of English

The dictionary entry above already gives a fairly good impression of what localization implies in a sensor network setting. A priori, a sensor node has no positional awareness. It neither knows its absolute position nor its location relative to the other nodes of the network. Many applications require certain knowledge of the underlying network structure, though. Examples include intrusion detection and data gathering [WGM06] as well as mundane services like efficient routing within the network [FGG06, RRP+03] or event detection [DLL09] through changes in the network structure, e.g. due to fires or collapsing structures.

This demand can be met in many ways. Geographic coordinates provide the most accurate localization, but they need absolute positioning systems like GPS, a central infrastructure, or setting them manually. This is often not a viable solution, e.g. due to cost or energy constraints. Virtual coordinates offer a good substitution for many applications. Based on the network topology, they provide a rough sense of relative location between the sensor nodes. Sometimes even more abstract information about node positions in relation to the network is sufficient.

This chapter focuses on the last alternative—in particular, on how to efficiently decide for a sensor node whether it is in the interior of the network or on its fringes. Ideally, this classification is performed distributed and location-free, without relying on positional information.

References. The contents of this chapter are based on joint work with Markus Völker and Dorothea Wagner [SVW11a, SVW11b]. Contributions to the EC-BR algorithm were provided mainly by Markus Völker. Wordings of the above publications are used in this thesis.

4.1 Introduction

A lot of applications on sensor networks require certain knowledge of the underlying network topology, especially of the holes and boundaries. Examples include the tasks listed in the chapter preface as well as the surveillance algorithms described in Chapter 3. Having identified a band of nodes around the fringes of the network, intrusions can be detected reliably while the network lifespan is maximized. In many situations, holes can be used as indicators for insufficient coverage or connectivity within the network. Especially in dynamic settings, in which nodes can run out of energy, fail, or move, an online detection and update of holes and boundaries is inevitable.

Many different algorithms for boundary detection have been developed in the past. However, most of them come with certain disadvantages. Some algorithms rely on oversimplified assumptions concerning the communication model, or they require knowledge about absolute or relative node positions, both of which are usually not available in a large-scale sensor network. Other algorithms are not distributed or require information exchange over long distances, and therefore they do not scale well with the network size. Algorithms that rely only on local information usually produce many misclassifications. Furthermore, many of the existing algorithms are too complex for an actual implementation on real sensor nodes. Thus, there is still demand for simpler and more efficient algorithms for boundary detection.

4.1.1 Related Work

As there is a wide range of applications that require boundary detection, there is an almost equally large number of approaches to detect and classify network boundaries and holes. Based on the underlying principles, these approaches can be classified roughly into three categories: Geometrical approaches, statistical approaches, and topological approaches. Furthermore, there are multiple classification schemes for boundary detection in the previous work.

Geometrical Approaches. Algorithms use information about node positions, distances between nodes, or angular relationships to detect network boundaries and holes. Accordingly, these approaches require appropriate sensing equipment such as GPS devices to be available. Unfortunately, in many realistic settings this is either not the case, or the existing equipment is not accurate enough (e.g. when inferring distances from signal strength).

In [MS04], Martincic and Schwiebert describe an algorithm which requires each node to know the positions and communication links of its 2-hop neighborhood. With this information, the node determines whether it is surrounded by a circle of other nodes. If such a circle exists, it is used as witness that the node is located in the interior of the network. Deogun et al. [DDHG05] only require a node to be able to determine the

distances to its direct neighbors. The node selects four of its neighbors that are close to the node but far from each other. Then it checks whether three of these neighbors fully surround itself or not. The approach by Fang et al. [FGG06] needs nodes to know their position. Using a Delaunay graph and local searches, holes are identified. Zhang et al. [ZZF09] apply localized Voronoi polygons. Each node has to collect the positions of its direct neighbors. The work by Shirsat and Bhargava [SB11] only requires a node to know a clockwise order of its neighbors. Each node checks for empty cones and chordless paths in the connectivity graph of its 2-hop neighborhood. Luthy et al. [LGDH12] propose an algorithm that draws an actual image to decide whether the border of a node's communication range is covered wholly by the communication ranges of its neighbors. Node coordinates are required for this approach.

Statistical Approaches. This type of algorithms tries to exploit statistical properties such as node degrees to detect boundary nodes. As long as nodes are evenly distributed, this approach works quite well since boundary nodes usually have less neighbors than interior nodes. However, as soon as node degrees fluctuate noticeably, most statistical approaches produce many misclassifications. Besides, these algorithms often require unrealistically high average node degrees.

Prominent statistical approaches are by Fekete et al. [FKP+04, FKKL05] and Bi et al. [BTG+06]. Fekete et al. first analyze node degree distribution in a theoretical work. Their implementation in a second work requires data gathering over the entire network to compute a histogram of node degrees. Using the histogram, they determine a threshold value by which each node can classify itself as inner node or boundary node. In the approach by Bi et al., nodes only need information of their local neighborhoods. Each node compares its node degree with the average node degree of its 2-hop neighbors to decide whether it is on the network boundary.

Topological Approaches. Algorithms using this approach concentrate on information given by the connectivity graph and try to infer boundaries from its topological structure. They often require nodes to gather information of a large neighborhood or entail complex algorithms with high computational cost.

Funke [Fun05] and Funke and Klein [FK06] describe algorithms that construct isocontours and check whether those contours are broken. If a node detects that a contour is broken, it classifies the corresponding contour end-points as boundary nodes. The first algorithm requires that the whole network is flooded starting from some seed nodes. The second algorithm works distributed, based on 6-hop neighborhoods. The methods proposed by Ghrist and Muhammad [GM05] and De Silva and Ghrist [dSG06] detect holes by utilizing algebraic homology theory. They are centralized and rely on restrictive assumptions on the communication model. The algorithm of Kröller et al. [KFPF06] works by identifying complex combinatorial structures called flowers. Such flowers exist with high probability under some assumptions on the communication

model if the average node degree is above 20. The algorithm requires that every node knows its 8-hop neighborhood. An algorithm that works well even in networks with low average node degree is given by Wang et al. [WGM06]. The algorithm involves multiple steps, some of which require that the whole network is flooded. In [SSGM10], Saukh et al. propose an algorithm that tries to identify distinct patterns in the neighborhood of a node. Under certain conditions, they can guarantee that all nodes which are classified as inner nodes lie inside of the network. The algorithm is distributed and every node only needs information of its k-hop neighborhood. The radius h depends on the node density. For low density, $k = 6$ is used. For higher densities, it is possible to use smaller neighborhoods. The algorithms by Dinh [Din09] and Chu and Ssu [CS12] show similarities to our own EC-BR algorithm (see Section 4.4). Each node constructs a graph induced by its neighbors in exactly 2-hop distance. They check whether these graphs form closed circles. This is done by verifying the connectivity of subgraphs in Dinh's case or by tree construction and analysis in the approach Chu and Ssu. They also provide a proof of correctness of their method. Dong et al. [DLL09] describe a distributed algorithm that is based on topological transformations of the connectivity graph. It is especially aimed at locating small holes. The approach by Li and Hunter [LH09] needs to decide whether the sensing ranges of two nodes overlap by some means (e.g. by comparing sensing results). The graph induced by this information is used by their algorithm. Boundary node candidates are determined by finding circles in their 1-hop neighborhood. Using knowledge of their k-hop neighborhood, holes of up to $4k + 2$ hops in perimeter are detected. Yan et al. [YMD11] focus on detecting small coverage holes. They assume the communication range of each node to be two times its sensing range and that nodes know or can easily determine whether they are located on the outer boundary of the network. Their theoretical reasoning applies the topological Čech and Rips complexes. In their distributed implementation, each node needs to find a Hamilton cycle in its 2-hop neighborhood. In another work, Dong et al. [DLL$^+$12] also focus on discovering small coverage holes. They make the same assumptions as Yan et al. regarding communication ranges and periphery awareness.

Classification Schemes. Until recently, most boundary (or hole) definitions were based on an embedding of the connectivity graph. In [FGG06], Fang et al. determine the Delaunay triangulation of the node positions and remove edges of length greater than one. They classify faces of this reduced Delaunay graph with at least four edges as holes of the network. Boundary nodes are those nodes that induce these faces. In [KFPF06], Kröller et al. define boundaries according to a decomposition of the plane into faces based on the embedded connectivity graph. A face is called a hole if the perimeter of its convex hull exceeds a minimum value. Since the vertices of a face usually do not correspond to network nodes, the authors define boundary nodes to be the nodes on a cycle in the network graph surrounding this face. The approaches by Fekete et al. [FKP$^+$04, FKKL05] only apply a basic boundary definition for the

continuous case. Given a set of holes, a closed cycle is called a boundary if it separates the area of a hole from the area occupied by the network. A mapping of the continuous boundary to network nodes is not defined. Saukh et al. [SSGM10] classify a node as boundary node if there exists any feasible embedding of the connectivity graph in which this node is located on the boundary of the embedded graph. In [DLL09], Dong et al. propose a topological boundary definition. They define a cycle in the connectivity graph to be a topological boundary if, given an arbitrary embedding of the graph, the embedded cycle can be continuously transformed into a boundary of the embedded graph. Similar to us, Luthy et al. [LGDH12] base their boundary definition on actual node coordinates. They paint an image of the network in which each node is drawn as a colored disk of radius equal to its communication range. If the disk is adjacent to the background color, the corresponding node is considered as boundary node.

4.1.2 Contribution

We present a novel boundary detection algorithm that allows a node to decide solely based on the *connectivity information* of its *local neighborhood* whether it is a boundary node. Our approach uses multidimensional scaling to compute a two-dimensional embedding of the 2-hop neighborhood of each node. Based on this information, we reconstruct opening angles between the node and its neighbors to decide whether the node is surrounded by other nodes or located at a network boundary. The approach can be adjusted to either mark thin boundary outlines or broader bands—halos—around network boundaries. We show that this classification can be performed efficiently and in a distributed way.

The presented algorithm has several benefits over existing approaches. Unlike many other algorithms, it is strictly local and suited for distributed application. As it only uses connectivity information, no knowledge about absolute or relative node positions is needed. It does not require the underlying network to be based on a unit disk graph or rely on other simplistic assumptions. Our approach is very robust to non-uniform node deployment and variations in node degree. It can be used equally well to detect extensive boundaries or small network holes that can occur when single nodes fail or move. Above all, it is much easier to understand and implement than most existing approaches. All of these features make our algorithm perfectly suited for application in large-scale networks.

We compare our approach qualitatively and quantitatively to multiple previous approaches. In order to allow for an objective comparison, we provide an intuitive definition of *network holes* and classify network nodes into three distinct groups, *mandatory boundary nodes*, *optional boundary nodes*, and *interior nodes*. The classification scheme is based on actual node positions in relation to network holes and boundaries. Extensive simulations show that our algorithm, despite its simplicity, outperforms the other algorithms in most settings by detecting a higher percentage of boundary nodes correctly and, at the same time, misclassifying less interior nodes.

4.2 Models

Before detailing our boundary detection approach and presenting the results of our simulations, we need to introduce the models that we use to describe our sensor networks and to analyze our algorithms. We show how to construct the sensor networks that we use in our simulations, and we define the classification scheme used in the evaluation of our simulation results.

4.2.1 Network Model

We assume the nodes of our sensor network to be placed in the two-dimensional plane according to some *distribution strategy*. The connectivity graph $G(V, E)$, with graph nodes $v \in V$ corresponding to sensor nodes and graph edges $(u, v) \in E$, $u, v \in V$, to communication links between sensor nodes, is induced by a *communication model*. It describes which nodes can communicate with each other. An embedding $\mathbf{p} : V \mapsto \mathbb{R}^2$ of the connectivity graph G assigns a two-dimensional coordinate $\mathbf{p}(v) \in \mathbb{R}^2$ to each node $v \in V$. For easier reading, we normalize distances to the maximum possible communication range between sensor nodes.

Communication Models. A communication model determines whether two nodes can communicate directly with each other depending on their relative position and possibly on their surroundings. It can provide further information on the induced communication links, such as the expected *signal strength*. Our simulations consider two models that are frequently found in the literature.

In the *unit disk graph* (UDG) model, two sensor nodes $u, v \in V$ can communicate with each other, i.e. there exists a communication link between them if the distance between them, $||\mathbf{p}(u) - \mathbf{p}(v)||$, is at most 1. This model is widely used and its properties have been thoroughly analyzed in the literature, see e.g. [CCJ90].

The *d-quasi unit disk graph* (d-QUDG) model was first introduced by Kuhn et al. in [KWZ08]. The sensor nodes $u, v \in V$ can to communicate directly if $||\mathbf{p}(u) - \mathbf{p}(v)|| \leq d$ for $d \in [0, 1]$. For distances $||\mathbf{p}(u) - \mathbf{p}(v)|| > 1$, communication is infeasible. In between, communication is possible with a probability of 50%. Choosing $d = 1$ corresponds to the UDG model. The d-QUDG model was conceived as it is considerably closer to reality than the unit disk graph model while still being simple enough to be theoretically analyzed.

Both models are still far away from accurately reflecting the complex conditions found in real-life sensor networks. However, they can be (and are) used to assess the quality of algorithms in simulated settings with great success.

Signal Strength. A sensor node receives messages from a neighboring node as electromagnetic waves. The amplitude of these waves is denoted as signal strength. In

a vacuum, signal strength decreases with the second power of the distance. However reflections and loss due to obstacles as well as random effects impact the received signals and cause their strength to diminish faster.

We can still try to exploit signal strength information to improve our knowledge of the sensor network and, in turn, the results of our boundary detection. In one simulation setting, we analyze the impact of utilizing signal strengths. As signal strength is a very sensitive quantity, depending on many exterior factors such as node orientation or signal interference, we opt to only apply a very simple model. We differentiate between two states, strong signals and weak signals. We assume the signal strength between two nodes $u, v \in V$ to be strong if $||\mathbf{p}(u) - \mathbf{p}(v)|| \leq 0.5$ and weak otherwise, as long as they can still communicate with each other.

Distribution Strategies. A distribution strategy describes the spatial deployment of sensor nodes, i.e. where they are placed on the plane. This can be an independent process or depend on the positions of previously placed nodes. Our simulations examine two node distribution strategies.

By using the *random placement* strategy, nodes are distributed uniformly at random in the plane. This strategy models applications in which sensor nodes are arbitrarily scattered in the environment, e.g. when dropped from an aircraft.

The *perturbed grid placement* strategy places sensor nodes on a regular grid with grid spacing 0.5, translated by an additional offset chosen uniformly at random from $[-0.25, 0.25] \times [-0.25, 0.25]$. Each grid position is chosen once before any grid position is used for a second time. This strategy guarantees a more uniform node distribution with less variance in node degree compared to random placement. It models node deployments in which sensor nodes are placed in a regular pattern without having to watch closely for the exact placement. This strategy is frequently found in the literature on boundary detection.

4.2.2 Hole and Boundary Model

To evaluate and compare boundary detection algorithms quantitatively, well-defined definitions of holes and boundaries are required. We decided to take a very practical and intuitive approach at what to label as holes and boundaries. In short, we call large areas with no communication links crossing them *holes* and nodes on the fringes of these areas *boundary nodes*.

Hole Definition. We base our hole definition on the true embedding of the considered sensor network. All faces induced by the edges of the *embedded connectivity graph* $\mathbf{p}(G)$ are hole candidates. Similarly to Kröller et al. [KFPF06], we define holes to be those faces of $\mathbf{p}(G)$ with a minimum perimeter of h_{min}. The exterior of the network can be considered as an infinite face for our purposes. To avoid special cases, we treat

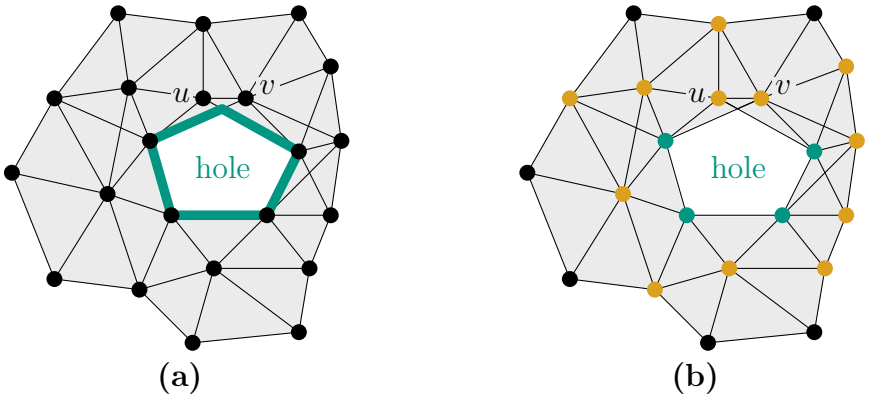


Figure 4.1: *Illustration of hole and boundary model. (a) Hole definition: Hole border (green). (b) Boundary node definition: Mandatory boundary nodes (green), optional boundary nodes (orange), and interior nodes (black).*

it as a hole during computation and evaluation. Figure 4.1(a) depicts a hole and its border according to our definition.

We believe that our choice, basing the hole definition on the network embedding, reflects reality better than solely basing the definition on topological properties as several other authors propose. Naturally, we only take advantage of real node positions for evaluation purposes. They are not used but to define our classification scheme. Our boundary detection algorithms work solely on connectivity information.

Boundary Node Definition. As seen in Figure 4.1(a), hole borders and node positions do not have to align. This leads us to the question, which nodes to classify as boundary nodes. For example, we could argue both ways whether to consider nodes u and v as boundary nodes. The correct choice might even depend on the underlying application. To circumvent this problem, we divide nodes into three categories:

- *Mandatory Boundary Nodes.* Nodes that are found exactly at the border of a hole are always considered to be boundary nodes.
- *Optional Boundary Nodes.* Nodes not located exactly at the border of a hole but within one maximum communication range of it may, but are not required to, be called boundary nodes.
- *Interior Nodes.* All other nodes must not be classified as boundary nodes.

The resulting node classification is shown in Figure 4.1(b). Mandatory boundary nodes form thin bands around holes, interrupted by topological structures as seen previously for nodes u and v. Together with the optional boundary nodes, they form a halo around each hole. Any node within the halo is at most one maximum

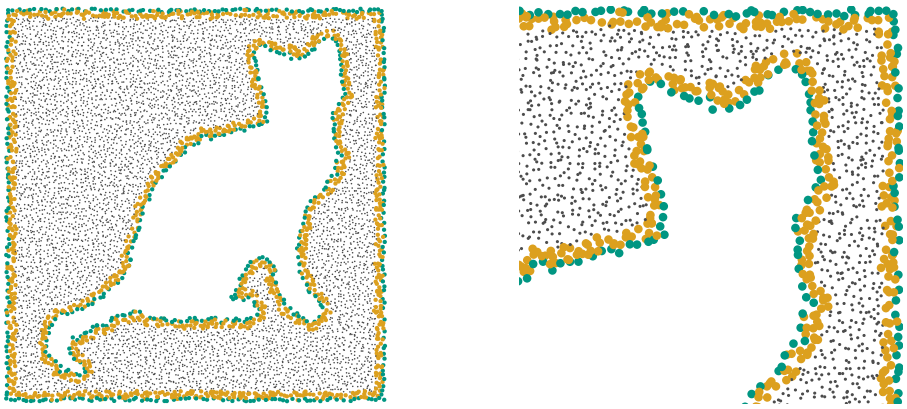


Figure 4.2: *Node classification example. Border outline of mandatory boundary nodes (green), halo of optional boundary nodes (orange), and interior nodes (grey). Full network and magnified upper right corner are shown.*

communication range away from the border of the enclosed hole. A sample classification of one of our network topologies is depicted in Figure 4.2.

We believe that this threefold classification scheme allows for a fairer comparison between different boundary detection algorithms. It is not tailored towards the strengths of our algorithm but solely built on observations of different network topologies. Only the classification of mandatory boundary nodes and interior nodes is strict. We think that their definition is reasonable enough so that their correct classification can be required. The introduction of optional boundary nodes provides some leeway to compensate for the unique nature of each boundary detection algorithm. Naturally, there might be applications that require a completely different classification scheme, but the one presented here should be sufficient for a wide range of interesting problems.

4.3 Multidimensional Scaling Boundary Recognition (MDS-BR)

Our novel algorithm for detecting network boundaries, *Multidimensional Scaling Boundary Recognition* (MDS-BR), can be seen as a hybrid approach. It combines elements from both, topological and geometrical approaches. Each node considers its local neighborhood and approximates the positions of these nodes based on connectivity information alone, using multidimensional scaling (see Section 2.3.2). Having obtained virtual coordinates via the network topology, the node can verify some simple geometric properties to decide whether it is located at a network border.

Our approach works distributed and location-free. All nodes decide independently whether they are a boundary node or an interior node by applying the base algorithm, followed by a possible refinement step.

4.3.1 Base Algorithm

At first, each node u gathers the connectivity information of its local 2-hop neighborhood $N_2(u)$ and constructs the neighborhood graph $G_2(u) = (N_2(u), E_2(u))$ induced by the communication links. Classical multidimensional scaling is applied as described in Section 2.3.2 to compute a two-dimensional embedding $\mathbf{p}(G_2(u))$ of the neighborhood graph. We approximate the required distances between all pairs of nodes by their respective distances in $G_2(u)$ (measured in hops). They are determined with the Floyd-Warshall algorithm of Section 2.2.2. Classical multidimensional scaling is well-suited for the embedding task as we only consider 2-hop neighborhoods, i.e. graphs with a diameter of at most four hops. We therefore do not need to compensate for drifting or folding effects that may occur when embedding large graphs.

With virtual coordinates given by the embedding, node u declares itself as a boundary node if it can verify some geometric properties of $\mathbf{p}(G_2(u))$. Below, we describe two classification strategies that offer structurally different classification results. The properties tested by them only depend on angular information of the embedded graph.

MDS-BR1. The first strategy leads to a thin outline of nodes marked as boundary nodes around each hole structure. This classification result is desirable e.g. to detect coverage holes in the network and to assess the extent of the entire network. We call our algorithm MDS-BR1 when using this classification strategy.

First, node u sorts its direct neighbors in clockwise order. It determines the maximum opening angle α between itself and two consecutive neighbors v, w as depicted in Figure 4.3(a). If α is smaller than a threshold value α_{min}, node u considers itself as an interior node. Otherwise, it is verified that nodes v, w have no common neighbors other than u in the cone spanned by the edges (uv) and (uw). If this is fulfilled, node u marks itself as a boundary node, otherwise as an interior node. Figure 4.3(b) illustrates this requirement.

The first requirement models our observation that boundary nodes exhibit a large gap in their immediate neighborhood due to the presence of a hole, whereas interior nodes are completely surrounded by other nodes. The second requirement is aimed at detecting and filtering miniature holes that are framed by four nodes with a perimeter of up to four maximum communication ranges. This topology is depicted in Figure 4.3(c). If an application is interested in this type of holes, we can omit the second requirement.

MDS-BR2. Our second strategy marks broader bands of nodes around holes as boundary nodes. These halos have a width of about one maximum communication range. They are useful in settings in which e.g. communication around network borders or load balancing in boundary structures become important. To distinguish from our first classification strategy, we speak of MDS-BR2 when using this approach.

Node u only considers nodes that are exactly two hops away. Similar to our first strategy, u sorts these nodes in clockwise order and determines the maximum opening

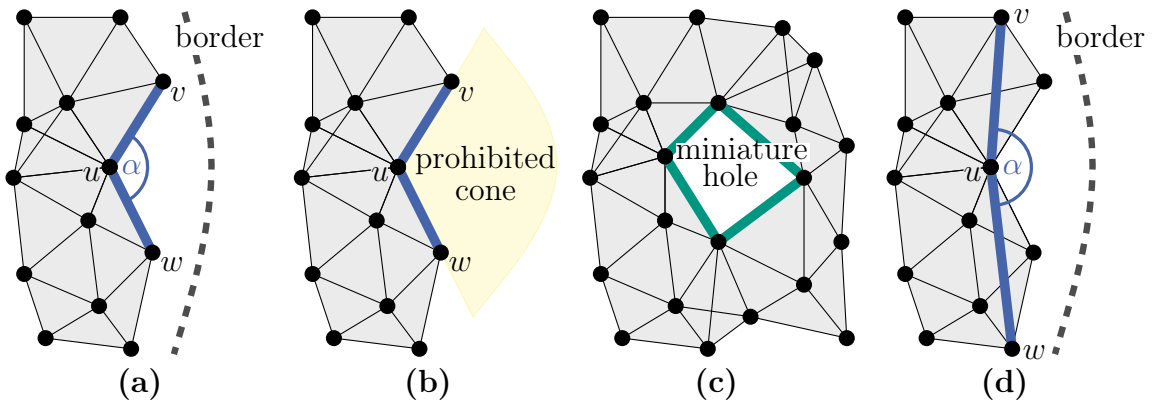


Figure 4.3: *Geometric properties used by MDS-BR. (a) Maximum opening angle to neighboring nodes. (b) Cone not containing common neighbors of nodes v, w. (c) Miniature hole. (d) Maximum opening angle to nodes in 2-hop distance.*

angle α between itself and two consecutive neighbors as in Figure 4.3(d). If α is larger than a threshold α_{min}, node u marks itself as boundary node.

Just as before, this angular requirement models the observation that boundary nodes show a large gap in their neighborhood. Considering opening angles to nodes that are further away from node u is less susceptible to classifying small, sparse structures as boundary nodes. This also allows us to omit a check for miniature holes as they are completely enclosed by our 2-hop neighborhoods. However, this strategy declares nodes as boundary nodes that are further away from the actual hole border—which can be a benefit as discussed above.

The running time of the base algorithm is dominated by the computation of the embedding. The asymptotic time complexity of multidimensional scaling is $\mathcal{O}(n^3)$, with $n = |N_2(u)|$ the number of considered nodes, as discussed in Section 2.3.2. Determining the pairwise distances needed as input for MDS with the Floyd-Warshall algorithm has the same asymptotic running time (Section 2.2) but much lower constant factors in practice. Thus, using a more efficient technique is not required unless the embedding can be computed more efficiently, too. Verifying the geometric properties takes $\mathcal{O}(n \log n)$ due the sorting involved. This amounts to a total running time of $\mathcal{O}(n^3)$ at each node u. We can generalize the result using $|N_2(u)| = \mathcal{O}(d_{max})$ [PG04], with d_{max} the maximum node degree of G. We obtain an asymptotic running time of $\mathcal{O}(d_{max}^3)$, which only depends on general network properties. Communication is limited to collecting the connectivity information of the 2-hop neighborhoods. If we assume synchronous communication in rounds and data aggregation, each node has to send at most two messages of size $\mathcal{O}(d_{max})$.

Our base algorithm offers further synergies with other applications running on the sensor network. If one of them computes virtual node coordinates that give reasonable estimates of the real positions, we can utilize them and skip the embedding process.

Linear Time Implementation. As our algorithm runs distributed on each node and only has to consider a 2-hop neighborhood, its running time is not very important. Still, computing embeddings becomes more and more time-consuming on dense graphs as the neighborhood size grows. We present a filtering step that runs in $\mathcal{O}(m)$, with $m = |E_2(u)|$, on each node u and allows the time complexity of MDS-BR to become asymptotically independent of the graph size. The basic observation is that dense graphs contain much more information than we actually need to find a good embedding. As long as the topological structure remains largely unchanged, each node u can thin out its neighborhood graph $G_2(u)$ to an average node degree d_{avg}. This approach is very effective in reducing the runtimes of MDS-BR as its time complexity grows with the third power of the neighborhood size. Moreover, as we shrink the neighborhood graphs to a fixed average degree d_{avg}, the time complexity of MDS-BR itself becomes asymptotically independent of the graph size.

Our proposed filtering step is very simple but highly efficient. After constructing its neighborhood graph $G_2(u)$ from connectivity information, node u begins the reduction process. First, it randomly removes nodes from its 1-hop neighborhood $N_1(u)$ until only d_{avg} nodes remain. Next, nodes are considered that are exactly two hops away from u, $N_{2\setminus1}(u)$. They are removed at random until $N_{2\setminus1}(u)$ is reduced to d_{avg}, scaled by the ratio $|N_{2\setminus1}(u)|/|N_1(u)|$ at their original sizes. The scaling guarantees that the ratio between nodes in one and two hops distance does not change. This aids in keeping the topology largely intact as does reducing $N_1(u)$ and $N_{2\setminus1}(u)$ separately. The filtering step runs in $\mathcal{O}(m)$ since we have to consider each edge at most once. Our simple approach is very effective, but it changes the topological structure of $G_2(u)$. As our simulations have shown, a reduction to $d_{avg} = 15$ still produces premium results. By reducing the number of nodes any further, classification quality starts to deteriorate as the embeddings become too much distorted.

If we accept longer runtimes for the filtering step, we can reduce the average node degree of $G_2(u)$ even further to $d_{avg} = 10$ while retaining very good results. We can prevent topological changes to the graph and, in turn, distorted embeddings by a *contraction process* during node removal. When a node u with adjacent edges (u, v), (u, w) is removed, we insert a new edge (v, w) into the graph. We also keep track of the number of hops that each of these edges represents in the original graph. This is required for multidimensional scaling to yield good embedding results. If we treated them only as single hops, MDS would compute a very distorted embedding based on these false assumptions. This approach is no longer linear in the number of edges, though. For each node removal, we have to look at $\mathcal{O}(n^2)$ pairs of nodes, with $n = |N_2(u)|$. Assuming $\mathcal{O}(n)$ node removals on a dense graph with $m = \Theta(n^2)$, we obtain a time complexity of $\mathcal{O}(m^{1.5})$.

To distinguish between both filtering approaches, we speak of *random filtering* in the former case and *contraction-based filtering* in the latter when discussing them in our section on simulational results (Section 4.6.5).

4.3.2 Refinement

Our base algorithm already gives very good results as shown in Figure 4.4(a). However, some isolated nodes remain marked as boundary nodes. This "noise" is highlighted in Figure 4.4(b). It is caused by a sensitivity of our algorithm towards very small holes, one might not be interested in, and by some real misclassifications. The effect is mainly pronounced for MDS-BR1. If desired, we can use a refinement step to remove most of these artifacts. The results of this procedure are depicted in Figure 4.4(c).

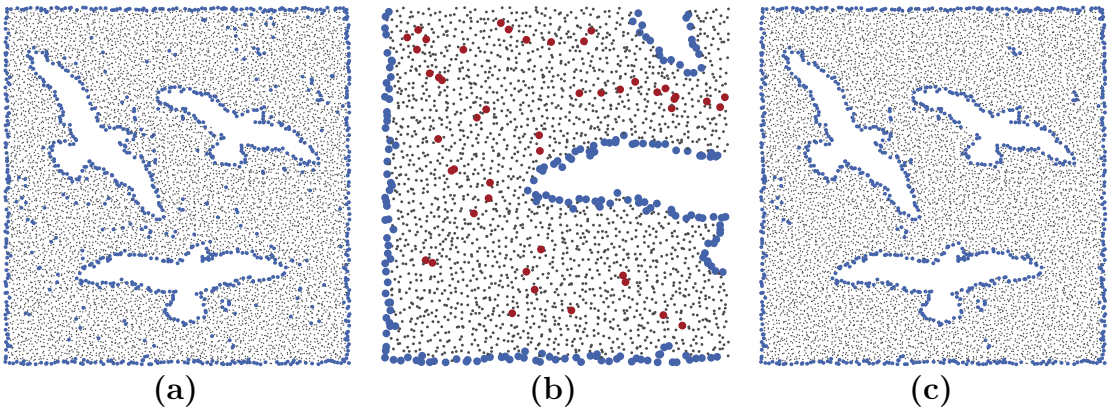


Figure 4.4: *Classification results of MDS-BR1 on a sample network. Boundary nodes are marked in blue. (a) Results of the base algorithm. (b) Zoom of lower left region. "Noise" is highlighted in red. (c) Results after refinement.*

The refinement is performed distributed on the set of nodes marked as boundary nodes by the base algorithm. We refer to these nodes as *tentative boundary nodes*. Each tentative boundary node u gathers the connectivity information of its r_{min}-hop neighborhood $\tilde{N}_{r_{min}}(u)$, restricted to nodes marked by the base algorithm. The graph induced by the this information is called $\tilde{G}_{r_{min}}(u)$. Node u computes a shortest path of maximum length between two nodes v, w in $\tilde{G}_{r_{min}}(u)$ that also contains u. If the length of the found path is less than r_{min}, node u reclassifies itself as interior node.

Our refinement step exploits the observation that artifacts occur isolated and only removes tentative boundary nodes that are not part of some larger boundary structure. The desired minimum size of the structures that we want to keep is specified by r_{min} as illustrated in Figure 4.5. Respectively to the base algorithm, only connectivity information is required by this procedure.

The time complexity of our proposed refinement step is bounded by $\mathcal{O}(n^3)$ for each node u, with $n = |\tilde{N}_{r_{min}}(u)|$. We assume that all pairwise shortest paths of $G_{r_{min}}(u)$ are computed with the Floyd-Warshall algorithm (see Section 2.2). Paths containing node u can be identified during the execution of this algorithm. The running time is not critical even for large values of r_{min} as the number of considered nodes is small. Figure 4.4(a) supports this statement. The base algorithm yields thin outlines of

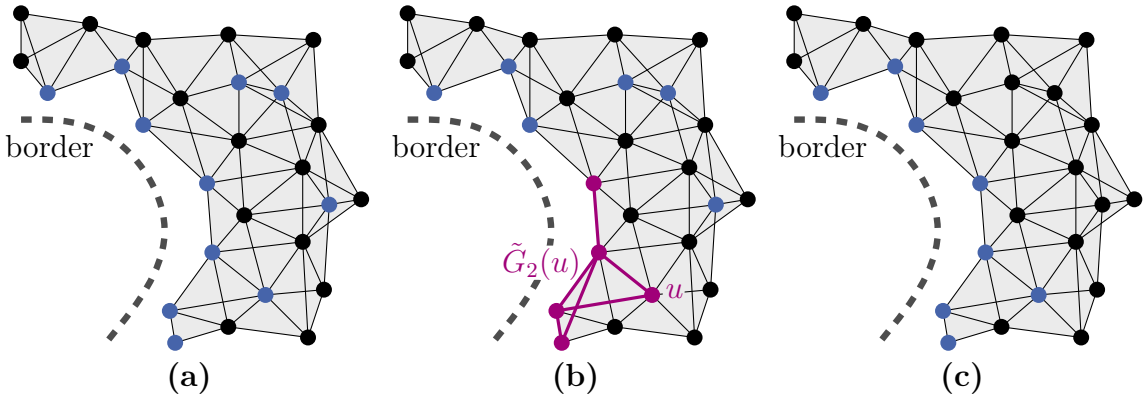


Figure 4.5: *(a) Tentative boundary nodes marked by MDS-BR1 (blue). (b) Graph induced by tentative boundary nodes reachable by u in two hops, $\tilde{G}_2(u)$ (purple) . (c) Boundary nodes remaining after refinement with $r_{min} = 2$ (blue).*

tentative boundary nodes around holes and scattered groups of isolated marked nodes elsewhere. Larger connected structures of tentative boundary nodes only occur within these boundary bands, and evidently they are much smaller than a complete r_{min}-hop neighborhood. The results of our simulations in Section 4.6.5 suggest that the growth rate of $\tilde{N}_{r_{min}}(u)$ is linear in r_{min}. This is not surprising as the tentative boundary bands can be considered as one-dimensional structures, whereas a complete neighborhood would be clearly two-dimensional. Taking into account these empirical results, this implies a running time of $\mathcal{O}(r_{min}^3)$, independent of network properties. Communication is, again, only required to gather connectivity information of a small neighborhood. With synchronous communication in rounds and data aggregation, each node has to send at most r_{min} messages. The maximum message size is given by the size of the marked neighborhood. It is in $\mathcal{O}(n)$ or, more precisely, in $\mathcal{O}(r_{min})$ as reasoned above.

4.3.3 Graph Embedding Strategies

Our boundary detection algorithm depends heavily on the quality of the computed graph embeddings. To improve the embedding results of MDS, i.e. to make them more accurate, we can take into account additional information. This yields better classification results but comes at the cost of a higher running time or with the issue of how to obtain this information. We introduce three graph embedding strategies that show what can be done at what additional cost. We could also look into other techniques that provide us with realistic angular relations between nodes. However, as multidimensional scaling is already a well-studied method that is as simple as efficient, we focus on the possibilities it has to offer.

If real node positions are known, there is no need to compute an embedding, and we can use them directly. This turns our algorithm into a geometrical approach. It is

not practical for deployment on most sensor networks, though, due to similar concerns as for other geometrical approaches. However, we can use this approach to gauge the performance of our classification strategies during simulations, independent of the embedding quality. We call this optimum strategy MDS_{opt}.

If we want to improve our results while still only relying on connectivity information, we can consider larger neighborhoods during the embedding process. However, this comes with an increased computation time. We opt to use 3-hop neighborhoods as the induced jump in processing time is still reasonable, while the gain in quality is already clearly noticeable. Moreover, the problems that multidimensional scaling faces on large graphs are still not very pronounced on this scale. The strategy is labeled MDS_3.

In a third, and most interesting, strategy, we incorporate signal strengths to better approximate node distances in $G_2(u)$. We only distinguish between strong and weak signals in our model. If node u receives a weak signal from another node v, we assume that they are far apart and set their distance to 1.0. Otherwise, we assume that the nodes are closer together and use 0.5 for their approximate distance. Distances between non-neighboring nodes are derived from these values as before. Multidimensional scaling uses these distance approximations as input to compute a more accurate embedding. We refer to this strategy as MDS_{SS}.

We have to take into account, though, that signal strength is a very sensitive quantity, depending on many external factors that make its evaluation prone to errors. To gauge the effects of making mistakes, we introduce two variants of MDS_{SS} that deliberately make mistakes when estimating node distances from signal strength values. One variant simply interprets a signal randomly as weak or strong. We label it MDS_{SSrnd}. The other variant models a worst case scenario, in which we assume to always obtain incorrect signal strength values. If the true signal was strong, we interpret it as a weak signal, and vice versa. This variant is called MDS_{SSerr}.

4.3.4 Performance Guarantees

Some boundary detection approaches give theoretical guarantees on their classification performance. They show that a certain percentage, or all, of the marked nodes are located close to a network border and, respectively, that unmarked nodes are nowhere near the edge of the network. For these guarantees to hold, certain assumptions on the network structure have to be fulfilled, e.g. concerning the communication model or the node placement strategy. The hole definition has to be chosen appropriately as well. Our work, in contrast, is not tailored towards a specific setting, and our hole definition tries to be as natural as possible. One might argue that the other algorithms can achieve good results even for settings in which their assumptions do not apply. However, for their proofs to work, they need to collect large neighborhoods (5 hops or even more), which entails a large communication volume and computational overhead. As we focus on efficient heuristics that can be used in real-world applications on large-scale sensor networks, we want to consider as little information as possible,

i.e. only the connectivity information of a 2-hop neighborhood. This makes it rather difficult to give any theoretical guarantees. The necessity to quantify the deviation of the computed embedding to the real node positions further complicates this task. However, if we assume known node positions and do not consider a possible refinement step, we can at least show some properties of our technique.

Unfortunately, we cannot provide guarantees on the correctness of our classification results. It is easy to construct degenerate network topologies in which a boundary node is classified as interior node, and vice versa. Figure 4.6 gives examples for both cases. However, all of these topologies require carefully placed nodes with long, uninterrupted communication links. This becomes less likely with growing node density as nodes move closer together and additional communication links emerge, interrupting degenerate structures and preventing classification errors that would be caused by them. As we cannot guarantee correctness, we resort to discussing the effects of erroneous classifications and try to assess upper bounds on the misclassification ratios. Our analysis is divided into a section discussing the possible misclassification of boundary nodes and one concerned with interior nodes. We assume $\alpha_{min} = 0.5\pi$ (90°) in either case. This is the same value we use in most of our simulations.

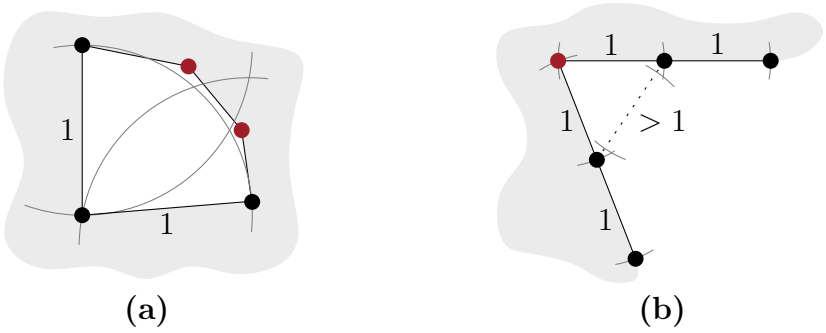


Figure 4.6: *Misclassification examples. Respective nodes are marked in red. Communication ranges are indicated by grey arcs. (a) Miniature holes give rise to falsely marked interior nodes. (b) Hastate structures cause boundary nodes to be classified incorrectly.*

Misclassification of Boundary Nodes. Any hastate structure as in Figure 4.6(b) gives rise to falsely classified boundary nodes. But how often can this occur in succession when considering an extensive boundary outline? We find that at most two consecutive boundary nodes are classified as interior nodes by MDS-BR1. This is illustrated in Figure 4.7. A boundary structure with two successive nodes classified as interior nodes comprises four nodes and requires at least three of them to be far apart as depicted in Figure 4.7(b). If we extend the structure by another node and try to construct a third consecutive interior node, there are two options. We can place the node in a position

that generates communication links to previous nodes and have the structure coil up on itself. This shifts the position of the network border and the two misclassified nodes become actual interior nodes as shown in Figure 4.7(c). Otherwise, the opening angle to the newly placed node is larger than $\alpha_{min} = 0.5\pi$. This results in the third node to be classified as boundary node as we see in Figure 4.7(d). To summarize, in a worst-case scenario at most two thirds of all mandatory boundary nodes are misclassified by MDS-BR1. However, as our sample topology requires precise positioning, this structure arises less likely with growing network density as the average node distance decreases.

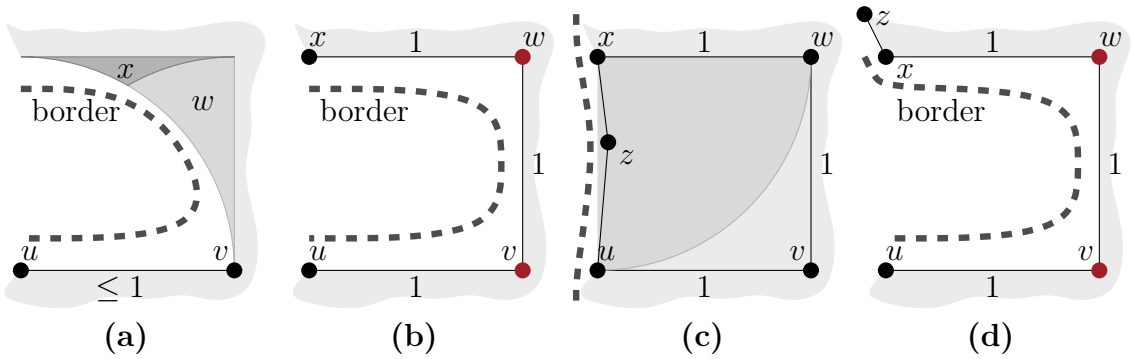


Figure 4.7: *Misclassifications of boundary nodes by MDS-BR1. At most two consecutive (mandatory) boundary nodes are classified as interior nodes. (a) Given nodes u and v, the position of node w is restricted to the light grey area if v should be classified as interior node. Respectively, the dark grey area gives possible positions for node x depending on the placement of w. With u, v moving closer together, the colored areas become smaller. (b) Possible node positions with v, w falsely classified as interior nodes (red). We choose extreme positions to maximize the placement options for node z. (c) Node z can be placed inside the light grey area to classify x as interior node. This placement permits communication links back to previous nodes. The boundary structure coils up and creates a new borderline with v, w becoming interior nodes. (d) Otherwise, node x is classified as boundary node, and we retain only two misclassifications in a row.*

We cannot give similar bounds for MDS-BR2 in general. As shown in Figure 4.8(a), all nodes around the hole are classified as interior nodes since the angle to their neighbors in 2-hops distance is at most 0.5π. However, this is only possible for small, closed structures. In an extensive boundary structure there has to be a large angle at some point as indicated by Figure 4.8(b), otherwise the structure coils up on itself. Here, we can assess an upper bound similar to Figure 4.7(c). By doubling each node and translating it by a small amount, we obtain a topology with similar properties. This is shown in Figure 4.8(c). The same reasoning as for MDS-BR1 gives an upper bound of 4 on the number of consecutive misclassifications of boundary nodes.

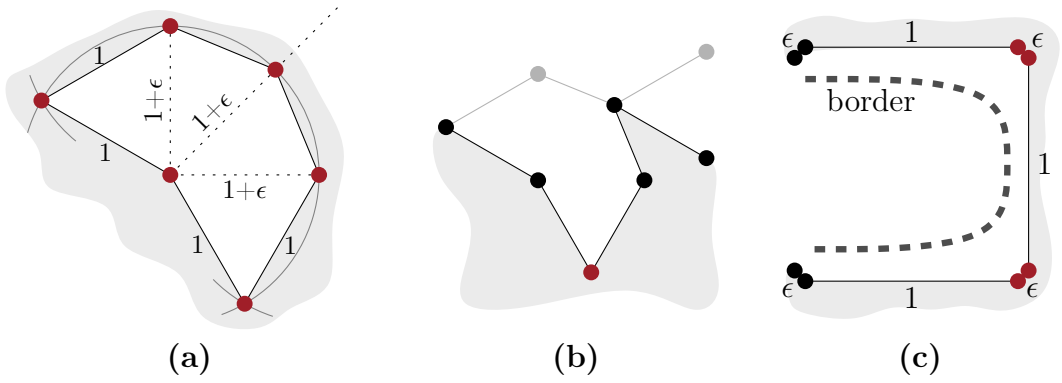


Figure 4.8: *Misclassification of boundary nodes by MDS-BR2. (a) All nodes around the local boundary structure are falsely considered as interior nodes (red). Communication ranges are indicated by grey arcs. (b) Extended boundary structures only allow for few successive misclassifications. The respective closed structure is implied in grey. (c) Similar structure to Figure 4.7(b). All nodes are doubled and translated by an epsilon. Adding another node gives the same results as Figure 4.7.*

Misclassification of Interior Nodes. Interior nodes marked as boundary nodes are found at the border of miniature holes. Even though falsely classified according to our hole definition, they can aid in uncovering areas of sparse coverage. Moreover, they are easily removed by our refinement step as there are no large clusters of miniature holes blending into each other but for very sparse networks. Figure 4.9 gives a minimal example for each of our classification strategies. With this, we can assess lower bounds on the perimeter of miniature holes that cause misclassifications. We obtain $2+\epsilon'$ for MDS-BR1 and $2+2\sqrt{2-\sqrt{2}}+\epsilon'$ for MDS-BR2. Due to the larger bound, our second strategy is less prone to misclassify interior nodes in random, sparse structures.

The random network structure does not allow us to give meaningful upper bounds on the misclassification ratio of interior nodes. Miniature holes can be clustered or chained indefinitely but with at least one correctly classifed interior node at each hole.

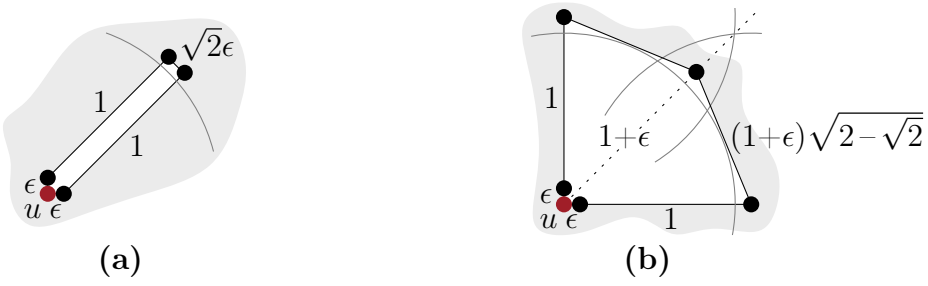


Figure 4.9: *Misclassification of interior nodes at the border of a miniature hole when using (a) MDS-BR1 or (b) MDS-BR2. Node u (red) is marked incorrectly. Communication ranges are indicated by grey arcs. Minimal holes are shown.*

4.4 Enclosing Circle Boundary Recognition (EC-BR)

In [SVW11b, SVW11a], we presented a second algorithm to detect network boundaries, *Enclosing Circle Boundary Recognition* (EC-BR). This thesis uses EC-BR mainly as another reference algorithm when assessing the performance of MDS-BR. We only introduce the basic concepts of the approach and refer to our previous publications and to the PhD thesis of Markus Völker [Vö12] for a more elaborate discussion.

The EC-BR algorithm is a topological approach but also contains some statistical elements. Its basic idea is simple and efficient. The nodes ignore their direct neighbors and only consider nodes that are exactly two hops away. For a node u, we denote the corresponding node set as $N_{2\setminus1}(u)$ and the induced subgraph as $G_{2\setminus1}(u) = (N_{2\setminus1}(u), E_{2\setminus1}(u))$. Based on the connectivity information in $G_{2\setminus1}(u)$, the node tries to decide whether it is surrounded by a closed path C. If such a closed path exists, one can be sure that the node is not a boundary node. Otherwise, this is seen as an indication that the node lies near a hole or border (compare Figures 4.10(a)-(c)).

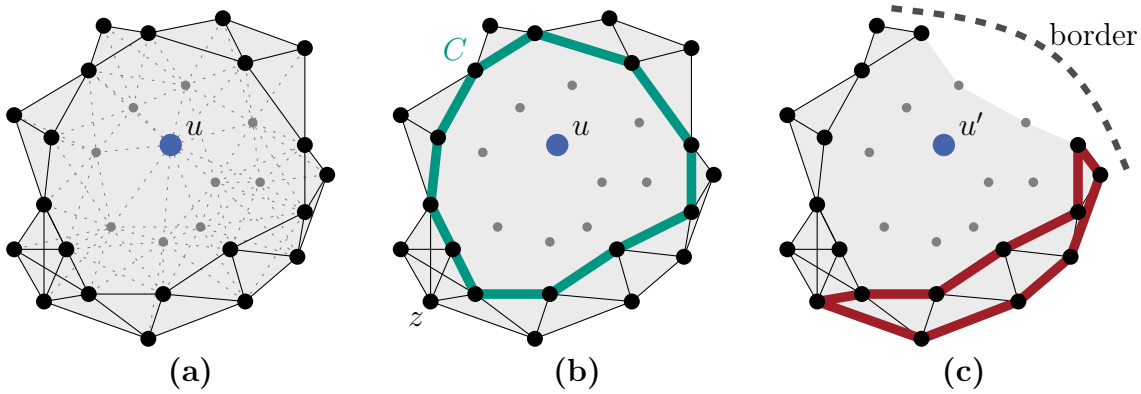


Figure 4.10: *Basic idea of EC-BR. (a) 2-hop neighborhood of node u (black). (b) Enclosing circle C (green). (c) Boundary node without enclosing circle. A non-enclosing circle is highlighted (red).*

4.4.1 Enclosing Circle Detection

How can we decide whether an enclosing circle exists? Knowing the actual node positions, this would be an easy task. However, we do not have this information and reconstructing node positions as in the previous section would be too imprecise for this task. In particular, we need to distinguish between an enclosing circle like in Figure 4.10(b) and a non-enclosing circle such as in Figure 4.10(c). The length of the circle is no sufficient criterion as both circles have the same length and only the first one is enclosing. However, there is a structural difference between both types of circles: The enclosing circle in Figure 4.10(b) sits around the hole like a tight rubber band.

There is no way to split it into smaller circles by using other edges of $E_{2\setminus 1}(u)$. On the other hand, it is easy to see that the circle in Figure 4.10(c) can be split into smaller circles. More formally, the first circle is a chordless circle. This implies, for each pair v, w of nodes on the circle, the shortest path between them using only circle edges is also a shortest path between them in $G_{2\setminus 1}(u)$. Thus, we simply need to look for a preferably long circle with this property in $G_{2\setminus 1}(u)$.

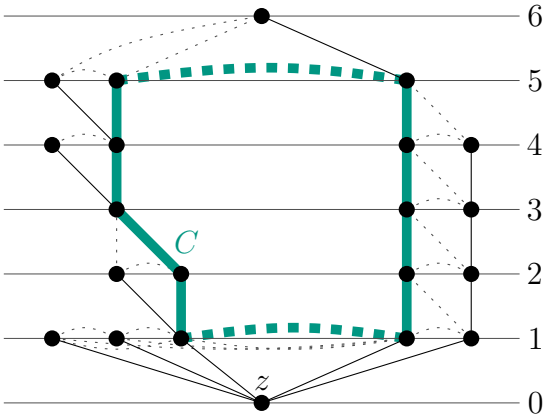


Figure 4.11: *Modified breadth-first search. Normal lines indicate the search tree, and dashed lines symbolize cross edges that close circles. Numbers denote distances in hops from z. The maximum circle C is marked in green.*

To find a maximum chordless circle, we can use a modified breadth-first search. The corresponding search tree for $G_{2\setminus 1}(u)$ of Figure 4.10(b) is depicted in Figure 4.11. We start the search at a random node z in $G_{2\setminus 1}(u)$ with maximum degree. In every step of the search, we maintain shortest path distances for all pairs of visited nodes. When a new edge is traversed, there are two possibilities: Either a new node is visited, or a previously encountered node is revisited. In the first case, we compute the shortest path distances between the new node and the previously visited ones. This can be done efficiently by inferring the distances from the distances to the parent node. In the second case, we have found a new circle in $G_{2\setminus 1}(u)$. The length of the circle is the current shortest path between the endpoints of the traversed edge plus one. Subsequently, we update the shortest path information of all nodes. During the search we keep track of the maximum length of the circles encountered so far. Depending on the maximum circle length that was found during the search, the considered node u is classified either as a boundary node or as an inner node.

In a unit disk graph, we are guaranteed that such a circle exists with length of at least 7 if node u is enclosed by other nodes. On the other hand, if node u lies somewhere near to a hole, it is not fully enclosed by other holes, and it is very unlikely that a large chordless circle exists. Figure 4.12 shows histograms of maximum circle lengths as found by our simulations on networks based on unit disk graphs and quasi

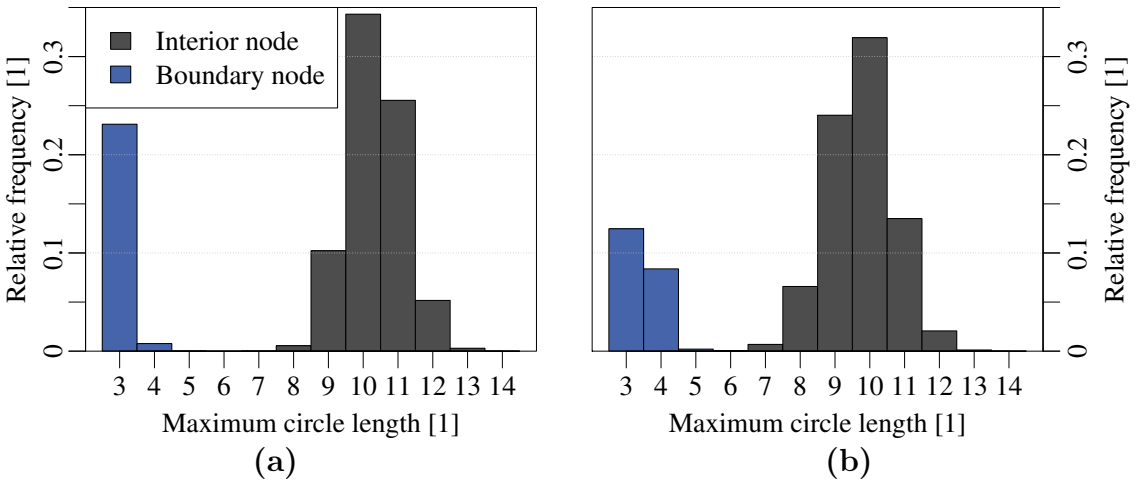


Figure 4.12: *Distribution of maximum circle lengths for different communication models: (a) UDG and (b)* 0.75*-QUDG. Colors depict classification results of EC-BR.*

unit disk graphs. We see two very well defined peaks that correspond to nodes with and without enclosing circles. Based on this distribution, we classify all nodes u with a maximum circle in $G_{2\setminus 1}(u)$ of length 6 or more as inner nodes and all other nodes as boundary nodes. Our simulations indicate that this statistical classification into nodes with and without enclosing circle works extremely well for both UDGs and QUDGs.

This kind of classification is extremely robust to variations in node degree. It does not matter whether $N_{2\setminus 1}(u)$ consists of a small number of nodes or hundreds of nodes, the same threshold 6 on the maximum circle length can be used to distinguish interior nodes from boundary nodes. The classification stays the same as long as we assume that the node density is sufficiently high so that inner nodes are actually surrounded by other nodes. This robustness distinguishes EC-BR from existing statistical approaches.

The asymptotic time complexity of the algorithm is in $\mathcal{O}(mn^2)$, with $m = |E_{2\setminus 1}(u)|$ and $n = |N_{2\setminus 1}(u)|$. Each edge in $G_{2\setminus 1}(u)$ is visited exactly once, and the update of all tentative pairwise distances is performed at most once at each visit.

Linear Time Implementation. The enclosing circle detection of EC-BR runs distributed on every single node and each node only has to consider its 2-hop neighborhood. Thus, its running time is virtually uncritical.

For the sake of completeness, we describe how to improve the search to run in linear time $\mathcal{O}(m)$ on each node u, with $m = |E_{2\setminus 1}(u)|$, if the underlying network has properties of a quasi unit disk graph. The key insight is that it does not make any difference for the classification if a node u is enclosed by thousands of nodes or by just enough nodes so that the circle is closed. Thus, in a first step, each node u filters $N_{2\setminus 1}(u)$ to obtain a small set of representatives. By considering each edge in $E_{2\setminus 1}(u)$ once, a maximal independent set $\mathcal{I}$ of $G_{2\setminus 1}(u)$ can be computed in time $O(m)$.

Based on packing arguments, the number of nodes in set $\mathcal{I}$ is bounded by a small constant for QUDGs. By iterating again over all edges, we assign each node v to the nodes of $\mathcal{I}$ that v is connected to. Next, two nodes in $\mathcal{I}$ are connected if there exists an edge $(v, w) \in E_{2\setminus 1}(u)$ with v and w assigned to these two nodes. As the size of $\mathcal{I}$ is asymptotically independent of the network size, this can be achieved in time $\mathcal{O}(m)$. Now, node u is enclosed by nodes in $\mathcal{I}$ if and only if it was enclosed in $G_{2\setminus 1}(u)$. Thanks to the constant size of $\mathcal{I}$, the time for the enclosing circle detection on $\mathcal{I}$ is asymptotically independent of the size of $G_{2\setminus 1}(u)$. The classification threshold γ has to be adjusted, though, as edges no longer correspond to 1-hop distances. Altogether, the enclosing circle detection can be done in time linear in the size of $E_{2\setminus 1}(u)$.

4.4.2 Classification Results

Figure 4.13 shows an example classification done by EC-BR. Nodes classified as boundary nodes are marked as blue dots. We can see that our approach recognizes broad boundaries, halos, meaning that even nodes which are only in the proximity of a hole or the outer boundary are classified as boundary nodes. The reason for this is that EC-BR checks whether the nodes in 2-hop distance form a closed circle. And even for nodes that are almost one hop away from the boundary such a closed circle does not exist. Another observation are the many small circles which do not belong to the large-scale boundaries. By looking at the magnification in Figure 4.13(b), we can see that the marked circles enclose small holes in the network.

We show how to remove both kinds of artifacts in the next section, the wide borders and the circles around tiny holes in the network. However, in many situations exactly this kind of information might be of interest. The detection of small holes, for instance, can be used to detect node failures or areas of insufficient coverage. And having a broader border might increase the fault tolerance and makes it easier to distribute messages along the border as it is guaranteed to be connected. Additionally, neighboring boundary nodes can divide their workload and thus extend the lifetime of their batteries.

4.4.3 Refinement

Sometimes one might not be interested in the tiny holes that occur in areas with low node density. We can extend EC-BR with a simple refinement technique that removes most of these small holes and also yields a thinner boundary. After the tentative boundary nodes have been marked with EC-BR, each marked node u checks whether a certain percentage γ of its neighbors are currently marked as boundary nodes. If this is true, the node remains a boundary node. Otherwise, the node changes its classification to being an interior node.

The basic insight behind this strategy is that a node that is near a hole is surrounded by other nodes that are marked as boundary nodes and by the hole itself. Under the idealized assumption that the connectivity graph is a unit disk graph, $\gamma = 100\%$ results

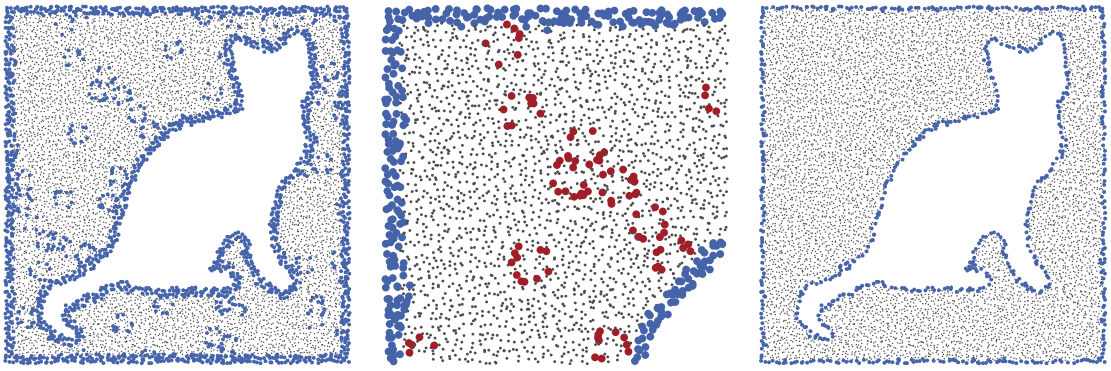


Figure 4.13: *Classification results of EC-BR. Boundary nodes are marked in blue. (a) Results of the base algorithm. (b) Magnification of upper left corner. Artifacts are marked in red. (c) Results after refinement.*

in very precise boundaries. For more realistic communication models, a threshold $\gamma \approx 70\%$ is more reliable. The results of this refinement strategy are depicted in Figure 4.13(c). Apparently, all nodes but the ones near large-scale holes are marked as inner nodes and the border is very precise.

4.5 Non-Local Network Structures

Our algorithm, MDS-BR, works on a local scale and only provides a single bit of information for each node—whether it is a boundary node or not. An underlying application might have more specific requirements on a boundary detection algorithm and require information about non-local network structures. We describe two extensions to our algorithm that make it possible to determine larger structures.

4.5.1 Large-Scale Holes

Our algorithm is sensitive to very small holes, but an application might be interested only in large-scale holes of a minimum perimeter p_{min}. To meet this demand, we perform MDS-BR to determine a tentative boundary. As the length of a shortest path around such holes is at least $\lfloor p_{min}/2 \rfloor$, we apply our refinement step with $r_{min} = \lfloor p_{min}/2 \rfloor$ to filter small holes. All boundary structures that do not permit shortest paths of length r_{min} are removed, and only nodes around large holes remain marked.

This approach can be used to recognize hole structures of arbitrary size with very little communication overhead. Only information about a small subset of all nodes, the boundary nodes, is communicated. The approach is no longer strictly local as we need to consider neighborhoods that can contain holes of the desired minimum size.

4.5.2 Connected Boundary Cycles

Some applications need to know connected cycles of boundary nodes around holes. We can extend our algorithm to identify these structures and relay this information. Due to their nature, boundary cycles cannot be determined locally. Fortunately, we only have to consider nodes previously marked as boundary nodes. The strategy is quite simple. We begin by determining boundary nodes with MDS-BR. Next, we select an arbitrary boundary node and perform a shortest path query within its halo to find an enclosing circle. As the halo consists of roughly the 1-hop neighborhood of the boundary, we are all but guaranteed a connected component around the hole. After a closed cycle is found, all nodes within the 1-hop neighborhood are classified as interior nodes and the cycle structure is stored. This is repeated until all nodes either belong to a connected boundary cycle or are classified as interior nodes. As a precaution to guarantee the existence of connected components around holes in difficult network instances, we can mark all nodes adjacent to a boundary node to enlarge the halos before commencing with the shortest path queries. Special care has to be taken if two large holes are less than one maximum communication range away from each other. In this case, the two boundary cycles might have to share some nodes.

This approach exhibits similar properties in terms of communication overhead and sense of locality as seen in the last section. Both ideas can be combined to reliable report only boundary cycles around large holes.

4.6 Simulations

We conclude the chapter by providing extensive simulations. We compare MDS-BR qualitatively and quantitatively to other previous approaches before analyzing its properties in more detail. All simulations were performed in parallel on Machine B.

4.6.1 Simulational Setup

Network Setting. We generate the topologies of our simulated sensor networks by iteratively placing nodes on an area of 50×50 maximum communication ranges according to one of the distribution strategies described in Section 4.2, perturbed grid placement and random placement. After each node is placed, communication links are added according to the UDG or d-QUDG model. Nodes are added until an average node degree d_{avg} is reached and at least 1 000 nodes have been placed. To generate holes, we apply hole patterns such as the ones in Figure 4.14.

We consider multiple distinct network settings in our simulations. Each setting is defined by the node distribution strategy, the average node degree, and the communication model. Our default setting uses perturbed grid placement, the UDG model, and $d_{avg} = 12$. Average network sizes for each setting are listed in Appendix B.

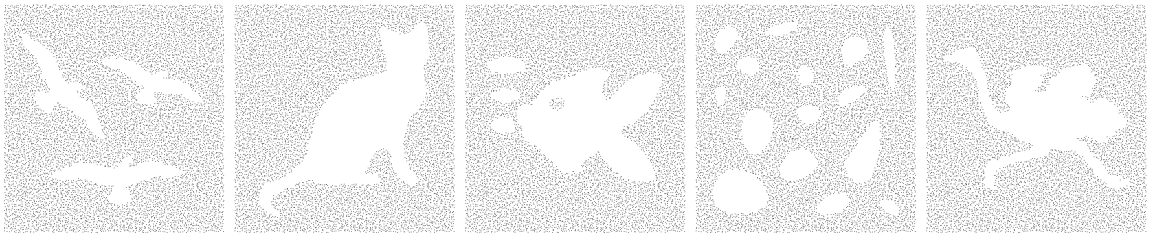


Figure 4.14: *Hole patterns. Sample node distributions are shown for our default setting with perturbed grid placement and an average node degree* $d_{avg} = 12$.

Measurement Procedure. We evaluate each of our network settings 100 times with different random seeds for each hole pattern in Figure 4.14. Faces with a perimeter $h_{min} \geq 4$ are considered as holes, e.g. squares of edge length 1 with no communication links crossing them. Our quantitative analysis gives mean misclassification ratios (false negatives) of mandatory boundary nodes and interior nodes in percent. For optional boundary nodes, we state the percentage of nodes classified as interior nodes.

We present our results as box-and-whisker plots. Boxes denote the range between the first and the third quartile of all test instances. Median values are marked. Whiskers give maximum and minimum values safe for outliers which are hidden. Entries with poor classification quality, i.e. with median values above 35% for most plots, are indicated by a star symbol. Network settings are distinguished by hue and saturation. We provide numerical values along with further results in Appendix B.

Different approaches use slightly different hole and boundary definitions. This might imply that these algorithms perform worse under our classification scheme. However, we believe that our distinction in mandatory and optional boundary nodes helps to achieve a fair comparison. Only nodes that are immediately at a network border or at least one hop away from any border are considered when determining the classification quality. In our opinion, it is reasonable to expect that these nodes are classified correctly by any algorithm. For all other nodes, it depends on the underlying application whether they should be classified as boundary nodes or not, and thus we do not rate them.

When assessing the classification quality of an algorithm, it is important to consider both, boundary nodes and interior nodes. With only one type, one cannot determine the quality of a boundary detection algorithm reliably. For example, by marking all nodes as boundary nodes, we would achieve a perfect classification of them while all interior nodes would be marked incorrectly. An effective algorithm has to balance the misclassification ratios of both types of nodes. Therefore, we evaluate the classification results for both, boundary nodes and interior nodes, in comparison.

Considered Algorithms. We compare the performance of our approach, MDS-BR, quantitatively to six well-known boundary detection algorithms: The geometrical approach by Martincic and Schwiebert [MS04], the statistical algorithm by Fekete

et al. [FKP+04], the centralized and distributed topological methods by Funke [Fun05] and Funke and Klein [FK06], the statistical approach by Bi et al. [BTG+06], and the EC-BR algorithm of our previous publications [SVW11a, SVW11b]. Our qualitative comparison in Section 4.6.2 also includes classification results of the global algorithm by Wang et al. [WGM06] and the topological approach by Saukh et al. [SSGM10]. We label each algorithm by its first author followed by the publication year, except for our algorithms. For the analysis, we provide our implementations of all algorithms according to the description in the respective publication and use the recommended parameter settings. Unless noted otherwise, our algorithms use these default settings: Both MDS-BR classification strategies use $\alpha_{min} = 0.5\pi$. MDS-BR1 further applies our refinement step with $r_{min} = 2$. We take a closer look at these choices in Section 4.6.5. EC-BR is used with and without refinement. The refinement threshold is $\gamma = 100\%$.

The number of existing approaches makes it impossible to include all of them in our simulations. Therefore, we tried to primarily select algorithms that assume similar conditions and constraints as our approach. We cannot compare ourselves directly to many of the other existing algorithms for various reasons. They might use additional information like absolute or relative node positions or the connectivity information of large neighborhoods up to the whole network. Some rely on certain network properties such as high average node degrees or the UDG communication model, while others require expensive operations like flooding the whole network or centralized computation. Some of these approaches might even achieve better classification results by utilizing more information or more expensive operations. However, our goal is to show that connectivity information of nearby nodes is sufficient to achieve impressive classification results with simple yet highly efficient algorithms and without any further assumptions on the underlying graph. For a comprehensive analysis, though, we include some geometrical and statistical approaches in our comparison.

4.6.2 Visual Comparison

We start by giving a visual comparison of the classification results of the considered algorithms in Figure 4.15. The network representations depict nodes that have been classified as boundary nodes in blue. The set of mandatory boundary nodes of the considered setting is marked in green as seen in Figure 4.15(l).

Both, MDS-BR1 and MDS-BR2, return faithful representations of the inner and outer network borders with almost no artifacts. Similar results are obtained by EC-BR. While the classification before refinement resembles MDS-BR2 with broad boundary bands, after refinement, the boundary is the same precise outline as for MDS-BR1. There are two striking differences, though. The results without refinement in Figure 4.15(c) also mark circles around miniature holes. When considering the results after refinement in Figure 4.15(d), we see that the classification by EC-BR is less precise in regions in which hole borders are near each other. Here, the boundary halos of both borders overlap, and the refinement routine can no longer differentiate between them both.

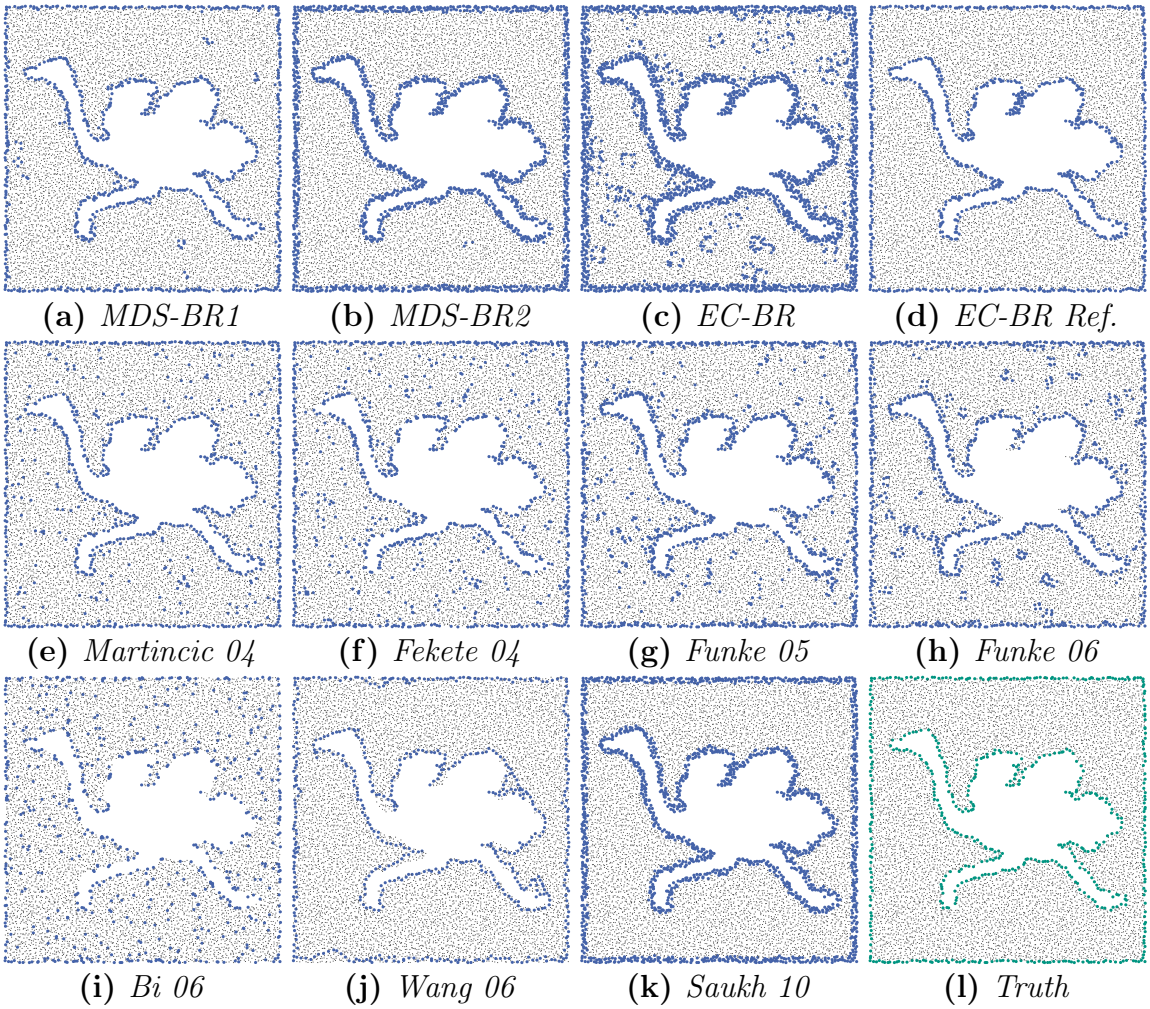


Figure 4.15: *Visual comparison of multiple boundary detection algorithms.*

Funke06 correctly identifies the boundaries with some artifacts. Similar to EC-BR, the apparent "noise" is caused by small holes which are surrounded by marked nodes. The results of Martincic04, Fekete04, and Funke05 show many random artifacts, though Martincic04 offers the most precise boundary outline. Moreover, Fekete04 failed to detect some mandatory boundary nodes. The results of Bi06 are extremely poor. The average node degree of our network is obviously too small for this statistical approach to work properly. Only some boundary nodes are marked correctly, and there are a lot of artifacts. We show the results of Wang06 in Figure 4.15(j). Their algorithm yields closed boundary circles with no artifacts, but due to its nature, marked boundaries are not always at the fringes of the network but shifted inwards. This characteristic led us to not include this algorithm into our quantitative analysis as it would result in an unfairly poor rating compared to the other algorithms. Still, the algorithm by Wang

et al. provides a fairly good sense of the general location of boundaries and performs satisfying for even lower average node degrees than our approach. The algorithm by Saukh et al. gives a result similar to MDS-BR2 and EC-BR without refinement. We see broad bands marked as boundary nodes and no artifacts. Unfortunately, their very impressive classification quality comes at a steep price. The algorithm suffers from a high computational overhead and requires each node to collect a large neighborhood. Including it in our quantitative analyses would have resulted in prohibitively long runtimes, even higher than the combined runtimes of all other algorithms.

Refinement. In order to convey a feeling for the influence of the refinement, we present additional classification results of our algorithm before and after refinement in Figure 4.16. Later in Section 4.6.4, we consider the question whether our refinement procedure can be used to improve the other algorithms. We also take a closer look at the refinement step of EC-BR.

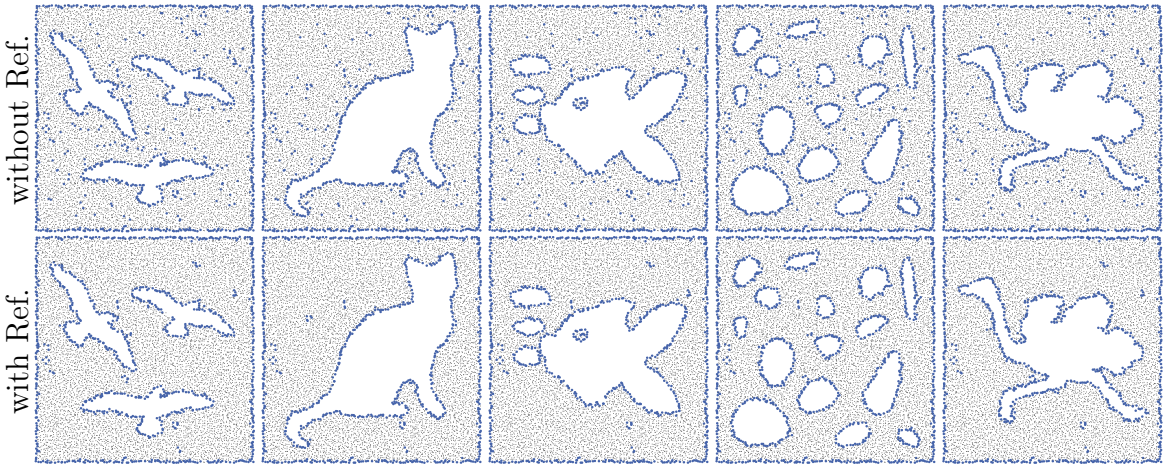


Figure 4.16: *Visual comparison of MDS-BR1 with and without refinement.*

4.6.3 Quantitative Analysis

Network Density. This paragraph considers how classification performance depends on the average node degree d_{avg} of the network. We provide a quantitative comparison in Figure 4.17, in which the percentages of false classifications for mandatory boundary nodes and interior nodes are shown with increasing values of d_{avg}. Most remarkably, MDS-BR2 classifies almost all nodes correctly except for the smallest node degree. This extreme setting, however, causes problems for all algorithms as additional small holes arise due to the overall sparse connectivity. EC-BR with refinement fares slightly worse, but it is still better than the other previous approaches. Only Martincic04 comes close, but good classification results are expected of a geometrical approach.

Our algorithms are still on par in case of MDS-BR1 or perform much better as seen for MDS-BR2. The falsely classified interior nodes of EC-BR before refinement are mostly due to the algorithm detecting very small holes that fall below our set threshold. The performance of MDS-BR1 on mandatory boundary nodes is surpassed by Fekete04 on denser graphs. Interior nodes are still detected more accurately by our approach, though. The improvement of Fekete04 on these graphs is reasonable since, as a statistical approach, it is optimized for dense networks. Funke06 achieves better results for interior nodes on denser graphs, but the results for mandatory boundary nodes deteriorate even more. As expected, the centralized variant, Funke05, performs much better in this respect. Bi06 offers a good classification of interior nodes, but this comes at the cost of a very poor performance on boundary nodes. Only for very dense networks its overall quality improves.

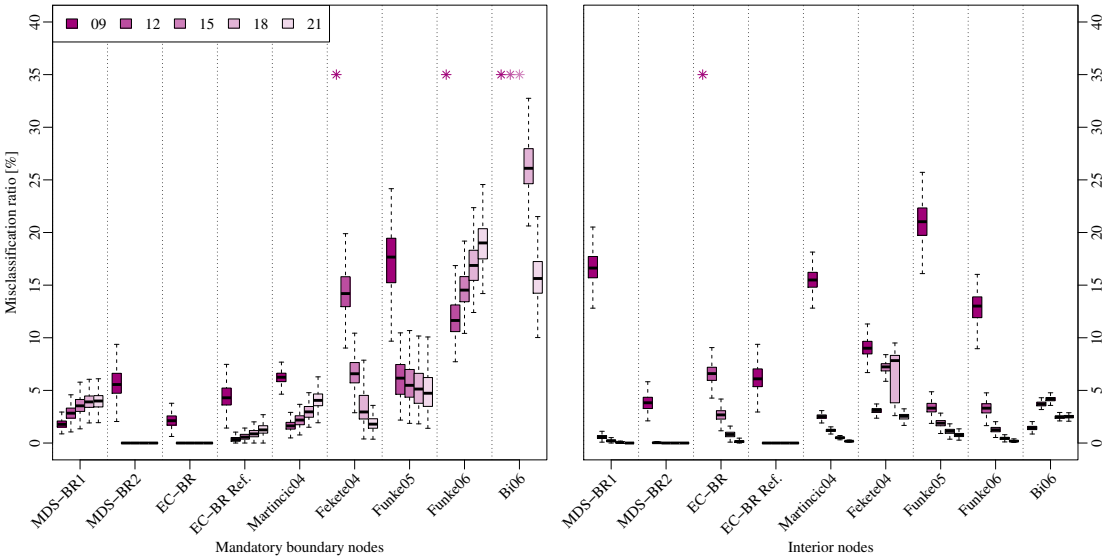


Figure 4.17: *Misclassification ratios (false negatives) in percent for average node degrees between* 9 *and* 21*.*

Figure 4.18 gives results for optional boundary nodes. We state how many nodes within one maximum communication range of a hole are not classified as boundary nodes, excluding mandatory boundary nodes. Here, the results of EC-BR are of particular interest. Before refinement, EC-BR classifies almost all of the optional nodes as boundary nodes while still providing a strict separation to the interior nodes. This makes the algorithm well suited for situations in which one is not only interested in the outermost boundary nodes but also in nodes close to holes. A similar behavior is seen for our algorithm, MDS-BR2, though the halo of boundary nodes is less distinct as only three quarters of the optional nodes are marked as boundary nodes. The other algorithms could approximate similar boundary bands using their results, but this would require an additional step, while here the solution is provided directly. With

MDS-BR1, we offer the other extreme, over all node degrees it classifies less optional nodes as boundary nodes than most of the other approaches, while still recognizing almost all mandatory boundary nodes correctly. A similar behavior can be seen for EC-BR after refinement and Martincic04. Especially the results of the latter are noteworthy. Martincic04 classifies almost all optional nodes as interior nodes while still providing a good separation to the mandatory boundary nodes and thus offering very precise boundary outlines. But we already expected premium results of an approach that takes into account node positions.

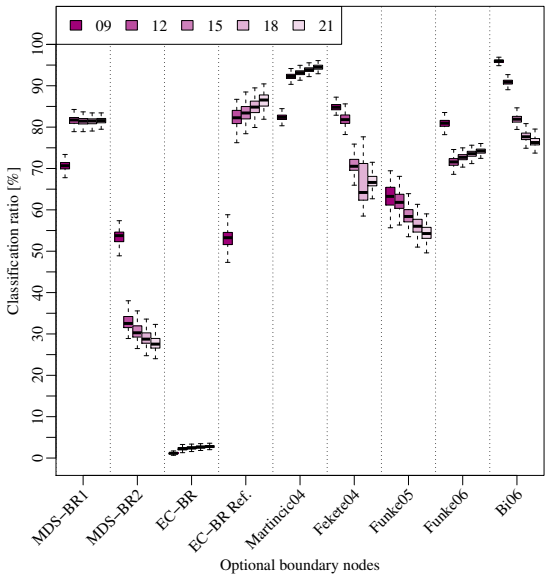


Figure 4.18: *Classification ratios of optional boundary nodes as interior nodes in percent for average node degrees between* 9 *and* 21*.*

We conclude the paragraph with a visual impression of the considered algorithms in Figures 4.19 and 4.20. The number of apparent classification artifacts increases rapidly for average node degrees of 10 and below. One reason is that for low node densities, many miniature holes emerge as seen for the true classification of the mandatory boundary nodes. Thus, most nodes are close to a hole and (correctly) classified as boundary nodes. Accordingly, for such sparse networks different algorithms and boundary definitions might be more appropriate. All algorithms we are considering exhibit the same principle behavior. However, apart from our two approaches, EC-BR with refinement, and Saukh10, all of them show a lot of "noise", i.e. interior nodes falsely marked as boundary nodes. Saukh10 offers very good results, but there are still more random misclassifications than for MDS-BR2. EC-BR before refinement is obviously not suited for sparse networks as it is too sensitive to miniature holes. Especially Fekete04 and Funke05 have problems detecting the boundary correctly. Furthermore, Bi06 is clearly unable to cope with our sparse networks. Only few nodes are marked as boundary nodes and many of them by mistake.

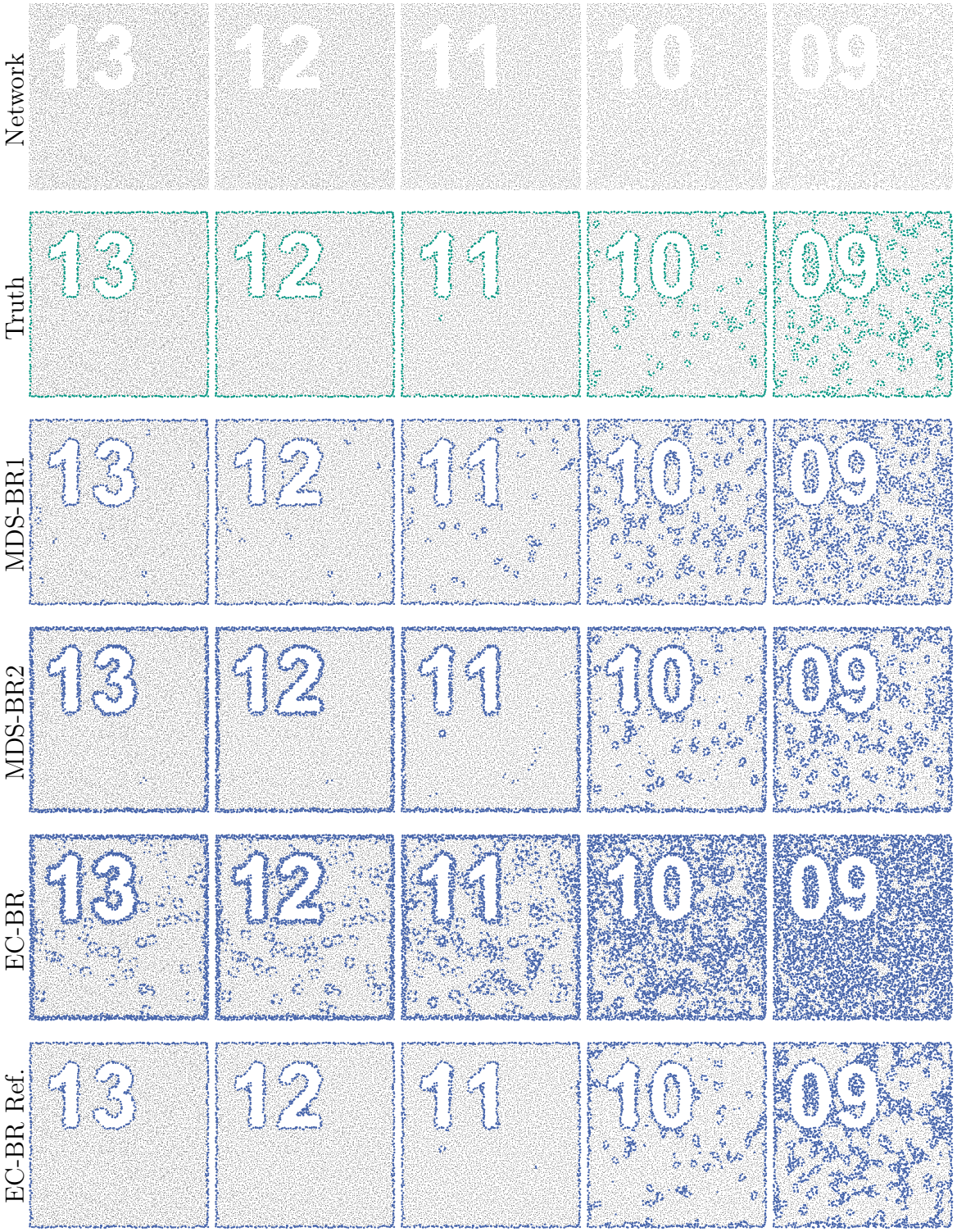


Figure 4.19: *Influence of average node degree on classification results. The number states the average node degree of the respective network.*

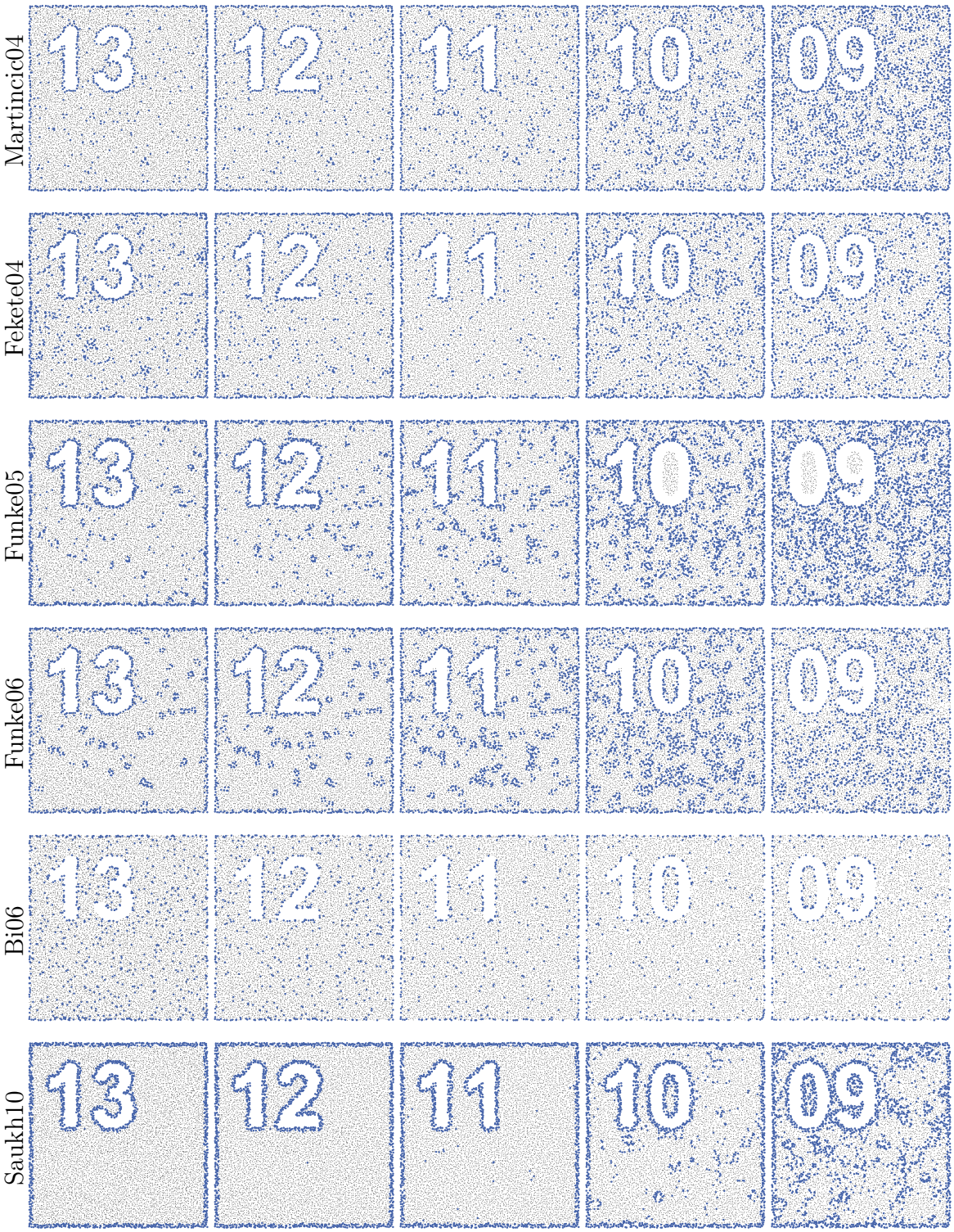


Figure 4.20: *Influence of average node degree on classification results. The number states the average node degree of the respective network.*

Random Placement. Perturbed grid placement ensures low variance in node degree over the entire network and few very small holes. Random node placement, however, leads to many gaps in the network that have to be detected. Figure 4.21 gives an impression of the expected mandatory boundary nodes in both settings. This paragraph considers the performance of the algorithms in this more difficult setting. A regular network structure is especially well suited for statistical boundary detection algorithms. For this reason, we expect Fekete04 and Bi06 to yield many misclassifications in a random placement setting, while our algorithms should still produce reasonable results.

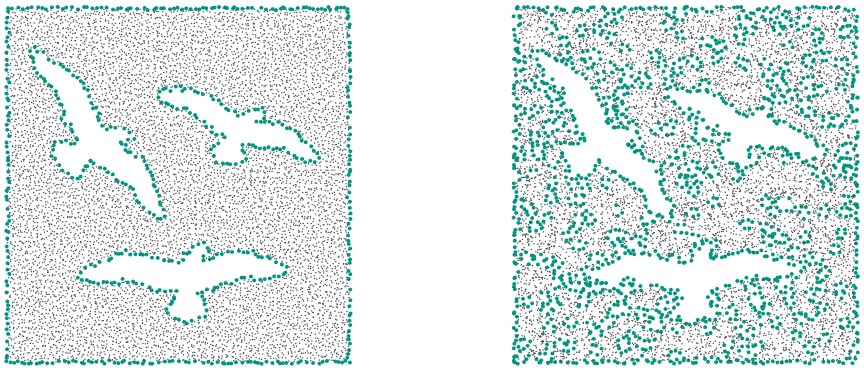


Figure 4.21: *Comparison of network boundaries depending on the node placement strategy: (a) Perturbed grid placement, (b) random placement.*

Figure 4.22 compares the performance of all algorithms for both placement strategies and $d_{avg} = 12$. The performance of the existing algorithms decreases drastically compared to perturbed grid placement. The results of MDS-BR and EC-BR, however, only decrease noteworthy for interior nodes. The classification of these nodes is still on par with or even better than for the other approaches, though. For mandatory boundary nodes, our results together with EC-BR remain much better than the next best competitor. Only Martincic04 comes close, but as it uses node positions, it should yield good results. The increased misclassifications of interior nodes are partially due to our algorithms detecting very small holes with a perimeter of less than four maximum communication ranges that occur frequently in a random placement setting.

In addition, we consider the influence of the network density when using the random placement strategy. Figure 4.23 shows the classification results. We see that all algorithms perform better with increasing network density. For some algorithms only the quality ratio of the mandatory boundary nodes or of the interior nodes improves, but the other one does not deteriorate. It is striking that Funke06 performs very poorly in this setting compared to the other algorithms. Only the classification quality of Bi06 is worse. However, for the highest node degree, Bi06 surpasses the results of Funke06 as the network starts to be dense enough for this approach to work properly. Overall, EC-BR and MDS-BR dominate the other algorithms for both, mandatory boundary

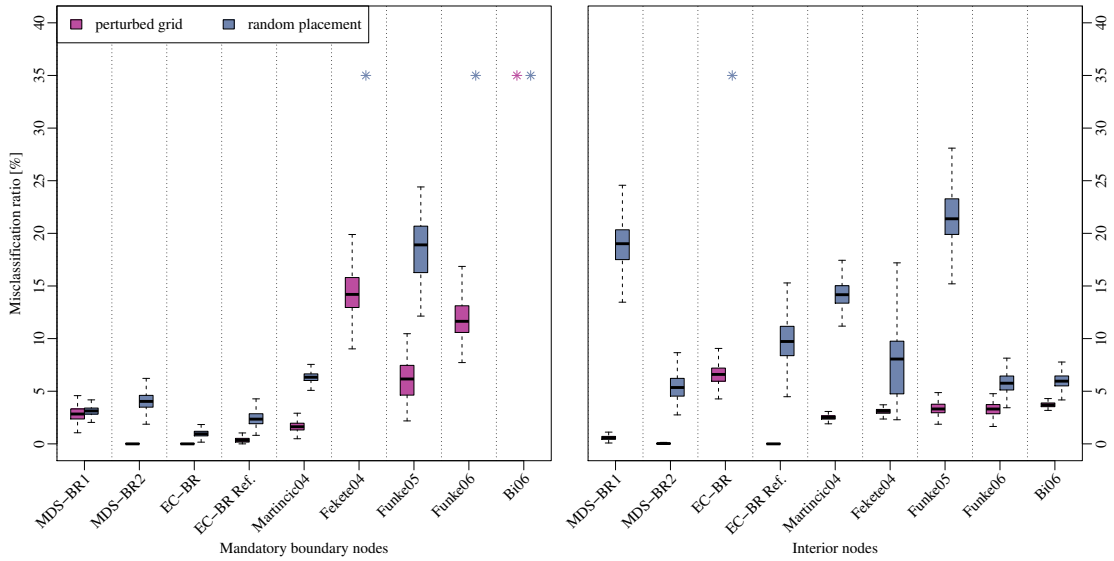


Figure 4.22: *Misclassification ratios (false negatives) in percent for perturbed grid placement and random node placement with* $d_{avg} = 12$.

nodes and interior nodes, with MDS-BR2 giving the best results. For sparse networks, we see an increased misclassification of interior nodes with a very high variance. As before, this is partly caused by detecting very small holes. Still, MDS-BR2 is able to classify more interior nodes correctly than all of the other approaches while retaining premium results for mandatory boundary nodes.

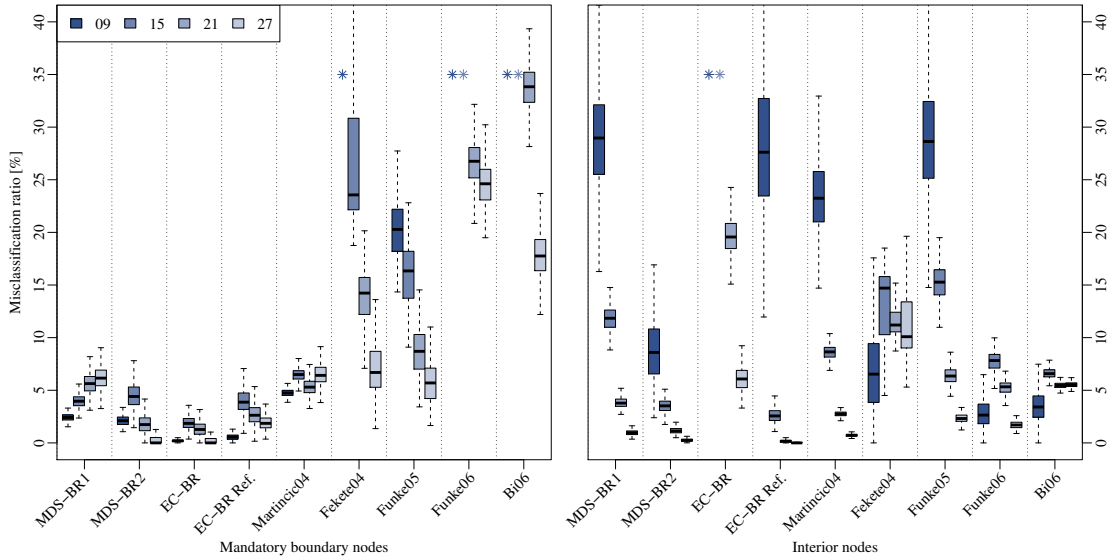


Figure 4.23: *Misclassification ratios (false negatives) in percent for random node placement and average node degrees between* 9 *and* 27.

Beyond Unit Disk Graphs. Unit disk graphs are frequently used for theoretical analyses and in simulations. They are motivated by the fact that under good-natured conditions, each sender has a transmission range which is roughly fixed. However, under realistic assumptions, the transmission range depends on environmental conditions and obstacles as well as on unpredictable effects such as interference and signal reflections. In this paragraph, we evaluate the algorithms under more realistic conditions. Uncertainties are taken into account by the use of the d-quasi unit disk graph model, which integrates the observation that short-range transmissions are usually successful, while long-range transmissions have some random behavior. Unlike for random node placement, the basic hole structure remains unchanged when using more uncertain communication models as we show in Figure 4.24.

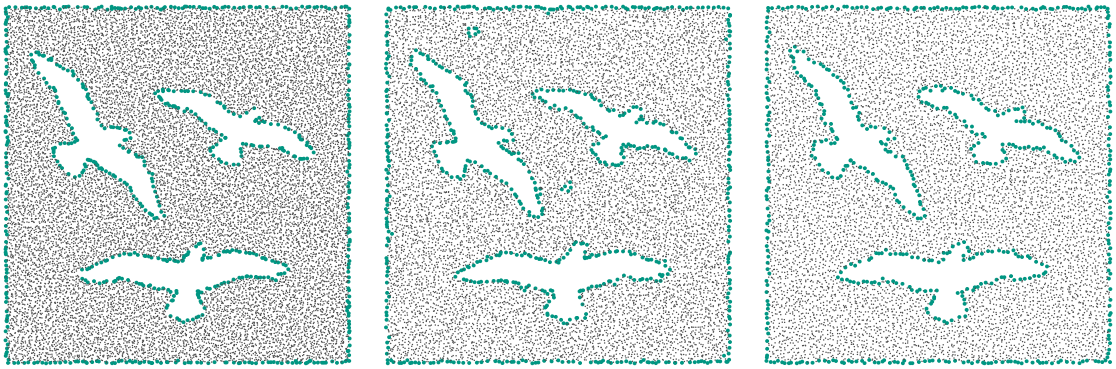


Figure 4.24: *Comparison of network boundaries depending on the communication model: (a)* 0.05*-QUDG, (b)* 0.75*-QUDG, (c) UDG.*

Even though the general structure of what we want to classify as boundary nodes does not change, the network structure does. To compensate for the more random communication links, EC-BR already applies a refinement with a smaller threshold of $\gamma = 70\%$. A refinement with $\gamma = 100\%$ would perform poorly because in QUDGs mandatory boundary nodes are not necessarily surrounded by other boundary nodes. Similarly, we need to alter the parameters of MDS-BR2 to obtain premium results. Figure 4.25(a) shows the classification of MDS-BR2 on a 0.25-QUDG network. At first sight, the classification appears to be good, but a closer inspection reveals that many mandatory boundary nodes are missing. We find that the maximum opening angles are shifted to lower values for quasi unit disk graphs. Section 4.6.5 goes into more details on the distribution of opening angles. To compensate for this effect we choose a smaller threshold value $\alpha_{min} = 0.3\pi$ for networks with a high amount of uncertainty. The results in Figure 4.25(b) already look promising as most boundary nodes are found. An additional refinement step can help to remove the "noise" as seen in Figure 4.25(c). We do not apply the refinement step in this paragraph, though, but refer to Section 4.6.4 and Section 4.6.5 for additional information on using refinement. If one does not want

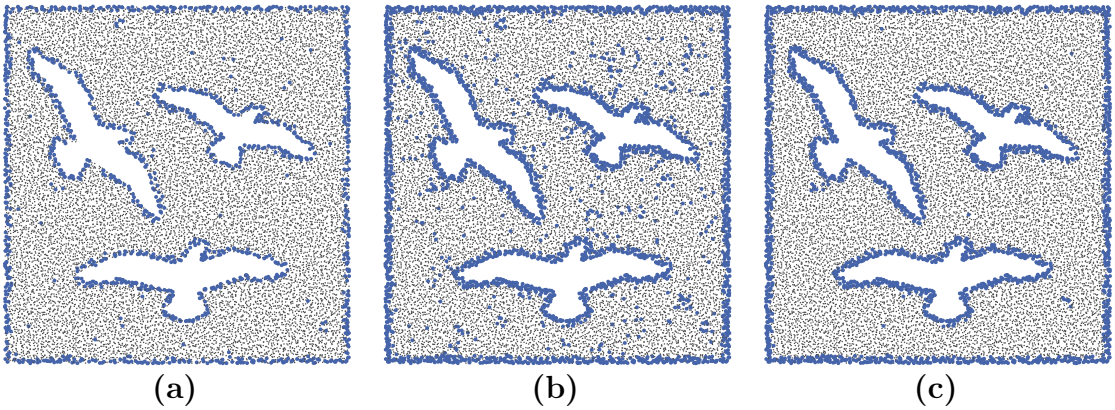


Figure 4.25: *Comparison of MDS-BR2 classifications on a* 0.25*-QUDG network: (a) Normal algorithm, (b) with* $\alpha_{min} = 0.3\pi$*, (c) and with additional refinement.*

to choose between two different threshold values, we propose to use $\alpha_{min} = 0.35\pi$ as a compromise. This offers almost the same results for 0.05-QUDG and 0.25-QUDG networks while still providing good results for 0.75-QUDGs and UDGs, especially when combined with refinement.

Figure 4.26 shows the performance of the algorithms for average node degrees between 9 and 18 in simulations on 0.25-QUDG networks. We can see that the results offered by MDS-BR2 are among the best. Only for the sparsest setting, the classification of interior nodes deteriorates noticeably. This is due to many nodes near miniature holes

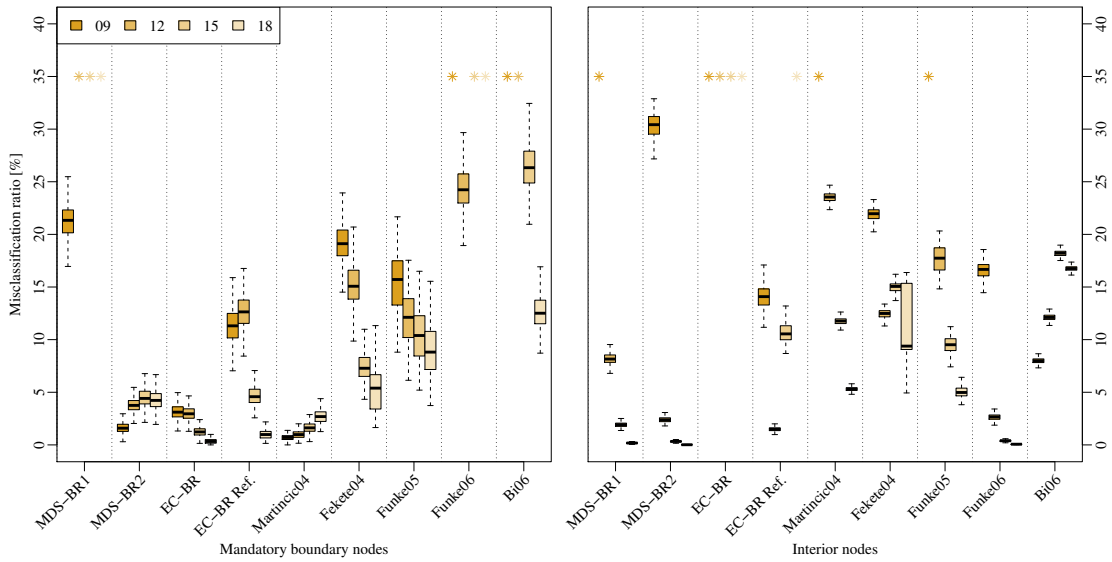


Figure 4.26: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with* $d = 0.25$ *and average node degrees between* 9 *and* 18*.*

being marked as boundary nodes. The increased error rate of MDS-BR1 for mandatory boundary nodes is a result of the base algorithm producing a tentative boundary set that is not necessarily connected. In consequence, the refinement classifies many correct boundary nodes as interior nodes as the connected substructures are not sufficiently large. EC-BR before refinement suffers from the same problem as MDS-BR2, just more pronounced and over the complete range of network densities. The result quality remains rather mixed, even after refinement. For $d_{avg} = 12, 15$, the classification is average but declines for both, very low and very high node degrees. Martincic04 detects boundary nodes well, but especially at low node degrees, many interior nodes are classified falsely. The other algorithms yield much higher misclassification rates than MDS-BR2 for mandatory boundary nodes and perform significantly worse at detecting interior nodes safe for the lowest node degree.

In Figure 4.27, we go a step further and compare the results on QUDG networks with a growing level of uncertainty to UDGs. As expected, all algorithms produce more misclassifications the more random the distribution of communication links becomes. Again, MDS-BR2 outperforms the other algorithms easily. Especially for QUDGs with a high amount of uncertainty, it detects boundary nodes very accurately while retaining a premium classification quality for interior nodes. EC-BR without refinement and Martincic04 offer better results for mandatory boundary nodes, but they fare far worse for interior nodes and thus yielding an altogether poor classification. Overall, all algorithms perform significantly worse than in our UDG simulations. Only MDS-BR2 is able to retain a similar classification quality for both, mandatory boundary nodes and interior nodes, for these test instances.

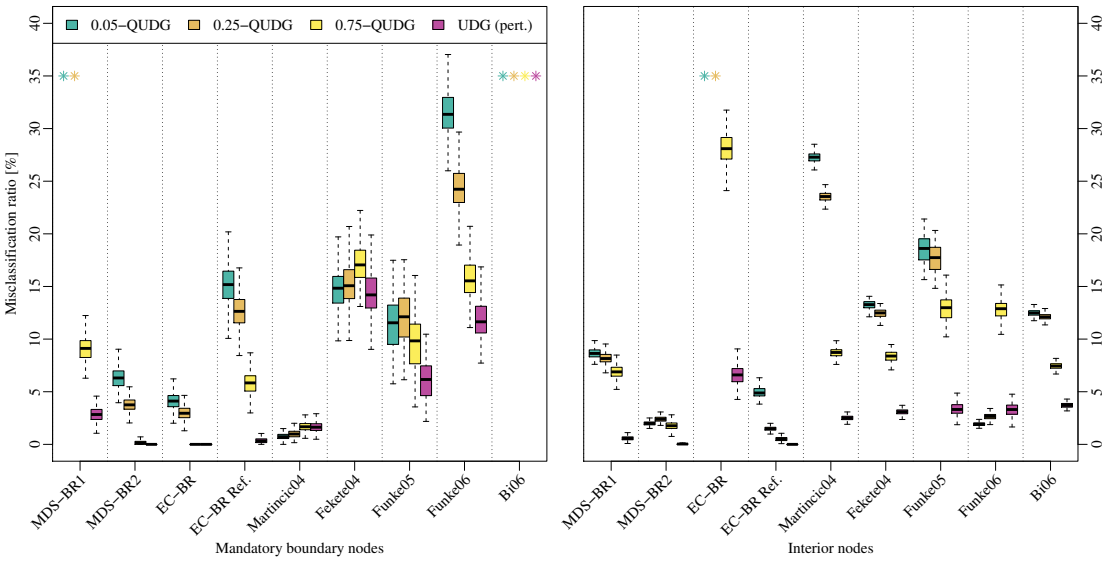


Figure 4.27: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree* 12 *and uncertainty levels between* 0.05 *and* 1.00 *(i.e. for unit disk graphs).*

4.6.4 Refinement

Our algorithm, MDS-BR, as well as EC-BR propose to apply a refinement heuristic to improve results after an initial boundary classification has been achieved. Both approaches use the heuristic to eliminate false positives, interior nodes marked as boundary nodes. In network visualizations, these nodes appear as small, random artifacts, similar to what we saw for most other algorithms in Figure 4.15. One might ask whether our refinement procedure could be used in conjunction with the other algorithms to improve their results. When comparing Figure 4.28 to the classification results in Figure 4.15, we see that the refinement procedure gets rid of most of the random artifacts but also of small—or in case of Fekete04 and Bi06 large—sections of the actual boundaries. The results of Funke06 remain mostly unchanged, boundaries around small holes are retained similar to EC-BR, only obvious, small artifacts are lost. The classifications of Martincic04 and Funke05 also seem to improve as most of the initial "noise" is removed. However, a closer inspection reveals that the boundaries are interrupted on short sections. The results of Fekete04 and Bi06 are obviously useless after refinement. In light of the already poor visual results, we refrain from performing extensive quantitative simulations.

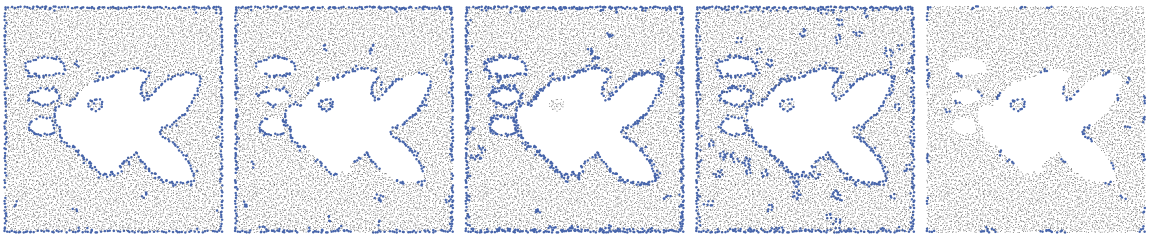


Figure 4.28: *Impact of MDS-BR refinement with $r_{min} = 2$ on different algorithms: (a) Martincic04, (b) Fekete04, (c) Funke05, (d) Funke06, (e) Bi06.*

As MDS-BR and EC-BR apply different refinement routines, but both aiming at the same goal, one might wonder how well each of them works on the solution provided by the other algorithm. We first take a look at how EC-BR fares when we apply MDS-BR refinement. The lower row of Figure 4.29 shows that our refinement procedure has little impact on the classification results of EC-BR, even when applying much more aggressive parameter values to consider larger neighborhoods. As EC-BR yields a broad halo around each hole, the refinement of MDS-BR almost always finds a path within this halo of the required length, and only few small structures can be changed to interior nodes. Figure 4.29(d) shows that our refinement removes some of the "noise" while retaining the broad boundary structures and the circles around small holes. However, EC-BR refinement with a smaller threshold value does a much better job at removing small artifacts while keeping the broad boundaries intact as depicted in Figure 4.29(b). We can achieve similar results with our refinement procedure tuned

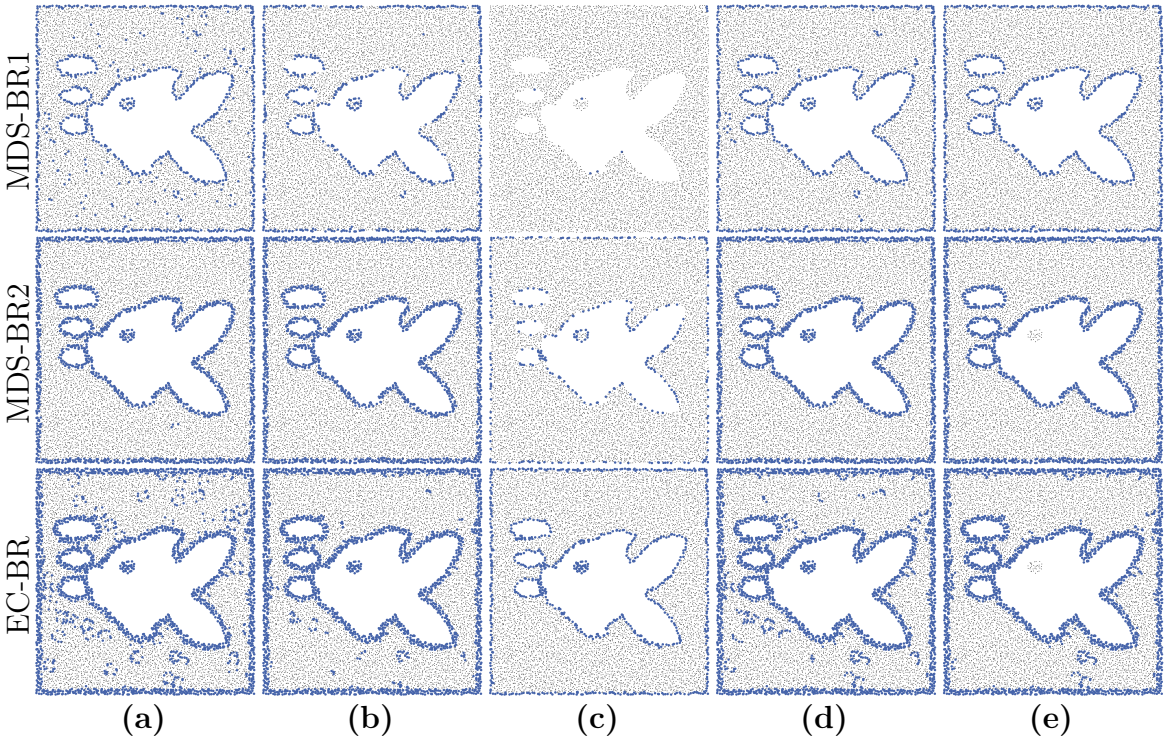


Figure 4.29: *Impact of using different refinement heuristics: (a) No Refinement, (b) EC-BR Refinement ($\gamma = 30\%$), (c) EC-BR Refinement ($\gamma = 100\%$), (d) MDS-BR Refinement ($r_{min} = 3$), (e) MDS-BR Refinement ($r_{min} = 6$).*

more aggressively, but we completely lose boundaries around small enclaves with a radius of less than r_{min}. This can be seen in Figure 4.29(e).

When considering EC-BR refinement for MDS-BR, we have to distinguish between MDS-BR1 and MDS-BR2. The classification results of MDS-BR2 are already almost optimal for our default setting as seen in the middle row of Figure 4.29. A refinement procedure cannot hope to improve upon them by much. Both, EC-BR refinement with a lower threshold and normal MDS-BR refinement, seem to be good candidates in this regard as they remove the remaining small artifacts while retaining the boundary bands. Using more aggressive settings for either refinement procedure leads to missed boundary nodes. We go into more details on choosing the best refinement parameter value for different network settings in Section 4.6.5. Note that EC-BR refinement is not able to yield the same thin boundary outlines for MDS-BR2 as for EC-BR. The boundary bands of MDS-BR2 are more rugged, and thus there exists no single threshold value that removes the broad band but retains the thin outlines. For this requirement, we offer a separate classification strategy with MDS-BR1.

For MDS-BR1, a refinement step is required to remove the random artifacts as seen in the upper row of Figure 4.29. Applying the refinement step of EC-BR returns poor

results. The small artifacts are gone but so are large parts of the actual boundary. This occurs as MDS-BR1 already yields a thin boundary outline, but the refinement of EC-BR requires boundary nodes to be completely surrounded by other boundary nodes to remain marked. The effect remains even for a lowered threshold value. Parts of the boundary are still missing, while some random artifacts start to reappear. Similar results would be achieved for the same reasons when using EC-BR refinement on the other approaches. We summarize that for MDS-BR1 only our refinement procedure helps in removing "noise" while retaining the actual boundary.

4.6.5 MDS-BR Properties

After comparing our algorithm to various competing boundary detection approaches, we focus on the properties of MDS-BR in this section. We start by studying alternative graph embedding strategies, followed by a discussion on embedding errors and angular distributions. This leads to an analysis of the optimal parameter values for MDS-BR. We conclude by taking a look at our proposed linear time implementation.

Graph Embedding Strategies. Our boundary detection approach depends on a good approximation of the relative angles between nodes to offer premium results. We deduce these angles from embeddings of local graphs. The actual node positions are not required, though. This paragraph considers the graph embedding strategies presented in Section 4.3.3 and studies their impact on boundary detection quality. Figure 4.30 shows the classification results for different network settings. We consider the UDG model with perturbed grid and random node distributions as well as quasi unit disk graphs with a growing amount of uncertainty. The analysis uses the MDS-BR2 classification strategy after computing the embedding, but our findings hold true for MDS-BR1 as well. Respective results are found in Appendix B along with results for $d_{avg} = 15$.

We see that all three alternate graph embedding strategies, MDS_3, considering 3-hop neighborhoods, MDS_{opt}, using true node positions, and MDS_{SS}, taking into account signal strength information, improve the results in the d-QUDG model compared to plain multidimensional scaling. They are able to deal with randomly missing communication links much better than our default approach. This leads to more accurate embeddings and, in turn, to a higher classification quality. The results for unit disk graphs have already been almost optimal and cannot be improved much further. Interestingly, MDS_3 and MDS_{opt} produce a very high amount of misclassifications for mandatory boundary nodes for random node distributions. Both strategies actually compute more accurate embeddings than plain MDS and MDS_{SS}. However, as the distribution of maximum opening angles does not allow for a clear distinction between inner and boundary nodes in this setting (see paragraph on angular distributions), classification quality is poor. The (imprecise) embeddings by our default embedding approach and MDS_{SS} actually help to better differentiate between both types of nodes.

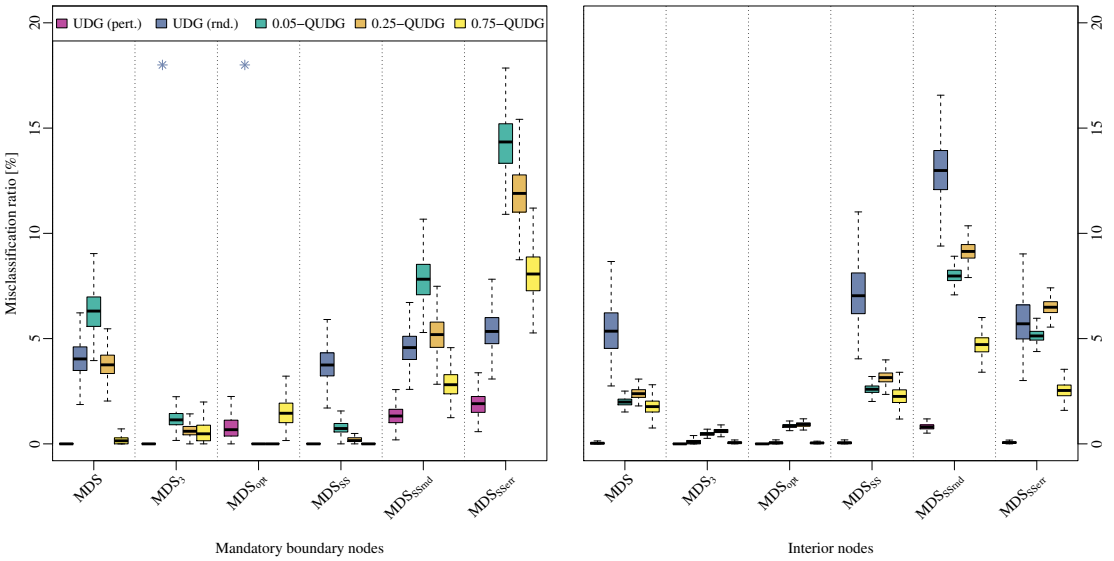


Figure 4.30: *Misclassification ratios (false negatives) in percent for different network settings with average node degree* 12 *and multiple graph embedding strategies.*

The results of taking into account signal strengths are particularly interesting. As already discussed, classification quality improves for quasi unit disk graphs and remains roughly the same for the UDG model with either node placement strategy. This alone is already impressive as we only apply a very rough distance estimation by distinguishing solely between weak and strong signals. However, we also consider the impact of making errors when estimating node distances from signal strengths. Randomly interpreting signal strength as weak or strong in case of MDS_{SSrnd}, the resulting classification becomes slightly worse compared to plain multidimensional scaling. Even when always estimating node distances poorly with MDS_{SSerr}, the results do not deteriorate much further. This shows that MDS_{SS} is very robust to fluctuating signal strengths.

Considering the performance of our graph embedding strategies, utilizing signal strength information seems to be the best option to improve the classification quality. The other strategies offer better results, but both entail further complications, while MDS_{SS} can be integrated easily. MDS_3 induces an additional communication and computation overhead, and the node locations required by MDS_{opt} are usually not readily available.

Embedding Quality. Using the perfect embeddings offered by MDS_{opt}, we are able to assess the quality of our own embeddings, comparing the computed maximum opening angles to the true values. Note that this is done in a simulated setting, in which we can easily offer access to node locations for the sake of analysis. Considering angles to nodes in 2-hop distance—similar to MDS-BR2—we obtain an average absolute deviation of 0.04π for plain multidimensional scaling in our default setting. By differentiating

between interior nodes and mandatory boundary nodes, we gain further insights. While the average absolute error of the former is quite small at 0.03π, it is considerable at 0.11π for the latter. MDS_3 reduces the deviation for mandatory boundary nodes to 0.10π, and by using MDS_{SS} we achieve an even smaller error of 0.08π. The deviation for interior nodes remains stable. The average absolute error for opening angles to neighboring nodes—as required by MDS-BR1—is much larger at 0.2π for plain MDS, 0.18π for MDS_3, and 0.13π for MDS_{SS} when considering boundary nodes. The deviation for interior nodes increases only slightly to 0.05π for either technique. Results for other network settings as well as relative error values can be found in Appendix B.

While these deviations seems to be quite high compared to our default threshold value for α_{min}, our simulations in Section 4.6.3 showed the maximum opening angle to be an excellent classification criterion for all considered settings. The next paragraph sheds some more light on these results by considering angular distributions and showing that the respective values for boundary nodes and interior nodes remain sufficiently separated even for imperfect embeddings.

Angular Distributions. We now take a closer look at the distributions of the true maximum opening angle and of the values that our embedding provides. We focus on opening angles to nodes in 2-hop distance as required by MDS-BR2. The same general observations hold for opening angles to neighboring nodes. These results are listed in Appendix B along with results for the alternate embedding strategies.

Figure 4.31 shows the results for the UDG model with the perturbed grid and random

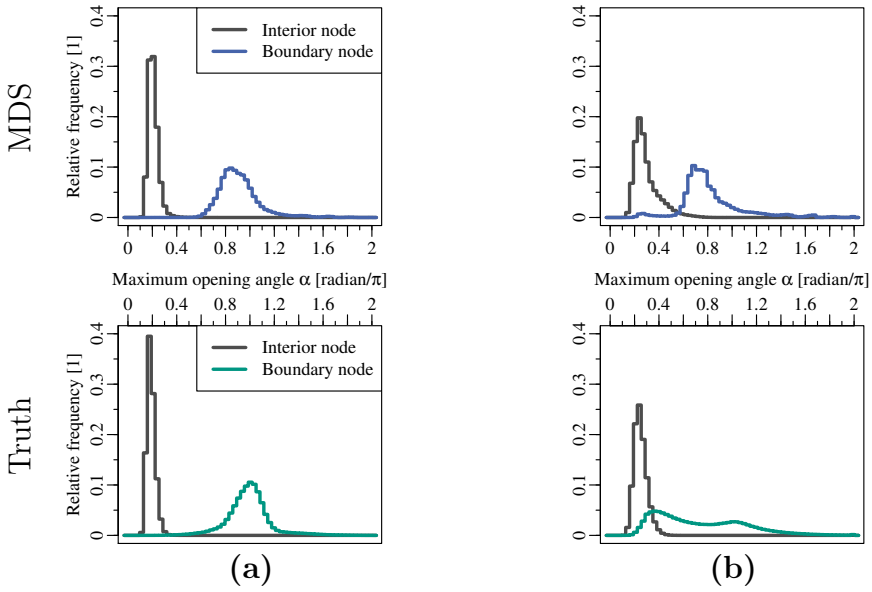


Figure 4.31: *Distributions of the maximum opening angle in the UDG model with (a) perturbed grid or (b) random node placement.*

placement strategies. We see a clear separation in the occurring angles for interior nodes and mandatory boundary nodes for perturbed grid placement. Our embedding retains the general separation, though the distribution of boundary nodes is slightly skewed to lower values. Setting the threshold α_{min} right between the two peaks obviously yields an almost perfect classification. The results for the random placement strategy paint a completely different picture, though. While the angular distribution for interior nodes remains sharply peaked, the distribution for mandatory boundary nodes is spread over a large range of angles. Obviously, this is not a good classification criterion in this setting. However, looking at the distributions produced by our embeddings, we again see two clear peaks. The topological properties of networks generated with either node distribution strategy seem to be similar on a local (2-hop) scale as the embedding yields similar results in either case. A more flat distribution only emerges when information of a larger neighborhood is considered by the embedding, and the random character of the network starts to have an impact. This is supported by the results of MDS_3 resembling those of MDS_{opt}, while distributions with MDS_{SS} look more like those obtained with plain multidimensional scaling (see Appendix B).

The results in Figure 4.32 show the distributions for the d-quasi unit disk graph model. Obviously, the true angular distributions are very similar to the one for unit disk graphs. This is in accordance to Figure 4.24 that shows similarly structured boundaries for both communication models. The flank for mandatory boundary nodes is slightly drawn-out towards lower values. This justifies using a lower threshold value for the maximum opening angle to obtain best classification results. The distributions

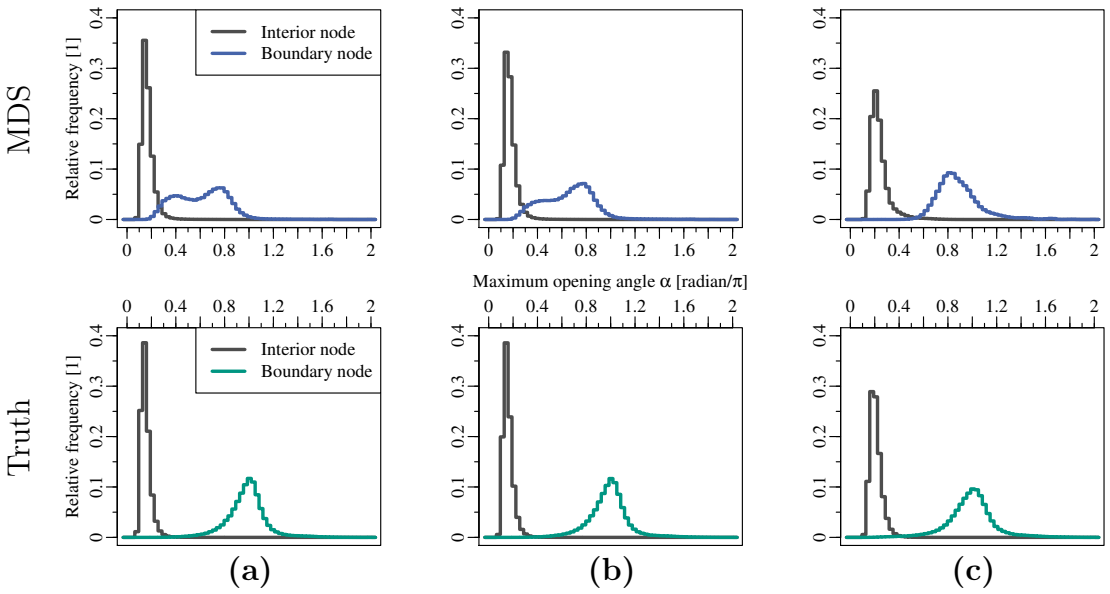


Figure 4.32: *Distributions of the maximum opening angle in the (a)* 0.05*-QUDG, (b)* 0.25*-QUDG, and (c)* 0.75*-QUDG model.*

look very different for our embeddings though. The higher the amount of uncertainty in the QUDG model, the more the distribution of mandatory boundary nodes is spread to smaller values. Results for the embedding strategies MDS_3 and MDS_{SS} remain similar to MDS_{opt} (see Appendix B). This indicates that our default embedding strategy has difficulties to compute an accurate embedding. Still, the distributions for both types of nodes remain sufficiently separate to allow for a good classification.

We still need to determine the best threshold value α_{min} to separate boundary nodes from interior nodes. The following parameter analysis determines this optimum threshold value. We will see that we can manage with one value if we are content with good but not optimum results. By using separate thresholds for networks based on the UDG model and the d-QUDG model, we can obtain best results, though.

Parameter Selection. Our previous analysis focused on one set of parameter values for MDS-BR, an opening angle threshold $\alpha_{min} = 0.5\pi$ and, in case of MDS-BR1, a refinement step with $r_{min} = 2$. We now take a closer look at how different parameter values impact the classification quality of our algorithm. As we have already seen in the previous paragraph on the distributions of maximum opening angles, our default threshold value for α_{min} seems to be chosen well. In addition, Section 4.6.4 showed that applying the refinement procedure of EC-BR with a lower threshold value for γ can be beneficial to our algorithm, too. Both observations are now examined in more detail and verified by our simulations. We focus on MDS-BR2 in the following analysis. The results for MDS-BR1 are similar and given in Appendix B.

We present our findings in form of heatmaps as seen in Figure 4.33. Each colored square represents the results of one set of parameter values, opening angle threshold and refinement amount. Red indicates a good classification quality, while white stands for poor results. The color scale is optimized for discriminability for each plot individually. We measure classification quality as the geometrical mean of the correct classification ratios of mandatory boundary nodes and interior nodes. Note the split x-axis. Single digit numbers to the left indicate r_{min} values for the refinement step of MDS-BR, while double digit numbers to the right indicate γ values in percentage for the refinement step of EC-BR. Running the algorithm without refinement is implied by $r_{min} = 0$.

The heatmaps depicting our results for the UDG model in Figure 4.33 show that the opening angle threshold $\alpha_{min} = 0.5\pi$ is the best choice when not using any refinement. The classification quality improves for larger values of r_{min} before it deteriorates again at $r_{min} = 5$. Here, small but required boundary structures are eliminated by the refinement process. This behavior does not emerge for random node placement as there is no particular decline of appropriately sized boundary structures. Using EC-BR refinement with increasing threshold values, we see that the sweet spot for α_{min} shifts to smaller values while the quality decreases in general. A small refinement threshold of $\gamma = 20$ - 30% yields good results but not significantly different than without refinement.

The heatmaps in Figure 4.34 indicate that the optimum value for the opening

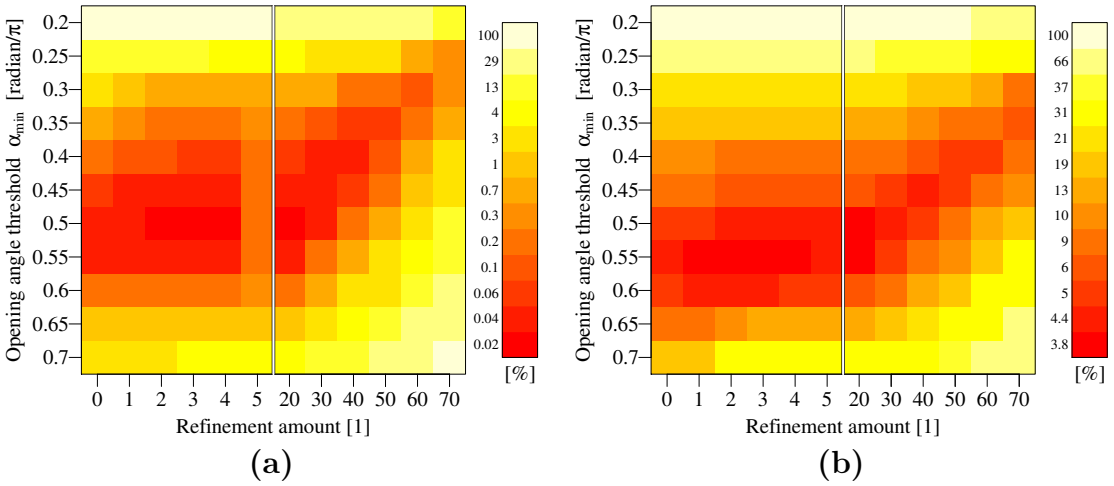


Figure 4.33: *Heatmap of classification results for different sets of parameter values in the UDG model: (a) Perturbed grid placement, (b) random placement.*

angle threshold shifts from $\alpha_{min} = 0.5\pi$ to $\alpha_{min} = 0.25\pi$ with a growing amount of uncertainty in the d-QUDG model. This effect is supported by the angular distributions shown in the last paragraph. It is much less pronounced for MDS-BR1, which is also in accordance to the respective angular distributions (see Appendix B). Apart from that we see the same general behavior as for unit disk graphs. Interestingly, using EC-BR refinement with small threshold values provides the best results. A value of $\gamma = 30\%$ seems to work best for all settings.

As we typically cannot assess the network structure in a real-life setting beforehand, we need to select one set of parameter values that works well for all cases. Based on our results, we deduce that $\alpha_{min} = 0.3\pi$ and $\gamma = 30\%$ provide a good compromise for any network setting. Though, if we know that the UDG model is a good approximation of the communication links in our network, $\alpha_{min} = 0.5\pi$ and $r_{min} = 0$ work even better.

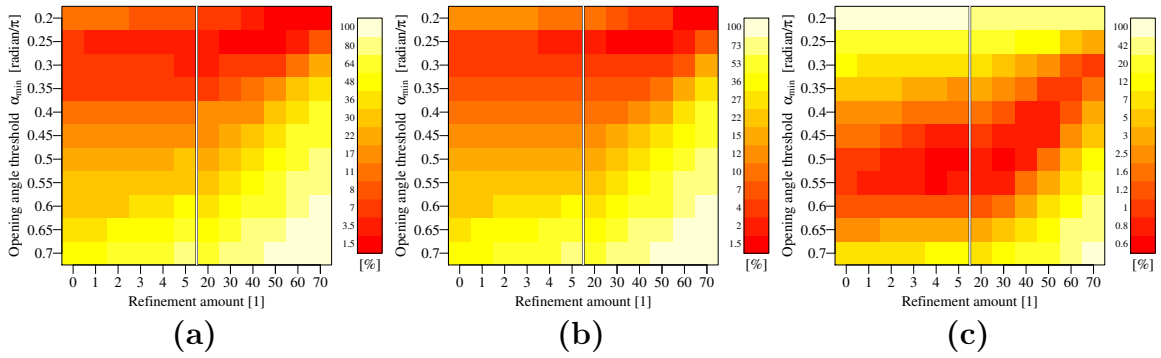


Figure 4.34: *Heatmap of classification results for different sets of parameter values on d-QUDGs: (a) $d = 0.05$, (b) $d = 0.25$, (c) $d = 0.75$.*

Refinement Costs. We argued in Section 4.3.2 that our refinement procedure runs efficiently as we only have to consider few nodes. Our simulations confirm this initial assumption. Using MDS-BR1 and varying r_{min}, we obtain the marked neighborhood sizes shown in Figure 4.35 and Figure 4.36. For our default setting, we see that the sizes increase linearly with r_{min} and the average node degree. This is expected behavior as the considered boundary is a very thin band, most often only one node wide. For the smallest node degree, the numbers are much higher, though, as the boundary structures start to overlap and we no longer have thin boundary bands. The same effect occurs for both, networks with random node placement and networks using the quasi unit disk graph model with a high amount of uncertainty. With higher average node degrees, the marked neighborhoods become smaller as the boundaries become more precise and better separated. For our default parameter setting, $r_{min} = 2$, the average neighborhood size that is considered stays at 25 or well below for all network settings. Thus, we conclude that the refinement costs are indeed negligible for MDS-BR1.

As our refinement step is not very effective in combination with MDS-BR2, we do not promote to apply it in this case. However, we give a brief summary of the respective refinement costs for the sake of completeness. Detailed results are listed in Appendix B. The average considered neighborhood stays below 30 nodes for all network settings and $r_{min} = 2$. Sizes increase quicker for larger values of r_{min} and average node degrees as MDS-BR2 marks broader bands of boundary nodes. We see the effect that low-degree networks exhibit large neighborhoods less often and less pronounced than for MDS-BR1 as the boundary structures are more distinct, even for difficult network settings.

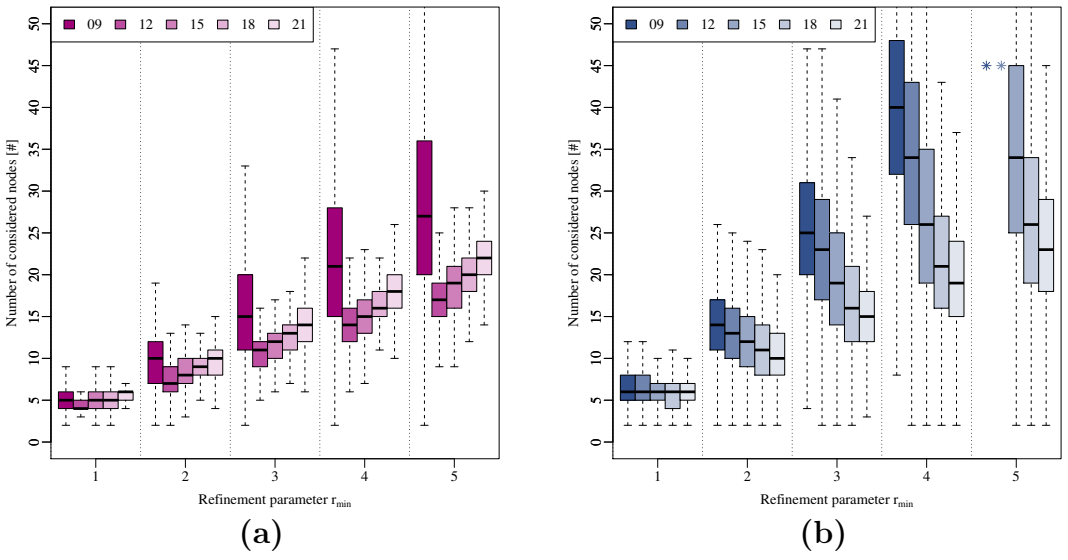


Figure 4.35: *Neighborhood size that our refinement strategy considers on UDG networks with average node degrees between 9 and 21 and (a) perturbed grid placement or (b) random node placement.*

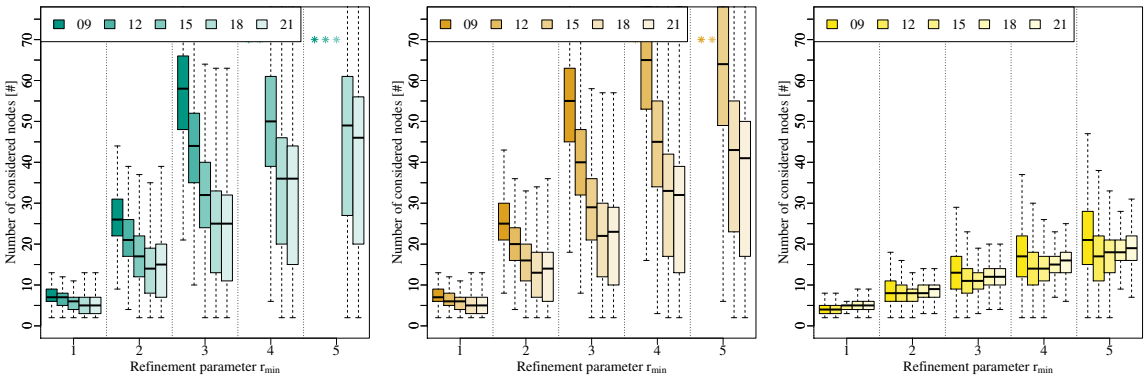


Figure 4.36: *Neighborhood size that our refinement strategy considers on d-QUDG networks with average node degrees between 9 and 21 and (a) $d = 0.05$, (b) $d = 0.25$, or (c) $d = 0.75$.*

Linear Time Implementation. We discussed a linear time implementation of MDS-BR that uses random filtering to reduce the considered neighborhoods to an average node degree $d_{avg} = 15$ in Section 4.3.1. We argued that the classification quality does not significantly suffer. Figure 4.37 compares the results of this approach to our normal algorithm and to the contraction-based approach on high-degree networks. Again, we only consider MDS-BR2 as the general findings are the same for MDS-BR1.

Obviously, our default algorithm that considers complete neighborhoods yields the

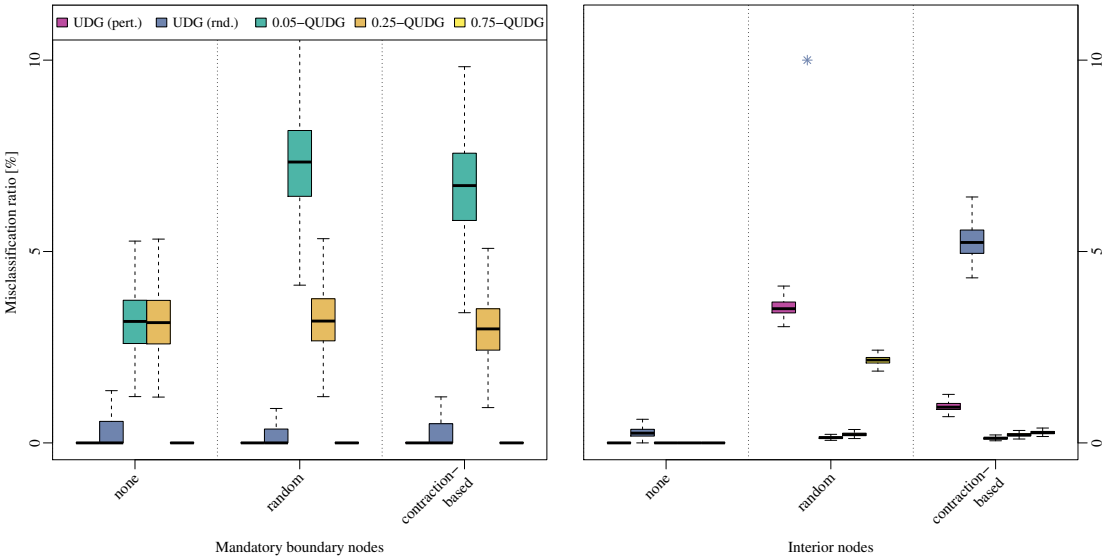


Figure 4.37: *Misclassification ratios (false negatives) in percent for different network settings with average node degree 27 and multiple MDS filtering techniques.*

best results. They get worse when we apply our linear time implementation. However, this is to be expected as the embedding of the filtered neighborhood is less precise. The classification quality is still largely on par with what to expect from an actual network with $d_{avg} = 15$. The results for interior nodes on networks with random node placement are the only outliers. It is already difficult to embed complete neighborhoods in this setting as seen previously. Removing nodes at random without compensating for them only aggravates this task. When applying our contraction-based implementation, contracting the neighborhood to an average node degree of 10, we observe an improved classification quality compared to the linear time implementation. As we do not simply remove nodes but retain the topological structure of the reduced neighborhoods, the computed embeddings are more accurate and therefore yield better classifications. The approach still fares worse than our normal algorithm but also offers better asymptotic running times even if they are not quite linear time.

4.7 Concluding Remarks

We presented a novel algorithm, MDS-BR, for location-free boundary detection in sensor networks. Our approach works distributed and only needs to gather connectivity information of small, local neighborhoods around each node. This is a huge improvement over most existing algorithms that either need to query much larger neighborhoods or rely on additional network properties. The resulting low communication overhead makes MDS-BR an excellent choice for boundary detection in large-scale sensor networks. It also makes our approach well suited for scenarios that include mobility or dynamic changes of the network topology. Depending on the requirements of the underlying application, we can offer two classification strategies to either mark a precise outline of the network boundary or to give broader bands around the fringes of the network. The latter option is of particular interest when building a sensor network to monitor the perimeter of a larger area. Together with the results of Chapter 3, we can identify a broad boundary band around the area and compute a schedule that maximizes the network lifetime. As nodes start to fail, we can even update the halo and have it move inwards to further prolong the monitoring activity.

Extensive simulations showed that our approach is very robust to different network densities, communication models, and node distributions. Despite its simplicity and low communication overhead, MDS-BR outperformed the other considered approaches significantly, especially in challenging network settings. Additionally, we offer much lower computational complexity than many existing approaches. Our simulations are complemented by an analysis of the properties and variants of our algorithm.

Outlook. Our approach depends on the embedding procedure to return an accurate representation of the angular relationships between nodes. This makes it worthwhile to look into alternate embedding strategies in the future. There are two possible benefits

to gain, a higher precision in reconstructing angles and an accelerated computation. The first leads to better classification results, while the latter helps the nodes to conserve their scarce energy reserves.

Moreover, MDS-BR offers synergies with other applications running on the sensor network. If they also compute virtual node coordinates and provide a reasonable estimate of the real positions, we could utilize them and avoid the embedding process entirely. A worthwhile candidate to consider is the algorithm MDS-MAP(F) by Katz [Kat09] that finds embeddings for large-scale networks of high quality.

Our refinement step only considers boundary nodes and turns them into interior nodes if some conditions are not fulfilled. A revised procedure should be able to identify falsely classified interior nodes as well and mark them correctly.

5 Chapter 5

Determining Efficient Paths in Large-Scale Sensor Networks

Communication. The imparting or exchanging of information by speaking, writing, or using some other medium.

— Oxford Dictionary of English

The exchange of information is a central aspect in sensor networks. Without sharing data, each node in the network is just an isolated system with very limited capabilities. Only in interaction with other sensor nodes, more complex tasks can emerge such as the detection of network boundaries discussed in Chapter 4, traffic guidance systems that provide early warnings of arising and potentially dangerous situations [KSC06], or the study of cosmic rays at the Pierre Auger Observatory [Kie10].

As the above dictionary definition implies, communication, i.e. the exchange of information, can be done over diverse media. In a sensor network context, communication is usually handled by wireless transceivers, but tethered sensor networks exist, too. The information is relayed over multiple intermediate nodes before arriving at the intended target. There are numerous, sometimes even opposing demands on how to relay this information between nodes. For example, if response times are crucial, preferably direct connections are used. Energy consumption is often equally important as most sensor nodes are energy-constraint. Which demands are set depends on the network structure and may even vary between different applications on the same sensor network.

This chapter focuses on finding efficient paths in sensor networks that meet the above demands. Our efforts are aimed at the quick computation of these paths in static settings with respect to a high frequency of queries as encountered in simulation frameworks or within large-scale infrastructural networks.

References. The contents of this chapter are based on previous publications. Results on approximate queries are based on joint work with Robert Geisberger [GS10] who gave the respective proofs. Alternative connections are based on joint work with Dennis Luxen [LS12a, LS14]. Wordings of the above publications are used in this thesis.

5.1 Introduction

Routing information through a network, i.e. finding some best path from a sender to the intended receiver, is of such fundamental importance as motivated in the chapter preface that it is worthwhile to consider this aspect isolated on its own. We can distinguish between *online* routing approaches that use little to no information to relay a message from node to node following a usually simple protocol on each sensor node and *offline* routing algorithms that often require a significant amount of preprocessing. While the latter can also be deployed on sensor networks, they are most often found in auxiliary applications such as simulation frameworks that run on classical systems. The reasons for this are obvious and manifold. Foremost, offline algorithms are not distributed. Thus, preprocessing has to be handled at some central point or replicated at each sensor node. This implies that the network topology has to be known in advance or gathered before running the preprocessing. Later, the processed data has to be spread again to all sensor nodes. Both operations cause a substantial amount of communication over the whole network. Moreover, the amount of computation and auxiliary data required at each node can become significant, especially if the preprocessing is replicated at all sensor nodes. To make matters worse, changes in the network topology may induce at least a partial reprocessing of the auxiliary data.

Offline algorithms have their own benefits, though. They focus on processing large amounts of routing queries in a very short timeframe and determine the complete relaying information, not just the next hop on the path to the target. Moreover, they provide optimal solutions or at least come with tight error bounds. These properties are beneficial in auxiliary applications such as the aforementioned simulation frameworks or general analysis toolkits. Both of them face their own challenges, though. As they are implemented on classical systems, there is only a limited amount of parallelism available compared to a sensor network. However, they still have to deal with potentially large networks and process communication volumes (i.e. routing queries) that grow rapidly with the network size. Thus, a quick processing of single queries is crucial, especially when dealing with large-scale sensor networks. To assess the capabilities of these networks, we obviously need (near) optimal paths, i.e. shortest paths with respect to the requirements of the considered setting, but reasonable alternative paths to them are important to consider as well. They serve, for example, to spread the communication load more evenly over the entire network and provide choices in case of (simulated) node failures or overload in the considered scenarios.

Even sensor networks, as long as they do not suffer from the above limitations, can profit from offline algorithms. A good example are tethered infrastructural networks along roads and highways that are unlikely to change and come with ample processing and power reserves. Optimal routing strategies, even for some exceptional cases, can be precomputed and distributed to all nodes in the network, guaranteeing, for example, shortest communication paths in the network.

While there has been a tremendous amount of work on shortest path techniques over the last decade, most of these efforts focus on transportation networks alone. Previous studies like [BDS+10] even show that techniques that work well on road networks do not permit equally impressive results on sensor networks. Thus, even in the offline setting there is still a gap to close.

5.1.1 Related Work

Determining a path that is efficient in some sense is a similar problem to finding a shortest path in a network. In fact, many such problems involve the computation of shortest paths as a subproblem. We therefore focus on shortest path techniques before considering further publications on alternative paths. As there is a tremendous amount of work in this area, especially from the last decade, we can only list the most prominent techniques in our overview on the related work.

For an extensive overview on routing techniques, we refer to the recent survey article by Bast et al. [BDG+14], even though they only focus on traffic networks. In [Som14], Sommer provides a more general overview on shortest path techniques in static networks from both a theoretical and a practical perspective. Zwick [Zwi01] further considers approximate techniques. These articles are completed by a history on the shortest path problem given by Schrijver in [Sch12].

Shortest Path Techniques. The computation of shortest paths dates back to the middle of the last century, with Dijkstra's algorithm [Dij59] likely the most prominent solution to this problem and the basis of many recent algorithms. In a graph without negative edge costs, it finds shortest path distances from one source node to all other nodes. It searches radially around the source by iteratively settling the node with minimal distance from the source. If just a single point-to-point distance is needed, it may stop early once encountering the target. The algorithm can be augmented to yield shortest paths and shortest path trees. Even though its running time is polynomial, it does not scale well to large networks. An obvious way to accelerate at least point-to-point queries is to perform a *bidirectional search* with Dijkstra's algorithm (BD) [Dan63]. Two searches start simultaneously, one from the source and one from the target. A shortest path is found once the same node is settled by both searches. In practice, this approach is about twice as fast as running plain Dijkstra's algorithm.

More involved techniques prune the *search space* of Dijkstra's algorithm, i.e. the number of nodes it visits, for a considerable speed-up. These approaches can be divided into *goal-directed* and *hierarchical* techniques and *combinations* thereof. They usually allow for bidirectional search in some form, and most of them require a preprocessing step in which data is aggregated to accelerate queries at the expense of some memory. This scheme is effective if there are many queries to answer on the same graph.

Below, we consider these types of techniques as well as algorithms for *batched queries*, *theoretical results* on finding shortest paths, *approximate algorithms*, and *futher results*.

Goal-directed Techniques. Whereas Dijkstra's algorithm searches uniformly in all directions, goal-directed techniques try to focus the search in the direction of the target. A simple heuristic from artificial intelligence is the $A*$ search [HNR68]. It uses lower bounds on the shortest path distance to the target node, and in each iteration, it settles the node which minimizes the sum of its distance from the source and said lower bound. The method usually performs better than Dijkstra's algorithm, but slowdowns are reported in the presence of inadequate bounds [GH05]. This publication further reports on *ALT*, a variant of A* that uses a graph-theoretic lower-bounding heuristic based on landmarks and the triangle inequality. It selects a small set of well distributed nodes and precomputes distances to and from all other nodes. During a query, the distances are used to determine lower bounds with the triangle inequality. The obvious advantage of this approach to other heuristics is the independence on external data. It is also more than an order of magnitude faster than Dijkstra's algorithm. However, a lot of memory is required for the precomputed distances.

The *Arc Flags* (AF) approach [Lau04] partitions the graph into regions. Every edge is augmented by a label that holds a flag for each region, indicating whether there is a shortest path over this edge into the region. The query follows Dijkstra's algorithm but only considers edges with the flag of the target's region set. The approach has been revisited several times to consider different partitioning schemes or faster preprocessing methods [KMS05, MSS+07, HKMS09]. It is very quick with reported speed-ups to Dijkstra's algorithm of over 1 000 in the bidirectional case. However, preprocessing is costly as it requires the computation of shortest paths from each boundary node of a region to all other nodes, i.e. one complete Dijkstra run from each such node. Hilger et al. [HKMS09] suggest a technique to compute shortest paths of all boundary nodes of a region simultaneously, but it is memory intensive and still slow. By now, PHAST [DGNW13] allows for very fast computation of arc flags on certain types of graphs.

Hierarchical Techniques. These approaches exploit that some edges are more important than others. While this is apparent for road networks with city streets and highways, it is less obvious for others graphs. For example, in sensor networks one tries to avoid energy-intensive long-distance communication if battery power is limited, but if transmission speed is of the essence, they are actually preferred.

Contraction Hierarchies (CH) [GSSV12] is the pinnacle in a series of techniques based around the concept of shortcut edges [SS07, SS12a]. In a preprocessing step, nodes are removed iteratively from the graph in a heuristic order of importance, and shortcut edges are inserted to retain shortest path distances in the remaining graph. The query is a bidirectional variant of Dijkstra's algorithm with a modified stopping criterion. It runs on the original graph augmented by the shortcut edges and only proceeds to more important nodes due to the construction of the graph. Impressive results are achieved on sparse networks, exceeding even those of Arc Flags, but so far, CH performs less well on dense graphs like sensor networks. As this approach is the basis of our first contribution, we give a more elaborate description in Section 5.3.1.

The *Customizable Route Planning* (CRP) technique [DGPW13] takes a different approach and focuses on fast preprocessing of different metrics. It is almost an order of magnitude slower than CH but allows to process a new metric in the blink of an eye, especially with GPU support [DKW14]. In a first (slow) preprocessing step, a multi-level partitioning of the graph is generated. On each level, the respective boundary nodes induce an overlay graph. To maintain shortest path distances, each region is connected in a clique by shortcut edges. A second (fast) preprocessing step incorporates the metric by computing correct costs for all shortcut edges. The query is a (bidirectional) Dijkstra's algorithm that (implicitly) switches to the next higher level at each boundary node to proceed on the respective overlay graph. The basic idea has been studied before with some success, e.g. in [SWZ02, HSW08, DHM+09], but CRP profits greatly from clever engineering and PUNCH [DGRW11] to find tiny separators. In general, these approaches work best if the graph has small (and efficiently computable) separators. Unfortunately, this is not the case for sensor networks.

Transit Node Routing (TNR) [BFSS07] identifies a small set of important nodes during preprocessing and computes pairwise distances between these transit nodes. For all other nodes, it further determines a minimal set of transit nodes that covers all shortest path starting at them. The nodes in these sets are called access nodes. A query selects a path that minimizes the combined distances of source and target to their respective access nodes and between the access nodes. A locality filter further estimates whether the shortest path might avoid transit nodes entirely and runs a normal query as a fallback. This approach is almost two orders of magnitude faster than CH, but it needs an expensive preprocessing and a lot of memory. A variant solely based on graph theoretical concepts simplifies the approach and reduces the preprocessing overhead considerably [ALS13].

Recently, *Hub Labels* (HL) [ADGW11] have emerged as the fastest technique for computing shortest path distances. Queries take about the same time as five memory accesses. The algorithm stores a label at each node with distances to some other nodes. Preprocessing guarantees that a shortest path between two nodes goes over a common node in both of their labels. Thus, finding a shortest path distance is reduced to scanning these labels and adding distances. However, the algorithm comes with hefty preprocessing and memory requirements. Subsequent publications relax these requirements at the expense of query performance [ADGW12, DGW13, AIKK14]. Though the basic labeling algorithm [Pel00b] has long been known, only recent advances in the theoretical understanding of speed-up techniques [ADF+13], especially with respect to transportation networks, allowed for these impressive results. The approach is extended to denser networks in a recent publication [DGPW14].

Combined Techniques. The combination of goal-directed techniques and hierarchical methods holds a lot of promise and is consequently studied in multiple publications on single combinations [BD09, DSSW09] and in more extensive surveys [SWW00, HSWW05, BDS+10]. The latter publication introduces two very successful techniques,

Core-ALT (CALT), which combines ALT with a single-level overlay graph, and *CHASE*, combining Contraction Hierarchies with Arc Flags. Both of them apply a similar idea to different approaches. The respective goal-directed technique is only applied to a small core of the graph containing the most important nodes. This saves a lot of preprocessing time and memory while retaining most of the performance compared to applying the goal-directed technique on the whole graph. Queries are performed in two phases. The first phase only uses the hierarchical technique and explores the graph up to the core. The second phase is restricted to nodes in the core and activates goal direction. In practice, this leads to a speed-up of about an order of magnitude compared to the single techniques. With the recent advances in computing and storing arc flags, CHASE now commonly uses arc flags on the whole graph. This also simplifies the query as only one phase is needed. Bauer et al. [BDS+10] further describe a combination of TNR with Arc Flags, which was the fasted approach until the recent emergence of Hub Labels.

Batched Techniques. Some applications, like the introduced preprocessing routines, require the computation of shortest paths from one node to all others. While Dijkstra's algorithm already solves this problem asymptotically optimal, it is neither cache-efficient nor can it exploit parallelism well. The *PHAST* algorithm [DGNW13] uses the augmented graph of Contraction Hierarchies to remedy these shortcomings. It performs a search from the source like CH, but instead of a search from a target, it scans linearly over all nodes once the initial search finishes. This can be done cache-efficiently and in parallel. The sequential execution is already an order of magnitude faster than Dijkstra's algorithm. The approach efficiently computes multiple queries at once, and it can be adapted to only use a subset of all nodes as targets [DGW11].

Some applications may require shortest paths between multiple sources and targets, though. This is easily done in parallel with point-to-point queries or PHAST from above, but we can do even better with a *bucket-based approach* [KSS+07]. The algorithm only needs time linear in the sum of sources and targets instead of their product. Using any hierarchical approach, like CH, we first perform all searches from the target nodes, storing for each node its tentative distance to all targets in a bucket. Next, we run the respective searches from the source nodes. By scanning the buckets, we can efficiently determine shortest paths for each pair of source and target node. A variant using CRP shows how to further exploit the existence of separators in [DW13].

Theoretical Results. Apart from a multitude of highly engineered algorithms to compute shortest paths, the last years also provided us with several interesting theoretical results. The following two are of particular interest to us. All preprocessing routines of the above algorithms have a heuristic component, be it the choice of landmarks, the partitioning of the graph, or the order of node contraction. Bauer et al. [BCK+10] show that making optimal choices is $\mathcal{NP}$-hard. The introduction of *highway dimension* [ADF+13] gave a theoretical justification why recent hierarchical techniques perform so

well on some types of graphs. Roughly speaking, a graph has low highway dimension, and in turn the above techniques work well, if at any scale, all shortest path of a certain length are hit by set of nodes that is locally sparse. This seems to be the case for at least transportation networks and similarly structured graphs.

Approximate Shortest Path Techniques. So far, we only considered algorithms for computing exact shortest paths. However, there is a whole area of research that focuses on approximate shortest path techniques. They forego exact results in favor of a faster computation or lower memory requirements but usually still demand some guarantee on the computed paths. For instance, paths should not become arbitrarily long.

An interesting topic in this area are ϵ-spanners. An ϵ-spanner is a spanning subgraph of a graph such that any shortest path is at most $(1+\epsilon)$ times longer than the respective one in the original graph. For example, for a complete graph on n nodes in k dimensions, there exists an ϵ-spanner with $\mathcal{O}(\epsilon^{-k}n)$ edges [Vai91]. Clearly, correctness is traded for lower memory requirements. However, this may implicitly lead to shorter runtimes, too. Zwick provides an overview on spanner algorithms in [Zwi01].

Algorithms that focus on a quick computability often originate from the field of artificial intelligence. For example, Pohl introduces the *weighted A** search in [Poh70]. It applies more aggressive bounds than plain A* and only guarantees a certain error on shortest path distances. [Pea84] gives an overview on related techniques from this field.

Further Results. Most of the above publications focus on transportation networks alone. While [BDS+08] shows that techniques that work well on these networks do not permit equally impressive results on sensor network instances, the results in [DGPW14] are promising that some of the previous techniques can be adapted for sensor networks. Studies that concentrate on other types of networks often only provide theoretical results, though, focusing on asymptotic time complexities but not on constant factors or measured runtimes. For example, Beier et al. [BFMS11] apply techniques from computational geometry to compute energy-efficient paths in radio networks under different constraints like a bounded number of hops or certain energy cost models. Their approaches are theoretically efficient, but their simulations in [FMS08] show that the required preprocessing is already substantial for tiny networks.

With respect to sensor networks, there are also many publications on routing schemes for the distributed application on sensor nodes. Each node decides with locally available information alone where to relay a message next for an efficient delivery. The main goal of these routing schemes is to keep the information stored at each node low compared to the network size while guaranteeing that the found paths do not become arbitrarily long. Not requiring node positions or similar labels is a further goal. [AGGM06] and [KRX07] present asymptotically (near) optimal results for a certain class of networks using hierarchical decompositions of the network. The work by Sarkar et al. [SZG13] is interesting as well. At each node, they store routing information to a few distant nodes similar to shortcut edges and guarantee near optimal routes. Moreover, this information

is generated in a distributed and unsupervised way. The book by Peleg [Pel00a] and the surveys by Peleg and Gavoille [Gav01, GP03] provide further background and details on routing schemes. Next to these theoretically results, [AKK04] and [GG12] give an overview on practically used routing techniques for sensor networks. Neither of these theoretical results nor any of the distributed algorithms is applicable to our use case, though, the computation of efficient paths for large sensor networks on classical systems. We therefore do not go into further details on these topics.

Alternative Path Techniques. Next to the abundant research in shortest path techniques, there exists quite a lot of work that can be considered for finding alternative paths. An intuitive approach is to compute the k-th shortest path between two nodes s and t in a network. Yen and Eppstein study the general *k-shortest path problem* in their respective articles [Yen71, Epp98]. Unfortunately, reasonable alternatives are usually not among the first few hundred or even thousand shortest paths as they only offer minimal deviations from the actual shortest path. Consider a path with l minor detours as seen in Figure 5.1. There are 2^l possible paths, each with a similar distance between s and t. Unfortunately, none of these paths is a reasonable alternative as all of them have a high amount of pair-wise overlap. Even for a moderate number l of such minor detours, the parameter of a k-shortest path algorithm must be chosen exponentially high in l to yield any reasonable alternative. Moreover, these algorithms are not fast enough to be considered practical.

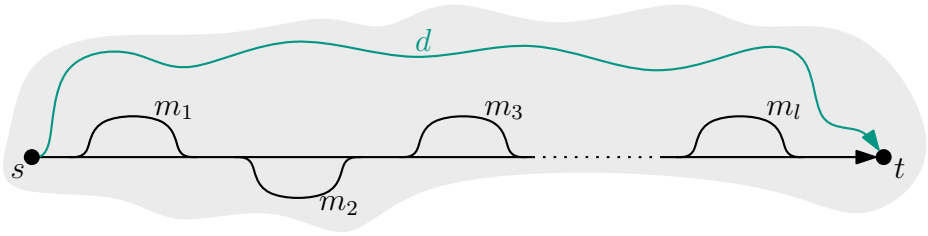


Figure 5.1: *Path from s to t with l minor detours m_i (black), $i \in \{1, \ldots, l\}$, and one long detour d (green). Neither combination of minor detours yields a reasonable alternative, whereas detour d represents a distinct alternative. However, it is much longer than even taking all l minor detour from s to t.*

Another natural approach applies *multiple cost functions*, e.g. energy consumption or hop count. In a multi-criteria search, a maximum set of incomparable paths, i.e. a Pareto set, is computed that hopefully contains several distinct paths. Two paths are incomparable if neither dominates the other in all cost functions. This problem is first considered by Hansen [Han80] and later revisited in [Mar84, MHW01, DW09]. As long as the Pareto set is small this can be done efficiently with variants of Dijkstra's algorithm and even in parallel [SM13, EKS14]. However, we are not guaranteed to obtain a reasonable alternative. Another option to handle multiple cost functions is to

apply linear combinations. By varying the ratio between them, different and possibly distinct paths are obtained. A variant of Contraction Hierarchies provides paths that are optimal with respect to all linear combinations of two [GKS10] or more [FS13] cost functions. But again, it is not evident that reasonable alternatives are among them.

The *disjoint path problem* is similar to the k-shortest path problem, except that paths may not overlap. Scott et al. [SPJB97] propose a combination of both methods to find reasonable alternative paths. They compute k-similar paths, i.e. shortest paths with the additional restriction to only have k edges in common with the unrestricted shortest path. However, their linear programming based heuristic is very time-consuming.

Choice Routing by Camvit, though not entirely published, provides alternatives of good quality in practice. This approach is also referred to as *plateau method* as it searches for plateaus in the shortest path trees from a source s and to a target t. A *plateau* $P_{u,v}$ from node u to v is a path of maximal length that appears in both trees. They give candidates for natural alternative paths, i.e. follow the forward tree from s to u, then the plateau, and then the backward tree from v to t. The general concept is first formalized by Abraham et al. in [ADGW13]. They model the alternative paths that are found by the plateau method as a combination of two shortest paths over a *via node*. As considering all plateaus is too costly, the authors introduce several heuristics based on popular speed-up techniques for Dijkstra's algorithm. Their approaches form the basis for our alternative path algorithms and are therefore elaborated in more detail in Section 5.4.1. The plateau method is also studied by Kobitzsch [Kob13]. He exploits the structure of the augmented graph of Contraction Hierarchies in his HiDAR approach to efficiently consider all plateaus at once. This is less efficient than the method of Abraham et al. for one alternative path but soon gets better when multiple alternatives are requested.

Chen et al. [CBB07] introduce a different approach for finding reasonable alternatives to the shortest path with the *penalty method*. They iteratively compute shortest paths while increasing a number of edge costs on the computed paths. A variant thereof that also penalizes edges leaving an joining the computed paths is applied by Bader et al. in [BDGS11]. Their main focus is on computing sparse subgraphs of the considered network that encode many alternative paths, though. They compute multiple paths with the penalty method and choose several of them for inclusion in these *alternative graphs*. Bader et al. further define quality measures for the alternative graphs and introduce filters that can be applied to thin out these graphs. However, they leave open how to extract good alternative paths from them. Kobitzsch et al. and Paraskevopoulos and Zaroliagis revisit the alternative graph problem in [KRS13, PZ13]. The former focus on reducing the runtimes of the penalty method with their CRP-π approach using a modified and parallelized CRP algorithm. In addition, they show how to extract good alternative paths from their alternative graphs. The latter suggest modifications to the penalization scheme to obtain alternative graphs of higher quality. They further introduce a pruning technique based on ALT that yields shorter runtimes than the original method based on Dijkstra's algorithm, but it remains slower than CRP-π.

5.1.2 Contribution

This chapter considers the computation of efficient paths for large sensor networks on classical systems, e.g. as part of a simulation framework or to be used for static networks. We present a heuristic shortest path algorithm based on Contraction Hierarchies that performs well on dense sensor networks. It is an approximation scheme, i.e. computed paths are at most $(1 + \epsilon)$ times longer than a shortest path. Its performance gains are achieved by selectively avoiding the insertion of shortcut edges during the preprocessing compared to the original method. Moreover, this also reduces the memory requirements of our algorithm. Our non-trivial contributions are to ensure that errors do not stack during preprocessing and modifying the query algorithm to remain efficient. We also consider combinations with other speed-up techniques to further improve runtimes at the cost an increased memory overhead.

As efficient paths are not limited to the shortest path, we further study the computation of alternative paths. We show how to engineer previous algorithms found in [ADGW13] to achieve a substantial speed-up. We observe that reasonable alternatives between distinct regions of the network pass over few intermediate nodes. This observation allows us to further decrease runtimes considerably by determining these nodes in advance. We combine our contributions on approximate and alternative path computation for additional gains on dense sensor networks. Moreover, we show how to extend our approach to an online setting without the need for a dedicated preprocessing and how to compute alternative graphs.

In extensive simulations, we show that our shortest and alternative path techniques fare better than previous work from the literature. Our results prove to be robust to multiple distinct network settings.

5.2 Models and Concepts

Before going into the details of our algorithms, we need to describe the models that we apply to represent sensor networks and introduce the notations and concepts used throughout this chapter. We formally define a fundamental routing problem and present basic algorithms for solving it.

5.2.1 Network Model

We consider a sensor network consisting of n nodes $v_i \in V$, with $i \in \{1, \ldots, n\}$. Node positions are arbitrary. The (reliable) communication links between these nodes are described by a connectivity graph $G(V, E)$. This graph is induced by the underlying communication model. Edge $(u, v) \in E$ implies that node v is in the communication range of node u. There are $m = |E|$ communication links in total. Links may be unidirectional. Edge costs model various properties of the communication.

Communication Model. As implied above, a communication model defines whether two sensor nodes can communicate directly with each other depending on their relative positions and possible further conditions, e.g. obstacles or interfering nodes. The model determines whether a node receives signals emitted from another node at a large enough signal strength so that they can be decoded. It induces the connectivity graph $G(V, E)$ associated with the sensor network.

The unit disk graph model requires the connectivity graph to be a unit disk graph as introduced in Section 2.2.1. Two nodes are assumed to be able to communicate (reliably) with each other if their distance is below a certain threshold. Otherwise, communication is considered to be impossible. The threshold is uniform for all pairs of nodes. Other models may introduce multiple threshold values for each pair of nodes and decrease the probability for a reliable communication link to exist between them, the further they are apart. Differing threshold values for each pair of nodes are another possible extension to the basic model.

Edge Cost Model. We utilize edge costs to describe properties of the communication links between nodes. Often, these properties are correlated to the distance between the nodes. Here, we set the cost of each edge $(u, v) \in E$ to the Euclidean distance between nodes u and v to the power of p. This choice covers a wide range of models, especially in context of wireless communication.

For $p = 0$, our edge costs are uniformly set to one and model hop counts. A shortest path distance in this metric corresponds to the minimum number of transmissions required for two nodes to exchange messages. For $p = 1$, we obviously obtain Euclidean distances, which describe signal latencies as signal propagation is roughly uniform at the speed of light in the considered medium (e.g. air, fiber optics). Additional delay in relay nodes can be added as an offset. We use $p = 2$ to model the energy requirements of free-space communication. Imagine the signal as the surface of a growing sphere. The total signal strength is constant, but for an area of fixed size, it shrinks quadratically with the radius of the sphere, i.e. the distance to the sender. Higher values of p in $(2, 6]$ are commonly used to model energy requirements in the presence of signal absorption, reflection, or interference [Rap02]. While values up to 4 are often used to model outdoor communication, e.g. in the flat world model where signals may reflect from the surface and interfere with themselves, higher values up to 6 are only required for complex indoor environments.

5.2.2 Problem Definition

A basic problem that naturally occurs in routing applications is the *shortest path problem*. Many other applications can be broken down to shortest path problems or at least require them as a subroutine, like our alternative path algorithms. The problem was first formally defined in the 1950s by multiple independent authors according to

Schrijver's discourse on the history of the shortest path problem [Sch12]. Formally, we define the problem as

Definition 5.1 (Single Source Shortest Path Problem)**.** *Given a directed graph $G(V, E)$ with edge cost function c, source node $s \in V$, and target node $t \in V$, find the shortest path $P_{s,t}$ between these two nodes.*

The problem is only well-defined in the absence of negative cycles. Otherwise, one can iterate over such a cycle to obtain arbitrarily small results. While the basic problem asks for the actual point-to-point shortest path, variations thereof may only ask for the shortest path distance $d(s, t)$. Algorithms solving this problem are called *distance oracles*. Other variants ask for shortest paths between multiple nodes. These batched problems are solved by *one-to-many*, *many-to-one*, or *many-to-many* queries.

5.2.3 Basic Algorithms and Concepts

We now revisit some of the basic algorithms and concepts introduced in our overview on the related work. In particular, we describe Dijkstra's algorithm, the A* search, and the Arc Flags approach in more detail as well as the partitioning of graphs since these concepts form the basis of the algorithms in the following sections.

Dijkstra's Algorithm. The seminal algorithm by Dijkstra [Dij59] computes shortest path distances from a given source node s to all other nodes in a directed graph $G(V, E)$ with non-negative edge costs c. The algorithm maintains a label $\mu(u)$ for each node $u \in V$ with a tentative distance from s, which is an upper bound on the shortest path distance. Nodes can be unreached, reached, or settled. Unreached nodes are not yet encountered, reached nodes represent the current search horizon and are managed in a priority queue Q with $\mu(u)$ as key for node u, and settled nodes have their correct distance already computed.

The query is initialized by setting $\mu(s) = 0$ and inserting it into Q. All other tentative distances are set to infinity. In each step, node u with the minimal distance from s is removed from the queue and becomes *settled*. The algorithm then scans all outgoing edges (u, v) of u and *relaxes* them, i.e. if $\mu(u) + c(u, v) \leq \mu(v)$ holds, the tentative distance $\mu(v)$ is updated and v is either inserted into the priority queue or its key is decreased. The query terminates once the priority queue is empty. We then have $\mu(u) = d(s, u)$ for all $u \in V$. Algorithm 5.1 summarizes this procedure.

If we are only interested in shortest path distances between two nodes s and t, we can stop the algorithm once t becomes settled. All remaining nodes in Q have a larger tentative distance than $\mu(t)$, and since the graph only contains positive edge costs, $\mu(t)$ cannot be decreased later. This is called a *point-to-point query* in contrast to the regular *one-to-all* query. We can further reduce runtimes by performing a *bidirectional* query from s and t simultaneously [Dan63]. The *forward search* from s only relaxes

Algorithm 5.1 Dijkstra's algorithm

Input: Graph $G(V, E)$, edge cost function c, source s
Output: Array μ of shortest path distances between s and all $u \in V$

```
1:  for all v ∈ V do μ(v) = ∞ end for                    ▷ initialize query
2:  μ(s) ← 0
3:  Q.insert(s, 0)
4:  while not Q.empty() do
5:      u ← Q.deleteMin()                                ▷ settle node u
6:      for all (u, v) ∈ E do
7:          if μ(u) + c(u, v) ≤ μ(v) then                ▷ relax edge (u, v)
8:              μ(v) = μ(u) + c(u, v)
9:              Q.update(v, μ(v))                        ▷ insert v or decrease its key
10:         end if
11:     end for
12: end while
13: return μ                                             ▷ return all shortest path distances
```

outgoing edges and computes distances μ_f from s. The *backward search* from t only relaxes incoming edges and finds distances μ_b to t. We alternate between both search directions. Once a *meeting node* u becomes settled in both directions, we have found the shortest path distance $d(s,t) = \mu_f(u) + \mu_b(u)$. In practice, this approach takes roughly half the time of a unidirectional search with Dijkstra's algorithm. If we think of a normal query as growing a ball of settled nodes around s until it reaches t, the bidirectional search grows two balls of half the radius around s and t, see Figure 5.2(a) and 5.2(b). These balls indicate the *search space* of the query, i.e. the settled nodes.

If we are not only interested in shortest path distances but actual shortest paths, we can augment the algorithm by storing the predecessor of each node, i.e. if $\mu(v)$ is updated while relaxing an edge (u, v), we set u as predecessor of v. By following these predecessors back to s from a node t, we can reconstruct the shortest path from s to t. In the bidirectional case, we have to follow the predecessors from meeting node u back to s and t. The union of all predecessors computed by a full run of Dijkstra's algorithm from s induces the *search tree* for node s.

The time complexity of Dijkstra's algorithm depends on the actual implementation of the priority queue and the running times of its operations. In general, we have $T_{Dijkstra} = \mathcal{O}(m \cdot T_{decreaseKey} + n \cdot (T_{deleteMin} + T_{insert}))$. The algorithm inserts and removes each node exactly once from the priority queue, and each edge results in the decrease of at most one key. *Speed-up techniques* to Dijkstra's algorithm as introduced in Section 5.1.1 try to minimize the number of queue operations. Their search horizons and thus the number of elements stored in the priority queues usually remain very small. The time complexity of queue operations is therefore less important in practice.

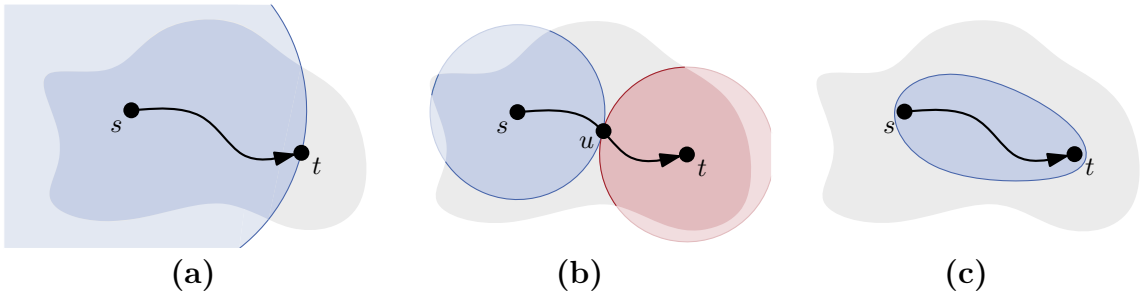


Figure 5.2: *Forward (blue) and backward (red) search spaces for a query of $P_{s,t}$ with (a) plain and (b) bidirectional Dijkstra's algorithm, and (c) A* search.*

A* Search. The A* algorithm [HNR68] is a simple heuristic to Dijkstra's algorithm for point-to-point queries. It applies additional knowledge to reduce the number of settled nodes before reaching target node t. This information is formalized by a *node potential* function $\pi : V \mapsto \mathbb{R}$, which estimates the distances from each node $u \in V$ to target t. We use these estimates to modify the order in which nodes are processed by sorting the nodes in the priority queue by the sum $\mu(u) + \pi(u)$ for each node u. Thus, nodes which are on a path of lowest estimated total cost, i.e. nodes that potentially lead closer to the target, are settled first. This is equivalent to performing Dijkstra's algorithm on a graph with reduced edge costs $c'(u, v) = c(u, v) + \pi(v) - \pi(u)$, i.e. nodes are settled in the same order.

We cannot choose the potential function π arbitrarily, though. The above formulation implies that the reduced edge costs $c'(u, v)$ of all edges $(u, v) \in E$ have to be non-negative. This property is called *feasibility*. It is a necessary condition for Dijkstra's algorithm to remain correct. If potential function π is feasible, so is $\pi'(u) = \pi(u) - \pi(t)$, and $\pi'(u)$ is a lower bound on the distance $d(u, t)$ for all $u \in V$. Consider a shortest path $P_{s,t} = \langle s, v_1, \ldots, t \rangle$. We have $\pi'(s) \leq c(s, v_1) + \pi'(v_1) \leq \ldots \leq c(P_{s,t}) + \pi'(t) = c(P_{s,t})$. Inequalities hold as π' is feasible. The A* search is often faster than Dijkstra's algorithm, but its performance depends on the quality of the lower bounds of π'. Its search space can be visualized by an ellipsoid as shown in Figure 5.2(c). This is already an indicator for shorter runtimes when compared to the ball representing Dijkstra's algorithm.

The A* search can be performed bidirectionally as well with potentials π_f and π_b for the forward and, respectively, backward search direction. In general, a query cannot be stopped once a node u becomes settled in both directions, though. We either have to apply a more complicated stopping criterion [Poh71] or use *consistent* potential functions [IHI+94]. They can be obtained from any two feasible potentials π_f and π_b as $(\pi_f - \pi_b)/2$ for the forward and $(\pi_b - \pi_f)/2$ for the backward direction. While the former approach may use better potential functions, the latter is faster in practice as we may stop once both search spaces meet.

A common choice for the node potentials are Euclidean distances if node positions are known, especially if the edge cost function also models Euclidean distances or a

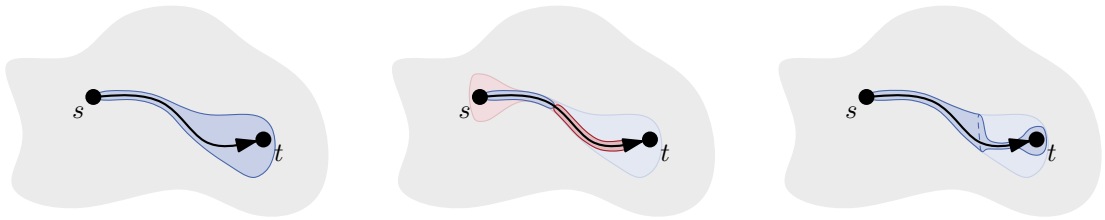
correlated value. The ALT approach by Goldberg and Harrelson [GH05] uses a graph-theoretic lower-bounding heuristic based on landmarks and the triangle inequality. It only applies information inherent to the graph and, in particular, does not need node positions. In a preprocessing step, the approach selects a small set of nodes L and computes distances to and from all other nodes. We have $d(u,t) \geq d(u,l) - d(t,l)$ and $d(u,t) \geq d(l,t) - d(l,u)$ for any node $u \in V$ and landmark $l \in L$. The maximum of these estimates over all landmarks is used as a lower bound. The corresponding potential function is feasible. The quality of the heuristic depends on the landmarks' positions. In general, they should be well spread over the network. Goldberg and Werneck [GW05] study several selection strategies. Their *avoid* strategy is a good compromise between quality and computability. The authors further discuss using a small, adaptive set of active landmarks for performance reasons.

The *weighted A** algorithm [Poh70] is a more aggressive variant of A*. To speed up the query, the condition on feasibility is relaxed at the expense of optimality. It is obtained by exchanging the feasible potential function π for $\pi'(u) = (1+\epsilon) \cdot \pi(u)$, with $\epsilon \geq 0$. This node potential no longer has to be feasible or a lower bound on the distances to t. When settling target t, we can only guarantee that $\mu(t)$ is at most $(1+\epsilon)$ times longer than the shortest path distance $d(s,t)$. The reasoning behind this approach is to preferably settle nodes that are closer to the target and thus arrive sooner at a provably good solution.

Arc Flags. The Arc Flags approach [Lau04] is another extension to Dijkstra's algorithm that uses precomputed information to accelerate point-to-point queries. This aggregated data is like road signs, preventing the query to look in an obviously wrong direction. More formally, the algorithm considers a partitioning of the graph into k regions and stores a label of k flags, one for each region, at every edge $e \in E$. A flag denotes whether a shortest path exists into the respective region over this edge.

A query based on Dijkstra's algorithm only relaxes edges for which the flag of the target's region is set. This simple modification reduces the search space significantly. Closer to the target, however, we experience a coning effect as more and more edges have the flag to the target's region set until it is set for all edges inside the target region as shown in Figure 5.3(a). There are two obvious solutions to this problem. The first is to use a bidirectional search with separate arc flags needed for each search direction. The coning effect is reduced by a great deal as the searches stop once the search spaces meet. The other solution uses a multi-level partitioning of the graph [MSS+07]. Once the search gets closer to the target, it switches to a finer partitioning with a different set of arc flags, narrowing the cone. The search spaces of both approaches are depicted in Figure 5.3(b) and 5.3(c), respectively.

The simplest way to compute arc flags is to perform a one-to-all search in the reverse graph from every node. When setting a node, the appropriate flag is set on the edge incoming from its predecessor, i.e. the flag for the region of the source of this search.



(a) (b) (c)

Figure 5.3: *Forward (blue) and backward (red) search spaces for a query of $P_{s,t}$ with (a) normal, (b) bidirectional, or (c) multi-level Arc Flags. Light colors indicate full AF search spaces. The dashed line marks the switch to the finer partitioning.*

However, this is not practical but for very small graphs. An easy improvement considers only the nodes at region boundaries as every shortest path into a region has to pass them. This is much faster but still rather slow. Hilger et al. [HKMS09] propose to compute the shortest path trees for all boundary nodes of a region at once using a modified Dijkstra's algorithm that stores tentative distances to all boundary nodes. This further reduces preprocessing times but requires a lot of memory during the computation. PHAST [DGNW13] now allows us to efficiently compute many one-to-all searches in parallel, significantly reducing preprocessing times.

In practice, one often only computes shortest path distances instead of shortest path trees. This is more efficient as predecessors do not have to be managed. A subsequent scan over all edges $(u, v) \in E$ is used to set the arc flags by comparing the difference of the computed distances to u and v to $c(u, v)$. This has the side effect that arc flags are set for all shortest paths if multiple equivalent ones exist. This is not strictly needed for the Arc Flags approach and even slightly reduces its query performance, but it simplifies the combination with other techniques as one does not have to take care that both techniques compute the same shortest path.

Storing flags at each edge is very expensive. As many of them are equal, though, they can be managed more efficiently in a lookup table, e.g. in a hash map. They can be further compressed by setting some unset flags, yielding more equivalent labels. This does not impact optimality but slows down the query as slightly more edges have to be relaxed [BDGW10].

Partitioning. Arc Flags and multiple of the algorithms in Section 5.1.1 require a partitioning of the network. We see this subproblem as largely orthogonal to our efforts on shortest path techniques and therefore do not focus on it. We only provide a short overview of the general concepts and notations and refer to the numerous partitioning schemes proposed in the literature, see [BMS+13] for an overview.

A partitioning of $G(V, E)$ is a set of k pairwise disjoint subsets $R_i \subseteq V$, $i \in \{1, \ldots, k\}$ such that the union of all subsets yields V. We refer to these subsets as *regions*. The

subgraph of G induced by each region does not have to be connected. We write R_u for the region of node $u \in V$. Node u is a *boundary node* if there is an edge $(u, v) \in E$ with $R_u \neq R_v$. The number of boundary nodes and the *edge cut*, i.e. the number of edges $(u, v) \in E$ with $R_u \neq R_v$, describe the quality of a partitioning. We further require all regions to be of roughly equal size, i.e. $|R_i| \leq (1 + \epsilon) \cdot \lceil |R_i|/k \rceil \ \forall i \in \{1, \ldots, k\}$, with $\epsilon \geq 0$ the *imbalance* parameter.

5.3 Approximate Queries

We first consider the computation of approximate shortest paths since our approach can be used as a basis for the subsequent alternative path techniques. As it is based on Contraction Hierarchies [GSSV12], we provide an overview of this method before detailing our own contributions. We show how to adapt preprocessing and query algorithms for the approximate case and introduce an additional optimization for dense graphs. Combinations with previous speed-up techniques for shortest path queries complete our contributions.

5.3.1 Baseline Algorithm

Contraction Hierarchies by Geisberger et al. [GSSV12] is a very efficient speed-up technique for Dijkstra's algorithm and the basis for many other algorithms. It applies a preprocessing step to generate an augmented *search graph* $G^*(V, E^*)$ from the original graph $G(V, E)$, which is later used to run queries on.

During preprocessing, nodes are heuristically ordered by some measure of importance, denoted by function $I : V \mapsto \{1, \ldots, n\}$, and *contracted* in this order. Contracting a node u implies removing it (temporarily) from the graph without changing shortest path distances between the remaining, more important nodes. This is ensured by preserving shortest path distances between the neighbors of u. Given two of its neighbors, v and w, we may have to insert a *shortcut edge* (v, w) with cost $c(v, u) + c(u, w)$. A *witness search* verifies whether this is actually needed. It runs a local Dijkstra's algorithm in the remaining graph without u to find a shortest path between v and w. If such a path exists with less or equal cost than the potential shortcut edge, it is a *witness* that the shortcut edge is not needed. Otherwise, we insert the shortcut edge to retain shortest path distances. Witness searches can be performed simultaneously for all pairs of neighbors of a node by a modified Dijkstra's algorithm that stores tentative distances from all source nodes. They do not have to run exhaustively as false negative results do not impact correctness. Unnecessary shortcut edges only increase the search space of a query. We can therefore prune the witness searches, e.g. after settling a given number of nodes. Once all nodes are contracted, the search graph G^* is defined as the union of the original graph G and all generated shortcut edges. The final node order, i.e. the computed importance values, defines a *hierarchy* on the nodes of G^*.

The order in which nodes are contracted is determined by an online heuristic since the computation of an optimal order, e.g. minimal in the number of shortcut edges or in the query search space, is $\mathcal{NP}$-hard [BCK+10]. We assign an initial importance $I(u)$ to each node $u \in V$ prior to any contraction. It measures how attractive it is to contract a node, with the least important nodes contracted first. We describe the importance by a linear combination of several terms, which are computed during a simulated contraction of u. While the original publication uses multiple complicated terms, Vetter [Vet10] introduces much simpler and more efficient ones. We have

$$\alpha \cdot \text{edge quotient} + \beta \cdot \text{original edges quotient} + \gamma \cdot \text{hierarchy depth}\,, \tag{5.1}$$

with $\alpha, \beta, \gamma \in \mathbb{N}$. The *edge quotient* is the number of shortcut edges added during simulation divided by the number of (any) removed edges. It tries to keep the graph sparse. The *original edges quotient* represents the same measure but counts shortcut edges by the number of edges they comprise in the original graph. It attempts to keep shortcut edges small in terms of original edges they represent. The *hierarchy depth* stresses a uniform distribution of contracted nodes and keeps the height of the search graph small. It is the maximum number of hops from a previously contracted node using only (shortcut) edges to more important nodes. Updating the importance values during contraction takes the most amount of time as performing each simulation costs as much as actually contracting the node. Thus, only the neighbors of a contracted node are updated after it is removed from the graph. This heuristic does not catch all changes, though. One can therefore perform *lazy updates*, i.e. update a node right before it is contracted. If it is no longer the node of least importance, this is repeated. If this happens too often, the importance values of all remaining nodes are updated.

Vetter [Vet09] shows that the preprocessing can be parallelized by selecting *independent sets* of nodes and iteratively contracting them in parallel until the whole graph is processed. Independent implies that each node in the set can be contracted without interfering with the contraction of any of the other nodes. The nodes of each independent set are heuristically chosen so that they are minimal in terms of their importance values within a 2-hop neighborhood. This maintains a similar contraction order as in the sequential case, in particular with respect to a uniform distribution. The 2-hop radius further ensures that the node contractions do not interfere with each other. Lazy updates are not applied by Vetter.

The query algorithm is essentially a bidirectional variant of Dijkstra's algorithm on the search graph $G^*(V, E^*)$. However, it only relaxes edges to more important nodes and requires a modified stopping criterion. We keep track of the tentative shortest path distance and abort the search in either direction once the minimum key in the respective priority queue is greater than this distance. More formally, we define an *upward (downward) graph*

$$G^{\uparrow} = (V, E^{\uparrow}), \quad E^{\uparrow} = \{(u, v) \in E^* \mid I(u) < I(v)\}$$
$$(G^{\downarrow} = (V, E^{\downarrow}), \quad E^{\downarrow} = \{(u, v) \in E^* \mid I(u) > I(v)\})$$

of (original and shortcut) edges that lead to more (less) important nodes. Each path in this graph only leads "upward" ("downward") in the hierarchy and is therefore called an *upward* (*downward*) path. We further define an *up-down path* which consists of an upward path followed by a downward path. The forward search of the CH query algorithm only considers $G^{\uparrow}$ and the backward search only $G^{\downarrow}$. Figure 5.4(a) gives an example of the respective search spaces. The tentative shortest path distance for a query between s and t is given by the minimum over all $\mu_f(s,u) + \mu_b(u,t)$, with meeting node u settled in both directions. The (final) shortest path is an up-down path over the meeting node u, which induced the tentative shortest path distance. Geisberger et al. [GSSV12] show that such a path exists in the search graph for each shortest path in the original graph and that they have the same length.

In practice, one often stores the search graph as $G^*(V, E^{\uparrow} \cup \overline{E^{\downarrow}})$ since the backward search considers incoming edges and therefore operates on the reverse graph. This graph is the union of all (normal and reverse) edges that go "upward" in the hierarchy defined by the node order, and it represents a directed acyclic graph (DAG). We therefore may simply speak of an upward path when we actually mean either an upward path in $G^{\uparrow}$ or a downward path in $G^{\downarrow}$.

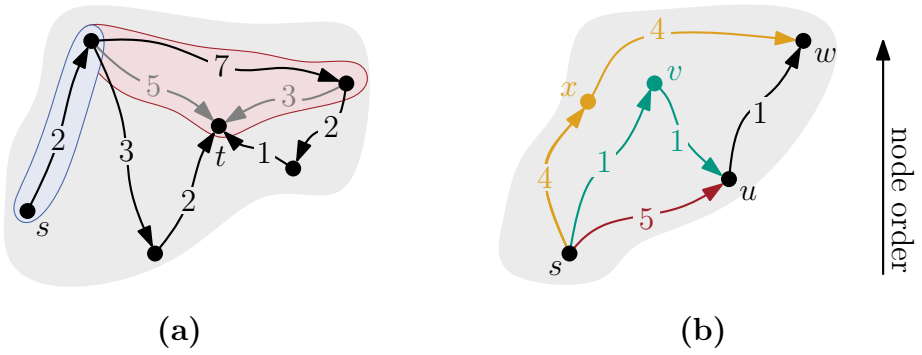


Figure 5.4: *(a) Search graph of Contraction Hierarchies with shortcut edges (grey). Forward (blue) and backward (red) search spaces of a CH query between s and t are shown. (b) Stall-on-demand technique. When u is settled over $\langle s, u\rangle$ (red), it is stalled by v—path $\langle s, v, u\rangle$ (green) costs less. Its neighbor w is also stalled as it is already reached over $\langle s, x, w\rangle$ (orange), but the path over (v, u) costs less.*

Stall-On-Demand Technique. Contraction Hierarchies may settle nodes with a suboptimal distance during a query. While a regular search with Dijkstra's algorithm would never do that, it can happen for Contraction Hierarchies since we do not relax edges leading to less important nodes. Continuing the query from such nodes does not yield a shortest path. We can therefore prune the search at them to reduce the search space and prevent unnecessary work. The following explanation covers the forward search; the backward search is handled symmetrically.

It is easy to verify whether a node u is reached via a suboptimal upward path $p_{s,u}$ when settling it. The general situation is depicted in Figure 5.4(b). For each more important neighbor v of u with edge (v, u) that is already reached by an upward path $p_{s,v}$, we inspect path $p_{s,v,u} = \langle s, \ldots, v, u \rangle$. As $p_{s,v,u}$ is no upward path, it could be shorter than $p_{s,u}$. In particular, if we find $c(p_{s,v,u}) < c(p_{s,u})$, we *stall* u, i.e. we do not relax its incident edges. This is correct as the CH query is correct and as the suboptimal path $p_{s,u}$ is never part of an optimal path. We further try to stall the reached neighbors w of u if the path via v is shorter than their current tentative distance. For correctness, unstalling such reached nodes w can be necessary when the search later finds a shorter upward path than the path via v. The propagation of stalling information is handled in various ways in the literature. While the original publication [Sch08b] conducts a breath-first search to consider even more nodes, [Lux13] does not even propagate the stalling information to the neighbors of u.

This technique is particularly well-suited for pruning search spaces in dense graphs as there are usually a lot of reached but not settled nodes during a query. However, in combination with other techniques that already decrease the overall search space size, the overhead of stalling may become too high in practice [BDS+10].

Shortest Path Retrieval. While the shortest path distances returned by Contraction Hierarchies are equivalent to the ones from Dijkstra's algorithm in the original graph, the resulting path may contain shortcut edges. They have to be *unpacked* to reconstruct the shortest path in the original graph. This is done by a recursive procedure that replaces any shortcut edge with the two edges from which it was built. Storing a *middle node* with each shortcut edge, i.e. the contracted node from the creation of the shortcut edge, is sufficient for the reconstruction of the original path. This information may be omitted if we are only interested in shortest path distances.

5.3.2 Approximation Algorithm

Our heuristic shortest path algorithm, *approximate Contraction Hierarchies* (apxCH), no longer provides exact shortest paths but guarantees that the found path is at most $(1 + \epsilon)$ longer than the exact shortest path for some $\epsilon \geq 0$. It applies a modified preprocessing step to generate an *approximate search graph* $G^*_\epsilon(V, E^*_\epsilon)$ subject to ϵ. The query remains the same as for normal Contraction Hierarchies. However, we later show that the stall-on-demand technique has to be adapted.

During preprocessing, we no longer preserve shortest path distances in the remaining graph when contracting a node, but we still have to guarantee an error bound. Intuitively, when we contract node u, we do not add a shortcut edge between its neighbors v and w if the witness search finds a path $p_{v,w}$ that is just a bit longer than $\langle v, u, w \rangle$. However, we need to ensure that the errors do not stack when we later contract a node on $p_{v,w}$ to guarantee a maximum relative error of ϵ. We therefore introduce a second edge cost $\tilde{c}$ that has to adhere to

Lemma 5.1. *For each edge* $(u, v) \in E^*_\epsilon$ *the inequality*

$$\frac{c(u,v)}{1+\epsilon} \leq \tilde{c}(u,v) \leq c(u,v)$$

holds, with $\epsilon \geq 0$ *and* $\tilde{c} : E^*_\epsilon \mapsto \mathbb{N}_0$ *the* witnessed cost *of an edge.*

The witnessed cost $\tilde{c}(p)$ of a path p denotes the minimal length of a shortcut edge that this path prevented as a witness. We initialize $\tilde{c}$ with the original edge costs. When contracting node u, we add no shortcut edge (v, w) if our witness search finds a path $p_{v,w}$ with cost $c(p_{v,w}) \leq (1+\epsilon) \cdot \big(\tilde{c}(v,u) + \tilde{c}(u,w)\big)$. In this case, we have to update the witnessed edge costs on the (approximate) witness path $p_{v,w}$, though. We distribute the cost difference between witness and omitted shortcut edge proportionally among all edges of the witness. Let $\gamma = \big(\tilde{c}(v,u) + \tilde{c}(u,w)\big)/c(p_{v,w})$, we then set $\tilde{c}(e) = \min\{\gamma \cdot c(e), \tilde{c}(e)\}$ for each edge $e \in p_{v,w}$ on the witness path. This strategy maintains the requirements of Lemma 5.1. The first inequality holds as $\gamma \geq (1+\epsilon)^{-1}$ due to our constraint when to omit a shortcut edge. The second inequality holds as we initialize the witnessed costs with the original edge costs and only decrease them in the following. The strategy further implies that $\tilde{c}(p_{v,w}) \leq \tilde{c}(v,u) + \tilde{c}(u,w)$. We may also use other strategies to distribute the cost difference as long as they are compatible with the above lemma. If we have to insert a shortcut edge (v, w), we set $c(v,w) = c(v,u) + v(u,w)$ and $\tilde{c}(v,w) = \tilde{c}(v,u) + \tilde{c}(u,w)$.

The query algorithm of our approximate Contraction Hierarchies remains the same as for exact Contraction Hierarchies. Similar to CH, it is performed on $G^*_\epsilon(V, E^*_\epsilon)$, the original graph augmented by the shortcut edges of the modified contraction process. In order to prove that apxCH finds paths that are at most $(1+\epsilon)$ times longer than corresponding shortest paths, we first show

Lemma 5.2. *Given a search graph* $G^*_\epsilon = (V, E^*_\epsilon)$ *for* G *with* $\epsilon \geq 0$ *and an arbitrary path* $p_{s,t}$ *in* G^*_ϵ*. There exists an up-down path* $q_{s,t}$ *in* G^*_ϵ *with* $\tilde{c}(q_{s,t}) \leq \tilde{c}(p_{s,t})$*.*

Proof. Let $P_{s,t}$ be a shortest path in G^*_ϵ. If it is an up-down path, we are done and $P_{s,t}$ corresponds to $q_{s,t}$. Otherwise, we iteratively construct an up-down path as Figure 5.5 illustrates. We select a locally minimal node $u \in P_{s,t}$, i.e. a node whose predecessor v and successor w on $P_{s,t}$ have a higher importance, of overall minimal importance (obviously excluding s and t). During preprocessing, it is contracted before v and w, and therefore either a shortcut edge (v, w) or a witness path $p_{v,w}$ of more important nodes and cost $c(p_{v,w}) \leq (1+\epsilon) \cdot \big(\tilde{c}(v,u) + \tilde{c}(u,w)\big)$ exists. By construction, the witnessed cost of either is no more than $\tilde{c}(\langle v,u,w\rangle)$. Thus, we can replace subpath $\langle v,u,w\rangle$ on $P_{s,t}$ by either of the two to obtain path $q_{s,t}$ with cost $\tilde{c}(q_{s,t}) \leq \tilde{c}(P_{s,t})$. It consists only of nodes more important than v. Since $n < \infty$, we can iterate this process only finitely many times until there exists no more node u with the above property to select. The resulting path $q_{s,t}$ is therefore an up-down path, and $\tilde{c}(q_{s,t}) \leq \tilde{c}(P_{s,t}) \leq \tilde{c}(p_{s,t})$ holds. □

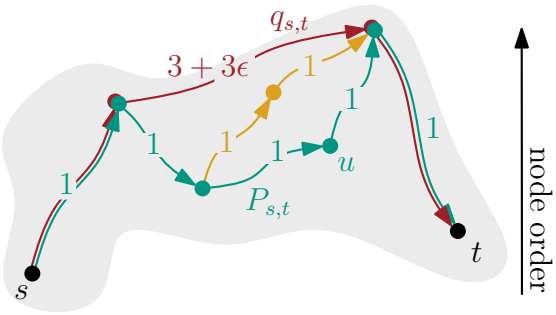


Figure 5.5: *Illustration for Lemma 5.2. We can iteratively replace subpath $\langle v, u, w \rangle$ on the shortest path $P_{s,t}$ (green) by a path of more important nodes than u (orange), always complying with the cost constraints, until we obtain up-down path $q_{s,t}$ (red).*

We are now ready to show that our query algorithm has an approximation ratio of $(1+\epsilon)$ with respect to the cost of the "shortest" path that it returns when used with an approximate search graph $G^*_\epsilon(V, E^*_\epsilon)$ as introduced above. We have

Theorem 5.1. *Given a directed graph $G(V, E)$ with edge cost function c, source node $s \in V$, and target node $t \in V$. Let $\tilde{d}(s,t)$ be the distance computed by the apxCH algorithm with $\epsilon \geq 0$ and let $d(s,t)$ be the optimal (shortest) distance in the original graph. Then $d(s,t) \leq \tilde{d}(s,t) \leq (1+\epsilon) \cdot d(s,t)$.*

Proof. Let $G^*_\epsilon = (V, E^*_\epsilon)$ be an approximate search graph for G and ϵ. By construction, the shortest path distance between s and t in G^*_ϵ is the same as in the original graph. Thus, the first inequality holds. Moreover, every shortest path $P_{s,t}$ in the original graph also exists in G^*_ϵ. Our query algorithm, however, only finds up-down paths. But if there exists a shortest path $P_{s,t}$ in G^*_ϵ, there exists an up-down path $q_{s,t}$ with $\tilde{c}(q_{s,t}) \leq \tilde{c}(P_{s,t})$, as per Lemma 5.2. We can further assess $\frac{c(q_{s,t})}{1+\epsilon} \leq \tilde{c}(q_{s,t})$ and $\tilde{c}(P_{s,t}) \leq c(P_{s,t})$ with Lemma 5.1. Altogether, we obtain $c(q_{s,t}) \leq (1+\epsilon) \cdot c(P_{s,t})$. Thus, the second part of our initial inequality holds as well. Our query algorithm either finds $q_{s,t}$ or another path that is no longer than $q_{s,t}$ as a CH query finds a smallest up-down path. □

Just as the proof of correctness of exact Contraction Hierarchies, see [GSSV12], our proof of Theorem 5.1 does not depend on the order in which nodes are contracted. Moreover, while the proof uses witnessed edge costs $\tilde{c}$, the actual query algorithm does not use them at all. They are only required during preprocessing and therefore do not need to be stored afterwards.

Our algorithm is an approximation scheme according to the definition in Section 2.1.2 as it has an approximation ratio of $(1+\epsilon)$. If we further consider [ADF+13], which states a time complexity for the preprocessing and query algorithm of Contraction Hierarchies that is polynomial in the highway dimension and the diameter of the graph (which are both polynomial in the graph size), we may label our algorithm as a

polynomial time approximation scheme (PTAS). As we only omit shortcut edges for $\epsilon > 0$, which speeds up preprocessing and query, we could even consider our algorithm as a fully polynomial time approximation scheme (FPTAS). However, no proof is given.

Stall-On-Demand Technique. As stressed before, the stall-on-demand technique is an important ingredient for a practically efficient implementation of the Contraction Hierarchies query. However, when using approximate search graphs, we have to be careful not to destroy the correctness of our query algorithm as it no longer computes optimal paths. Figure 5.6 gives an example in which a query with stall-on-demand does not find a shortest path.

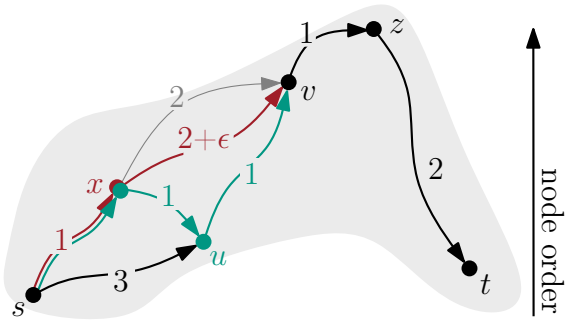


Figure 5.6: *A query with stall-on-demand does not find a shortest path from s to t. Node u is stalled by x when settled—path $\langle s, x, u\rangle$ costs less than $\langle s, u\rangle$. Its neighbor v is also stalled as it is already reached by $\langle s, x, v\rangle$ (red), but the path over (x, u) (green) costs less. The forward search therefore never reaches z. Note that shortcut (x, v) (grey) is omitted in the approximate search graph $G^*_{0.1}(V, E^*_{0.1})$ during contraction.*

To maintain the correctness of our algorithm, i.e. to ensure that it finds a path between any pair of nodes s and t that is at most $(1+\epsilon)$ times longer than the respective shortest path in G, we need to modify the stall-on-demand technique. When settling a node u over upward path $p_{s,u}$, we consider all of its more important neighbors x with edge (x, u) that are already reached over another upward path $p_{s,x}$. If we find

$$c(p_{s,x}) + (1 + \epsilon) \cdot c(x, u) < c(p_{s,u})\,, \tag{5.2}$$

we stall node u and propagate the stalling information to its reached neighbors v. They are stalled if the path via x is shorter than their current tentative distance. However, we consider the cost of the subpath from x to v as scaled by a factor of $(1 + \epsilon)$. The symmetric approach applies to the backward search. For $\epsilon = 0$, this corresponds to the stalling condition of the exact query algorithm. To show that our *approximate stall-on-demand* condition (5.2) is correct, we iteratively construct a new up-down path from a stalled one by applying Lemma 5.3. Figure 5.7 illustrates a single iteration.

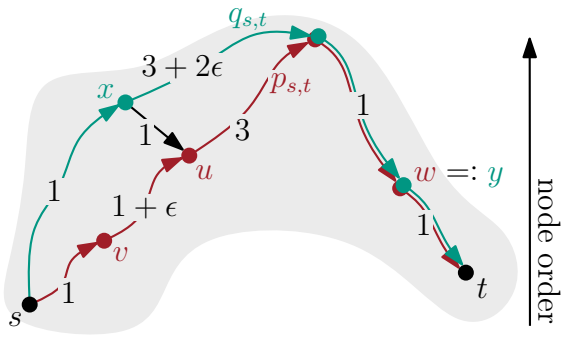


Figure 5.7: *Illustration for Lemma 5.3. Node u on up-down path $p_{s,t}$ (red) is stalled by x. A new up-down path $q_{s,t}$ (green) can be computed from the concatenation of paths $p_{s,x}$, (x,u), and $p_{u,t} \subseteq p_{s,t}$ with Lemma 5.2.*

Lemma 5.3. *Let $(p_{s,t}, v, w)$ be a* stall state triple *(SST), with $p_{s,t}$ an up-down path, node $v \in p_{s,t}$ settled by the forward search over $p_{s,v}$ and not stalled and node $w \in p_{s,t}$ settled by the backward search over $p_{w,t}$ and not stalled. We further define*

$$g(p_{s,t}, v, w) = c(p_{s,v}) + (1+\epsilon) \cdot \tilde{c}(p_{v,w}) + c(p_{w,t})\,.$$

If one of the nodes in $p_{v,w} \subseteq p_{s,t}$ becomes stalled, there is an SST $(q_{s,t}, x, y)$ with

$$g(q_{s,t}, x, y) < g(p_{s,t}, v, w)\,.$$

Proof. Let $u \in p_{v,w}$ be the node that is stalled. We assume w.l.o.g. that $p_{v,u} \in p_{s,t}$ is an upward path, i.e. the stalling occurs in the forward search. Let u be stalled by x, which is reached over upward path $p_{s,x}$. Let $r_{x,w}$ be the up-down path constructed from the concatenation of (x,u) and $p_{u,w} \subseteq p_{s,t}$ following Lemma 5.2. Set $y := w$ and let $q_{s,t}$ be the concatenation of $p_{s,x}$, $r_{x,w}$, and $p_{w,t}$. By construction, $(q_{s,t}, x, y)$ is an SST with the above property since

$$\begin{aligned}
g(q_{s,t}, x, y) &= c(q_{s,x}) + (1+\epsilon) \cdot \tilde{c}(q_{x,y}) + c(q_{y,t}) \\
&= c(p_{s,x}) + (1+\epsilon) \cdot \tilde{c}(r_{x,w}) + c(p_{w,t}) && \text{(by definition)} \\
&\leq c(p_{s,x}) + (1+\epsilon) \cdot (\tilde{c}(x,u) + \tilde{c}(p_{u,w})) + c(p_{w,t}) && \text{(Lemma 5.2)} \\
&\leq c(p_{s,x}) + (1+\epsilon) \cdot c(x,u) + (1+\epsilon) \cdot \tilde{c}(p_{u,w}) + c(p_{w,t}) && \text{(Lemma 5.1)} \\
&< c(p_{s,u}) + (1+\epsilon) \cdot \tilde{c}(p_{u,w}) + c(p_{w,t}) && \text{(5.2)} \\
&= c(p_{s,v}) + c(p_{v,u}) + (1+\epsilon) \cdot \tilde{c}(p_{u,w}) + c(p_{w,t}) \\
&\leq c(p_{s,v}) + (1+\epsilon) \cdot \tilde{c}(p_{v,u}) + (1+\epsilon) \cdot \tilde{c}(p_{u,w}) + c(p_{w,t}) && \text{(Lemma 5.1)} \\
&= g(p_{s,t}, v, w)
\end{aligned}$$

If u is stalled through the propagation of stalling information, we replace edge (x,u) by the path from x to u, with $p_{s,x}$ still a maximal upward path starting at s. □

With this lemma, we can show that our query algorithm is correct on $G^*_\epsilon(V, E^*_\epsilon)$, even when using the approximate stall-on-demand technique. We state

Theorem 5.2. *Theorem 5.1 holds when using approximate stall-on-demand (5.2).*

Proof. We iteratively construct an SST with Lemma 5.3, starting with the up-down path $p_{s,t}$ found by a query between s and t without stall-on-demand as in the proof of Theorem 5.1. Obviously, at the beginning of the query, both nodes s and t are settled and not stalled. Thus, $(p_{s,t}, s, t)$ is an SST and

$$\begin{aligned} g(p_{s,t}, s, t) &= c(p_{s,s}) + (1+\epsilon) \cdot \tilde{c}(p_{s,t}) + c(p_{t,t}) \\ &= (1+\epsilon) \cdot \tilde{c}(p_{s,t}) \\ &\leq (1+\epsilon) \cdot d(s,t)\,, && \text{(Theorem 5.1)} \end{aligned}$$

holds. After a finite number of applications of Lemma 5.3, we obtain an SST $(q_{s,t}, x, y)$ so that path $q_{s,t}$ is found by our query with stalling. For this path $q_{s,t}$, we have

$$\begin{aligned} c(q_{s,t}) &= c(q_{s,x}) + c(q_{x,y}) + c(q_{y,t}) \\ &\leq c(q_{s,x}) + (1+\epsilon) \cdot \tilde{c}(q_{x,y}) + c(q_{y,t}) && \text{(Lemma 5.1)} \\ &= g(q_{s,t}, x, y) \\ &\leq g(p_{s,t}, s, t) && \text{(Lemma 5.3)} \\ &\leq (1+\epsilon) \cdot d(s,t)\,. \end{aligned}$$

As our graph is finite and due to the strict relation in Lemma 5.3, we can apply Lemma 5.3 only a finite number of times. The final SST $(q_{s,t}, x, y)$ is found by our query algorithm as x is settled in the forward search and not stalled and as y is settled in the backward search and not stalled. Since this is the final SST, no node on the path $q_{x,y}$ is stalled. Thus, our query finds path $q_{s,t}$ or a shorter path. □

As before, we only require the witnessed edge costs $\tilde{c}$ for our proofs. They are not used in the actual query algorithm and therefore still do not have to be stored once the preprocessing is done.

While Theorem 5.2 guarantees that the approximation ratio $(1+\epsilon)$ of Theorem 5.1 also holds for our query algorithm when applying the approximate stall-on-demand technique, it does not ensure that we obtain the same path and therefore the same error. Figure 5.8 gives an example in which the observed error increases when using the stall-on-demand technique.

Node Ordering. We introduce a slight modification to the heuristic node ordering process. While not strictly required for approximate Contraction Hierarchies, it is a necessary ingredient to handle dense graphs efficiently.

The preprocessing routine of exact Contraction Hierarchies updates the importance values of all neighbors of a contracted node. In addition, it performs lazy updates for

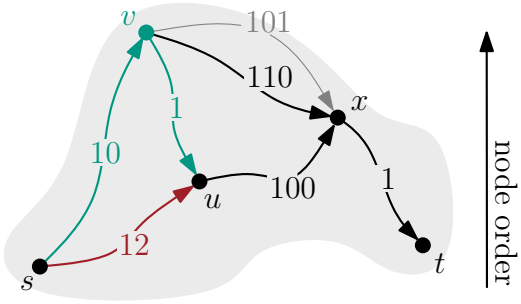


Figure 5.8: *Stalling may increase the observed error. An apxCH query without stalling finds up-down path $\langle s, u, x, t\rangle$ with an error of 0.9%. With stalling, only the longer path $\langle s, v, x, t\rangle$ with an error of 8.0%, is found since u is stalled by v—$\langle s, v, u\rangle$ (green) costs less than $\langle s, u\rangle$ (red). Note that shortcut (v, x) (grey) is omitted in the approximate search graph $G^*_{0.1}(V, E^*_{0.1})$ during contraction.*

the node of least importance before contracting it, and it recomputes the importance of all remaining nodes from time to time. Updating neighbors is very expensive in a dense graph, though. Not only are there many neighbors, for which importance values have to be recomputed, the required simulated node contraction is also much more expensive than on sparse graphs. While we can limit local searches, the number of relaxed edges remains high if we want to find any witness path (e.g. consider an average node degree of 20, even reaching a node in 2 hops distance may require 400 edge relaxations). We therefore opt not to update neighbors of contracted nodes, and since each update is expensive, we also forego updating the remaining nodes after too many lazy updates. We only perform lazy updates.

For parallel preprocessing, we proceed slightly differently than for normal lazy updates. After selecting an independent set of nodes for contraction, we recompute the importance values of these nodes. Next, we check which of these nodes remain locally minimal with respect to their updated importance. The ones that remain minimal are contracted (in parallel) before the next independent set is determined.

5.3.3 Combination with Other Techniques

In general, we can use the approximate search graph of apxCH with any technique that utilizes the exact CH search graph without any modifications as long as only information of up-down paths is considered, like in many-to-many queries or PHAST. Otherwise, however, we can encounter shorter paths than our approximate query is able to find. This may lead to inconsistencies and thus to complications similar to what we described for the (exact) stall-on-demand technique before. The general issue is illustrated in Figure 5.9.

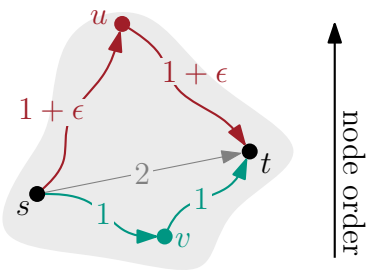


Figure 5.9: *By considering edge (s, v), which goes to a less important node, we may encounter path $p_{s,v,t}$ (green), which is shorter than path $p_{s,u,t}$ (red). However, an apxCH query between s and t only find the latter path. The shortcut edge (grey) required for correctness in an exact CH search graph is omitted when using apxCH.*

Combinations of goal-directed and hierarchical speed-up techniques have been very successful in the past, see Bauer et al. [BDS+10], trading a little bit of preprocessing time and memory for a substantial gain in query times. We therefore consider two combinations of approximate Contraction Hierarchies with previous goal-directed techniques in more detail.

apxCHASE. First, we consider the CHASE algorithm, which combines Contraction Hierarchies with Arc Flags. The original implementation in [BDS+10] only applies arc flags on a small core of the graph consisting of the most important nodes. This is done as their preprocessing was expensive at that time and the overhead of storing them at each edge substantial. By now, PHAST allows us to quickly compute arc flags even for large graphs and by storing them not at the edges but in a hash map, we can save a lot of memory. We therefore apply arc flags on the whole graph. Preprocessing for CHASE computes a CH search graph and arc flags as for the individual techniques. However, boundary nodes are now inferred from the search graph instead of the original graph and arc flags are also computed for shortcut edges. The CHASE query algorithm remains largely equivalent to the one of CH, but it checks whether an edge has a flag set for the appropriate target region before relaxing it. Thus, the forward search prunes all edges not on a shortest path to region R_t, and the backward search respectively prunes all edges not coming from R_s, the region to which source s belongs.

To adapt CHASE for approximate queries, we exchange CH for apxCH. We call the resulting heuristic algorithm *apxCHASE*. Preprocessing starts by computing the approximate CH search graph, which is then used for preprocessing arc flags. We need to modify the required graph searches for the latter to only consider up-down paths, though. As they normally run on a flattened search graph, i.e. they relax edges to both more and less important nodes, this may lead to the issues described above and thus to inconsistencies between the shortest paths flagged by the arc flags and the paths apxCH finds. However, when using PHAST to determine arc flags, this is automatically

guaranteed as the shortest paths returned by PHAST are the same paths a CH (or apxCH) query would return.

The apxCHASE query uses the changes to the stall-on-demand technique detailed in Section 5.3.2 for apxCH. In addition, a path p may only stall an upward path if the flag for the appropriate target region is set on all edges of p. However, the stall-on-demand technique is usually switched off in CHASE (and apxCHASE) queries as the computational overhead outweighs the performance gain in practice.

With these changes to CHASE we have

Theorem 5.3. *Given a directed graph $G(V, E)$ with edge cost function c, source node $s \in V$, and target node $t \in V$. Let $\tilde{d}(s,t)$ be the distance computed by the apxCHASE algorithm with $\epsilon \geq 0$ and let $d(s,t)$ be the optimal (shortest) distance in the original graph. Then $d(s,t) \leq \tilde{d}(s,t) \leq (1+\epsilon) \cdot d(s,t)$.*

apxCHALT. On some networks, ALT proves to be superior to using the Arc Flags approach. We therefore propose our second combination, *apxCHALT*, which combines apxCH with ALT. The pattern is the same as before with apxCHASE. We first compute an apxCH search graph and then apply ALT on this graph. This time, however, we only apply ALT on a small core like in the original CHASE algorithm or CALT. The reason for this deviation to our above choices is the huge impact on memory consumption that applying ALT on the whole graph would entail. Formally, the core consists of the top fraction of nodes in the apxCH search graph with respect to the node order. ALT preprocessing selects a small but fixed set of landmark nodes from within this core and computes shortest path distances from and to all other nodes in the core.

The apxCHALT query algorithm is performed in two phases. The first phase is a pure apxCH query without ALT that stops at nodes belonging to the core. The second phase, if required, starts from the core nodes that have been settled during the first phase of the query and only runs on the core. We continue with an apxCH query, but now guided by ALT node potentials. The conditions when to stop each phase are the same as for the CALT query. We further apply the same approach as CALT to determine node potentials if source or target are not in the core, i.e. using appropriate proxy nodes.

Overall, we obtain

Theorem 5.4. *Given a directed graph $G(V, E)$ with edge cost function c, source node $s \in V$, and target node $t \in V$. Let $\tilde{d}(s,t)$ be the distance computed by the apxCHALT algorithm with $\epsilon \geq 0$ and let $d(s,t)$ be the optimal (shortest) distance in the original graph. Then $d(s,t) \leq \tilde{d}(s,t) \leq (1+\epsilon) \cdot d(s,t)$.*

Similarly to the weighted A* algorithm of Section 5.2.3, we may choose to also approximate the ALT part of the query by scaling the node potentials by a factor of $(1+\epsilon')$. The approximation guarantees of both algorithms, apxCH and the weighted ALT search, are multiplicative. We therefore can only guarantee that the computed distances are no longer than $(1+\epsilon) \cdot (1+\epsilon')$ times the shortest path distance.

5.4 Alternative Connections

We now turn to the computation of alternative paths. As already mentioned in the related work section, our approach is based on the modelling and algorithms introduced by Abraham et al. in [ADGW13]. We therefore begin with an overview of their method, which we also refer to as *baseline algorithm*, before introducing our own contributions. We develop query variants and show how to perform the required preprocessing efficiently. An online setting with *on-the-fly* preprocessing and the computation of alternative graphs complete our contributions.

5.4.1 Baseline Algorithm

Abraham et al. [ADGW13] model alternative paths between two nodes $s, t \in V$ as a concatenation of two shortest paths $P_{s,v}$ and $P_{v,t}$. They are fully specified by the *via node* v. The concatenated path $P_{s,v,t}$ is a *via path*. A via path has to be reasonable to be considered as a viable alternative, though. The authors describe a class of *admissible* alternative paths in which a via path $P_{s,v,t}$ has to obey three heuristic but natural conditions: First, the via path has to be significantly different from the shortest path. Second, every local decision along the via path has to make sense. And finally, every via path should only be a fraction longer than a shortest path. Figure 5.10 gives example alternative paths that do not respect the above conditions.

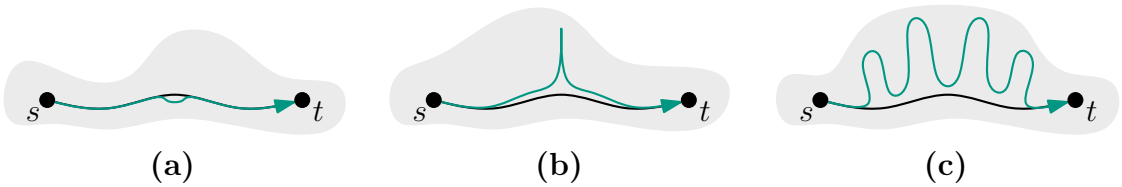


Figure 5.10: *Examples for alternative paths (green) to $P_{s,t}$ (black) that are not admissible due to the condition in Definition 5.2 on (a) limited sharing, (b) local optimality, or (c) uniformly bounded stretch.*

Before formalizing these conditions and the notion of an admissible alternative in Definition 5.2, we need to introduce two properties of via paths: A via path $P_{s,v,t}$ is *T-locally optimal (T-LO)* if two conditions hold: (a) Every sufficiently short subpath $p' \subseteq P_{s,v,t}$ with $c(p') \leq T$ has to be a shortest path. (b) Let p' be a subpath of $P_{s,v,t}$ and p'' obtained from p' by removing the ending nodes of p' on both sides, then p' has to be a shortest path if $c(p') > T$ and $c(p'') < T$. A via path $P_{s,v,t}$ is said to have $(1+\epsilon)$ *uniformly bounded stretch* (UBS) if every subpath $p_{u,w} \subseteq P_{s,v,t}$ is at most $(1+\epsilon)$ times longer than the respective shortest path $P_{u,w}$.

Given the above properties, Abraham et al. [ADGW13] introduce

Definition 5.2 (Admissible Alternative)**.** *With $\alpha, \gamma \in [0,1]$ and $\epsilon \geq 0$, a path $P_{s,v,t}$ between s and t is called an admissible alternative to $P_{s,t}$ if it satisfies the following three conditions:*

1. $c(P_{s,t} \cap P_{s,v,t}) \leq \gamma \cdot c(P_{s,t})$, *(limited sharing)*
2. $P_{s,v,t}$ *is T-locally optimal for $T = \alpha \cdot c(P_{s,t})$, and* *(local optimality)*
3. $P_{s,v,t}$ *has $(1+\epsilon)$-UBS.* *(uniformly bounded stretch)*

The summed cost of all edges common to both paths is denoted by $c(P_{s,t} \cap P_{s,v,t})$. The above measures require a number of shortest path queries to be verified that is quadratic in the number of nodes of $P_{s,v,t}$. Generally speaking, this is not practical for a setting in which many queries have to be answered in a short timeframe.

A Practical Algorithm. When requesting an alternative to a shortest path $P_{s,t}$, we obviously cannot consider all nodes in the graph and check whether they induce an admissible alternative. We cannot even consider a sizeable amount of them since the conditions that have to be checked are expensive to compute. In order to still obtain reasonable alternatives in an efficient manner, one has to resort to heuristics.

Abraham et al. [ADGW13] give a practical solution based on a bidirectional Dijkstra's algorithm, called *X-BDV*, to compute single via paths that are reasonable and good alternatives. The heuristic incorporates ideas from the plateau method and works as follows: An *exploration query* identifies potential alternative paths. A (forward) shortest path tree is grown from s and another (backward) tree from t until all nodes are settled that are not farther than $(1+\epsilon) \cdot c(P_{s,t})$ away from the root of their respective tree. This bound is tight as no admissible path can be any longer than this threshold. Next, each node v that is settled in both search trees becomes a *via node candidate*. Three measurements are computed in linear time for each of the candidates: $c(P_{s,v,t})$, the length of via path $P_{s,v,t}$, $\sigma(P_{s,v,t})$, the amount of sharing of $P_{s,v,t}$ with the optimal route, and $pl(P_{s,v,t})$, the length of a longest plateau containing v. These more practical measures are used to sort all candidates in non-decreasing order according to the priority function $f(P_{s,v,t}) = 2 \cdot c(P_{s,v,t}) + \sigma(P_{s,v,t}) - pl(P_{s,v,t})$. The first via path $P_{s,v,t}$ is returned that is *approximately admissible*. If no viable alternative can be found, we report this as a negative result.

Before formalizing the notion of approximate admissibility in Definition 5.3, we introduce a 2-approximation for T-local optimality—called *T-test*—that is easy to compute in linear time. Given a via path $P_{s,v,t}$ and a parameter T, let u be the closest node on $P_{s,v}$ that is at least T away from v or s. Likewise, w is the closest node on $P_{v,t}$ that is also at least T away or s. Such a path $P_{s,v,t}$ is said to pass the T-test if the portion of $P_{s,v,t}$ between u and w is a shortest path. If $pl(P_{s,v,t}) > T$, the T-test is always successful.

Given the above approximation, Abraham et al. [ADGW13] specify

Definition 5.3 (Approximately Admissible)**.** *With $\alpha, \gamma \in [0,1]$ and $\epsilon \geq 0$, a path $P_{s,v,t}$ between s and t is an approximately admissible alternative to $P_{s,t}$ if the following three conditions hold:*

1. $\sigma(P_{s,v,t}) < \gamma \cdot c(P_{s,t})$, *(limited sharing)*
2. *successful T-test for $T = \alpha \cdot c(P_{s,v,t} \backslash P_{s,t})$, and* *(local optimality)*
3. $c(P_{s,v,t} \backslash P_{s,t}) < (1+\epsilon) \cdot c(P_{s,t} \backslash P_{s,v,t})$. *(small stretch)*

We have $c(P_{s,v,t} \backslash P_{s,t}) = c(P_{s,v,t}) - \sigma(P_{s,v,t})$ and $c(P_{s,t} \backslash P_{s,v,t}) = c(P_{s,t}) - \sigma(P_{s,v,t})$. Note that Abraham et al. define local optimality and stretch with respect to the detour of the alternative to the shortest path and not with respect to the entire path. This is done for practical reasons, see [ADGW13] for their reasoning.

We may exclude nodes that are not approximately admissible before sorting the remaining ones according to the above priority function as all required measures are already available during the exploration query. We further need to ensure that forward and backward search consider the same paths in the presence of multiple equidistant nodes. Otherwise, no meaningful plateaus are found. If this is not an option, we have to omit the plateau term in our priority function and perform the T-test explicitly.

The X-BDV approach can be iterated to compute multiple via paths that represent an alternative to the shortest path and to all previous alternatives. In the presence of multiple alternatives, $\sigma(P_{s,v,t})$ denotes the overlap to $P_{s,t}$ and to all previous alternatives. The other measures remain the same as for one alternative.

A Faster Variant. Abraham et al. [ADGW13] further apply the query of Contraction Hierarchies to the above method in order to obtain a faster algorithm, *X-CHV*, which we describe next. The forward and backward (CH) search spaces of nodes s and t are fully explored. Nodes v in the forward search space are reached over path $p_{s,v}$ with *forward distance* $\mu_f(v)$ and, respectively, nodes in the backward search space are reached over path $p_{v,t}$ with *backward distance* $\mu_b(v)$. These paths do not have to be shortest paths and may still contain shortcut edges. For each node v that occurs in both search spaces, a preselection similar to the conditions in Definition 5.3 is done. Nodes are discarded for which the sum of forward and backward distance is longer than a certain fraction of the length of the shortest path, i.e. $\mu_f(v) + \mu_b(v) \geq (1+\epsilon) \cdot c(P_{s,t})$. These distances are not necessarily correct but at least upper bounds. Thus, we have $\mu_f(v) + \mu_b(v) \geq c(P_{s,v,t})$. We further check whether the *approximated overlap* $\sigma^{apx}(p_{s,v,t})$ of the concatenated paths $p_{s,v}$ and $p_{v,t}$ in the forward and backward search spaces is at most as long as a certain fraction of the shortest path distance, i.e. $\sigma^{apx}(p_{s,v,t}) < \gamma \cdot c(P_{s,t})$. Paths $p_{s,v}$ and $p_{v,t}$ are unpacked as needed. In addition, a condition regarding the stretch of these paths must hold, i.e. $c(p_{s,v}) + c(p_{v,t}) - \sigma^{apx}(p_{s,v,t}) < (1+\epsilon) \cdot \big((c(p_{s,t}) - \sigma^{apx}(p_{s,v,t})\big)$. The remaining candidates are ranked according to the same priority function as X-BDV except that approximated overlap and distance values are used. Moreover, the plateau

length is always zero for methods based on Contraction Hierarchies as their forward and backward search spaces only meet but do not overlap. We compute the exact via path $P_{s,v,t}$ for nodes v in the order of the ranking. The first node for which the conditions of Definition 5.3 hold is selected as via node.

The success rate of X-CHV is inferior to X-BDV since search spaces are much narrower. To cope with these smaller success rates, Abraham et al. introduce a *relaxed exploration* phase. This relaxed query is allowed to search more nodes than the plain CH query to (hopefully) find more viable via node candidates. While the latter prunes all edges to lower-order nodes in the search graph hierarchy, the relaxed query may look to less important nodes under certain conditions. Let $\text{pred}_i(u)$ be the i-th ancestor of u in the search tree. The *x-relaxed CH query* prunes an edge (u, v) if and only if v precedes all vertices $u, \text{pred}_1(u), \ldots, \text{pred}_x(u)$ in the order of the hierarchy. If u has fewer than k ancestors v is never pruned. A 0-relaxed CH query corresponds to plain Contraction Hierarchies. Relaxed CH search spaces may contain plateaus due to this "downward" search, but the priority function continues to use $pl(P_{s,v,t}) = 0$. For our implementation, we apply a slightly modified relaxation as it achieves superior runtimes with only a small impact on the found alternatives. Our variant prunes an edge (u, v) if and only if v precedes node $\text{pred}_x(u)$ in the order of the hierarchy. It explores less than a third of the nodes compared to the original relaxation technique.

This algorithm, the x-relaxed variant of X-CHV, is the starting point of our work.

Engineering the Baseline Algorithm. The X-CHV algorithm can be further accelerated by exchanging some of its algorithmic components by more efficient ones. While this is mostly straight-forward, we give a description for the sake of completeness. This engineered variant has not been described in the previous work and is used later in our simulations in Section 5.5.3.

Recall that the algorithm is a two-step method. A bidirectional *exploration query* searches for via node candidates that are subsequently tested using a number of point-to-point shortest path queries, which we call *target queries*. The obvious approach is to handle path searches by faster methods than Contraction Hierarchies. One has to keep in mind, though, that the optimization goals for target and exploration queries are conflicting. The first one should be as fast as possible and therefore settle as few nodes as possible that are not in the shortest path, while the latter one has to explicitly explore nodes that are not on the shortest path.

For instance, we apply CHASE as introduced in Section 5.1.1 for the target queries. Albeit any path oracle could be used in principle, CHASE is a natural choice as we show in the following sections. The only additional data necessary are a partitioning of the input graph and precomputed arc flags. While the point-to-point queries of CHASE are very fast, their search spaces are too narrow to be used for exploration as a comparison in Appendix C confirms. In addition to that, we store all shortcut edges of the CH search graph pre-unpacked to replace the recursive path unpacking procedure

by a simple table lookup. Both optimizations have equal impact on the runtimes. The combination of these two steps speeds up the computation by roughly a factor of two when compared to a simple CH query. We refer to the baseline algorithm X-CHV that uses CHASE for target queries and pre-unpacked shortcut edges as *X-CHASEV*.

5.4.2 Preprocessed Candidate Nodes

The analyses of Abraham et al. [AFGW10] show that speed-up techniques to Dijkstra's algorithm work especially well on certain classes of graphs where most shortest paths leaving a region go through a small set of nodes, i.e. all shortest paths out of that region are *covered* by this small node set. Their analyses lead us to the following similar assumption for alternative paths:

Assumption 5.1 (Limited Number of Alternative Paths)**.** *If the number of shortest paths between any two sufficiently far away regions of a network is small [AFGW10], so is the number of plateaus for the Choice Routing algorithm [Cam05]. Likewise, the number of admissible alternative paths of the algorithms in [ADGW13] is small and can be covered by few nodes.*

As any path between two regions of a partitioned graph has to go through the boundary nodes of the regions, it should suffice to consider only alternative paths between the boundary nodes. As distances between (non-neighboring) regions are much longer than within a region, the expected error due to this simplification should be minimal. Preliminary simulations support our assumptions. We observe that although positions of via nodes are (sometimes substantially) different, the number of good alternative paths between the two regions is rather limited. This variation in via node positions is a consequence of any node on a plateau being a via node for the same alternative over this plateau. In the light of these observations, the above assumption leads to the natural question whether the set of via nodes for entire regions is small and can be used to speed up the computation of an alternative path.

In the following paragraphs, we are interested in two things. First, we investigate how to find good candidate sets for via nodes between pairs of regions and second, how to actually extract good alternative paths from these candidates. Moreover, we would like sets that can be efficiently precomputed in a preprocessing step and for which the resulting alternative paths are approximately admissible and of good quality.

Single-Level Via Node Candidates. We split the generation of alternative paths into a preprocessing and a query phase. The preprocessing generates a small amount of additional data by *bootstrapping* that is exploited by the query phase. Bootstrapping implies that the query algorithm for computing alternative paths is used during preprocessing as well. Our working assumption is that the number of admissible alternative paths between any two nodes and likewise the number of such paths

between (connected) regions of the network is rather limited and that they can be represented by a very small set of via nodes. Hence, we partition the graph and apply bootstrapping to generate *via node candidate sets* for pairs of regions $\{R_1, R_2\}$, labelled $C(R_1, R_2)$. We assume the pairs to be ordered since shortest paths between two nodes and thus alternatives to them can be different in either direction.

Assume that for each pair of non-neighboring regions, we have already precomputed a set of via node candidates. Since candidates are already present, we do not need to identify them during an exploration phase. Computing an alternative path to a given shortest path $P_{s,t}$ becomes straight-forward. We iterate over all nodes v in the via node candidate set $C(R_s, R_t)$ of the pair of regions to which s and t belongs. For each v we check whether $P_{s,v,t}$ is approximately admissible according to Definition 5.3. The first approximately admissible path is returned as the result of the procedure in a greedy fashion. If there exists no appropriate node in the via node candidate set or the set is empty, we report this as a negative result. Figure 5.11(a) illustrates the query process.

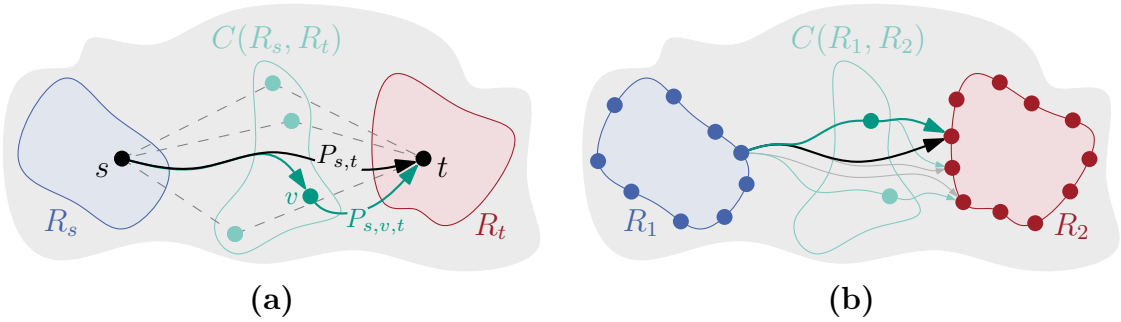


Figure 5.11: *(a) Single-level query between s and t. The regions of s and t are marked in blue and red, the associated via node candidate set $C(R_s, R_t)$ in green. The selected via node v and alternative path $P_{s,v,t}$ are highlighted. (b) Preprocessing. Each boundary node (blue) computes alternative paths to all boundary nodes of the other region (red). Processed shortest and alternative paths are shown as well as the already found via nodes (green). The currently considered path is highlighted.*

When considering a query between neighboring regions or within a single region, we perform X-CHASEV as a fallback. The reason for this procedure is our observation that the number of candidates is often too numerous in these cases. We observe average candidate set sizes of well above thirty for neighboring regions and even more within a single region. It is faster to use the baseline algorithm than to verify the pregenerated node sets in most of these instances. Figure 5.12(a) depicts an example. The same effect may even emerge between regions that are not adjacent if the average region size is chosen too small as non-neighboring regions remain close to each other.

Precomputing via node candidate sets starts with a partitioning of the underlying network. As the graph is already partitioned for CHASE, we can reuse this data. A set of via node candidates is generated greedily for each pair of regions by computing

alternative paths for all pairwise combinations of their boundary nodes. We store a tentative set of via node candidates for each pair of regions that keeps track of the candidates that have been identified thus far during preprocessing. To compute alternatives, we apply the above query algorithm with the tentative node set as bootstrapping. When an approximately admissible alternative is found over a node in this set, we continue with the next pair of boundary nodes. Otherwise, we run the baseline algorithm X-CHASEV to identify a new one. Whenever such a fallback query results in a new via node, it is added to the set of tentative via nodes. This continues until all pairs of boundary nodes have been considered. Figure 5.11(b) depicts a snapshot of a preprocessing run.

If we want to support multiple alternatives, we need separate via node candidate sets for each subsequent alternative. The query algorithm applies the set corresponding to the requested alternative with the overlap computation respecting all previous alternatives. During preprocessing, we compute the desired number of alternatives for each pair of boundary nodes. We may independently fall back to X-CHASEV for each one, e.g. we may require X-CHASEV to obtain a second alternative but still find a good one with the tentative candidate set for third alternatives. Whenever we find a new via node with our fallback query, we store it in the appropriate candidate set.

Multi-Level Via Node Candidates. So far, our algorithm does not compute via node candidate sets for neighboring pairs of regions and within a single region. We propose a second, fine partitioning to handle this shortcoming. Our network is partitioned into an order of magnitude more regions. The fine partitioning should respect the coarse one in the sense that all nodes of a fine region belong to exactly one of the coarse regions. While not strictly required, it reduces preprocessing times and memory requirements as discussed below. We denote the fine region to which node v belongs by M_v. Fine via node candidate sets are indicated by fine regions, i.e. $C(M_1, M_2)$. Figure 5.12(b) and 5.12(c) give examples of the expected number of via node candidates.

We do not run a full precomputation for all pairs of fine regions. Albeit technically feasible, this would induce a large number of additional alternative path computations, quadratic in the number of fine regions. This, in turn, would translate into roughly two orders of magnitude more work in practice. As our algorithm performs well for most pairs of coarse regions, we run the same preprocessing algorithm as before only on a subset of all pairs of fine regions. More precisely, we preprocess each non-neighboring pair of fine regions that either belongs to the same coarse region or to a pair of neighboring coarse regions, i.e. to those pairs of regions for which X-CHASEV is used during a single-level query. This results in a linear number of pairs of regions that have to be considered. If the fine partitioning does not respect the coarse one, preprocessing becomes more costly as more fine regions have to be considered for each coarse region, even some that only extend into the coarse region with a single node.

A query recurses to the fine partitioning for nodes in neighboring coarse regions

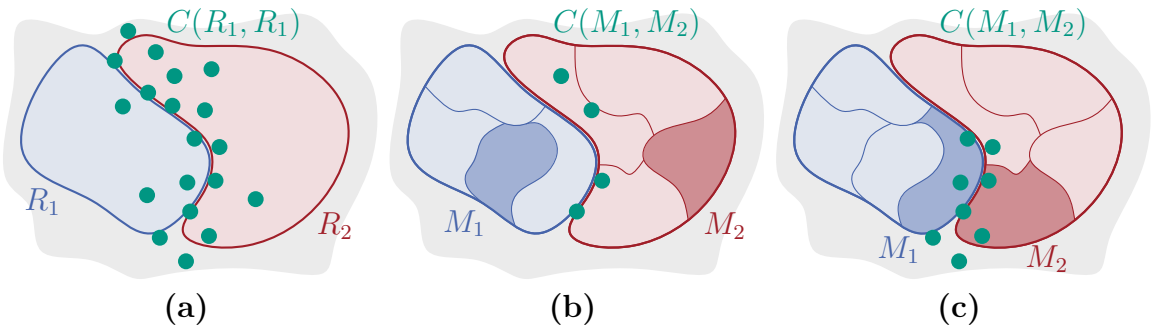


Figure 5.12: *(a) Coarse via node candidate set $C(R_1, R_2)$ between neighboring coarse regions R_1, R_2 is very large, while (b) fine via node candidate set $C(M_1, M_2)$ between fine regions M_1, M_2 is of a manageable size. (c) Fine via node candidate set $C(M_1, M_2)$ between neighboring fine regions M_1, M_2 is again very large.*

or within the same coarse region. When source and target are within the same or in neighboring fine regions, plain X-CHASEV is run as a fallback. As fine regions are much smaller, source and target are generally very close to each other. Thus, even the fallback query runs fast and poses no time penalty. In theory, though, it is possible that another level of partitioning is beneficial for very large graphs or regions. Algorithm 5.2 gives a more formal description of the multi-level query algorithm.

Algorithm 5.2 Multi-Level Query Algorithm with Candidate Sets

Input: Graph $G(V, E)$, source s, target t, via node candidate sets C, coarse regions R_s, R_t, fine regions M_s, M_t
Output: Approximately admissible alternative path $P_{s,v,t}$ ($\emptyset$, if none exists)

1: **if** $R_s \neq R_t$ and not neighboring **then** ▷ single-level query
2: $C' \leftarrow C(R_s, R_t)$
3: **else if** $M_s \neq M_t$ and not neighboring **then** ▷ multi-level query
4: $C' \leftarrow C(M_s, M_t)$
5: **else** ▷ fallback query
6: $C' \leftarrow ExplorationQuery(s, t)$ ▷ identify nodes in search space overlap
7: $C' \leftarrow filter(C')$ ▷ discard unsuitable nodes
8: $C' \leftarrow sort(C')$ ▷ rank nodes by $f(P_{s,v,t})$
9: **end if**
10: **for all** $v \in C'$ **do**
11: **if** $isApproxAdmissible(P_{s,v,t})$ **then** ▷ see Definition 5.3
12: **return** $P_{s,v,t}$ ▷ viable alternative found
13: **end if**
14: **end for**
15: **return** $\emptyset$ ▷ no viable alternative found

Further Engineering. After introducing our bootstrapping technique, we now turn to engineering its performance. There are many opportunities for further improvements of the preprocessing as well as the query algorithm. All subsequent ideas are applicable to both the single-level and multi-level variant of our approach.

We begin by considering extensions to our preprocessing step: The computation of via node candidate sets is easily adaptable to *shared-memory parallelism* as the processing of each pair of regions does not depend on the other pairs. This parallelization scales almost linearly with the number of processors until the memory bandwidth is saturated. It has no effect on the quality of the via node candidate sets as the selection of via nodes stays the same. We may further accelerate the preprocessing step by *sampling*. Most alternative path queries only verify the existence of a via node and do not result in a new node that is added to the tentative via node candidate set. Thus, a natural approach is to consider only a subset of all pairs of boundary nodes for each pair of regions instead of doing a full precomputation. Given a sampling rate s, we perform sampling by choosing $1/s$ boundary nodes of each region at random and computing alternatives only for this fraction of nodes. This effectively decreases the runtime of the preprocessing step while retaining the quality of the candidate sets if the sample rate is chosen reasonably. We further observe that a substantial amount of the computational effort during preprocessing is spent in search space exploration. Preprocessing times can be reduced by about a factor of three by storing forward and backward *search spaces of the boundary nodes* since they are required multiple times during preprocessing. However, the impact on memory consumption is not negligible. Preprocessing for each pair of regions involves a large number of target queries between the boundary nodes and the nodes in the tentative via candidate set. Thus, running the precomputation with a *many-to-many technique* like [KSS+07] seems to be reasonable. However, the computation takes about the same time as the above method when storing search spaces. When not expending memory for the storage of search spaces, though, the many-to-many technique is superior. We already argued that the sizes of the via node candidate sets are too large for queries between neighboring regions or within a single region. In principle, this may occur for any pair of regions, but it is more likely on dense graphs and between regions that are close to each other. To counter this issue, we may introduce an upper bound V_{max} for the *size of the via node candidate sets*. As soon as the preprocessing computes this many via nodes for a pair of regions, the processing of this pair is stopped and it is marked. The query considers marked pairs of regions as neighboring regions and always uses the fallback query.

We now move on to extensions to our query algorithm: Similar to the preprocessing step before, we can accelerate our query algorithm by storing forward and backward *search spaces of the via nodes in the candidate sets*. This effectively halves the duration of the target and exploration queries that involve these via nodes. As already mentioned before, we may further store shortcut edges pre-unpacked to reduce runtimes. Another tuning parameter is the *order* in which via node candidates are stored. We keep the (tentative) candidate sets ordered by the number of how often the node is used as a

via node during the preprocessing step. This order is not necessarily a best one as it heavily depends on the sequence in which the pairs of boundary nodes are processed. Fully sorting, i.e. computing some best among all possible orders, independent of the processing sequence, is technically feasible and leads to slightly superior runtimes for our query algorithm, but it is also computationally expensive to generate. Our query algorithm greedily chooses the first viable via node from the candidate set. We could, however, opt to select the via node that yields a *best quality* alternative. This is, of course, much more expensive as all nodes in the via node candidate set have to be examined. Note that the choice of the via node has a direct impact on whether viable subsequent alternatives can be found. Consider the example in Figure 5.13. The via node selected for the first alternative may yield a via path so that all possible choices of via nodes for a subsequent alternative overlap substantial parts of this path. Thus, none of them is admissible due to the overlap with the first alternative, but both of them would have been admissible by choosing a different first alternative.

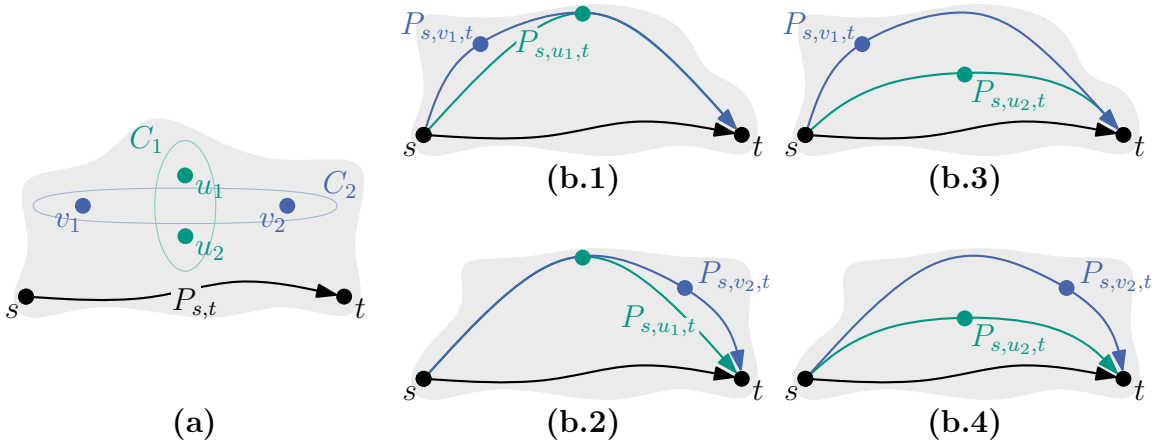


Figure 5.13: *(a) Shortest path $P_{s,t}$ with two sets of via node candidates, C_1 (green) for the first and C_2 (blue) for the second alternative. (b) Possible choices for the first and second alternative. As seen in Figure (b.1) and (b.2), choosing the upper via node candidate u_1 for the first alternative results in too much overlap. By choosing the lower via node candidate u_2 as in Figure (b.3) and (b.4), however, the first and second alternative path do not share any subpath.*

For both, preprocessing and query algorithm, we can exchange the underlying shortest path algorithms for our approximate variants apxCH and apxCHASE if the considered network is too dense. This decreases success rates as the search space of apxCH is much smaller than that of normal Contraction Hierarchies. However, as before, this can be amended by a relaxed query at only a small impact on runtimes. Our approximate algorithms work right out of the box, but one should make sure that negative distance differences are handled correctly, especially during relaxed queries that not only consider upward paths in either search direction.

Complexity. Preprocessing and query times of our approach depend on the baseline algorithm and the underlying shortest path algorithm. They determine the size of the via node candidate sets and the cost of shortest path queries. Moreover, the quality of graph partitioning has an impact on preprocessing. We denote the time complexity of the baseline algorithm by $T_{baseline}$ and the time complexity of the test for approximate admissibility by T_{check}. The running time of the latter is dominated by four point-to-point shortest path queries.

The preprocessing algorithm tries to compute an alternative route for each pair of boundary nodes for all region pairs. In the worst case, each of these queries finds a new via node after unsuccessfully testing all previous via node candidates for approximate admissibility. Consider the graph in Figure 5.14(a). Each boundary node of a region is connected to exactly one boundary node of the other region over a direct edge as well as over an intermediate node. Thus, assuming R regions and at most B boundary nodes per region, we obtain $T_{prepro} = \mathcal{O}\big(R^2B^2 \cdot (T_{baseline} + B^2 \cdot T_{check})\big)$ as time complexity of the preprocessing step. We may replace the factor B^2 by V_{max}, the maximum cardinality over all via node candidate sets. By applying sampling to consider only $\sqrt{B}$ of the boundary nodes of each region, we can subtract another factor of B from the running time. As seen in our respective simulations in Section 5.5.3, this has little impact on quality and even larger reductions remain practical.

The query algorithm applies a baseline algorithm if the regions of source and target node are neighboring or the same. This has the same time complexity as the baseline algorithm since the overhead for retrieving regions and candidate sets is constant. Otherwise, we have to check all via node candidates of the considered pair of regions for approximate admissibility in the worst case, e.g. if no viable alternative is found. Again, overhead due to accessing regions and candidate sets is negligible. This gives us the time complexity $T_{query} = \mathcal{O}(T_{baseline} + V_{max} \cdot T_{check})$ of the query algorithm.

Considering plain X-CHV as baseline algorithm for the exploration query, via node candidates can only be those nodes at which forward and backward search spaces meet. In the worst case, both search spaces are equal besides the root nodes as depicted in Figure 5.14(b). Thus, all nodes become via node candidates. The size of Contraction Hierarchies search spaces is bound by $\mathcal{O}(h \log D)$ according to Abraham et al. [ADF+13], with h the highway dimension h and D the diameter of the considered graph. This yields a *very rough* upper bound for the maximum size V_{max} of the via node candidate sets. In practice, we only see the square root of these values as usually only the borders of the search spaces touch and have to be considered. However, a tight analysis remains an open question.

Abraham et al. bound the time complexity of a Contraction Hierarchies query by $\mathcal{O}\big((h \log D)^2\big)$ [ADF+13]. The same holds for the time complexities of our target and exploration query when using plain CH and thus for $T_{baseline}$ and T_{check}. No theoretical analysis exists for Arc Flags and consequently neither for CHASE. The time complexity of Contraction Hierarchies can serve as a rough upper bound for CHASE, though, as the CHASE query degenerates to plain CH if we assume that all arc flags are set.

In conclusion, we obtain $T_{query} = \mathcal{O}\big((h \log D)^3\big)$ for 0-relaxed, single-level X-CHASEV queries and $T_{prepro} = \mathcal{O}\big((RB)^2 \cdot T_{query}\big)$ for the respective preprocessing. The multi-level variant has the same query time complexity, while preprocessing times are increased by the additional pairs of regions to be considered. We cannot reasonably assess search space sizes and query times for relaxed CH queries, though. Considering multiple alternatives introduces a constant factor to both running times.

Note that the cited complexities by Abraham et al. assume an optimal CH preprocessing, which is hard [BCK+10]. With a more realistic polynomial time preprocessing, each term $(h \log D)$ has to be replaced by $(h \log h \log D)$ [ADF+13].

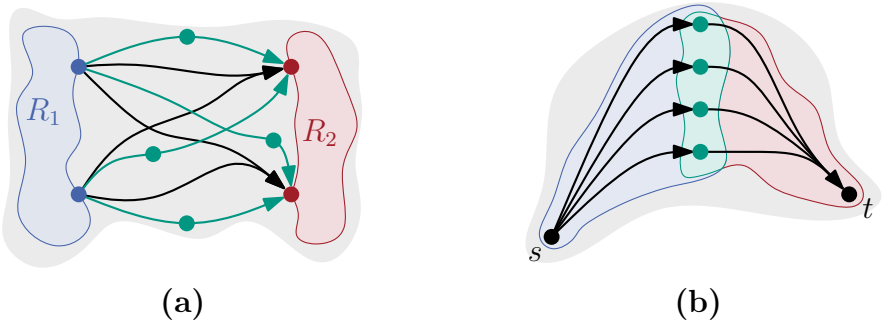


Figure 5.14: *(a) Worst-case via node candidate set size: Each pair of boundary nodes (blue, red) between region R_1 and R_2 induces a unique via node (green). (b) Worst-case CH search space overlap: Forward (blue) and backward (red) search spaces from s and to t overlap in all nodes but the root nodes (green). Recall that the query only follows edges to more important nodes.*

Correctness. To complete the description and analysis of our approach, we argue that its results are correct with respect to the underlying baseline algorithm.

If our query algorithm returns an alternative path, it is always approximately admissible according to Definition 5.3—just as the results of a baseline algorithm. This is due to the construction of our algorithm. If the regions to which source and target belong are neighboring or the same, the result of a baseline algorithm is returned which guarantees approximate admissibility. Otherwise, all nodes in the via node candidate set of the respective pair of regions are checked for approximate admissibility until one is found or all have been tested. This is done with the same routine as for the baseline algorithm. Thus, a reported alternative is guaranteed to be approximately admissible.

However, if our query algorithm fails to find an alternative, there is still a chance that the baseline algorithm would find one. This may happen as our preprocessing routine only computes alternatives between the boundary nodes of each pair of regions. As actual queries usually originate from within the regions, they are slightly longer compared to queries between boundary nodes, and thus the approximate admissibility criteria may become violated and via paths rejected in turn.

We can guarantee to find approximately admissible alternatives for all queries that the baseline algorithm does by applying it as a fallback to all queries that did not yield an alternative after testing all via node candidates. This is the same approach as during the learning phase of our online algorithm in the following section. Average runtimes increase since the baseline algorithm is much slower than our approach, but this is partially amortized as it is only needed in few queries. Moreover, our evaluation of the online algorithm in Section 5.5.3 shows that running this fallback query has virtually no (positive) impact on the success rate and average solution quality. Thus, it can be entirely omitted in the presence of preprocessed via node candidate sets.

5.4.3 Applications

We now move on to two applications that our approach can be naturally extended to. We consider an online setting in which our algorithm does not require explicit preprocessing of via node candidate sets and improves over time. We further discuss the efficient generation of alternative graphs from our candidate sets.

Online Algorithm. Our method can be easily adapted to an online setting. In this case, via node candidate sets are not computed in advance but instead learned on-the-fly from a stream of queries. The resulting online algorithm is well-suited to be added on top of a legacy system that already implements one of the algorithms of Section 5.4.1 or another method based on via nodes. The only other prerequisite is a partitioning of the graph of the considered network. We regard this requirement as a minor issue and largely orthogonal to the problem at hand. The online algorithm is further suitable for applications in which the network data changes on a regular basis. Under such circumstances, one would not want to spend much time on preprocessing additional data as its life-time is very limited. Our online algorithm is an extension to the bootstrapped preprocessing routine proposed in Section 5.4.2 and as such easy to implement, which we demonstrate now.

As mentioned above, the via node candidates are not precomputed but learned on-the-fly while answering queries for alternative paths. Thus, the candidate sets for each pair of regions are empty at first. When answering a query for an alternative path to $P_{s,t}$, we look up the via node candidate set $C(R_s, R_t)$ of the pair of regions to which s and t belongs. This set may still be empty or already filled with via nodes from previous queries. If one of the nodes in the set yields an approximately admissible alternative to $P_{s,t}$, we report the respective via path. Otherwise or if the set was empty, we use—similar to our preprocessing step—some baseline algorithm as a *fallback method* to compute an approximately admissible alternative. If this is successful, we store the via node in the respective candidate set. Thus, the fallback method gradually fills the via node candidate sets. When enough candidates have been learned from the stream of queries, no further fallback computations are performed. This decision is made independently for each candidate set. Our simulations in Section 5.5.3 show

that the sets saturate quickly for coarse via node candidates. About $t = 60$ queries for each pair of regions suffice to gather enough information to be competitive with respect to success rate and quality compared to explicit preprocessing. If we cannot find an approximately admissible alternative for a query at this point, we report this as a negative result. Multiple alternatives can be handled in the same way as before.

Alternative Graphs. Another application we want to highlight is the computation of alternative graphs. As briefly introduced among our related publications, the concept of an *alternative graph* aims to encode many alternatives at once. Due to their sparseness, these subgraphs of the considered network may be later processed efficiently by more expensive path finding algorithms. Our method is a natural extension to our via node candidate sets and aims at quickly providing alternative graphs in the absence of other dedicated methods or supplementing these techniques. The approach is simple and easy to implement as we show now.

For a given query between source s and target t, we first compute a shortest path $P_{s,t}$ and add it to our alternative graph, which is initially empty. Next, we consider the via node candidate set $C(R_s, R_t)$ of the regions to which s and t belong. Via paths are computed iteratively for all nodes in this set and added to the alternative graph unless they are longer than a certain threshold. We stop when the via node candidate set is exhausted or a maximum number of via paths have been added to the alternative graph. Figure 5.15 depicts an example graph.

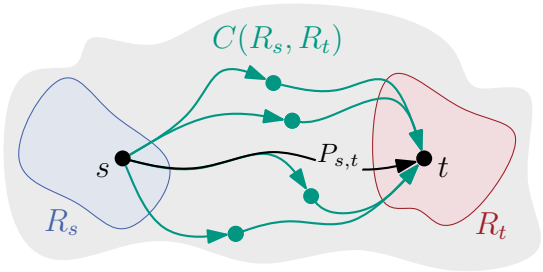


Figure 5.15: *Alternative graph between s and t constructed from via node candidate set $C(R_s, R_t)$. It consists of the shortest path $P_{s,t}$ (black) and four alternative paths over the via nodes in $C(R_s, R_t)$ (green).*

We distinguish between single-level alternative graphs that are generated only from our coarse via node candidate sets and multi-level alternative graphs that further consider the fine candidate sets. Alternative graphs for nodes in neighboring or within the same region only contain the shortest path as the respective candidate sets are not computed during preprocessing and thus remain empty.

Approximate admissibility is not verified for the added via paths as the established quality measure for alternative graphs in [BDGS11] does not consider it. If this or other properties are required, though, they can be easily verified on the sparse alternative

graph. However, we still check each via path $P_{s,v,t}$ for self-overlapping subpaths as in Figure 5.16(a) and remove them. Due to the construction of via paths such overlaps only occur around via node v. Thus, to remove these appendages, it suffices to traverse $P_{v,t}$ in forward and $P_{s,v}$ in backward direction until both paths diverge. All nodes but the last common node v' are pruned. $P_{s,v',t}$ remains a via path due to the subpath optimality of shortest paths and can be added to the alternative graph, see Figure 5.16(b) for the results.

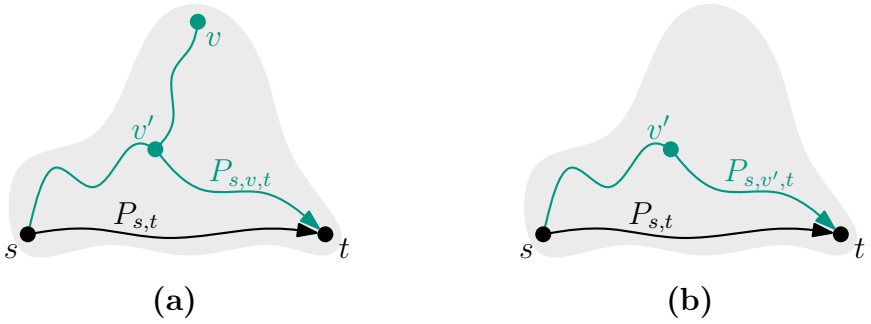


Figure 5.16: *(a) Subpath $P_{v',v,v'}$ of $P_{s,v,t}$ is not locally optimal and therefore pruned. (b) The remaining via path $P_{s,v',t}$ is added to the alternative graph.*

The computation can be accelerated by running a one-to-many search [KSS+07] from s to all nodes in the via node candidate set and a subsequent many-to-one search from these nodes to t. The cited publication applies a deprecated technique, Highway Hierarchies, but the general approach is directly applicable to Contraction Hierarchies and CHASE. Runtimes can be further improved by storing additional data. As suggest in Section 5.4 as an option for further engineering, we may store the forward and backward search spaces for all nodes of the via node candidate sets. All paths that are (possibly) added to an alternative graph by our approach could then be identified by a forward search from s and a backward search from t only. Pruning operations have little impact on runtimes as they only take time linear in the length of the appendage and can be performed while unpacking and reporting the path.

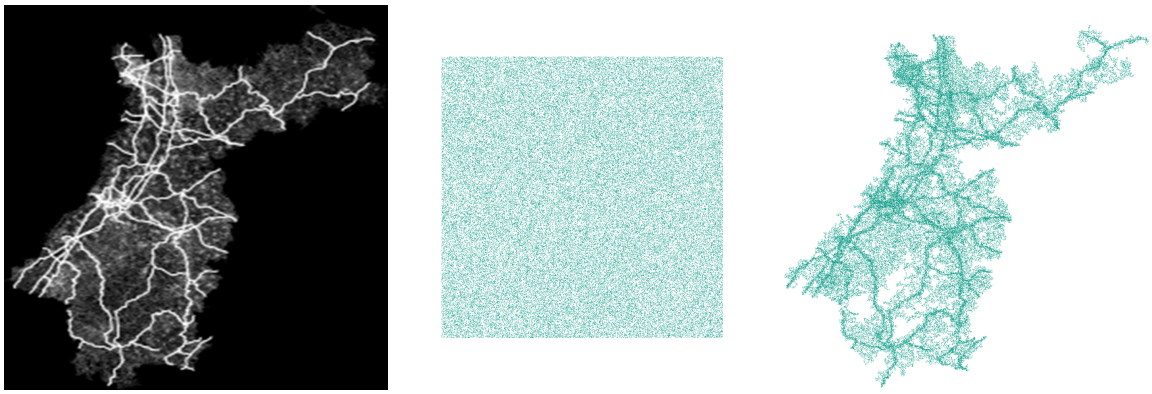
5.5 Simulations

We conclude the chapter by providing extensive simulations in which we study the computation of (approximate) shortest paths and reasonable alternatives to them. We compare the performance of our algorithms to previous techniques in various aspects. Most of our simulations use one core of Machine A. The preprocessing of Contraction Hierarchies and all instances of PHAST were performed in parallel on machine A, with PHAST also applying SSE instructions. Preprocessing for via node candidate sets was done in parallel on machine B.

5.5.1 Simulational Setup

Network Setting. We consider two distinct types of networks, unstructured random networks (*box*) and infrastructural networks that model roads and population densities in some sense (*road*). We generate the topologies of our simulated sensor networks by iteratively placing nodes on a squared area. Node positions are chosen uniformly at random. To model the above types, we consult a probability map to decide whether a node is actually placed. We stop after having placed 1 000 000 nodes. Edges are set so that the resulting graph is a unit disk graph (see Section 2.2.1), with distances scaled to obtain an average node degree d_{avg} within a tolerance of 1%. We assume bidirected edges. Edge costs are set to the Euclidean distance between nodes to the power of p. We extract the largest strongly connected component of this graph. Node positions are only used during graph generation. The algorithms in our simulations are only aware of the graph topology. The resulting graph sizes are found in Table C.2.

The probability map for network type "box" is uniform. The map used for type "road" is depicted in Figure 5.17(a). Lighter areas denote higher probability for node placement, and vice versa. The map is modelled after the OpenStreetMap[1] data for *Regierungsbezirk Karlsruhe*. Appendix C goes into more details on how we extract OpenStreetMap data and compute the probability map. Examples for network topologies using either network type are given in Figure 5.17(b) and 5.17(c).



(a) **(b)** **(c)**

Figure 5.17: *(a) Probability map for network type "road". Network topologies with* $d_{avg} = 10$ *for network type (b) "box" and (c) "road".*

We apply three different edge cost functions in our simulations. We use $p = 0$ to model hop counts, $p = 1$ for signal latencies, and $p = 2$ gives us the energy costs for free-space communication. Since the shortest path problem only gets easier to solve for larger values of p, see [BDS+10], we do not consider values above $p = 2$.

[1] `http://www.openstreetmap.org/`. Accessed: 2014-08-06.

We consider multiple distinct network settings in our simulations. Each setting is defined by the network type, the average node degree, and the power used for the edge costs. Our default setting is of network type "road" and uses $d_{avg} = 10$ and $p = 1$.

Measurement Procedure. All reported query measures are based on 10 000 random but fixed queries unless otherwise stated. We report average values. We present several results as plots against the Dijkstra rank, with each data point representing 1 000 queries. The *Dijkstra rank* of a query is the number of nodes Dijkstra's algorithm would settle. Statistics for the partitionings of each network setting, i.e. edge cut value, number of boundary nodes, and neighboring regions are listed in Table C.2.

Shortest Path Techniques. We measure the performance of shortest path techniques by the three values preprocessing time, query time, and memory overhead compared to a bidirectional Dijkstra's algorithm. For approximate techniques, we further state the average error. Usually, there is not a single best algorithm but several pareto-optimal ones providing different tradeoffs between the aforementioned measures. Note that the preprocessing time does not include the time required for partitioning the graphs as we can set an arbitrary time limit with little impact on the quality of the final partitioning as reasoned in Appendix C when discussing our network instances.

Alternative Path Techniques. In analogy to [ADGW13], we measure the performance of alternative path techniques in terms of efficiency and quality. Efficiency comprises query time and success rate. Quality is defined by the three measures uniformly bounded stretch, sharing, and detour-based local optimality of Definition 5.2. The time needed to compute these measures is not taken into account. Appendix C describes how to determine them efficiently for evaluation purposes.

For our approaches, we list further statistical values with respect to via node candidate sets. We state preprocessing time and memory overhead for the candidate sets and list their average size and the fraction of empty sets with respect to pairs of regions for which we perform preprocessing. Query statistics include the average number of tested via node candidates as well as the rates of fallback queries and queries that consider via node candidates. Details that only apply to the implementation of our online setting are described in the respective section of our simulational evaluation.

Considered Approaches. We compare the performance of our contributions, approximate Contraction Hierarchies and via node candidate sets for alternative paths, to multiple other approaches. We provide our own implementations of all algorithms according to the respective publication. Graphs are stored explicitly in main memory as adjacency arrays. Priority queues are based on binary heaps. Both data structures are described more closely in Appendix C. Shortest path techniques that are used in the evaluation of both approximate and alternative path queries apply the same settings and preprocessed data in either case. Partitioning is detailed in Appendix C.

Shortest Path Techniques. We consider bidirectional Dijkstra's algorithm [Dan63], Arc Flags [Lau04], and Core-ALT [BDS+10] as purely exact techniques and apply ALT [GH05], Contraction Hierarchies [GSSV12], and the combinations CHASE [BDS+10] and CHALT with and without approximation. ALT based algorithms use 64 landmarks computed with the avoid strategy. Parallel CH preprocessing is considered without lazy updates (nlu) and with only lazy updates, the default case. Nodes are ordered as per Equation (5.1) with $\alpha = 2$, $\beta = 4$, and $\gamma = 1$. Witness searches are pruned after 2 000 settled nodes (1 000 during simulation). Arc Flags based algorithms use a partitioning into 128 regions. The flags are computed in parallel with PHAST [DGNW13], 64 trees per sweep, and stored in a hash map. CHASE applies arc flags on the whole graph, while CHALT applies landmarks only on the 5% most important nodes. CH and CHALT use stall-on-demand. We do not reorder graphs for better cache-locality.

We do not consider further, more recent techniques for various reasons. Customizable Route Planning [DGPW13] requires a small number of boundary nodes to be efficient. However, this is not possible for dense sensor network instances. Hub Labels [ADGW11], on the other hand, should reflect the capabilities of Contraction Hierarchies on these instances, though at much shorter query times. However, the required preprocessing times and the memory overhead are likely much higher than for any of our considered techniques. We also do not consider distributed routing schemes as they are not aimed at quick computability in a centralized setting.

Alternative Path Techniques. We compare our approach with coarse and fine candidate sets to the algorithms X-BDV and X-CHV in [ADGW13] as well as to X-CHASEV. We apply the same parameter values as in the above publication. Minimum local optimality is set to $\alpha = 0.25$, maximum sharing to $\gamma = 0.8$, and maximum stretch to $\epsilon = 0.25$. X-BDV uses an explicit T-test for $p = 0$. CH based techniques are considered unrelaxed ($x = 0$) and 3-relaxed ($x = 3$). Shortcut edges are stored pre-unpacked in case of X-CHASEV. The coarse partitioning corresponds to the Arc Flags partitioning with 128 regions. The fine partitioning uses 1 024 regions and does not respect the coarse partitioning. During the preprocessing of via node candidates, search spaces are stored and reused. Sampling of boundary nodes is considered in a separate study. Further engineering techniques are not applied in our simulations.

We do not compare ourselves to other previous techniques for various reasons. CRP-π, the fastest approach based on the penalty method, is still two orders of magnitude slower than X-CHV. They focus on alternative graphs, though, and not on single alternative paths. HiDAR, which is based on the plateau method like our approach, is a more serious contender, but for few alternatives it remains slower than X-CHV. As our results supersede the algorithms by Abraham et al. in query times and as our focus is on at most three alternative paths, we do not consider CRP-π or HiDAR.

For an evaluation of our approach to generate alternative graphs, we refer to [LS14] for results on road networks. To summarize them briefly, we observe about 10% lower quality measures but up to two orders of magnitude shorter runtimes than even CRP-π.

5.5.2 Approximate Queries

We start the discussion of our simulational results on finding efficient paths in large-scale sensor networks by considering the performance of our heuristic shortest path algorithm, approximate Contraction Hierarchies. Table 5.1 lists results for all considered techniques in unstructured random networks (network type "box"). We focus on this network type at first and consider the latency cost model ($p = 1$) with average node degrees of $d_{avg} = 10$ and 20. Results for the other edge cost models together with this network type are listed in Appendix C. In the following, we only study different edge cost models in our default setting, infrastructural networks (network type "road") with an average node degree of $d_{avg} = 10$. The respective results are presented in Table 5.2. Further results for this network type are found in Appendix C.

Improved Node Ordering. We first focus on the modified node ordering process for the preprocessing phase of Contraction Hierarchies that we introduced in Section 5.3.

Table 5.1: *Performance of all considered (exact and approximate) algorithms using the network type "box" and the latency cost model ($p = 1$).*

		Prepro.		**Query**		**Prepro.**		**Query**	
		time [s]	overhead [B/n]	time [ms]	error [%]	time [s]	overhead [B/n]	time [ms]	error [%]
	algorithm	node degree $d_{avg} = 10$				node degree $d_{avg} = 20$			
	Bidir. Dijkstra	0	0	156.804	–	0	0	231.287	–
	Arc Flags	1 363	95	2.373	–	9 279	182	4.658	–
	Core-ALT	205	164	1.474	–	778	432	4.850	–
ALT	exact	215	512	2.985	–	268	512	4.756	–
ALT	apx ($\epsilon = 0.01$)	215	512	1.714	0.1	268	512	1.947	0.2
ALT	apx ($\epsilon = 0.10$)	215	512	1.310	1.0	268	512	1.441	1.2
ALT	apx ($\epsilon = 0.21$)	215	512	1.262	1.9	268	512	1.408	1.9
CH	exact (nlu)	5 415	−2	2.325	–	266 749	29	14.643	–
CH	exact	895	1	2.477	–	30 492	30	14.831	–
CH	apx ($\epsilon = 0.01$)	400	−5	2.238	0.2	8 967	−8	12.615	0.2
CH	apx ($\epsilon = 0.10$)	177	−19	2.182	2.2	1 277	−51	7.223	1.8
CHASE	exact	6 216	97	0.042	–	62 761	243	0.225	–
CHASE	apx ($\epsilon = 0.01$)	4 878	86	0.038	0.2	27 863	169	0.167	0.2
CHASE	apx ($\epsilon = 0.10$)	3 120	61	0.028	2.2	9 540	84	0.073	1.8
CHALT	exact	927	26	0.415	–	30 586	56	2.522	–
CHALT	apx ($\epsilon = 0.01$)	427	20	0.351	0.2	9 028	17	1.798	0.2
CHALT	apx ($\epsilon = 0.10$)	198	7	0.292	2.2	1 312	−25	0.843	1.8
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	198	7	0.124	3.6	1 312	−25	0.379	3.4

While the original (parallel) approach always updates the importance of all neighbors of a contracted node and performs no lazy updates (nlu), we *only* perform lazy updates. Comparing the preprocessing times of these two exact Contraction Hierarchies implementations, we observe that our approach is an order of magnitude faster for networks of $d_{avg} = 10$ and 20. Using only lazy updates effectively avoids a lot of (potentially) unnecessary updates of node importance values. However, the quality of the generated search graph suffers as the order of node contraction is likely less optimized for the same reason. We observe that the search graph gets slightly denser as reflected by the increased memory overhead. Query times increase likewise.

Consulting Table 5.2, we see that the preprocessing speed-up gained by using only lazy updates is most pronounced for the latency cost model ($p = 1$). For the other two models, we still see an improvement by a factor of 2 ($p = 2$) and, respectively, by at least 50% ($p = 0$). Once we have considered further results for all three cost models, we will see that the latency cost model is the most demanding model with respect to query times. Thus, by avoiding witness searches, which essentially are a variant of Dijkstra's algorithm, we gain the most benefits under the latency edge cost model. We discuss later why it is more demanding than the other two models.

Our approximate Contraction Hierarchies technique, which we consider next, already applies the modified node ordering process with only lazy updates by default.

Table 5.2: *Performance of all considered (exact and approximate) algorithms using the network type "road" with average node degree $d_{avg} = 10$ under different edge cost models ($p \in \{0, 1, 2\}$).*

		Prepro.		**Query**		**Prepro.**		**Query**		**Prepro.**		**Query**	
		[s]	[B/n]	[ms]	[%]	[s]	[B/n]	[ms]	[%]	[s]	[B/n]	[ms]	[%]
	algorithm	hop count ($p = 0$)				latency ($p = 1$)				energy consumption ($p = 2$)			
	Bidir. Dijkstra	0	0	118.149	–	0	0	141.151	–	0	0	148.387	–
	Arc Flags	500	92	2.720	–	522	92	0.835	–	477	91	1.022	–
	Core-ALT	237	170	5.249	–	239	170	2.702	–	239	175	4.089	–
ALT	exact	164	512	7.153	–	178	512	3.666	–	191	512	5.155	–
ALT	apx ($\epsilon = 0.01$)	164	512	6.523	0.2	178	512	3.047	0.0	191	512	4.578	0.0
ALT	apx ($\epsilon = 0.10$)	164	512	3.872	1.3	178	512	2.354	1.1	191	512	4.186	0.7
ALT	apx ($\epsilon = 0.21$)	164	512	3.218	2.5	178	512	2.342	2.5	191	512	4.174	2.5
CH	exact (nlu)	144	−36	0.170	–	1 071	−14	0.482	–	204	−33	0.109	–
CH	exact	92	−34	0.177	–	254	−10	0.440	–	113	−30	0.103	–
CH	apx ($\epsilon = 0.01$)	92	−34	0.183	0.0	166	−17	0.381	0.2	111	−31	0.102	0.1
CH	apx ($\epsilon = 0.10$)	94	−34	0.202	0.7	116	−29	0.322	2.1	111	−34	0.099	1.2
CHASE	exact	1 025	25	0.012	–	2 086	72	0.013	–	1 116	31	0.006	–
CHASE	apx ($\epsilon = 0.01$)	1 026	25	0.012	0.0	1 680	59	0.011	0.2	1 096	29	0.006	0.1
CHASE	apx ($\epsilon = 0.10$)	997	25	0.012	0.7	1 147	34	0.009	2.1	950	23	0.006	1.2
CHALT	exact	106	−8	0.120	–	275	16	0.244	–	126	−5	0.077	–
CHALT	apx ($\epsilon = 0.01$)	106	−8	0.121	0.0	185	9	0.213	0.2	124	−5	0.076	0.1
CHALT	apx ($\epsilon = 0.10$)	107	−8	0.138	0.7	131	−3	0.171	2.1	123	−8	0.074	1.2
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	107	−8	0.100	1.2	131	−3	0.109	2.8	123	−8	0.061	1.5

Approximate CH. By allowing a small one-sided error when answering queries, we are able to further decrease preprocessing times than with the modified node ordering process alone. Especially on the dense network, we observe a considerable speed-up by a factor of 30 when accepting an error of up to 10% ($\epsilon = 0.1$) in our queries. The actually measured error rates, however, are about an order of magnitude lower than the guaranteed maximum error. This already seems to be a good trade-off, but there are even more benefits. By not inserting certain shortcut edges into the search graph, it remains much sparser than the search graph of an exact CH. We even observe, again in Table 5.1 and for $d_{avg} = 20$, a large *negative* memory overhead. This is also reflected in shorter query times. For the sparse network, the reduction in query times is less pronounced, but the reduction in memory overhead is still significant.

At this point, we should briefly explain how a negative memory overhead can be achieved and what it implies. As mentioned in Section 5.5.1 when discussing our simulational setup, we measure memory overhead compared to a bidirectional Dijkstra's algorithm. This algorithm stores an edge at each of its nodes to facilitate the lookup of incoming and outgoing edges. In Contraction Hierarchies, however, we only need to store an edge at the node of less importance. Thus, if we add less shortcut edges than there are edges in the original graph, we achieve a negative memory overhead. Less memory consumption further translates into a better cache locality, which in turn improves query times.

When we consider the other edge cost models in Table 5.2, we only see a negligible improvement in any considered measure for $p = 2$ and even a decline for the hop count cost model ($p = 0$) that worsens with growing values of ϵ. This effect also holds true for the dense graphs of network type "road". In general, our approach is most effective on graphs with many similar paths of similar but not equal lengths. By allowing some error, we can omit a lot of shortcut edges, which in turn keeps the graph sparse during contraction and facilitates preprocessing. Paths in the latency cost model ($p = 1$) offer this property. Thus, the improvement between exact and approximate preprocessing is very pronounced. When considering energy consumption ($p = 2$), there are only few similar paths of similar cost as even small differences in the distances between nodes translate to large variations in the edge costs. In addition, we observe a more distinct hierarchy, with edges between close nodes being preferred, i.e. paths of many hops are more desirable. This is the exact opposite of the hop count cost model ($p = 0$), which tries to minimize the number of hops and therefore has a preference for edges between distant nodes. In this model, there are many equivalent (shortest) paths, which is already handled well by exact Contraction Hierarchies. Thus, graphs that apply either of these two edge cost models offer only few paths that our approximate algorithm can exploit. We therefore see no improvement when switching from the exact to our approximate algorithm. Exact Contraction Hierarchies already works well for these settings. The performance decrease for $p = 0$ is explained by an inferior search graph. If we omit a shortcut edge, we may later have to add further shortcut edges that would otherwise not be required due to witness paths that include the omitted shortcut edge.

In general, approximate Contraction Hierarchies can help to process graphs that are less hierarchically structured, and we should see a comparable performance to our network settings under the latency cost model. Moreover, our approach can be easily combined with other techniques that are based on Contraction Hierarchies. In addition to the already discussed benefits of our approach, this is particularly helpful if the other technique has significant costs related to the number of edges. As we reduce the number of shortcut edges, this can bring a significant improvement. In the following, we study two such examples, query algorithms that apply a goal-directed technique, either Arc Flags or ALT, on top of Contraction Hierarchies.

apxCHASE & apxCHALT. We are able to reduce the preprocessing times and the memory overhead of Contraction Hierarchies by using a modified node ordering and approximation. However, it remains questionable whether the general (hierarchical) technique is the right tool for our considered networks. When we regard common goal-directed techniques, namely Arc Flags and variants of ALT, in Table 5.1, we observe that they are already on par or even better than our approach with respect to query times, which is arguably the most important aspect for our desired application setting. There are some trade-offs, though. The ALT variants have a very high memory overhead and the preprocessing of Arc Flags takes long.

However, if we combine either of these techniques with (approximate) Contraction Hierarchies, we not only observe a huge improvement in query times by one to two orders of magnitude, the memory consumption also decreases compared to the pure goal-directed methods. While this is expected behavior from previous studies like [BDS+10], it shows that our technique is the best tool for this job given the right support. For example, apxCHALT with $\epsilon = 0.1$ dominates all other ALT-based methods for $d_{avg} = 10$. The average observed errors might be higher than for ALT with $\epsilon = 0.1$, but both approaches only guarantee the same maximum error. On dense networks, though, our preprocessing times remain higher. The average error rates of approximate Contraction Hierarchies, apxCHASE, and apxCHALT are equal for the same value of ϵ as approximation is only applied to the hierarchical component of the query. If we also allow approximation in the goal-directed component of our query, we obtain even higher speed-ups in query time at a slightly increased error rate. Consider apxCHALT with $\epsilon, \epsilon' = 0.1$. Its approximation guarantee corresponds to ALT with $\epsilon = 0.21$. The shortest query times are obtained by apxCHASE, though its preprocessing takes the most time, even when applying approximation. However, we observe that the preprocessing overhead compared to Contraction Hierarchies greatly decreases for higher values of ϵ as does the memory overhead. Since there are fewer edges in the approximate case, there are less flags (or indices to the corresponding lookup table) to store, but there are also less boundary nodes and therefore less shortest path trees to compute for Arc Flags. However, when only storing unique flags, the memory consumption could increase as there are more distinct flags in the approximate case.

However, in our simulations this is always compensated by the reduced number of edges. The beneficial effects for CHASE in the approximate setting are more pronounced on our dense networks as more edges are omitted.

If we consider the other edge cost models once more, Table 5.2 tells us that Contraction Hierarchies already performs well in these network settings, dominating all of the goal-directed techniques. In combination with them, though, we can further decrease query times. However, also applying approximation has almost no effect—similar as for plain Contraction Hierarchies. Only the preprocessing times for apxCHASE decrease by up to 10%. As stated before, our approximate technique does not have much to work with in these settings. There are simply few shortcut edges that can be omitted.

Our results show that the preprocessing times of CHASE are high in any network setting and remain so, even when applying approximation. However, when considering combinations of goal-directed techniques with (approximate) Contraction Hierarchies, we have the choice between two general approaches. We can either apply goal-direction to the whole graph or only to a core of important nodes. In our simulations, we choose to use ALT only on the core and Arc Flags on the whole graph to cover both options. If we would choose to only compute arc flags for the core, which actually conforms to the original implementation of the CHASE technique, we would see a reduction in preprocessing time and memory overhead by a large amount with only a small impact on query times as previously studied in [BDS+10].

To conclude our main results on approximate queries, we repeat that this technique works best in the presence of many similar paths of similar but not equal cost. Independent thereof, the modified node ordering seems to be beneficial for any dense network. We obtain vastly shorter preprocessing times at roughly the same query times. Finally, we want to stress that none of our techniques requires node positions, neither actual positions nor virtual coordinates. Unlike many other techniques that are explicitly designed with sensor networks in mind, only the communication graph of the considered sensor network is needed.

Further Network Models. We briefly considered further communication models and node distributions in our simulations. The main result of these trial studies is that, next to the average node degree, the distribution of node degrees has the most significant impact on measured runtimes. In our network settings that follow the unit disk graph model, we observe a normal distribution of node degrees with small variance. However, when using different and potentially more realistic communication models, the distribution may become heavy-tailed with a significant amount of nodes of high degree, even though the average node degree remains small. Such network instances are already more demanding to process. However, the average node degree remains a main indicator for the complexity of an instance. Overall, we can state that the more high degree nodes there are in the communication graph of a network, the more demanding this network instance is to be handled by one of our studied methods.

5.5.3 Alternative Connections

After discussing (heuristic) shortest path algorithms in the previous section, we now turn to the evaluation of our approaches for determining alternative connections to the best one. We focus on the default network setting (type "road", $d_{avg} = 10$, $p = 1$) for the majority of our studies. While we present results for the other two edge cost models, we do not consider network type "box" and networks of a higher average node degree. The reason behind this decision is that these network settings offer (too) many (approximately) admissible alternative paths according to our Definition 5.3. Thus, no elaborate technique is required to find them efficiently. A comprehensive compilation of the results of all considered network settings and all applied alternative path techniques and their variants is given in Appendix C.

As the results of this section are obtained on two different machines, we compare their respective performances for target and exploration queries in preliminary simulations. These results are listed in Table C.3. The corresponding section in Appendix C further discusses the search spaces sizes of the considered algorithms and explains why CHASE is not applicable for exploration queries.

Engineered Baseline Algorithm. We start our evaluation by comparing our engineered baseline algorithm, X-CHASEV, to two methods by Abraham et al. [ADGW13], X-BDV, which applies a bidirectional Dijkstra's algorithm for target and exploration queries, and X-CHV, which runs on top of Contraction Hierarchies. At first, we limit ourselves to exact query algorithms. Tables 5.3 and 5.4 report on the query performance and alternative path quality of these algorithms with normal exploration queries ($x = 0$) and relaxed ones ($x = 3$). As they do not use via node candidate sets, the respective

Table 5.3: *Query performance for finding alternative paths $a = \{1, 2, 3\}$ under the latency cost model ($p = 1$). Exact queries without relaxation ($x = 0$) are used.*

		Performance		**Path Quality**			**Candidate Sets**		
a	algorithm	time [ms]	success rate [%]	UBS [%]	sharing [%]	locality [%]	via.cand. rate [%]	fallback rate [%]	tested [#]
1	X-BDV	871.229	98.5	7.7	29.2	63.5	–	–	–
	X-CHV	7.354	91.7	5.3	40.1	67.1	–	–	–
	X-CHASEV	3.936	91.7	5.3	40.1	67.1	–	–	–
2	X-BDV	925.930	96.3	7.7	48.8	57.8	–	–	–
	X-CHV	17.994	86.8	5.6	52.8	60.6	–	–	–
	X-CHASEV	6.552	86.8	5.6	52.8	60.6	–	–	–
3	X-BDV	981.484	92.0	7.4	60.3	54.7	–	–	–
	X-CHV	35.115	81.8	5.8	57.6	58.3	–	–	–
	X-CHASEV	10.574	81.8	5.8	57.6	58.3	–	–	–

columns remain empty. Moreover, X-BDV has no relaxation by design.

As stated in Section 5.4.1, our engineered algorithm is faster than X-CHV by about a factor of two while providing alternative paths of the same quality. Table 5.3 confirms this statement. These results are rather expected, judging from the performance of the individual algorithmic components that we added on top of X-CHV. The speed-up even grows when computing second or third alternative paths as the fraction of fast target queries that can use CHASE increases. X-BDV offers the highest success rates but obviously also the longest query times by several orders of magnitude. However, the relative amount of additional work to determine further alternative paths is rather small compared to the CH-based algorithms whose runtimes double with each subsequent alternative path. Note that the listed query time for an alternative path a includes the time for computing any previous alternative path. The path quality measures are very similar for all algorithms and identical for X-CHV and X-CHASEV by design. The success rates of all three algorithms drop with each subsequent alternative path, though they remain high for X-BDV. They are measured relative to the actual number of queries for a first, second, or third alternative path and not relative to the total number of queries. By considering the rate of failed queries instead of the success rate, we realize that X-BDV actually performs much better than X-CHV/X-CHASEV. For example, whereas the former only fails to find a third alternative path in 8% of all queries that request one, the CH-based techniques do not provide one for over 18% of all queries. This is expected behavior as X-BDV encounters many more nodes in the overlapping search spaces of its exploration query, whereas CH preprocessing already prunes many suboptimal paths and, moreover, its search spaces only meet but do not overlap. Still, this high success rate comes at the cost of long query times. X-BDV essentially explores a number of nodes linear in the network size.

Table 5.4: *Query performance for finding alternative paths $a = \{1, 2, 3\}$ under the latency cost model ($p = 1$). Exact queries with relaxation ($x = 3$) are used.*

		Performance		**Path Quality**			**Candidate Sets**		
a	algorithm	time [ms]	success rate [%]	UBS [%]	sharing [%]	locality [%]	via.cand. rate [%]	fallback rate [%]	tested [#]
1	X-BDV	871.229	98.5	7.7	29.2	63.5	–	–	–
	X-CHV	16.151	96.7	6.7	32.3	63.6	–	–	–
	X-CHASEV	10.553	96.7	6.7	32.3	63.6	–	–	–
2	X-BDV	925.930	96.3	7.7	48.8	57.8	–	–	–
	X-CHV	32.375	92.1	6.7	49.8	58.8	–	–	–
	X-CHASEV	14.654	92.1	6.7	49.8	58.8	–	–	–
3	X-BDV	981.484	92.0	7.4	60.3	54.7	–	–	–
	X-CHV	57.828	86.4	6.7	59.0	56.4	–	–	–
	X-CHASEV	20.874	86.4	6.7	59.0	56.4	–	–	–

Table 5.5: *Query performance of X-CHASEV when applying apxCHASE for target queries and apxCH for exploration queries w/o relaxation ($x = 0$). The latency cost model ($p = 1$) is considered.*

		Performance		Path Quality			Candidate Sets		
a	X-CHASEV	time [ms]	success rate [%]	UBS [%]	sharing [%]	locality [%]	via.cand. rate [%]	fallback rate [%]	tested [#]
1	exact	3.936	91.7	5.3	40.1	67.1	–	–	–
	apx ($\epsilon = 0.01$)	2.853	89.9	5.3	39.8	63.9	–	–	–
	apx ($\epsilon = 0.10$)	1.485	86.9	5.9	40.7	61.0	–	–	–
2	exact	6.552	86.8	5.6	52.8	60.6	–	–	–
	apx ($\epsilon = 0.01$)	4.894	82.5	5.0	52.9	59.3	–	–	–
	apx ($\epsilon = 0.10$)	2.635	73.4	5.5	52.5	57.0	–	–	–
3	exact	10.574	81.8	5.8	57.6	58.3	–	–	–
	apx ($\epsilon = 0.01$)	7.838	78.1	5.1	58.6	56.5	–	–	–
	apx ($\epsilon = 0.10$)	4.247	67.7	5.6	57.9	55.1	–	–	–

The values in Table 5.4 for results with relaxed ($x = 3$) exploration queries indicate that relaxation improves the average path quality as well as the success rate of the CH-based methods. The latter uniformly increase by about 5–6% for determining the first through third alternative path. Query times increase by a factor of two to three, though, which is a direct result of the larger number of nodes that are settled during exploration, compare Table C.3. Relaxation is as a viable trade-off between query times and success rates since the number of queries for which we find no alternative path is halved. This reduces the gap between the success rates of X-CHV (and X-CHASE-V) and X-BDV to 2% per considered alternative path.

We focus on approximate shortest path techniques next. Table 5.5 reports on our findings when we switch to apxCHASE for target queries and to apxCH for exploration queries in X-CHASEV. In this case, it suffices to exchange the original search graph for the approximate one since neither CHASE nor CH use the stall-on-demand technique when performing exploration queries. Query times decrease much more than what is expected from our previous results of these algorithms (see query times in Table 5.2). For example, computing a first alternative path becomes about three times faster with $\epsilon = 0.1$ than with exact queries. This is easily explained, though, when we consult the reported success rates. While they only decrease slightly for the first alternative path, there is a large drop when looking for a second or even a third alternative path. As the approximate search graph is smaller than the one of exact Contraction Hierarchies (compare the respective memory overheads in Table 5.2), so are the search spaces in an exploration query. Thus, less via node candidates are encountered and potentially useful nodes are missed. However, by using relaxation, we can boost the success rates again and to even higher levels than for exact but unrelaxed X-CHASEV

queries for first and second alternative paths. This is reported among further results on approximate queries in Appendix C. The quality of the reported alternative paths remains similar to the exact technique since when a via node is selected, it is often the same as in the exact case.

Overall, our engineered baseline algorithm, X-CHASEV, already beats all previous approaches with respect to query times, especially when requesting second or third alternative paths, and by applying relaxation, its success rates come close to those of X-BDV while query times remain low. They can be further decreased by applying approximation, but this comes at cost of lower success rates. We need to keep in mind, though, that the underlying CHASE algorithm requires additional preprocessing and needs to store additional data. However, both issues are much less of a concern than some years ago, with better techniques now available for computing and storing arc flags. Still, the engineering applied to X-CHV remains rather obvious. The next paragraphs therefore report on our actual contribution, precomputed via node candidate sets that offer even more substantial improvements.

Efficient Queries With Via Node Candidate Sets. We now evaluate our approach with precomputed coarse and fine candidate sets, denoted by *single-level* and *multi-level* in the respective tables. As it runs on top of X-CHASEV, we mainly compare ourselves to this engineered baseline algorithm. Table 5.6 gives results without relaxation ($x = 0$). We observe that query times are an order of magnitude faster than with the engineered baseline algorithm. Even for determining three alternative paths, the average runtime stays well below 2 ms, which is more than practical. We further see that our approach even improves the success rate of the query while the quality of the returned alternative paths remains at a high level. A fallback to X-CHASEV is used in 4% of all queries,

Table 5.6: *Query performance for finding alternative paths $a = \{1, 2, 3\}$ under the latency cost model ($p = 1$). Exact queries without relaxation ($x = 0$) are used.*

		Performance		**Path Quality**			**Candidate Sets**		
a	algorithm	time [ms]	success rate [%]	UBS [%]	sharing [%]	locality [%]	via.cand. rate [%]	fallback rate [%]	tested [#]
1	X-CHASEV	3.936	91.7	5.3	40.1	67.1	–	–	–
	single-level	0.424	93.6	5.8	42.1	65.7	94.2	4.7	1.8
	multi-level	0.353	93.8	5.8	42.2	65.7	98.2	0.5	1.9
2	X-CHASEV	6.552	86.8	5.6	52.8	60.6	–	–	–
	single-level	0.928	88.7	6.0	52.4	61.4	94.3	4.3	2.6
	multi-level	0.817	88.9	6.0	52.4	61.5	98.1	0.4	2.7
3	X-CHASEV	10.574	81.8	5.8	57.6	58.3	–	–	–
	single-level	1.775	84.2	6.1	58.2	58.3	94.9	3.7	4.1
	multi-level	1.585	84.3	6.0	58.2	58.3	98.3	0.3	4.2

which is not surprising as it only depends on the partitioning of the graph. This rate is reduced by an order of magnitude when introducing the multi-level partitioning. We are therefore able to consult precomputed via node candidate sets for over 99% of all considered queries. Thus, we briefly tried to omit the fallback query entirely while using the multi-level approach and observed that neither the quality measures nor the success rate degraded noticeably. We conclude that a third partitioning level does not give any significant improvement to the performance of the query in our settings.

The quality of the applied candidate sets is high. This is indicated by the small absolute number of via node candidates that are tested per query. On average, we need to test less than two nodes to obtain an (approximately) admissible alternative path. About three suffice for a second alternative path and four for the third one. This further implies that the via node candidates cover entire sets of alternative paths and not just individual ones. The quality of the candidate sets is further testified by the alternative path quality that remains on the same level as for X-CHASEV.

The results for using relaxation ($x = 3$) are given in Table 5.7. We observe a further improvement in success rates by up to 5% for the single- and multi-level approach. While the query times of the former increase by half, the multi-level approach still shows the same query time as without relaxation. Since the multi-level approach only performs X-CHASEV as a fallback for 0.5% of all queries, the increased cost of a relaxed exploration query is insignificant. The path quality measures become slightly worse compared to the baseline algorithm. In particular, we observe a higher sharing amount for the first and second alternative path. However, these measures are still better than those of X-CHASEV without using relaxation. Due to a larger choice of nodes when exploring relaxed search spaces, nodes are selected for inclusion in the via node candidate sets that yield alternative paths of higher quality.

Table 5.7: *Query performance for finding alternative paths $a = \{1, 2, 3\}$ under the latency cost model ($p = 1$). Exact queries with relaxation ($x = 3$) are used.*

		Performance		**Path Quality**			**Candidate Sets**		
		time	success	UBS	sharing	locality	via.cand.	fallback	tested
a	algorithm	[ms]	rate [%]	[%]	[%]	[%]	rate [%]	rate [%]	[#]
1	X-CHASEV	10.553	96.7	6.7	32.3	63.6	–	–	–
	single-level	0.659	96.8	6.6	38.1	63.8	95.0	4.7	1.8
	multi-level	0.361	96.8	6.6	38.6	63.9	99.2	0.5	1.9
2	X-CHASEV	14.654	92.1	6.7	49.8	58.8	–	–	–
	single-level	1.431	93.5	6.6	53.0	59.9	94.9	4.7	2.6
	multi-level	0.820	93.7	6.5	53.2	60.1	99.1	0.5	2.6
3	X-CHASEV	20.874	86.4	6.7	59.0	56.4	–	–	–
	single-level	2.637	88.8	6.7	60.0	57.1	95.3	4.5	3.9
	multi-level	1.531	89.0	6.7	59.9	57.0	99.3	0.4	4.0

Table 5.8: *Query performance under different edge cost models ($p \in \{0, 1, 2\}$). Exact queries without relaxation ($x = 0$) are used.*

		Performance		**Path Quality**			**Candidate Sets**		
p	algorithm	time [ms]	success rate [%]	UBS [%]	sharing [%]	locality [%]	via.cand. rate [%]	fallback rate [%]	tested [#]
0	X-BDV	769.008	100.0	–	33.2	44.0	–	–	–
	X-CHASEV	0.862	89.4	3.2	40.9	69.8	–	–	–
	single-level	0.195	91.7	2.8	43.5	69.6	93.7	4.7	1.6
	multi-level	0.178	91.9	2.8	43.6	69.6	97.7	0.5	1.6
1	X-BDV	871.229	98.5	7.7	29.2	63.5	–	–	–
	X-CHASEV	3.936	91.7	5.3	40.1	67.1	–	–	–
	single-level	0.424	93.6	5.8	42.1	65.7	94.2	4.7	1.8
	multi-level	0.353	93.8	5.8	42.2	65.7	98.2	0.5	1.9
2	X-BDV	1 018.658	94.8	9.6	38.8	71.0	–	–	–
	X-CHASEV	0.895	72.3	7.9	38.9	73.5	–	–	–
	single-level	0.298	76.8	8.3	43.5	72.9	88.2	4.7	1.7
	multi-level	0.282	77.1	8.3	43.7	72.8	92.1	0.5	1.8

Using precomputed candidate sets is faster by an order of magnitude than the engineered baseline algorithm and faster by far than the original method X-BDV. We identify two reasons for this result. First, a rather expensive (relaxed) exploration query has to be performed only in the rare case when a fallback is needed. Second, the number of via node candidates that have to be tested is small on average. Our approach also delivers consistently higher success rates than the engineered baseline algorithm on which it is based. This result is surprising at first, yet easy to explain. The nodes in each candidate set are chosen from the union of the search spaces of all boundary nodes of the corresponding pair of regions. This union of search spaces covers a much larger part of the graph than any search space of a single exploration query. Thus, the obtained via node candidates are more diverse and can (potentially) cover more alternative paths.

Next, we consider the impact of different edge cost models. Table 5.8 lists the respective results. We observe that the query times remain comparable between all models. The relative improvement between X-CHASEV and our approach is most pronounced for the latency cost model ($p = 1$), though. When we go back to Table 5.2 and consider the average runtimes of Contraction Hierarchies and CHASE under these models, the observed behavior becomes obvious. While the latter performs similarly for all models, CH queries are about four times slower for $p = 1$. Thus, X-CHASEV is also slower by a similar amount under this model. As our approach only applies target queries with CHASE (apart from a few fallback queries), there is no significant difference in query times between the different edge cost models. Considering success

rates, we observe a significant decrease under the energy consumption cost model ($p = 2$). While X-BDV yields slightly lower success rates, those of the CH-based techniques take a large hit. Consulting Table 5.2 once more, we infer from the memory overheads that the CH search graphs are very small for $p \in \{0, 2\}$ and therefore so must be the search space of an exploration query. Thus, we expect the success rate in either model to be low. While this is true for $p = 2$, the hop count cost model offers the highest success rate of all edge cost models. This is easily explained since there is an abundance of equivalent paths when edge costs are uniform as for $p = 0$. Finally, we want to remark on the missing stretch value for X-BDV in the hop count cost model. They were erroneous at a value of around 500%. We therefore opted to omit them.

We now briefly report on approximate queries in combination with our single- and multi-level approach. The respective numerical values are listed in Appendix C among our comprehensive results. We observe only a slight decrease in query time, on the same level as when comparing apxCHASE to CHASE in Table 5.2. These values are reasonable as our algorithm with precomputed candidate sets only performs target queries unless a fallback to X-CHASEV is needed. This only happens rarely, though, with source and target nodes in neighboring regions or within the same region. While the results with coarse via node candidate sets offer relatively high success rates close to those without approximation, the success rates decline substantially for the multi-level approach. We are not completely sure about the reason for this effect, but we assume that it can be entirely explained by poor fine candidate sets. Queries between distant regions remain the same as in the single-level setting. Only queries between neighboring coarse regions or within a single coarse region now consider (fine) candidate sets instead of performing a fallback query with X-CHASEV. Thus, X-CHASEV apparently provides much higher success rates for short-range queries than our fine candidate sets when using approximation. Concluding the discussion on approximate queries, we want to remark that they can also apply the via node candidate sets computed by exact methods, and vice versa. We did not consider this option further, though.

So far, we only considered average results over random queries. However, we also want to gain a better understanding of the performance of our approach with respect to the distance of a query. Figure 5.18 presents success rates for finding a first alternative path with and without relaxation for varying Dijkstra ranks. We opt to use the energy consumption cost model ($p = 2$) as the differences between our approach and X-CHASEV are more pronounced and therefore easier to study in this setting. Considering the results without relaxation ($x = 0$), we see consistently equal or higher success rates for our approach with coarse or fine candidate sets than for X-CHASEV. The success rates of X-BDV are given for reference as they mark a "gold-standard". They are obviously much higher than for the other approaches. However, with relaxation ($x = 3$) enabled, all curves shift much closer to each other and to the reference values of X-BDV. Our approach shows a clear advantage for long-range queries, starting at a Dijkstra rank of 2^{14} for a single-level of partitioning and, respectively, at 2^{11} for the multi-level approach. These numbers roughly correspond to the size of two regions

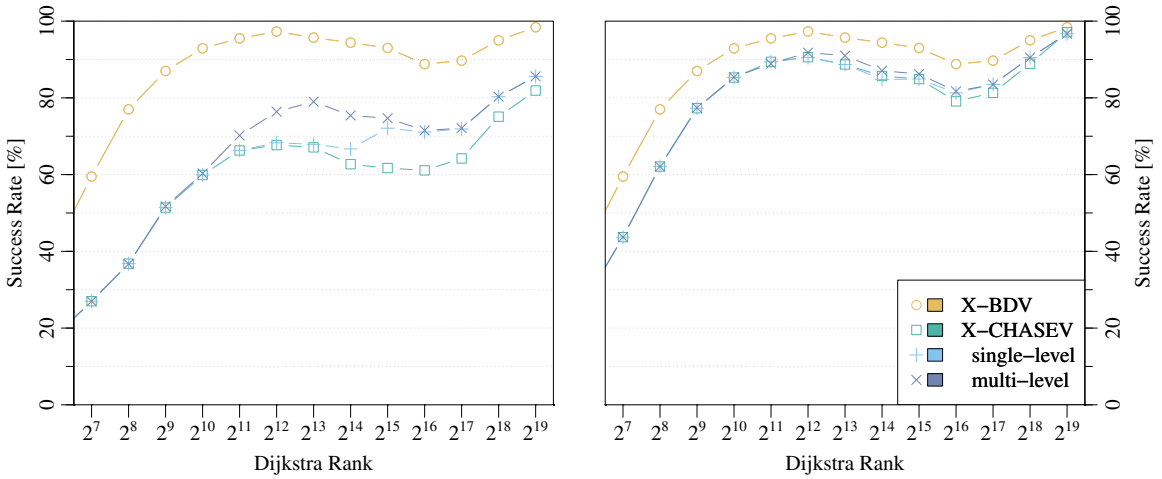


Figure 5.18: *Success rates against Dijkstra rank for the energy consumption cost model ($p = 2$) and exact queries w/o ($x = 0$), left, and with relaxation ($x = 3$), right.*

in the respective partitioning. This fits well to the fact that our approach does not consider via node candidate sets for queries between neighboring regions or within a single region. In general, the success rates rise with the distance between source and target with a dent at a Dijkstra rank of 2^{17} that is most likely caused by some peculiarities of the topology of our network. We infer that the further apart two nodes are in a network, the more likely it is to find an (approximately) admissible alternative path to the shortest path between them.

In summary, we can state that our approach with precomputed via node candidates offers a considerable boost to query times while increasing success rates at the same time. When using relaxation, we are almost on par with X-BDV with respect to success rates while our queries are over three orders of magnitude faster. Moreover, our results hold true for all considered edge cost models. It remains to be seen, though, how expensive the required preprocessing becomes. We consider this question next.

Preprocessing Via Node Candidate Sets. Having thoroughly discussed the query algorithm, we now report on the efficiency of the preprocessing that is required for both our single- and multi-level approaches. Table 5.9 lists results for the latency cost model with normal ($x = 0$) and relaxed ($x = 3$) exploration queries, while Table 5.10 gives additional results for $x = 0$ and the other edge cost models. We do not report on the preprocessing with approximate CH variants as their performance is virtually the same as when using exact algorithms. Results for all combinations of network settings and applied techniques are found in Appendix C.

In Table 5.9, we observe that preprocessing takes a substantial amount of time, especially when computing the via node candidate sets of a multi-level partitioning. The reported preprocessing time and memory usage of the latter even have to be

Table 5.9: *Preprocessing performance under the latency cost model ($p = 1$). Exact queries without ($x = 0$) and with ($x = 3$) relaxation are considered.*

		Performance		**Candidate Sets**					
				$a = 1$		$a = 2$		$a = 3$	
		time	size	empty	size	empty	size	empty	size
x	type	[h]	[kiB]	[%]	[#]	[%]	[#]	[%]	[#]
0	single-level	6.0	1 795	1.2	6.2	4.1	9.4	8.8	13.0
	+ multi-level	15.7	11 755	2.2	9.8	5.4	14.9	9.5	20.3
3	single-level	4.4	2 009	0.3	6.8	1.2	10.4	3.1	14.8
	+ multi-level	15.2	13 907	0.2	11.0	1.0	17.3	2.8	24.9

considered on top of the ones required for the coarse candidate sets. The following paragraphs therefore consider methods to reduce the preprocessing time or to avoid preprocessing entirely while affecting the quality of the computed sets as little as possible. Focusing on the other reported values for now, we see that the multi-level preprocessing yields a higher average number of via node candidates per pair of regions. This is to be expected as multi-level preprocessing only considers pairs of regions that are close to each other, which naturally have more distinct alternative paths—this is the very reason why we introduced a multi-level approach for neighboring coarse regions in the first place. The fraction of empty candidate sets, i.e. of pairs of regions for which we could not identify a single via node, grows with each subsequent alternative path since it is more difficult to identify reasonable second or third alternative paths. Recall that this value only includes pairs of regions for which we actually try to find via nodes, it does not include neighboring pairs of regions for instance. Our observation holds true for both single-level and multi-level preprocessing, but the actual numbers are higher for the latter. There are less other regions to consider per region, and therefore each single empty candidate set has a greater impact on the statistics. The memory requirements of our approach are well below 20 MiB, even with fine candidate sets. Hence, we conclude that the memory overhead of our approach is more or less irrelevant on current systems. The relative speed-up that we see on machine B scales linearly with the number of cores. However, we experienced hitting the memory bandwidth on some older machines, which limits scalability.

Comparing the results of using a normal exploration query to using a relaxed one, we observe an actual decrease in preprocessing time for the latter. This is surprising at first since a relaxed query settles a substantial amount of nodes more than an unrelaxed one, compare Table C.3. However, if we find better via nodes that are suitable for more pairs of region boundary nodes or any via node at all, there are less (fallback) queries to perform in total, which directly impacts the time required for preprocessing. The reduced number of empty candidate sets supports the latter assumption. Moreover, the average number of via node candidates stored in each set increases. This implies

Table 5.10: *Preprocessing performance under different edge cost models ($p \in \{0, 1, 2\}$). Exact queries without relaxation ($x = 0$) are used.*

		Performance		**Candidate Sets**					
				$a = 1$		$a = 2$		$a = 3$	
		time	size	empty	size	empty	size	empty	size
p	type	[h]	[kiB]	[%]	[#]	[%]	[#]	[%]	[#]
0	single-level	1.0	1 223	1.9	4.6	6.4	6.5	12.3	8.4
	+ multi-level	2.1	6 808	3.7	6.1	8.7	8.7	13.4	11.2
1	single-level	6.0	1 795	1.2	6.2	4.1	9.4	8.8	13.0
	+ multi-level	15.7	11 755	2.2	9.8	5.4	14.9	9.5	20.3
2	single-level	1.4	883	6.9	3.9	16.6	4.9	28.5	5.3
	+ multi-level	2.4	5 891	5.9	5.9	14.0	7.8	23.1	8.8

that we can now support alternative paths between nodes for which we did not find a reasonable via node before without using relaxation.

When we move on to different edge cost models, compare Table 5.10, we observe a similar situation as before. The latency cost model is by far the most demanding one to preprocess. The two other models require almost an order of magnitude less time for preprocessing. Our other observations are generally true for them, though. Interestingly, they both show a higher amount of empty sets, even substantially so in case of $p = 2$. Considering the performance of the baseline algorithm in Table 5.8, we see its performance reflected in the results of the preprocessing. Finally, note that the size of the candidate sets is lower by about a third for the hop count cost model ($p = 1$) than for the latency cost model ($p = 2$), even though the success rates are not that different. This implies that the set of reasonable alternative paths between two regions is easier to cover in the former model, i.e. a single node is a good intermediate stop for many alternative paths.

Overall, we conclude that while preprocessing can be substantial in case of the latency cost model, the gained benefits in query time and even in success rates more than compensate for this single shortcoming. Moreover, the following two paragraphs show that we can reduce this deficiency or get rid of it entirely.

Sampled Preprocessing. As mentioned in Section 5.4.2, sampling is a viable option to reduce the required amount of preprocessing. We present exemplary results for a single-level of partitioning without relaxation ($x = 0$) and focus on the latency cost model ($p = 1$) with exact queries as this is the most demanding setting with the highest amount of preprocessing time. We choose to skip either every second ($s = 2$) or every fourth ($s = 4$) boundary node. As expected, the preprocessing time decreases quadratically with s since we only consider this fraction of pairs of boundary nodes. Table 5.11 reports on our preprocessing results in more detail.

Table 5.11: *Preprocessing performance with sampling of boundary nodes. Single level candidates are computed. No relaxation ($x = 0$) is applied.*

		Performance		**Candidate Sets**					
				$a = 1$		$a = 2$		$a = 3$	
		time	size	empty	size	empty	size	empty	size
s	type	[h]	[kiB]	[%]	[#]	[%]	[#]	[%]	[#]
1	single-level	6.0	1 795	1.2	6.2	4.1	9.4	8.8	13.0
2	single-level	1.6	1 466	1.4	5.3	4.5	7.7	9.5	10.3
4	single-level	0.4	1 133	1.6	4.4	5.1	6.0	11.0	7.6

We further consider the impact of the sampled results on the performance of our queries. Table 5.12 shows the respective results. We observe that sampling yields via node candidate sets that offer only slightly inferior success rates compared to the results with full preprocessing. In particular, they remain better than those of our baseline algorithm for $s = 2$ and on par for $s = 4$. The quality of the alternative paths remains the same as without sampling. It is reasonable to assume that similar results hold for the candidate sets of a multi-level partitioning and for other network settings.

Overall, this approach introduces a space/time/quality trade-off. By sampling, the preprocessing is obviously much faster and, as seen in the above results, it requires less memory, but memory usage was already very low to begin with. However, we pay for these improvements with slightly inferior success rates. We think that this is a reasonable trade-off, though, especially for the latency cost model ($p = 1$). The online algorithm in the next paragraph is another viable option when trying to minimize the cost of preprocessing via node candidate sets.

Table 5.12: *Query performance with candidate sets from sampled preprocessing. Coarse candidates are used without relaxation ($x = 0$).*

		Performance		**Path Quality**			**Candidate Sets**		
		time	success	UBS	sharing	locality	via. cand.	fallback	tested
a	s	[ms]	rate [%]	avg. [%]	avg. [%]	avg. [%]	rate[%]	rate [%]	avg. [#]
1	1	0.424	93.6	5.8	42.1	65.7	94.2	4.7	1.8
	2	0.433	92.5	5.9	41.8	65.7	94.0	4.7	1.9
	4	0.431	90.2	5.9	41.9	65.5	93.9	4.7	1.8
2	1	0.928	88.7	6.0	52.4	61.4	94.3	4.3	2.6
	2	0.925	87.9	6.0	52.4	61.3	94.1	4.3	2.5
	4	0.916	85.9	6.0	51.9	61.4	93.6	4.4	2.4
3	1	1.775	84.2	6.1	58.2	58.3	94.9	3.7	4.1
	2	1.708	83.1	6.0	57.9	58.6	94.6	3.8	3.7
	4	1.646	80.7	6.0	57.3	58.3	93.8	4.0	3.3

Online Algorithm. We conclude the discussion of our approach for computing alternative paths by considering the online setting introduced in Section 5.4.3. We only present results for the single-level variant of our approach with no relaxation ($x = 0$) as the other variants show a similar behavior. Moreover, we focus on the results for the latency cost model ($p = 1$) and exact Contraction Hierarchies ($\epsilon = 0.0$) as this is the most demanding setting. Results for all possible combinations of p and ϵ are listed in Appendix C.

Candidate sets are learned from a stream of queries instead of explicitly preprocessing the network. We simulate a stream of 3 000 000 random queries. To demonstrate the behavior of the underlying baseline algorithm, X-CHASEV, we only perform fallback computations at first. No learning is involved at this point, and thus no via nodes are recorded. Only after 500 000 queries have been processed, we activate the learning phase. This phase continues until the via node candidate sets become saturated. We call a set saturated once $t = 60$ queries have been performed for the corresponding pair of regions. This threshold value is determined heuristically as we discuss below. Once it is reached, no further fallback computations are performed for that pair of regions.

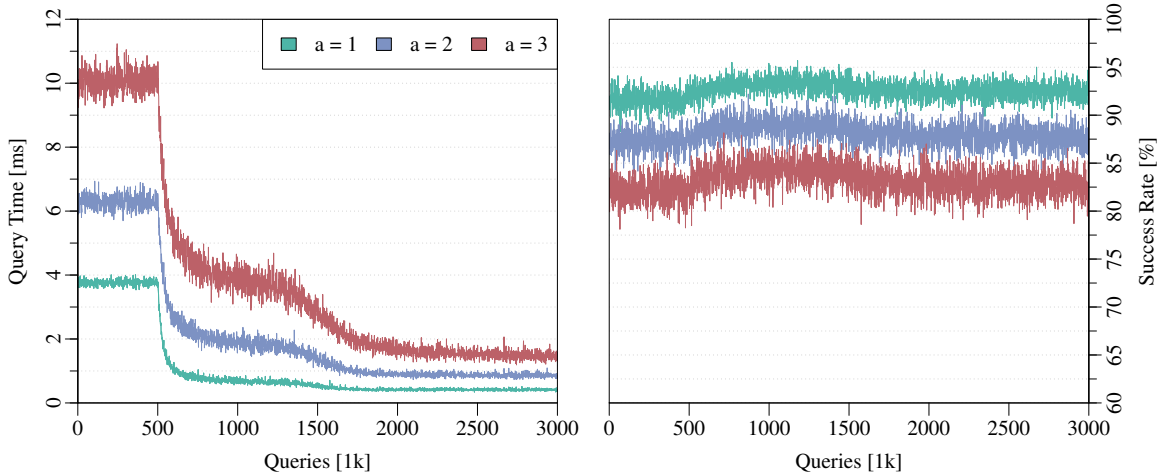


Figure 5.19: *Query time and success rate of our approach in the online setting. Exact queries ($\epsilon = 0.0$) and the latency cost model ($p = 1$) are used.*

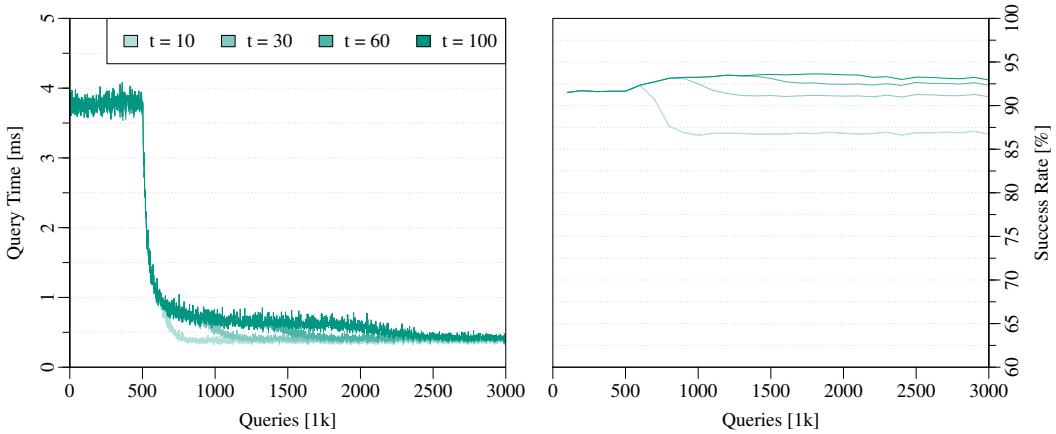
Figure 5.19 shows the development of the query time and success rate over the course of a simulation run. Values are averaged over 1 000 queries. We observe a particularly steep decline in query time during the first 100 000 queries after starting to learn via nodes. Since the number of via nodes that are required to sufficiently cover the alternative paths between two regions is small on average (see Table 5.9), we only need to learn few nodes to obtain competitive query times. Recall that we have 128 regions, and thus there are less than ten queries per pair of regions during this timeframe. The success rate likewise increases after starting to learn via nodes, but the effect is much less significant. It quickly reaches the same level as the single-level approach

with precomputed via node candidate sets. Thereafter, the query time remains stable with some variance. After roughly one million queries, less fallback computations are performed since the candidate sets start to become saturated. This induces the second decline in query time. However, we see that the success rate is virtually unaffected. After full saturation, query time and success rate are on about the same level as with explicit preprocessing. Considering the second and the third alternative path, we observe the same general behavior. The average query time rises and the average success rate decreases for each subsequent alternative path, which is in accordance to the behavior with explicit preprocessing.

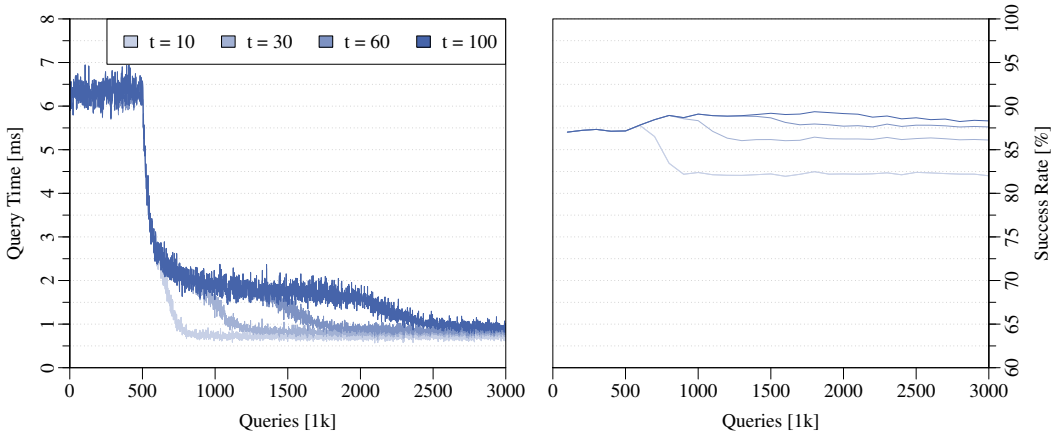
We determine a best threshold value for declaring a via node candidate set to be saturated in multiple simulation runs. Figure 5.20 shows the results of these simulations for the first through third alternative path. Query times on the left are averaged over 1 000 queries, while success rates on right correspond to 100 000 queries for clarity. We consider threshold values $t \in \{10, 40, 60, 100\}$. As before, we start learning only after the 500 000th query.

We observe a similar behavior for all alternative paths and therefore focus on the results of the first one in Figure 5.20(a) for now. The query time drops rapidly in the beginning for all values of t and shows a second decline after the via node candidate sets start to become saturated. It is more drawn-out, the higher the threshold value becomes as it obviously takes longer to saturate sets. Once fallback computations are no longer performed, query times are almost the same for any value of t, even though the stored sets are larger for higher threshold values as can be inferred from the higher success rates. This is due to the number of tested nodes being small on average, even with explicit preprocessing, compare Table 5.6. In fact, the query time after full saturation is on par with that of the single-level approach when using explicit preprocessing. We further observe that the success rates are higher for larger threshold values, but their relative differences get smaller. When comparing them to the results with precomputed candidate sets from above (see Table 5.6), we observe that they are on par. Surprisingly, we see that the success rates rise at first, only to decline later over the course of the simulation run. This effect is more pronounced for smaller threshold values, especially for $t = 10$. While this seems odd at first, a closer examination shows that the decline happens when the via node candidate sets start to become saturated. Thus, the reason for this behavior is that the candidate sets are prematurely saturated and the stored via nodes do not suffice for best success rates once the fallback method is no longer performed. For $t = 10$, the success rates even drop far below those of the baseline algorithm, i.e. of the fallback method.

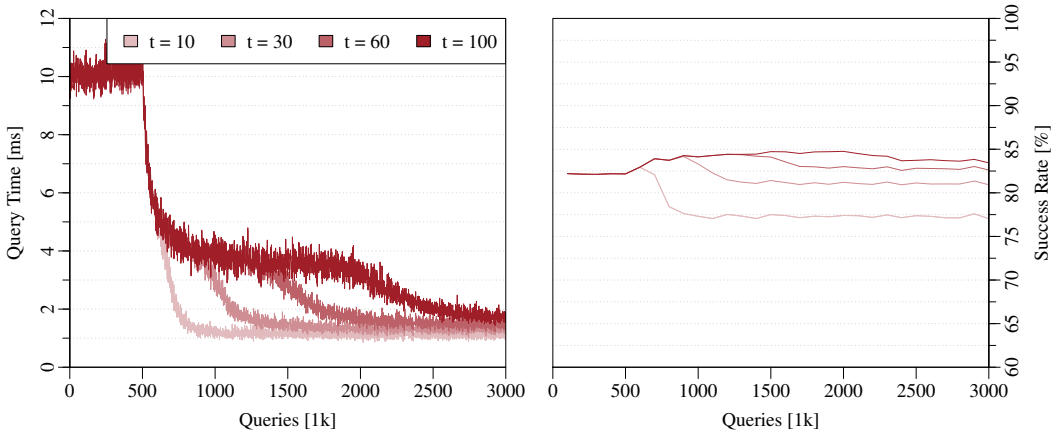
Our analysis of the saturation threshold for the first alternative path ($a = 1$) is generally true for the second and third one as well, see Figure 5.20(b) and 5.20(c). The most obvious differences are a higher query time and a lower success rate. This is expected behavior, though, as already seen with explicit preprocessing. Another difference is that the second decline in query time happens much later and is less pronounced. These "tails" are more drawn-out since second and third alternative paths



(a) *Query time and success rates for the first ($a = 1$) alternative.*



(b) *Query time and success rates for the second ($a = 2$) alternative.*



(c) *Query time and success rates for the third ($a = 3$) alternative.*

Figure 5.20: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Exact queries ($\epsilon = 0.0$) and the latency cost model ($p = 1$) are used.*

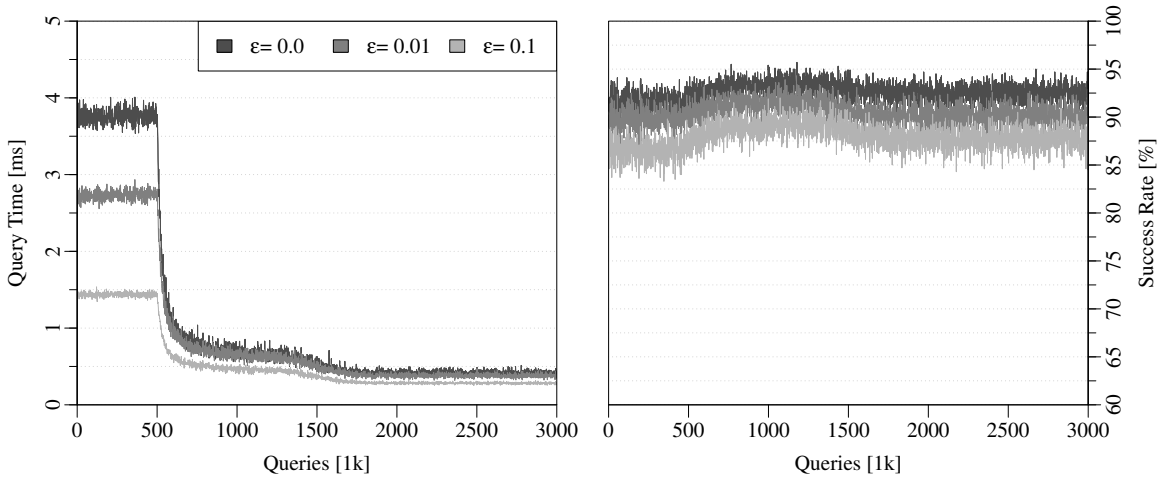


Figure 5.21: *Query time and success rate of our approach in the online setting for the first detour ($a = 1$). Exact and approximate queries are used with the latency cost model ($p = 1$).*

are requested less often—only when there is a first alternative path, searching for a second one makes sense. Thus, it takes more queries to reach the saturation threshold of the respective via node candidate sets. By comparing all results, we determine that a saturation threshold of $t = 60$ is a good trade-off between the final success rate and the required number of queries to saturate each via node candidate set.

The results for approximate Contraction Hierarchies with $\epsilon \in \{0.01, 0.1\}$ and other edge cost models, $p \in \{0, 2\}$, show a similar behavior and reflect the results that we already discussed for precomputed candidate sets. The differences in query time before starting to learn via nodes and after full saturation of the via node candidate sets are less pronounced, though, which is expected behavior when consulting the previous results. As we see in Figure 5.21 for the first alternative ($a = 1$), using approximate queries reduces the query time by almost a factor of three, while the success rate only decreases by roughly 5%, both with respect to $\epsilon = 0.1$. Moving on to Figure 5.22, again showing results for $a = 1$, it is obvious that the latency cost model ($p = 1$) is the most demanding one, but it also yields the highest success rates. The other two edge cost models offer roughly equal query times in the beginning, but after the learning starts, the hop count cost model ($p = 0$) is clearly easier to process. Moreover, it always offers higher success rates than the energy consumption cost model ($p = 2$). There are many equivalent paths in the hop count cost model. Thus, an (approximately) admissible alternative path is easy to find in this model, and only few via node candidates are required for each pair of regions. This explains the higher success rates and the shorter query times as less via nodes candidates implies that less of them have to be tested. Our reasoning is supported by the numerical results for precomputed candidate sets in Table 5.8. Results for all combinations of ϵ, p, and a are given in Appendix C.

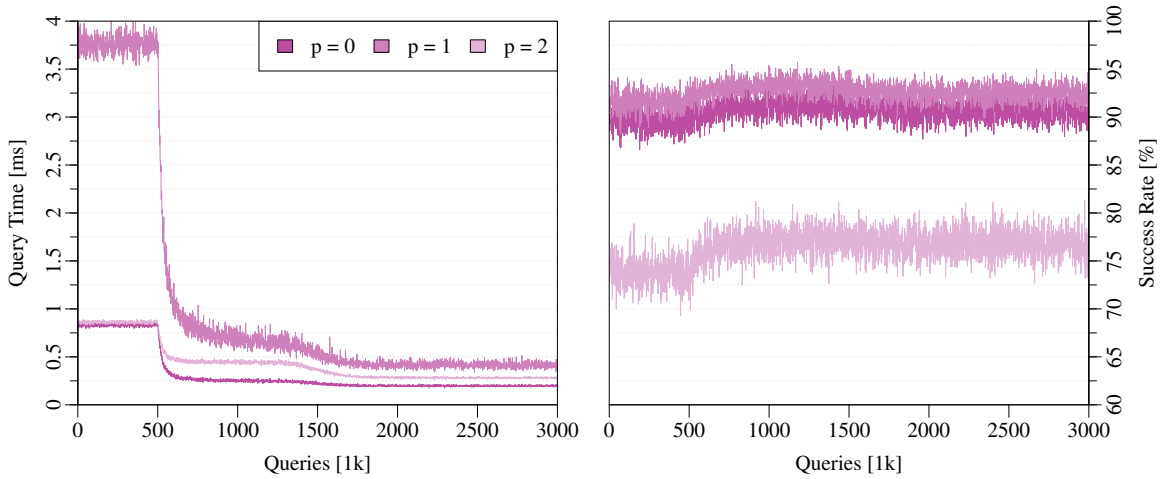


Figure 5.22: *Query time and success rate of our approach in the online setting for the first detour ($a = 1$). Exact queries ($\epsilon = 0.0$) and multiple edge cost models are used.*

We can even extract an observation from these simulations that justifies a choice that we made for our approach with preprocessed candidate sets. Once our online algorithm has found near optimal candidate sets compared to explicit preprocessing, performing a fallback query after not finding an alternative path has no significant impact on success rates. This reasoning is supported by comparing our results for $t = 100$ shortly before the second decline in query times starts to those after full saturation. We can therefore safely omit such fallback queries even in our general approach and report a negative result if a via node candidate set is empty or none of its nodes yields an (approximately) admissible alternative path.

Overall, the online algorithm offers an interesting means to completely avoid any (potentially high) preprocessing times compared to the sampling approach that only reduces preprocessing times. We can directly start to serve queries once some baseline algorithm is available and improve the query performance over time. Moreover, we could even opt to forget via nodes that are rarely used or that yield alternative paths of poor quality and reenable learning periodically.

5.6 Concluding Remarks

We presented two main results. The first contribution is an approximation scheme for Contraction Hierarchies. We showed that our approach guarantees a maximum stretch of $(1 + \epsilon)$ for the length of the computed paths, with even lower values seen in practice. By not adding selected shortcut edges to the CH search graph, we were able to save a substantial amount of runtime during both preprocessing and queries, which

is also reflected in a reduced memory overhead. This modification and changes to the node ordering process made our approach capable of handling dense sensor networks efficiently. We further decreased runtimes with Arc Flags and ALT, showing that combinations of techniques profit even more from the reduced size of the CH search graph. Moreover, we saw that other techniques based on Contraction Hierarchies like the many-to-many query in [KSS+07] or PHAST [DGNW13] can be directly used with approximate CH search graphs without any modifications as long as they only consider information of up-down paths.

The second contribution is centered around our assumption that there are few reasonable alternative paths between two distinct regions of a network and that they can be covered by few nodes. We showed that these via node candidates can be efficiently precomputed to facilitate alternative path queries. Our method is more than one order of magnitude faster and offers higher success rates than the previous results of [ADGW13]. It can be combined with our results on approximate queries to further improve runtimes on dense sensor networks. In addition, we showed how to adapt our algorithm for the generation of alternative graphs and to an online setting that does not need a dedicated preprocessing of via node candidates.

Both contributions are accompanied by an extensive simulational analysis. We compared their performance to other techniques in diverse sensor network settings and explored the impact of various parameters. Our simulations showed that our approaches are efficient and outperform the competition in all considered settings. For alternative paths, they further showed that the set of via node candidates is compact. This supports our assumption that few via nodes suffice in practice.

Outlook. Our contributions only consider centralized offline algorithms. It remains an open problem, though, how to effectively translate previous results on transportation networks to a distributed setting. A first step could be a semi-distributed variant of Contraction Hierarchies that uses centralized preprocessing. Queries would only use local information and shortcut edges would be stored distributed over all nodes covered by the edge. Nodes would relay a query to their direct neighbors and over shortcut edges. Arc flags could be used to effectively confine the search space, i.e. the nodes that are involved in the query process.

Considering approximate shortest path queries in general, it would be worthwhile to study whether similar techniques can be applied to other recent approaches like Customizable Route Planning or Hub Labels. Our own contraction process currently avoids adding shortcut edges in a greedy fashion. Using smarter approaches may further decrease the number of necessary shortcut edges and, in turn, shorten preprocessing and query times even more.

Our preprocessing routine for alternative paths determines via node candidate sets in a greedy fashion. A more sophisticated approach might be able to obtain smaller sets of similar quality. Generating via node candidate sets in reasonable time and of the same

or higher quality to what X-BDV yields is another open problem. Exploring theoretical guarantees for our method could help to assess our prospects in this direction. While we see high quality candidate sets in practice, we would like to capture the reason for this result in a theoretical analysis. A measurement related to highway dimension [ADF+13] might be the next step towards this goal.

Chapter 6
6 Discussion

The end. A final part of something, especially a period of time, an activity, or a story.

— Oxford Dictionary of English

In the future, ever growing sensor networks will emerge with thousands to even millions of nodes, be it networks in the conventional sense, established through initiatives in the line of the Smart Dust project [KKP99] and its successors, or networks that are part of the infrastructure and thus most likely connected to a steady power supply and a reliable communication infrastructure. We need strategies to efficiently deal with these vast structures, and we need them now, before they can ever become a problem. Our thesis approached the scalability issues that arise with large-scale sensor networks and showed various approaches to handle them in three diverse scenarios.

In Chapter 3, we introduced an efficient polynomial-time approximation scheme for the sensor network lifetime problem and also showed how to solve it optimally for small to medium-sized instances. These results are best used in auxiliary tools to study the capabilities of a sensor network. We can determine tight upper bounds on the maximal duration the sensor network is able to take data in a designated area. This can be further used to assess the quality of distributed algorithms for the same task. Moreover, if the network structure does not change and node failures are unlikely, we can even apply the computed schedules on an actual sensor network. This approach may also serve as a starting point for designing distributed algorithms as detailed in Section 3.7. The two main lessons to learn from this chapter, though, are to exploit the *redundancy* found in sensor networks and to consider *(efficient) approximation algorithms* for problems that are otherwise hard to solve. This holds true for both algorithms directly applied on sensor networks as well as for algorithms used in auxiliary applications that only have to run on classical systems.

Chapter 4 discussed a novel approach for determining boundaries and holes in a sensor network. Here, our solution is clearly aimed at applications running on sensor

networks. Each sensor node decides independently, only based on local connectivity information whether it is an interior node or on the fringes of the network. The suggested algorithm is agnostic to the network structure and gives remarkably good results for any network instance despite its simplicity. It can be further used to identify regions of sparse coverage and broader bands around network borders for applications like our scheduling algorithm. Our solution exemplifies several important design goals for algorithms that run on sensor networks: They need to be *simple* as the processing power of each node is limited. To compensate for this shortcoming, though, they can exploit the sheer number of sensor nodes and be massively *distributed*. However, the communication volume has to be kept low due to energy constraints. Thus, algorithms working with *local data* are to be preferred. Moreover, they should make *no assumptions* on the network structure to be generally applicable. We need to keep these guidelines in mind when designing efficient algorithms for sensor networks.

Finally, in Chapter 5, we considered algorithms for relaying messages on provably efficient routes through the sensor network. As these results are mainly in support of simulation frameworks, the respective algorithms are designed with execution on classical systems in mind. They have to be very efficient since there can be a lot of queries to answer with only little parallelism to exploit compared to an actual sensor network. We therefore proposed a fully polynomial-time approximation scheme for finding shortest paths. It is further used as a building block for determining good alternatives to an optimal routing. Both algorithms can rely on a preprocessing step as we assume the network not to change during a simulation run. Our solutions can be further used as a basis for other analysis tools, e.g. to assess bottlenecks in the network when all communication is done over the alleged best routes, or to compute equilibria if we can assess the communication volume as done in [LS11] for road networks. They are even applicable in actual, static settings such as expansive infrastructural networks. The lessons to take away in this case are that *algorithm engineering* can help to minimize the shortcomings of a hardware platform and that the *preprocessing* of information can be an efficient tool in accelerating repetitive tasks.

All of our results are naturally just examples of what can be done to counter an increase in network size with more sophisticated algorithms. Nonetheless, we highlighted three distinct fields of application in this thesis and made progress in all of them with diverse solution strategies. We named general techniques and guidelines to deal with ever growing sensor networks in applications that run directly on them as well as for algorithms that are part of auxiliary applications and executed on classical systems. As the previous chapters already provide detailed summaries and discussions on future work of our problems, we want to close this thesis with a more general résumé.

Outlook. Sensor networks are here to stay. They will become part of our daily life in one form or another if they not already have. For example, ad-hoc networks between mobile phones or other novelty gadgets can be regarded as one form of sensor networks

as can infrastructural networks erected alongside roads and highways. With growing network sizes and easier availability new and challenging problems will arise. Thus, research in sensor networks will remain highly relevant for the foreseeable future.

Our efforts must grow in multiple directions, though. Algorithms that run on sensor networks naturally have to remain a main focus, but we also need to consider algorithms that are part of auxiliary tools for network analysis and simulation. They run on classical systems and come with their own set of challenges. Orthogonally to this, research has to further expand in both theoretical and experimental directions, but also the middle ground, algorithm engineering for this new kind of systems, has to be considered and given sufficient attention.

An important prospect for the future is the availability of actual, large-scale sensor networks. We can do research in simulation frameworks and with theoretical models, and while this is certainly relevant and necessary, it can only give us so much insight. Ideally, there is a feedback loop between design, analysis, implementation, and experiments. At the moment, we do not have any options to close this loop, though, as the installed hardware platforms are just too small to require any sophisticated algorithms. However, this should be remedied over time.

Finally, we want to mention one aspect that we found especially frustrating when doing research in the field of sensor networks. It is the fragmentation of the community. There are many places to go to when doing purely theoretical research and even more when working with testbeds or simulators. However, when one is in the middle ground, doing algorithm engineering on sensor networks, it is very difficult to get accepted by either community. The one side wants proofs that cannot be given for a general model while the other side wants real experiments that cannot be done in the required scale. One is often relegated to general algorithm engineering conferences, and thus many interesting results are missed by the sensor network community.

Bibliography

[ABBC07] Arianna Alfieri, Andrea Bianco, Paolo Brandimarte, and Carla-Fabiana Chiasserini. Maximizing System Lifetime in Wireless Sensor Networks. *European Journal of Operational Research*, 181(1):390–402, 2007. [see pages 22, 24, and 43]

[ABCC07] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem*. Princeton University Press, 2007. [see page 48]

[ADF+13] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension and Provably Efficient Shortest Path Algorithms. Technical Report MSR-TR-2013-91, Microsoft Research, 2013. [see pages 121, 122, 138, 155, 156, and 185]

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *International Symposium on Experimental Algorithms (SEA'11)*, *LNCS*, vol. 6630, pp. 230–241. Springer, 2011. [see pages 121 and 162]

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *European Symposium on Algorithms (ESA'12)*, *LNCS*, vol. 7501, pp. 24–35. Springer, 2012. [see page 121]

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013. [see pages ix, 125, 126, 145, 146, 147, 149, 161, 162, 168, 184, and 250]

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *Symposium on Discrete Algorithms (SODA'10)*, pp. 782–793. SIAM, 2010. [see page 149]

[AGGM06] Ittai Abraham, Cyril Gavoille, Andrew V. Goldberg, and Dahlia Malkhi. Routing in Networks with Low Doubling Dimension. In *International Conference on Distributed Computing Systems (ICDCS'06)*, pp. 75:1–75:10. IEEE, 2006. [see page 123]

[AIKK14] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. Fast Shortest-Path Distance Queries on Road Networks by Pruned Highway Labeling. In *Meeting on Algorithm Engineering and Experiments (ALENEX'13)*, pp. 147–154. SIAM, 2014. [see page 121]

[AKK04] Jamal N. Al-Karaki and Ahmed E. Kamal. Routing Techniques in Wireless Sensor Networks: A Survey. *IEEE Wireless Communications*, 11(6):6–28, 2004. [see page 124]

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. Transit Node Routing Reconsidered. In *International Symposium on Experimental Algorithms (SEA'13)*, *LNCS*, vol. 7933, pp. 55–66. Springer, 2013. [see page 121]

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, first edition, 1993. [see page 248]

[AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990. [see page 248]

[Ans99] Kurt M. Anstreicher. Linear Programming in $\mathcal{O}(\frac{n^3}{\ln n}L)$ Operations. *SIAM Journal on Optimization*, 9(4):803–812, 1999. [see page 12]

[APM05] Ian F. Akyildiz, Dario Pompili, and Tommaso Melodia. Underwater Acoustic Sensor Networks: Research Challenges. *Ad Hoc Networks*, 3(3):257–279, 2005. [see page 17]

[BCK+10] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing Speed-Up Techniques Is Hard. In *International Conference on Algorithms and Complexity (CIAC'10)*, *LNCS*, vol. 6078, pp. 359–370. Springer, 2010. [see pages 122, 134, and 156]

[BCSZ04] Piotr Berman, Gruia Calinescu, Chintan Shah, and Alexander Zelikovsky. Power Efficient Monitoring Management in Sensor Networks. In *Wireless*

Communications and Networking Conference (WCNC'04), vol. 4, pp. 2329–2334. IEEE, 2004. [see page 21]

[BCSZ05] Piotr Berman, Gruia Calinescu, Chintan Shah, and Alexander Zelikovsky. Efficient Energy Management in Sensor Networks. In *Ad Hoc and Sensor Networks*, *Wireless Networks and Mobile Computing*, vol. 2, pp. 71–90. Nova Science Publishers, 2005.
[see pages 19, 21, 22, 23, 27, 31, 39, 42, 44, 45, 50, 61, 213, and 221]

[BD09] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, 2009. [see page 121]

[BDG+14] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Dorothea Wagner, and Renato F. Werneck. Route Planing in Transportation Networks. Technical Report MSR-TR-2014-4, Microsoft Research, 2014. [see page 119]

[BDGS11] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative Route Graphs in Road Networks. In *International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11)*, *LNCS*, vol. 6595, pp. 21–32. Springer, 2011.
[see pages 125, 158, and 250]

[BDGW10] Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner. Space-Efficient SHARC-Routing. In *International Symposium on Experimental Algorithms (SEA'10)*, *LNCS*, vol. 6049, pp. 47–58. Springer, 2010. [see page 132]

[BDS+08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In *International Workshop on Experimental Algorithms (WEA'08)*, *LNCS*, vol. 5038, pp. 303–318. Springer, 2008. [see page 123]

[BDS+10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, 2010.
[see pages 119, 121, 122, 136, 143, 160, 162, 166, and 167]

[BFMS11] Rene Beier, Stefan Funke, Domagoj Matijević, and Peter Sanders. Energy-Efficient Paths in Radio Networks. *Algorithmica*, 61(2):298–319, 2011. [see page 123]

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007. [see page 121]

[BG97] Ingwer Borg and Patrick Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer Series in Statistics. Springer, first edition, 1997. [see page 13]

[BGSV13] Gernot V. Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, 2013. [see page ix]

[BMS+13] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. Preprint available at arXiv:1311.3144 [cs.DS], Clemson University, Karlsruhe Institute of Technology, and Lawrence Berkeley National Laboratory, 2013. [see page 132]

[BT97] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, first edition, 1997. [see pages 12 and 44]

[BTG+06] Kun Bi, Kun Tu, Naijie Gu, Wan Lin Dong, and Xiaohu Liu. Topological Hole Detection in Sensor Networks with Cooperative Neighbors. In *International Conference on Systems and Networks Communication (ICSNC'06)*, pp. 31–35. IEEE, 2006. [see pages 69 and 92]

[Cam05] Cambridge Vehicle Information Tech. Ltd. Choice Routing, 2005. `http://camvit.com/camvit-technical-english/Camvit-Choice-Routing-Explanation-english.pdf`. Accessed: 2014-08-06. [see page 149]

[CBB07] Yanyan Chen, Michael G. H. Bell, and Klaus Bogenberger. Reliable Pretrip Multipath Planning and Dynamic Adaptation for a Centralized Road Navigation System. *IEEE Transactions on Intelligent Transportation Systems*, 8(1):14–20, 2007. [see page 125]

[CC94] Trevor F. Cox and Michael A. A. Cox. *Multidimensional Scaling*. Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, first edition, 1994. [see page 13]

[CCJ90] Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit Disk Graphs. *Discrete Mathematics*, 86(1–3):165–177, 1990. [see page 72]

[CD05] Mihaela Cardei and Ding-Zhu Du. Improving Wireless Sensor Network Lifetime Through Power Aware Organization. *Wireless Networks*, 11(3):333–340, 2005. [see page 20]

[CdDR12] Raffaele Cerulli, Renato de Donato, and Andrea Raiconi. Exact and Heuristic Methods to Maximize Network Lifetime in Wireless Sensor Networks with Adjustable Sensing Ranges. *European Journal of Operational Research*, 220(1):58–66, 2012. [see pages 23 and 43]

[Chr76] Nicos Christofides. Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. Management Sciences Research Report No. 388, Carnegie Mellon University, 1976. [see page 48]

[Chv79] Vašek Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. [see page 45]

[CK03] Chee-Yee Chong and Srikanta P. Kumar. Sensor Networks: Evolution, Opportunities, and Challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003. [see page 2]

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009. [see pages 10 and 45]

[CRSV13] Fabian Castaño, André Rossi, Marc Sevaux, and Nubia Velasco. A Column Generation Approach to Extend Lifetime in Wireless Sensor Networks with Coverage and Connectivity Constraints. *Computers & Operations Research*, 2013. Accepted for publication. [see pages 23 and 43]

[CS12] Wei-Cheng Chu and Kuo-Feng Ssu. Decentralized Boundary Detection without Location Information in Wireless Sensor Networks. In *Wireless Communications and Networking Conference (WCNC'12)*, pp. 1720–1724. IEEE, 2012. [see page 70]

[CT97] Marco Cesati and Luca Trevisan. On the Efficiency of Polynomial Time Approximation Schemes. *Information Processing Letters*, 64(4):165–171, 1997. [see page 10]

[CTLW05] Mihaela Cardei, My T. Thai, Yingshu Li, and Weili Wu. Energy-Efficient Target Coverage in Wireless Sensor Networks. In *Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, vol. 3, pp. 1976–1984. IEEE, 2005. [see pages 20, 23, 25, 29, 31, and 43]

[CW06] Mihaela Cardei and Jie Wu. Energy-Efficient Coverage Problems in Wireless Ad-Hoc Sensor Networks. *Computer Communications*, 29(4):413–420, 2006. [see page 19]

[Dan51] George B. Dantzig. Maximization of a Linear Function of Variables Subject to Linear Inequalities. In *Activity Analysis of Production and Allocation*, *Cowles Commision Monographs*, vol. 13, pp. 339–347. Wiley, 1951. [see page 12]

[Dan63] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, first edition, 1963. [see pages 12, 31, 119, 128, and 162]

[DDHG05] Jitender S. Deogun, Saket Das, Haitham S. Hamza, and Steve Goddard. An Algorithm for Boundary Discovery in Wireless Sensor Networks. In *International Conference on High Performance Computing (HiPC'05)*, *LNCS*, vol. 3769, pp. 343–352. Springer, 2005. [see page 68]

[Des11] Karine Deschinkel. A Column Generation based Heuristic for Maximum Lifetime Coverage in Wireless Sensor Networks. In *International Conference on Sensor Technologies and Applications (SENSORCOMM'11)*, pp. 209–214. IARIA, 2011. [see pages 22, 43, and 44]

[DF79] Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, 1979. [see page 248]

[DFF56] George B. Dantzig, Lester R. Ford, and Delbert R. Fulkerson. A Primal-Dual Algorithm for Linear Programs. In *Linear Inequalities and Related Systems*, *Annals of Mathematics Studies*, vol. 38, pp. 171–181. Princeton University Press, 1956. [see page 12]

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. [see pages 120, 122, 132, 162, and 184]

[DGPW13] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks, 2013. Preprint available at `http://research.microsoft.com/pubs/198358/crp_web_130724.pdf`. Accessed: 2014-08-06. [see pages 121 and 162]

[DGPW14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Robust Exact Distance Queries on Massive Networks. Technical Report MSR-TR-2014-12, Microsoft Research, 2014. [see pages 121 and 123]

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *International Parallel*

and Distributed Processing Symposium (IPDPS'11), pp. 1135–1146. IEEE, 2011. [see page 121]

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster Batched Shortest Paths in Road Networks. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, *OASIcs*, vol. 20, pp. 52–63. Dagstuhl Publishing, 2011. [see page 122]

[DGW13] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hub Label Compression. In *International Symposium on Experimental Algorithms (SEA'13)*, *LNCS*, vol. 7933, pp. 18–29. Springer, 2013. [see page 121]

[Dha12] Akshaye Dhawan. Maximum Lifetime Scheduling in Wireless Sensor Networks. In *Wireless Sensor Networks - Technology and Protocols*, pp. 25–48. InTech, 2012. [see pages 23 and 24]

[DHM+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *DIMACS Book Series*, vol. 74, pp. 73–92. American Mathematical Society, 2009. [see page 121]

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959. [see pages 119 and 128]

[Din09] Thanh Le Dinh. Topological Boundary Detection in Wireless Sensor Networks. *Journal of Information Processing Systems*, 5(3):145–150, 2009. [see page 70]

[DKMT11] Amol Deshpande, Samir Khuller, Azarakhsh Malekian, and Mohammed Toossi. Energy Efficient Monitoring in Sensor Networks. *Algorithmica*, 59(1):94–114, 2011. [see page 20]

[DKW14] Daniel Delling, Moritz Kobitzsch, and Renato F. Werneck. Customizing Driving Directions with GPUs. In *International Conference on Parallel Processing (Euro-Par'14)*, *LNCS*, vol. 8632, pp. 728–739. Springer, 2014. [see page 121]

[DLL09] Dezun Dong, Yunhao Liu, and Xiangke Liao. Fine-Grained Boundary Recognition in Wireless Ad Hoc and Sensor Networks By Topological Methods. In *International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'09)*, pp. 135–144. ACM, 2009. [see pages 67, 70, and 71]

[DLL+12] Dezun Dong, Xiangke Liao, Kebin Liu, Yunhao Liu, and Weixia Xu. Distributed Coverage in Wireless Ad Hoc and Sensor Networks by Topological Graph Approaches. *IEEE Transactions on Computers*, 61(10):1147–1428, 2012. [see page 70]

[DP10] Waltenegus Dargie and Christian Poellabauer. *Fundamentals of Wireless Sensor Networks: Theory and Practice*. Wiley Series on Wireless Communication and Mobile Computing. Wiley, first edition, 2010. [see page 4]

[dSG06] Vin de Silva and Robert Ghrist. Coordinate-free Coverage in Sensor Networks with Controlled Boundaries via Homology. *International Journal of Robotics Research*, 25(12):1205–1222, 2006. [see page 69]

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway Hierarchies Star. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *DIMACS Book Series*, vol. 74, pp. 73–92. American Mathematical Society, 2009. [see page 121]

[DVZ+06] Akshaye Dhawan, Chinh T. Vu, Alexander Zelikovsky, Yingshu Li, and Sushil K. Prasad. Maximum Lifetime of Sensor Networks with Adjustable Sensing Range. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'06)*, pp. 285–289. IEEE, 2006. [see page 21]

[DW60] George B. Dantzig and Philip Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–110, 1960. [see pages vii, 22, and 40]

[DW09] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In *International Symposium on Experimental Algorithms (SEA'09)*, *LNCS*, vol. 5526, pp. 125–136. Springer, 2009. [see page 124]

[DW13] Daniel Delling and Renato F. Werneck. Customizable Point-of-Interest Queries in Road Networks. In *International Symposium on Advances in Geographic Information Systems (GIS'13)*, pp. 490–493. ACM, 2013. [see page 122]

[DWW+12] Ling Ding, Weili Wu, James Willson, Lidong Wu, Zaixin Lu, and Wonjun Lee. Constant-Approximation for Target Coverage Problem in Wireless Sensor Networks. In *International Conference on Computer Communications (INFOCOM'12)*, pp. 1584–1592. IEEE, 2012. [see pages 21 and 46]

[EGK11] Thomas Erlebach, Tom Grant, and Frank Kammer. Maximising Lifetime for Fault-Tolerant Target Coverage in Sensor Networks. *Sustainable Computing: Informatics and Systems*, 1(3):213–225, 2011. [see page 21]

[EKS14] Stephan Erb, Moritz Kobitzsch, , and Peter Sanders. Parallel Bi-Objective Shortest Paths Using Weight-Balanced B-Trees with Bulk Updates. In *International Symposium on Experimental Algorithms (SEA'14)*, *LNCS*, vol. 8504, pp. 111–122. Springer, 2014. [see page 124]

[EM09] Thomas Erlebach and Matúš Mihalák. A $(4+\epsilon)$-Approximation for the Minimum-weight Dominating Set Problem in Unit Disk Graphs. In *International Workshop on Approximation and Online Algorithms (WAOA'09)*, *LNCS*, vol. 5893, pp. 135–146. Springer, 2009. [see page 25]

[Epp98] David Eppstein. Finding the k Shortest Paths. *SIAM Journal on Computing*, 28(2):652–673, 1998. [see page 124]

[FGG06] Qing Fang, Jie Gao, and Leonidas J. Guibas. Locating and Bypassing Holes in Sensor Networks. *Mobile Networks and Applications*, 11(2):187–200, 2006. [see pages 67, 69, and 70]

[FK06] Stefan Funke and Christian Klein. Hole Detection or: "How Much Geometry Hides in Connectivity?". In *Symposium on Computational Geometry (SCG'06)*, pp. 377–385. ACM, 2006. [see pages 69 and 92]

[FKK+07] Stefan Funke, Alexander Kesselman, Fabian Kuhn, Zvi Lotker, and Michael Segal. Improved Approximation Algorithms for Connected Sensor Cover. *Wireless Networks*, 13(2):153–164, 2007. [see page 25]

[FKKL05] Sándor P. Fekete, M. Kaufmann, Alexander Kröller, and N. Lehmann. A New Approach for Boundary Recognition in Geometric Sensor Networks. In *Canadian Conference on Computational Geometry (CCCG'05)*, pp. 84–87. University of Windsor, 2005. [see pages 69 and 70]

[FKP+04] Sándor P. Fekete, Alexander Kröller, Dennis Pfisterer, Stefan Fischer, and Carsten Buschmann. Neighborhood-Based Topology Recognition in Sensor Networks. In *International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS'04)*, vol. 3121, pp. 123–136. Springer, 2004. [see pages 69, 70, and 92]

[Flo62] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962. [see page 11]

[FMS08] Stefan Funke, Domagoj Matijević, and Peter Sanders. Constant Time Queries for Energy Efficient Paths in Multi-hop Wireless Networks. *Journal*

of Computing and Information Technology, 16(2):119–130, 2008. [see page 123]

[FR89] Thomas A. Feo and Mauricio G.C. Resende. A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem. *Operations Research Letters*, 8(2):67–71, 1989. [see page 23]

[Fri67] Ivan T. Frisch. An Algorithm for Vertex-Pair Connectivity. *International Journal of Control*, 6(6):579–593, 1967. [see page 19]

[FS13] Stefan Funke and Sabine Storandt. Polynomial-Time Construction of Contraction Hierarchies for Multi-Criteria Objectives. In *Meeting on Algorithm Engineering and Experiments (ALENEX'13)*, pp. 31–54. SIAM, 2013. [see page 125]

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, 1987. [see page 248]

[Fun05] Stefan Funke. Topological Hole Detection in Wireless Sensor Networks and its Applications. In *Joint Workshop on Foundations of Mobile Computing (DIALM-POMC'05)*, pp. 44–53. ACM, 2005. [see pages 69 and 92]

[Gav01] Cyril Gavoille. Routing in Distributed Networks: Overview and Open Problems. *SIGACT News*, 32(1):36–52, 2001. [see page 124]

[GB08] James E. Gubernatis and Thomas E. Booth. Multiple Extremal Eigenpairs by the Power Method. *Journal of Computational Physics*, 227(19):8508–8522, 2008. [see page 14]

[GG12] Jie Gao and Leonidas Guibas. Geometric Algorithms for Sensor Networks. *Philosophical Transactions of the Royal Society A*, 370(1958):27–51, 2012. [see page 124]

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Symposium on Discrete Algorithms (SODA'05)*, pp. 156–165. SIAM, 2005. [see pages 120, 131, and 162]

[GJ78] M. R. Garey and David S. Johnson. "Strong" NP-Completeness Results: Motivation, Examples, and Implications. *Journal of the ACM*, 25(3):499–508, 1978. [see page 10]

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, first edition, 1979. [see pages 7 and 13]

[GJLZ09] Yu Gu, Yusheng Ji, Jie Li, and Baohua Zhao. QoS-Aware Target Coverage in Wireless Sensor Networks. *Wireless Communications and Mobile Computing*, 9(12):1645–1659, 2009. [see pages 22, 45, and 51]

[GK07] Naveen Garg and Jochen Könemann. Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems. *SIAM Journal on Computing*, 37(2):630–652, 2007. [see pages vii, 21, 43, and 46]

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In *Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pp. 124–137. SIAM, 2010. [see page 125]

[GKT51] David Gale, Harold W. Kuhn, and Albert W. Tucker. Linear Programming and the Theory of Games. In *Activity Analysis of Production and Allocation*, *Cowles Commision Monographs*, vol. 13, pp. 317–329. Wiley, 1951. [see page 12]

[GLS81] Martin Grötschel, László Lovász, and Alexander Schrijver. The Ellipsoid Method and Its Consequences in Combinatorial Optimization. *Combinatorica*, 1(2):169–197, 1981. [see page 30]

[GLZ07] Yu Gu, Hengchang Liu, and Baohua Zhao. Joint Scheduling and Routing for Lifetime Elongation in Surveillance Sensor Networks. In *Asia-Pacific Service Computing Conference (APSCC'07)*, pp. 81–88. IEEE, 2007. [see pages 22 and 43]

[GM05] Robert Ghrist and Abubakr Muhammad. Coverage and Hole-Detection in Sensor Networks via Homology. In *International Symposium on Information Processing in Sensor Networks (IPSN'05)*, pp. 254–260. IEEE, 2005. [see page 69]

[Gom58] Ralph E. Gomory. Outline of an Algorithm for Integer Solutions to Linear Programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958. [see page 13]

[GP03] Cyril Gavoille and David Peleg. Compact and Localized Distributed Data Structures. *Distributed Computing*, 16(2–3):111–120, 2003. [see page 124]

[GR13] Monica Gentili and Andrea Raiconi. α-Coverage to Extend Network Lifetime on Wireless Sensor Networks. *Optimization Letters*, 7(1):157–172, 2013. [see pages 23 and 43]

[GS10] Robert Geisberger and Dennis Schieferdecker. Heuristic Contraction Hierarchies with Approximation Guarantee. In *International Symposium on Combinatorial Search (SoCS'10)*, pp. 31–38. AAAI Press, 2010. [see page 117]

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012. [see pages 120, 133, 135, 138, and 162]

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pp. 26–40. SIAM, 2005. [see page 131]

[GZDG06] Himanshu Gupta, Zongheng Zhou, Samir R. Das, and Quinyi Gu. Connected Sensor Cover: Self-Organization of Sensor Networks for Efficient Query Execution. *IEEE/ACM Transactions on Networking*, 14(1):55–67, 2006. [see pages 24 and 25]

[GZJL11] Yu Gu, Bao-Hua Zhao, Yu-Sheng Ji, and Jie Li. Theoretical Treatment of Target Coverage in Wireless Sensor Networks. *Journal of Computer Science and Technology*, 26(1):117–129, 2011. [see pages 22, 43, and 45]

[Han80] Pierre Hansen. Bricriterion Path Problems. In *Conference on Multiple Criteria Decision Making (MCDM'79)*, *LNEMS*, vol. 177, pp. 109–127. Springer, 1980. [see page 124]

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *DIMACS Book Series*, vol. 74, pp. 41–72. American Mathematical Society, 2009. [see pages 120 and 132]

[HM85] Dorit S. Hochbaum and Wolfgang Maass. Approximation Schemes for Covering and Packing Problems in Image Processing and VLSI. *Journal of the ACM*, 32(1):130–136, 1985. [see pages 21, 25, and 33]

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. [see pages 120 and 130]

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, 2008. [see page 121]

[HSWW05] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Combining Speed-Up Techniques for Shortest-Path Computations. *ACM Journal of Experimental Algorithmics*, 10(2.5):1–18, 2005. [see page 121]

[IHI+94] Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku, and Kunihiko Mitoh. A Fast Algorithm for Finding Better Routes by AI Search Techniques. In *Vehicle Navigation and Information Systems Conference (VNIS'94)*, pp. 291–296. IEEE, 1994. [see page 130]

[JOW+02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, pp. 96–107. ACM, 2002. [see page 17]

[Kar72] Richard M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pp. 85–103. Plenum Press, 1972. [see page 48]

[Kar84] Narendra Karmarkar. A Polynomial Algorithm in Linear Programming. *Combinatorica*, 4(4):373–396, 1984. [see page 12]

[Kat09] Bastian Katz. *Positioning and Scheduling of Wireless Sensor Networks—Models, Complexity, and Scalable Algorithms*. PhD thesis, University of Karlsruhe, Department of Informatics, 2009. [see page 115]

[KFPF06] Alexander Kröller, Sándor P. Fekete, Dennis Pfisterer, and Stefan Fischer. Deterministic Boundary Recognition and Topology Extraction for Large Sensor Networks. In *Symposium on Discrete Algorithms (SODA'06)*, pp. 1000–1009. ACM, 2006. [see pages 69, 70, and 73]

[Kha79] Leonid G. Khachiyan. A Polynomial Algorithm in Linear Programming. *Doklady Akademiia Nauk SSSR*, 224(S):1093–1096, 1979. English translation in *Soviet Mathematics Doklady*, 20(1):191–194. American Mathematical Society, 1979. [see page 12]

[Kie10] Roger M. Kieckhafer. Hard Real-Time Wireless Communication in the Northern Pierre Auger Observatory. In *Real Time Conference (RT'10)*, pp. 1–8. IEEE, 2010. [see page 117]

[KK99] George Karypis and Gautam Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999. [see page 251]

[KKP99] Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". In *International Conference on Mobile Computing and Networking (MobiCom'99)*, pp. 271–278. ACM, 1999. [see pages 2 and 187]

[KMS05] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *International Workshop on Efficient and Experimental Algorithms (WEA'05)*, *LNCS*, vol. 3503, pp. 126–138. Springer, 2005. [see page 120]

[Kob13] Moritz Kobitzsch. An Alternative Approach to Alternative Routes: HiDAR. In *European Symposium on Algorithms (ESA'13)*, *LNCS*, vol. 8125, pp. 613–624. Springer, 2013. [see page 125]

[KRS13] Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. Evolution and Evaluation of the Penalty Method for Alternative Graphs. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, *OASIcs*, vol. 33, pp. 94–107. Dagstuhl Publishing, 2013. [see page 125]

[KRX07] Goran Konjevod, Andréa W. Richa, and Donglin Xia. Optimal Scale-Free Compact Routing Schemes in Networks of Low Doubling Dimension. In *Symposium on Discrete Algorithms (SODA'07)*, pp. 939–948. SIAM, 2007. [see page 123]

[KSC06] Marcin Karpiński, Aline Senart, and Vinny Cahill. Sensor Networks for Smart Roads. In *International Conference on Pervasive Computing and Communications (PerCom'06)*, pp. 306–310. IEEE, 2006. [see pages 17 and 117]

[KSS+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pp. 36–45. SIAM, 2007. [see pages 122, 153, 159, 184, and 250]

[KWZ08] Fabian Kuhn, Rogert Wattenhofer, and Aaron Zollinger. Ad-Hoc Networks Beyond Unit Disk Graphs. *Wireless Networks*, 14(5):715–729, 2008. [see page 72]

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, vol. 22, pp. 219–230. IfGI prints, 2004. [see pages 120, 131, and 162]

[LD60] Ailsa H. Land and Alison G Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960. [see page 13]

[LD05] Marco E. Lübbecke and Jacques Desrosiers. Selected Topics in Column Generation. *Operations Research*, 53(6):1007–1023, 2005. [see page 40]

[LGDH12] Kyle Luthy, Edward Grant, Nikhil Deshpande, and Thomas C. Henderson. Perimeter Detection in Wireless Sensor Networks. *Robotics and Autonomous Systems*, 60(2):266–277, 2012. [see pages 69 and 71]

[LGR09] Jun Luo, André Girard, and Catherine Rosenberg. Efficient Algorithms to Solve a Class of Resource Allocation Problems in Large Wireless Networks. In *International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT'09)*, pp. 23–27. IEEE, 2009. [see pages 22, 43, and 58]

[LH09] Xiaoyun Li and David K. Hunter. Distributed Coordinate-Free Algorithm for Full Sensing Coverage. *International Journal of Sensor Networks*, 5(3):153–163, 2009. [see page 70]

[LK73] Shen Lin and Brian W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516, 1973. [see page 48]

[LS11] Dennis Luxen and Peter Sanders. Hierarchy Decomposition for Faster User Equilibria on Road Networks. In *International Symposium on Experimental Algorithms (SEA'11)*, *LNCS*, vol. 6630, pp. 242–253. Springer, 2011. [see pages ix and 188]

[LS12a] Dennis Luxen and Dennis Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, *LNCS*, vol. 7276, pp. 260–270. Springer, 2012. [see page 117]

[LS12b] Dennis Luxen and Dennis Schieferdecker. Doing More for Less—Cache-Aware Parallel Contraction Hierarchies Preprocessing. Preprint available at arXiv:1208.2543 [cs.DS], Karlsruhe Institute of Technology, 2012. [see page 248]

[LS14] Dennis Luxen and Dennis Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 2014. Accepted for publication. [see pages 117 and 162]

[Lux13] Dennis Luxen. *Building Blocks for Mapping Services*. PhD thesis, Karlsruhe Institute of Technology, Department of Informatics, 2013. [see page 136]

[LWCL09] Mingming Lu, Jie Wu, Mihaela Cardei, and Minglu Li. Energy-Efficient Connected Coverage of Discrete Targets in Wireless Sensor Networks. *International Journal of Ad Hoc and Ubiquitous Computing*, 4(3/4):137–147, 2009. [see pages 19, 23, and 24]

[Mar84] Ernesto Queiros Vieira Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 16(2):236–245, 1984. [see page 124]

[MCP+02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, pp. 88–97. ACM, 2002. [see page 17]

[Men31] Karl Menger. Bericht über ein mathematisches Kolloquium 1929/30. *Monatshefte für Mathematik und Physik*, 38(1):17–38, 1931. [see page 48]

[MH97] Nenad Mladenović and Pierre Hansen. Variable Neighborhood Search. *Computers & Operations Research*, 24(11):1097–1100, 1997. [see page 23]

[MHW01] Matthias Müller-Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *International Workshop on Algorithm Engineering (WAE'01)*, *LNCS*, vol. 2141, pp. 185–197. Springer, 2001. [see page 124]

[MIH81] Shigeru Masuyama, Toshihide Ibaraki, and Toshiharu Hasegawa. The Computational Complexity of the m-Center Problems on the Plane. *IECE Transactions*, E64(2):57–64, 1981. [see page 29]

[MS04] Fernando Martincic and Loren Schwiebert. Distributed Perimeter Detection in Wireless Sensor Networks. Technical Report WSU-CSC-NEWS/03-TR03, Wayne State University, 2004. [see pages 68 and 91]

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, first edition, 2008. [see pages 10 and 248]

[MSS+07] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11(2.8):1–29, 2007. [see pages 120 and 131]

[NS10] Amiya Nayak and Ivan Stojmenovic, editors. *Wireless Sensor and Actuator Networks: Algorithms and Protocols for Scalable Coordination and Data Communication*. Wiley, first edition, 2010. [see page 4]

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, first edition, 1984. [see page 123]

[Pel00a] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, first edition, 2000. [see page 124]

[Pel00b] David Peleg. Proximity-Preserving Labeling Schemes. *Journal of Graph Theory*, 33(3):167–176, 2000. [see page 121]

[PG04] Srinivasan Parthasarathy and Rajiv Gandhi. Fast Distributed Well Connected Dominating Sets for Ad Hoc Networks. Technical Report CS-TR-4559, University of Maryland, 2004. [see page 77]

[Poh70] Ira Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1(3):193–204, 1970. [see pages 123 and 131]

[Poh71] Ira Pohl. Bi-Directional Search. In *Machine Intelligence Workshop*, vol. 6, pp. 124–140. Edinburgh University Press, 1971. [see page 130]

[PR91] Manfred Padberg and Giovanni Rinaldi. A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. *SIAM Review*, 33(1):60–100, 1991. [see page 13]

[PZ13] Andreas Paraskevopoulos and Christos Zaroliagis. Improved Alternative Route Planning. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, *OASIcs*, vol. 33, pp. 108–122. Dagstuhl Publishing, 2013. [see page 125]

[Rap02] Theodore S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall, second edition, 2002. [see page 127]

[RG11] Andrea Raiconi and Monica Gentili. Exact and Metaheuristic Approaches to Extend Lifetime and Maintain Connectivity in Wireless Sensors Networks. In *International Network Optimization Conference (INOC'11)*, *LNCS*, vol. 6701, pp. 607–619. Springer, 2011. [see pages 23 and 43]

[RRP+03] Ananth Rao, Sylvia Ratnasamy, Christos Papadimitriou, Scott Shenker, and Ion Stoica. Geographic Routing without Location Information. In *International Conference on Mobile Computing and Networking (MobiCom'03)*, pp. 96–108. ACM, 2003. [see page 67]

[RSS12] André Rossi, Alok Singh, and Marc Sevaux. An Exact Approach for Maximizing the Lifetime of Sensor Networks with Adjustable Sensing Ranges. *Computers & Operations Research*, 39(12):3166–3176, 2012. [see pages 23, 43, and 58]

[SB71] Kenneth Steiglitz and John Bruno. A New Derivation of Frisch's Algorithm for Calculating Vertex-Pair Connectivity. *BIT Numerical Mathematics*, 11(1):94–106, 1971. [see page 19]

[SB11] Amit Shirsat and Bharat Bhargava. Local Geometric Algorithm for Hole Boundary Detection in Sensor Networks. *Security and Communication Networks*, 4(9):1003–1012, 2011. [see page 69]

[Sch89] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience Series in Discrete Mathematics. Wiley, first edition, 1989. [see page 12]

[Sch08a] Dennis Schieferdecker. Systematic Combination of Speed-Up Techniques for Exact Shortest-Path Queries. Diploma thesis, University of Karlsruhe, Department of Informatics, 2008. [see page 249]

[Sch08b] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, University of Karlsruhe, Department of Informatics, 2008. [see page 136]

[Sch12] Alexander Schrijver. On the History of the Shortest Path Problem. *Documenta Mathematica*, Optimization Stories:155–167, 2012. [see pages 119 and 128]

[SG76] Sartaj Sahni and Teofilo Gonzalez. P-Complete Approximation Problems. *Journal of the ACM*, 23(3):555–565, 1976. [see page 48]

[SM13] Peter Sanders and Lawrence Mandow. Parallel Label-Setting Multi-Objective Shortest Path Search. In *International Parallel and Distributed Processing Symposium (IPDPS'13)*, pp. 215–224. IEEE, 2013. [see page 124]

[Som14] Christian Sommer. Shortest-Path Queries in Static Networks. *ACM Computing Surveys*, 46(4):45:1–45:31, 2014. [see page 119]

[SP01] Sasa Slijepcevic and Miodrag Potkonjak. Power Efficient Organization of Wireless Sensor Networks. In *International Conference on Communications (ICC'01)*, vol. 2, pp. 472–476. IEEE, 2001. [see pages 19, 20, 22, 23, 43, 51, and 221]

[SPJB97] Kelley Scott, Glarycelis Pabón-Jiménez, and David Bernstein. Finding Alternatives to the Best Path. Preprint paper 970682, Transportation Research Board, 1997. [see page 125]

[SS05] Catherine Soanes and Angus Stevenson, editors. *Oxford Dictionary of English*. Oxford University Press, revised second edition, 2005. [see pages 1, 7, 17, 67, 117, and 187]

[SS07] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In *International Workshop on Experimental Algorithms (WEA'07)*, *LNCS*, vol. 4525, pp. 66–79. Springer, 2007. [see page 120]

[SS10] Peter Sanders and Dennis Schieferdecker. Lifetime Maximization of Monitoring Sensor Networks. In *International Workshop on Algorithms for Sensor Systems, Wireless Ad Hoc Networks, and Autonomous Mobile Entities (ALGOSENSORS'10)*, *LNCS*, vol. 6451, pp. 134–147. Springer, 2010. [see page 17]

[SS12a] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. *ACM Journal of Experimental Algorithmics*, 17(1):1–40, 2012. [see page 120]

[SS12b] Peter Sanders and Christian Schulz. Distributed Evolutionary Graph Partitioning. In *Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pp. 16–29. SIAM, 2012. [see page 251]

[SSGM10] Olga Saukh, Robert Sauter, Matthias Gauger, and Pedro J. Marrón. On Boundary Recognition without Location Information in Wireless Sensor Networks. *ACM Transactions on Sensor Networks*, 6(3):20:1–20:35, 2010. [see pages 70, 71, and 92]

[SVW11a] Dennis Schieferdecker, Markus Völker, and Dorothea Wagner. Efficient Algorithms for Distributed Detection of Holes and Boundaries in Wireless Networks. Karlsruhe Reports in Informatics 2011,8, Karlsruhe Institute of Technology, 2011. [see pages 67, 85, and 92]

[SVW11b] Dennis Schieferdecker, Markus Völker, and Dorothea Wagner. Efficient Algorithms for Distributed Detection of Holes and Boundaries in Wireless Networks. In *International Symposium on Experimental Algorithms (SEA'11)*, *LNCS*, vol. 6630, pp. 388–399. Springer, 2011. [see pages 67, 85, and 92]

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000. [see page 121]

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, *LNCS*, vol. 2409, pp. 43–59. Springer, 2002. [see page 121]

[SZG13] Rik Sarkar, Xianjin Zhu, and Jie Gao. Distributed and Compact Routing Using Spatial Distributions in Wireless Sensor Networks. *ACM Transactions on Sensor Networks*, 9(3):32:1–32:20, 2013. [see page 123]

[TG02] Di Tian and Nicolas D. Georganas. A Coverage-Preserving Node Scheduling Scheme for Large Wireless Sensor Networks. In *International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, pp. 32–41. ACM, 2002. [see page 24]

[TG05] Di Tian and Nicolas D. Georganas. Connectivity Maintenance and Coverage Preservation in Wireless Sensor Networks. *Ad Hoc Networks*, 3(6):744–761, 2005. [see page 19]

[Tor52] Warren S. Torgerson. Multidimensional Scaling: I. Theory and Method. *Psychometrika*, 17(4):401–419, 1952. [see pages viii and 13]

[TYIM05] Wataru Tsujita, Akihito Yoshino, Hiroshi Ishida, and Toyosaka Moriizumi. Gas Sensor Network for Air-Pollution Monitoring. *Sensors and Actuators B: Chemical*, 110(2):304–311, 2005. [see page 17]

[Vö12] Markus Völker. *Algorithmic Aspects of Communication and Localization in Wireless Sensor Networks.* PhD thesis, Karlsruhe Institute of Technology, Department of Informatics, 2012. [see page 85]

[Vai91] Pravin M. Vaidya. A Sparse Graph Almost as Good as the Complete Graph on Points in K Dimensions. *Discrete & Computational Geometry*, 6(1):369–381, 1991. [see page 123]

[Vaz02] Vijay V. Vazirani. *Approximation Algorithms*. Springer, first edition, 2002. [see pages 7, 10, and 25]

[Vet09] Christian Vetter. Parallel Time-Dependent Contraction Hierarchies. Student research project, Karlsruhe Institute of Technology, Department of Informatics, 2009. [see page 134]

[Vet10] Christian Vetter. Fast and Exact Mobile Navigation with OpenStreetMap Data. Diploma thesis, Karlsruhe Institute of Technology, Department of Informatics, 2010. [see page 134]

[vMPG29] Richard von Mises and Hilda Pollaczek-Geiringer. Praktische Verfahren der Gleichungsauflösung. *Zeitschrift für Angewandte Mathematik und Mechanik*, 9(1):58–77, 1929. [see page 14]

[vN47] John von Neumann. Discussion of a Maximum Problem, 1947. Unpublished working paper. Reprinted in *John von Neumann, Collected Works*, volume VI, pages 89–95. Pergamon Press, 1963. [see page 12]

[Wan11] Bang Wang. Coverage Problems in Sensor Networks: A Survey. *ACM Computing Surveys*, 43(4):32:1–32:53, 2011. [see page 19]

[Weg03] Ingo Wegener. *Komplexitätstheorie: Grenzen der Effizienz von Algorithmen*. Springer, first edition, 2003. [see page 7]

[WGM06] Yue Wang, Jie Gao, and Joseph S. B. Mitchell. Boundary Recognition in Sensor Networks by Topological Methods. In *International Conference on Mobile Computing and Networking (MobiCom'06)*, pp. 122–133. ACM, 2006. [see pages 67, 70, and 92]

[Wil79] Robert Williams. *The Geometrical Foundation of Natural Structure: A Source Book of Design*. Dover Publications, first edition, 1979. [see page 38]

[WW64] John William and Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. [see page 248]

[WW07] Dorothea Wagner and Roger Wattenhofer, editors. *Algorithms for Sensor and Ad Hoc Networks: Advanced Lectures*, *LNCS*, vol. 4621. Springer, first edition, 2007. [see page 4]

[XRC+04] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A Wireless Sensor Network For Structural Monitoring. In *International Conference on Embedded Networked Sensor Systems (SenSys'04)*, pp. 13–24. ACM, 2004. [see page 17]

[Yen71] Jin Y. Yen. Finding the k Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971. [see page 124]

[YMD11] Feng Yan, Philippe Martins, and Laurent Decreusefond. Connectivity-Based Distributed Coverage Hole Detection in Wireless Sensor Networks. In *Global Telecommunications Conference (GLOBECOM'11)*, pp. 1–6. IEEE, 2011. [see page 70]

[YMG08] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless Sensor Network Survey. *Computer Networks*, 52(12):2292–2330, 2008. [see pages 4 and 17]

[YWM05] Liyang Yu, Neng Wang, and Xiaoqiao Meng. Real-Time Forest Fire Detection with Wireless Sensor Networks. In *International Conference on Wireless Communications, Networking and Mobile Computing (WCNMC'05)*, vol. 2, pp. 1214–1217. IEEE, 2005. [see page 17]

[ZG08] Qun Zhao and Mohan Gurusamy. Lifetime Maximization for Connected Target Coverage in Wireless Sensor Networks. *IEEE/ACM Transactions on Networking*, 16(6):1378–1391, 2008. [see pages 21 and 24]

[ZH05] Honghai Zhang and Jennifer C. Hou. Maintaining Sensing Coverage and Connectivity in Large Sensor Networks. *Ad Hoc & Sensor Wireless Networks*, 1(1–2):89–124, 2005. [see pages 19 and 24]

[Zwi01] Uri Zwick. Exact and Approximate Distances in Graphs – A Survey. In *European Symposium on Algorithms (ESA'01)*, *LNCS*, vol. 2161, pp. 33–48. Springer, 2001. [see pages 119 and 123]

[ZWX+11] Feng Zou, Yuexuan Wang, Xiao-Hua Xu, Xianyue Li, Hongwei Du, Pengjun Wan, and Weili Wu. New Approximations for Minimum-Weighted Dominating Sets and Minimum-Weighted Connected Dominating Sets on Unit Disk Graphs. *Theoretical Compututer Science*, 412(3):198–208, 2011. [see page 25]

[ZZF09] Chi Zhang, Yanchao Zhang, and Yuguang Fang. Localized Algorithms for Coverage Boundary Detection in Wireless Sensor Networks. *Wireless Networks*, 15(1):3–20, 2009. [see page 69]

A Appendix A

Lifetime Maximization of Monitoring Sensor Networks

We complement our studies in Section 3.6 on solving SNLP to optimality by additional insights. We describe how to determine the entities required by our algorithm. This is followed by further simulational results to support our previous findings. We conclude with a small excursion on a set-based modelling of the coverage problem.

Converting Area Coverage to Target Coverage

As reasoned in Section 3.6.1, we have to partition the area covered by all sensor nodes into smaller, not necessarily connected regions, called *entities*, so that each entity is covered by a unique set of nodes. This decomposition allows us to define the area A that we want to have monitored by the sensor network. Moreover, the entities double as targets when converting from area coverage to target coverage. The conversion makes it easier to verify whether a set of sensor nodes covers area A. It was first described in [BCSZ05]. In the following, we present our approach for determining these entities.

Our approach considers the overlay of the outlines of the sensing areas of all nodes. Each face in this structure represents an entity. Faces covered by the same subset of nodes are combined into one logical unit, i.e. an entity. The procedure is summarized in Algorithm A.1. We first determine the intersections between all outlines (Algorithm A.2) before deducing the faces and thus the entities from them (Algortihm A.3).

Algorithm A.1 Compute Entities

Input: Set of sensor nodes $(\mathbf{v}, 1) \in V$, with node position $\mathbf{v}$ and sensing range 1
Output: Set of entities $e \in E$, with e described by the nodes covering it

1: $I \leftarrow computeIntersections(V)$ ▷ determine intersections of all sensing areas
2: $F \leftarrow computeFaces(V, I)$ ▷ determine all faces
3: $E \leftarrow removeDuplicates(F)$ ▷ remove duplicate faces
4: **return** E ▷ remaining faces correspond to entities

In the following, we give a detailed description of our procedure. Consider a set of sensor nodes V. We assume circular sensing areas with uniform radii of 1. First, we use Algorithm A.2 to compute the set of intersections I between the outlines of the sensing areas of all nodes in V. For each node $v \in V$, we determine its set of neighboring nodes N_v within a maximal distance of 2 (line 3). For each pair of nodes (v, n), $n \in N_v$, we compute the set of intersection points $P_{v,n}$ of the unit circles centered at the positions of s and n (line 5), i.e. of the outlines of their sensing areas. We store each intersection point $\mathbf{i} \in P_{v,n}$ and the set of nodes $N_\mathbf{i}$ inducing the intersection in I, see lines 7–11. If we encounter an intersection point $\mathbf{i}$ again, we only add the new nodes to $N_\mathbf{i}$.

Algorithm A.2 Compute Intersections

Input: Set of sensor nodes $(\mathbf{v}, 1) \in V$, with node position $\mathbf{v}$ and sensing range 1
Output: Set of intersections $(\mathbf{i}, N_\mathbf{i}) \in I$, with intersection point $\mathbf{i}$ and set of associated nodes $N_\mathbf{i}$ inducing the intersection

```
1:  I ← ∅                                              ▷ initialize set of intersections
2:  for all nodes v := (v, 1) ∈ V do
3:      N_v ← getNeighbors(v, 2, V)                    ▷ find nodes in V within distance 2 of v
4:      for all neighbors n := (n, 1) ∈ N_v do
5:          P_{v,n} ← intersectUnitCircles(v, n)       ▷ intersect unit circles centered at v, n
6:          for all i ∈ P_{v,n} do
7:              if exists (i, N_i) ∈ I then            ▷ check if intersection at i already exists
8:                  (i, N_i) ← (i, N_i ∪ {v, n})       ▷ if true, only add new nodes
9:              else
10:                 I ← I ∪ {(i, {v, n})}              ▷ otherwise, add new intersection
11:             end if
12:         end for
13:     end for
14: end for
15: return I
```

Once we have found all intersections I, we use Algorithm A.3 to determine the set of faces F. We consider each intersection $(\mathbf{i}, N_\mathbf{i}) \in I$. For each node $v \in N_\mathbf{i}$, we compute the inclination α of the tangent at $\mathbf{i}$ to the unit circle centered at its position (line 5), i.e. to the outline of its sensing area. Each tangent is viewed as two half-lines starting at $\mathbf{i}$ and stored as two tuples $(v, \alpha, \text{opening})$ and $(v, \alpha + 180°, \text{closing})$ in T, the set of tangents at $\mathbf{i}$. They are marked as "opening" and "closing", with the right-hand one, facing intersection point $\mathbf{i}$ from node v, marked as opening (line 6). The set of tangents T at $\mathbf{i}$ partitions the immediate area around $\mathbf{i}$ into $|T|$ regions, with each covered by a different set of nodes. When they have been computed, we iterate twice over T in clockwise order, with closing tangents before opening tangents to break ties. Meanwhile, we keep track of the set of active nodes N_{act}. It is initialized with all nodes

within a maximal distance of 1 of $\mathbf{i}$ that are not in $N_\mathbf{i}$, i.e. with all nodes that cover the immediate area surrounding $\mathbf{i}$ (line 9). When we encounter an opening tangent, we add the associated node to N_{act} (line 12). We remove it once we encounter the corresponding closing tangent (line 14). In either case, we add a new face $f_{v,t}$ to the set of faces F (line 16). It is described by the current set of active nodes N_{act}, i.e. by all nodes covering it. During the second iteration, we only consider closing tangents (line 20) and add nodes to the respective faces instead of creating new ones (lines 22). This addresses the nodes that should have been active at the start of the first iteration and guarantees, together with the sorting order, that we obtain the correct combination of nodes for each face. Figure A.1 illustrates the construction of tangents and faces.

Algorithm A.3 Compute Faces

Input: Set of sensor nodes $(\mathbf{v}, 1) \in V$, with node position $\mathbf{v}$ and sensing range 1, set of intersections $(\mathbf{i}, N_\mathbf{i}) \in I$, with intersection point $\mathbf{i}$ and set of associated nodes $N_\mathbf{i}$ inducing the intersection

Output: Set of faces $f \in F$, with f described by the nodes covering it

1: $F \leftarrow \emptyset$ ▷ initialize set of faces
2: **for all** intersections $(\mathbf{i}, N_\mathbf{i}) \in I$ **do**
3: $T \leftarrow \emptyset$ ▷ initialize set of tangents
4: **for all** nodes $v := (\mathbf{v}, 1) \in N_\mathbf{i}$ **do**
5: $\alpha \leftarrow getTangentAngle(\mathbf{i}, \mathbf{v})$ ▷ get angle of tangent at $\mathbf{i}$ to unit circle at $\mathbf{v}$
6: $T \leftarrow T \cup \{(v, \alpha, \text{opening}),\ (v, \alpha + \pi, \text{closing})\}$ ▷ store tangents
7: **end for**
8: $T \leftarrow sort(T)$ ▷ sort clockwise, closing tangents first to break ties
9: $N_{act} \leftarrow getNeighbors(\mathbf{i}, 1, V) \setminus N_\mathbf{i}$ ▷ find nodes in V within distance 1 of $\mathbf{i}$
10: **for all** tangents $(v, \cdot, type) \in T$ **do**
11: **if** $type =$ opening **then**
12: $N_{act} \leftarrow N_{act} \cup \{v\}$ ▷ opening tangent: add node
13: **else**
14: $N_{act} \leftarrow N_{act} \setminus \{v\}$ ▷ closing tangent: remove node
15: **end if**
16: $F \leftarrow \{f_{v,t} := N_{act}\}$ ▷ add new face consisting of active nodes
17: **end for**
18: **for all** tangents $(v, \cdot, type) \in T$ **do**
19: **if** $type =$ closing **then**
20: $N_{act} \leftarrow N_{act} \setminus \{v\}$ ▷ closing tangent: remove node
21: **end if**
22: $f_{v,t} \leftarrow f_{v,t} \cup N_{act}$ ▷ add active nodes to face
23: **end for**
24: **end for**
25: **return** F

After we have computed all of the faces surrounding each intersection point and stored them in the set of faces F, we remove duplicate entries (Algorithm A.1, line 3), i.e. faces that are covered by the same set of nodes. The resulting set E comprises all (unique) entities of the area covered by the sensor nodes in V that we are looking for.

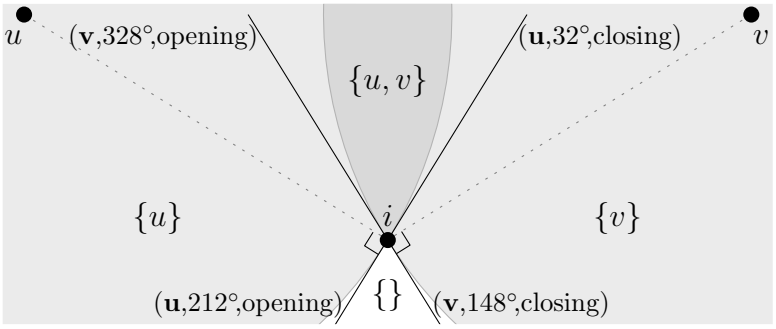


Figure A.1: *Sensor nodes u and v are shown with their respective circular sensing areas of radius* 1 *drawn in grey. The outlines of their sensing areas intersect at position* **i**. *Further depicted are the tangents to each sensing area at position* **i**. *The tuples at each tangent denote the entries stored for each half-line of the tangent starting at* **i**. *They consist of the position of the center of the circle that the half-line is tangent to, the angle of the half-line to the upward vertical axis, and an entry describing whether the half-line marks the beginning of the sensing area or the end, considered in clockwise order around* **i**. *The values in curly brackets denote the four faces induced by the tangents and describe the nodes covering each of them (other nodes that might also cover these faces are omitted for clarity).*

Our implementation applies several advanced data structures. We use a quadtree as geometric data structure to determine all nodes within some maximal distance of a given position. Intersections are stored in an associative array with intersection points as keys and the associated sets of nodes as values. The nodes comprising a face as well as the nodes associated with an intersection point are stored as an ordered set to avoid duplicates. Duplicate faces are filtered using a hash map.

We can simplify our procedure if we assume that only two circles can intersect at any one point. In this case, we do not have to construct or process any tangents. Each intersection point **i** is caused by just two nodes and yields exactly four faces. Each face consists of either both nodes, exactly one of them, or neither, as well as of all nodes within distance 1 of **i**. The special case of two circles only touching each other has to be considered separately, though.

Our approach computes all entities of the area the sensor network covers. However, we only consider a subset of these entities in our studies as explained in Section 3.6.1. Thus, the next section gives an overview on the average number of considered entities as well as further statistics on the monitored areas for each of our network settings.

Monitored Areas

To complement our studies in Section 3.6, we provide further information on the considered areas for each of the network settings. Table A.1 lists the average number of entities that have to be covered as well as the average number of nodes that can cover each entity in each network setting. In addition, the spatial connectivity of the monitored areas is considered. We list the total number of isolated subareas, called *components*, as well as the number of large components, i.e. components that encompass at least 1% as many entities as the largest one.

Table A.1: *Statistical information on the considered areas in each network setting. The average number of entities and the average amount of nodes covering one entity are listed. In addition, the number of isolated components of the monitored area is given. Large components contain at least* 1% *as many entities as the largest one.*

	Entities		Components	
nodes [#]	count [#]	avg. coverage [#/1]	total [#]	large [#]
100	2 110	7.9	39.2	6.4
300	7 285	8.4	13.1	1.1
500	12 705	8.5	3.4	1.0
1 000	26 609	8.7	1.4	1.0

	Entities		Components	
density [#/1]	count [#]	avg. coverage [#/1]	total [#]	large [#]
1.0	2 784	4.1	3.4	1.1
2.5	7 285	8.4	13.1	1.1
5.0	13 947	15.4	23.0	1.1
10.0	24 652	28.4	43.4	1.1

	Entities		Components	
node distr.	count [#]	avg. coverage [#/1]	total [#]	large [#]
g	7 177	8.0	1.5	1.0
pg	7 948	8.4	3.7	1.0
rnd	7 285	8.4	13.1	1.1

Simulational Results

In the following paragraphs, we present the results of additional simulations to support our previous findings. While we mainly focused on networks of 300 nodes with an average node density of 2.5 before, we now consider network sizes between 100 and 1 000 nodes and node densities between 1.0 and 10.0 nodes per unit square. All other parameters remain as in our default network setting of Section 3.6.1. The numbers confirm our previous findings in Section 3.6 to be robust in different network settings. As before, starred results for networks with 1 000 nodes have to be taken into account separately as they depend on values for which we reached our hard time limit of three hours and did not find an optimal solution.

We further show the impact of using different values for the error parameter ϵ in the Garg-Könemann approach while using a heuristic to solve its inner subproblem. The general trend that we observed before holds for all studied values.

Oracle Problem

Table A.2: *Results of the column generation approach with different methods for solving the oracle problem and with network sizes between* 100 *and* 1 000 *nodes. Using a greedy heuristic alone and in combination with an exact solver is shown as well as only using an exact solver. Starred results have to be regarded separately as they depend on values for which we reached our time limit of three hours.*

	heuristic				**heur. + exact**			**exact**		
nodes [#]	time [s]	error [%]	covers [#]	iter. [#]	time [s]	covers [#]	iter. [#]	time [s]	covers [#]	iter. [#]
100	0.1	0.4	65	1	1.6	61	15	1.7	61	14
300	1.6	0.6	181	1	20.1	180	26	20.2	181	25
500	6.5	1.6	264	1	234.2	282	91	232.4	284	91
1 000	31.2	2.8*	408	3	6 609.8*	529*	563*	6 553.0*	527*	561*

Table A.3: *Results of the column generation approach with different methods for solving the oracle problem and with node densities between* 1.0 *and* 10.0 *nodes per unit square. Heuristic and exact approaches are considered.*

	heuristic				**heur. + exact**			**exact**		
density [#/1]	time [s]	error [%]	covers [#]	iter. [#]	time [s]	covers [#]	iter. [#]	time [s]	covers [#]	iter. [#]
1.0	0.5	0.0	137	3	6.8	136	43	6.3	136	40
2.5	1.6	0.6	181	1	20.1	180	26	20.2	181	25
5.0	4.8	1.9	216	1	85.2	216	73	86.7	217	73
10.0	11.3	5.4	236	1	490.5	244	237	490.5	243	236

Garg-Könemann Approach

Table A.4: *Results of the column generation approach with different settings for the Garg-Könemann approach. Different values for error parameter ϵ are studied as well as using an exact solver for the inner subproblem of this approach.*

		Initialization			**CG (1 iter.)**			**Full CG**		
solver	ϵ [1]	time [s]	error [%]	covers [#]	time [s]	error [%]	covers [#]	time [s]	covers [#]	iter. [#]
heuristic	0.01	778.6	5.7	186 237	109.8	0.0	210	109.8	210	4
	0.02	66.9	6.4	46 536	35.2	0.0	215	35.2	215	6
	0.05	6.7	8.4	7 430	24.2	0.0	198	24.2	198	10
	0.10	1.5	12.1	1 839	20.2	0.0	181	20.2	181	25
	0.20	0.4	20.5	441	31.0	0.0	162	31.0	162	94
	0.50	0.1	48.4	54	68.3	0.0	167	68.3	167	482
	0.90	0.0	88.4	4	87.5	0.0	169	87.5	169	726
	0.99	0.0	96.9	1	89.1	0.0	171	89.1	171	768
	1.00	0.0	100.0	0	93.7	0.0	171	93.7	171	796
exact	0.10	63.2	22.7	649	46.3	0.0	161	46.3	161	194

Termination Condition

Table A.5: *Results of the column generation approach when terminating early. The algorithm is stopped if the relative improvement stays below* Δ *for* 10 *iterations. We consider network sizes between* 100 *and* 1 000 *nodes. Starred results have to be regarded separately as they depend on values for which we reached the time limit.*

		Initialization			**CG (1 iter.)**			**Full CG**			
nodes	Δ	time	error	covers	time	error	covers	time	error	covers	iter.
[#]	[%]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[%]	[#]	[#]
100	10.0	0.2	11.6	1 397	0.1	0.4	65	0.3	0.2	64	3
100	1.0	0.2	11.6	1 397	0.1	0.4	65	0.4	0.1	64	3
100	0.1	0.2	11.6	1 397	0.1	0.4	65	0.4	0.1	64	3
100	0.0	0.2	11.6	1 397	0.1	0.4	65	1.7	0.0	61	14
300	10.0	1.5	12.1	1 839	2.0	0.6	181	5.6	0.2	182	4
300	1.0	1.5	12.1	1 839	2.0	0.6	181	6.9	0.1	182	5
300	0.1	1.5	12.1	1 839	2.0	0.6	181	6.9	0.1	182	5
300	0.0	1.5	12.1	1 839	2.0	0.6	181	20.2	0.0	181	25
500	10.0	3.7	12.5	1 999	7.1	1.6	265	35.5	0.7	275	8
500	1.0	3.7	12.5	1 999	7.1	1.6	265	48.2	0.5	277	12
500	0.1	3.7	12.5	1 999	7.1	1.6	265	48.2	0.5	277	12
500	0.0	3.7	12.5	1 999	7.1	1.6	265	232.4	0.0	284	91
1 000	10.0	13.6	12.7	2 204	28.2	2.8	412	141.8	1.9	451	10
1 000	1.0	13.6	12.7*	2 204	28.2	2.8*	412	202.4	1.7*	463	15
1 000	0.1	13.6	12.7*	2 204	28.2	2.8*	412	202.3	1.7*	463	15
1 000	0.0	13.6	12.7*	2 204	28.2	2.8*	412	6 553.0*	0.0*	527*	561*

Table A.6: *Results of the column generation approach when terminating early and with node densities between* 1.0 *and* 10.0 *nodes per unit square.*

		Initialization			**CG (1 iter.)**			**Full CG**			
density	Δ	time	error	covers	time	error	covers	time	error	covers	iter.
[#/1]	[%]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[%]	[#]	[#]
1.0	10.0	0.7	9.1	1 015	0.5	0.0	137	0.7	0.0	137	2
1.0	1.0	0.7	9.1	1 015	0.5	0.0	137	0.7	0.0	137	2
1.0	0.1	0.7	9.1	1 015	0.5	0.0	137	0.7	0.0	137	2
1.0	0.0	0.7	9.1	1 015	0.5	0.0	137	6.3	0.0	136	40
2.5	10.0	1.5	12.1	1 839	2.0	0.6	181	5.6	0.2	182	4
2.5	1.0	1.5	12.1	1 839	2.0	0.6	181	6.9	0.1	182	5
2.5	0.1	1.5	12.1	1 839	2.0	0.6	181	6.9	0.1	182	5
2.5	0.0	1.5	12.1	1 839	2.0	0.6	181	20.2	0.0	181	25
5.0	10.0	4.5	16.2	3 285	5.4	2.0	216	15.8	0.8	216	8
5.0	1.0	4.5	16.2	3 285	5.4	2.0	216	20.6	0.5	216	12
5.0	0.1	4.5	16.2	3 285	5.4	2.0	216	20.6	0.5	216	12
5.0	0.0	4.5	16.2	3 285	5.4	2.0	216	86.7	0.0	217	73
10.0	10.0	18.2	22.6	6 110	12.5	5.4	236	40.5	3.3	237	11
10.0	1.0	18.2	22.6	6 110	12.5	5.4	236	72.0	2.2	240	24
10.0	0.1	18.2	22.6	6 110	12.5	5.4	236	72.0	2.2	240	24
10.0	0.0	18.2	22.6	6 110	12.5	5.4	236	490.5	0.0	243	236

Sensing Ranges

Table A.7: *Results of our approach with varying sensing ranges taken uniformly at random from* $[1, 1+R]$ *for each node and with network sizes between* 100 *and* 1 000 *nodes. Starred results have to be regarded separately as they depend on values for which we reached our hard time limit of three hours.*

		Initialization			CG (1 iter.)			Full CG		
nodes	R	time	error	covers	time	error	covers	time	covers	iter.
[#]	[1]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[#]	[#]
100	0.00	0.2	11.6	1 397	0.1	0.4	65	1.7	61	14
100	0.01	0.2	11.6	1 406	0.1	0.3	68	1.2	68	11
100	0.10	0.2	11.4	1 459	0.1	0.2	68	0.4	68	4
100	1.00	0.3	10.5	1 871	0.1	0.0	58	0.1	58	1
300	0.00	1.5	12.1	1 839	2.0	0.6	181	20.2	181	25
300	0.01	1.5	12.0	1 849	1.9	0.6	182	18.0	185	21
300	0.10	1.6	11.8	1 921	1.7	0.4	181	10.5	182	12
300	1.00	2.1	10.0	2 510	1.2	0.0	134	1.4	134	1
500	0.00	3.7	12.5	1 999	7.1	1.6	265	232.4	284	91
500	0.01	3.7	12.4	2 010	7.3	1.5	265	203.6	288	74
500	0.10	3.9	12.0	2 095	7.1	1.0	266	116.8	282	34
500	1.00	5.4	10.0	2 799	3.2	0.1	196	6.2	198	2
1 000	0.00	13.6	12.7*	2 204	28.2	2.8*	412	6 553.0*	527*	561*
1 000	1.00	19.3	9.2	3 038	11.5	0.1	281	64.4	286	9

Table A.8: *Results with sensing ranges taken uniformly at random from* $[1, 1+R]$ *for each node and with node densities between* 1.0 *and* 10.0 *nodes per unit square.*

		Initialization			CG (1 iter.)			Full CG		
density	R	time	error	covers	time	error	covers	time	covers	iter.
[#/1]	[1]	[s]	[%]	[#]	[s]	[%]	[#]	[s]	[#]	[#]
1.0	0.00	0.7	9.1	1 015	0.5	0.0	137	6.3	136	40
1.0	0.01	0.7	9.0	1 051	0.5	0.1	154	2.6	156	20
1.0	0.10	0.7	7.5	1 091	0.3	0.0	120	0.3	120	1
1.0	1.00	0.9	4.5	1 329	0.3	0.0	50	0.3	50	1
2.5	0.00	1.5	12.1	1 839	2.0	0.6	181	20.2	181	25
2.5	0.01	1.5	12.0	1 849	1.9	0.6	182	18.0	185	21
2.5	0.10	1.6	11.8	1 921	1.7	0.4	181	10.5	182	12
2.5	1.00	2.1	10.0	2 510	1.2	0.0	134	1.4	134	1
5.0	0.00	4.5	16.2	3 285	5.4	2.0	216	86.7	217	73
5.0	0.01	4.6	16.1	3 305	5.4	1.8	218	78.3	220	66
5.0	0.10	4.7	16.1	3 440	5.6	1.7	217	57.6	219	47
5.0	1.00	6.5	14.2	4 628	4.2	0.3	198	10.2	199	5
10.0	0.00	18.2	22.6	6 110	12.5	5.4	236	490.5	243	236
10.0	0.01	18.1	22.5	6 145	12.6	5.3	237	477.7	246	226
10.0	0.10	18.9	22.5	6 383	12.9	5.1	237	434.5	245	201
10.0	1.00	25.7	20.0	8 514	15.1	1.4	238	70.4	240	19

Set-Based Model for Coverage

In Section 3.2, we described the sensor network lifetime problem as a combinatorial, set-based problem as well as a linear program. We considered the attached coverage problem only implicitly at that time, though, as it was sufficient for our purposes. To complement the modelling of our problem, we now describe a set-based model for the coverage problem that is also used in the literature, e.g. in [SP01].

First, recall that area coverage can be reduced to target coverage, see [BCSZ05]. We can therefore concentrate on modelling the latter problem by sets. Consider a set of targets T that has to be covered by a set of sensor nodes V. We represent each node $v \in V$ by the subset of targets that it can cover. In consequence, each set of nodes is a subset of of the power set of T, i.e. $V \subseteq \mathcal{P}(T)$. A set of nodes $\mathbf{c}$ is a cover if the union of the sensor nodes contains all targets, i.e. $\bigcup_{v \in \mathbf{c}} v = T$. This completes the set-based model for (target) coverage and concludes our short excursion.

B Appendix B

Location-free Detection of Network Boundaries

We complement the results on boundary detection presented in Section 4.6 by results of additional simulations and numerical values. Individual network settings are not listed if they correspond to our default setting. Unless otherwise stated, tables list misclassification ratios of mandatory boundary nodes and interior nodes in percentage. Best values in each column are highlighted in bold. We also provide an overall *rating* for each algorithm to assess its general performance over all settings in the respective table. For each setting we calculate the geometric mean of the correct classification ratios of mandatory boundary nodes and interior nodes. The rating is then computed as (1 − the geometric mean of these values). Lower values denote better results.

Network Sizes

Table B.1: *Average network sizes for all considered settings. Numbers of sensor nodes and communication links are listed. All communication links are undirected.*

	d_{avg}	nodes	links
UDG (pert.)	09	6 573	59 157
	12	8 662	103 947
	15	10 608	159 129
	18	12 623	227 222
	21	14 682	308 328
	27	18 692	504 704

	d_{avg}	nodes	links
UDG (rnd.)	09	6 032	54 297
	12	8 042	96 516
	15	10 054	150 817
	18	12 062	217 130
	21	14 071	295 511
	27	18 093	488 530

	d_{avg}	nodes	links
0.05-QUDG	09	12 604	113 443
	12	16 700	200 406
	15	20 663	309 962
	18	24 729	445 131
	27	36 735	991 867

	d_{avg}	nodes	links
0.25-QUDG	09	12 101	108 915
	12	16 029	192 350
	15	19 714	295 720
	18	23 557	424 043
	27	34 867	941 415

	d_{avg}	nodes	links
0.75-QUDG	09	8 493	76 444
	12	10 954	131 455
	15	13 550	203 260
	18	16 217	291 916
	27	23 900	645 303

Quantitative Analysis

We show additional results for settings with random placement and with the d-QUDG model. This is followed by tables giving numerical values of all considered settings.

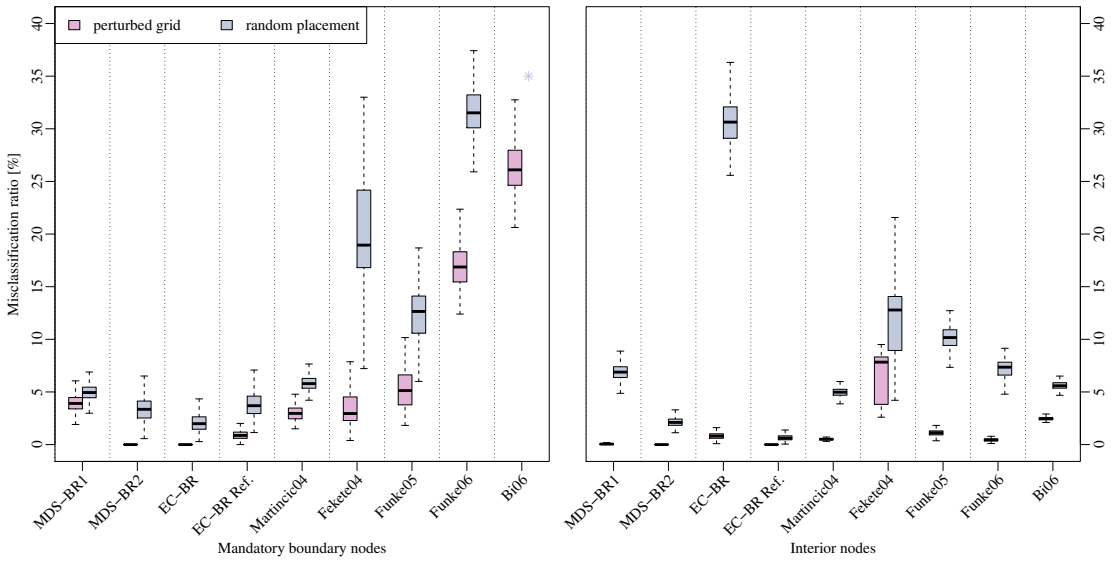


Figure B.1: *Misclassification ratios (false negatives) in percent for perturbed grid placement and random node placement with* $d_{avg} = 18$.

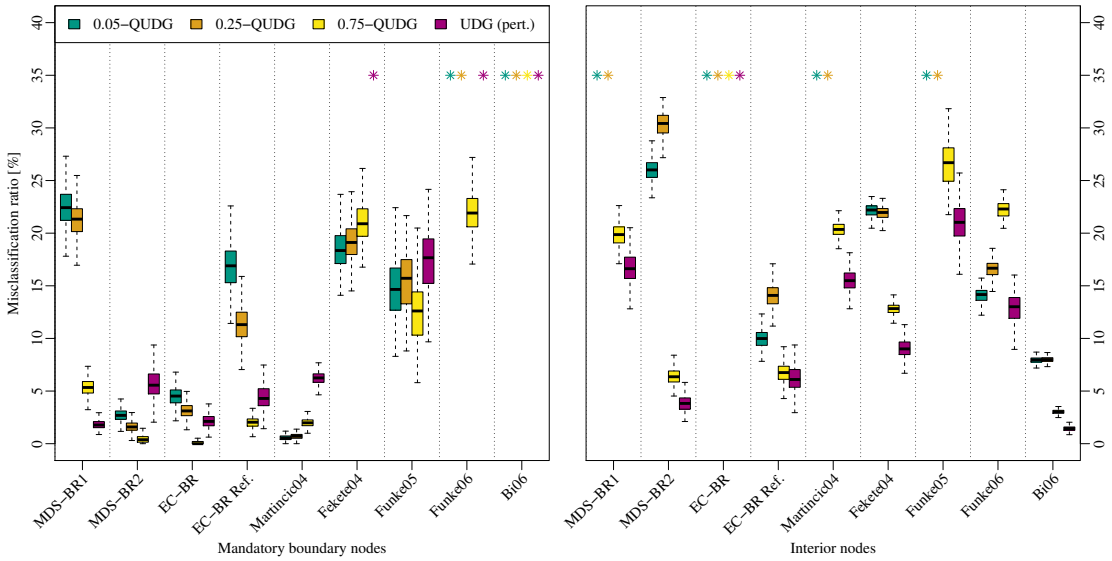


Figure B.2: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree* 9 *and uncertainty levels between* 0.05 *and* 1.00.

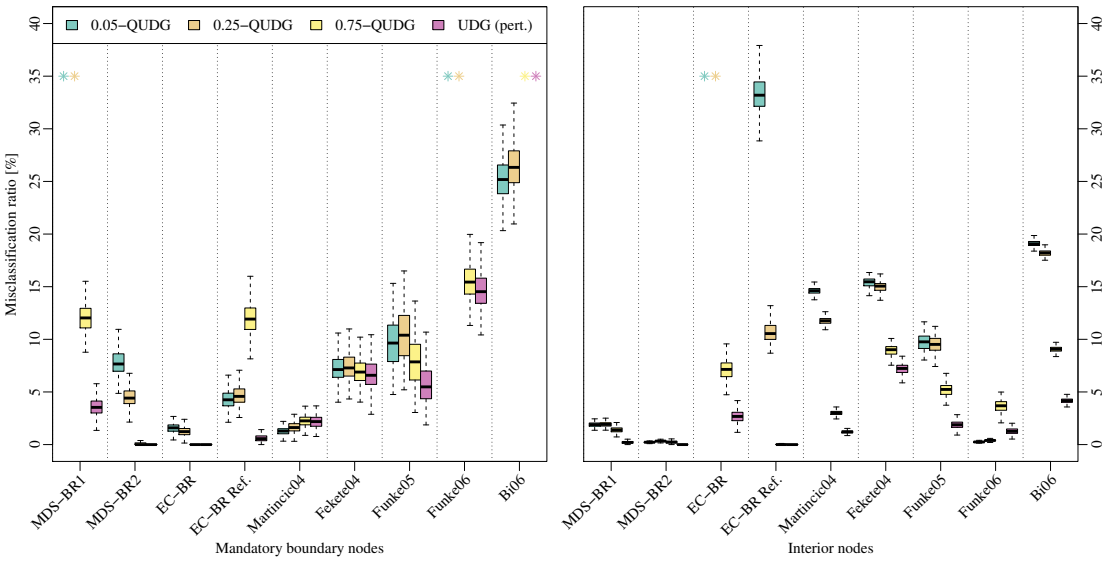


Figure B.3: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree* 15 *and uncertainty levels between* 0.05 *and* 1.00*.*

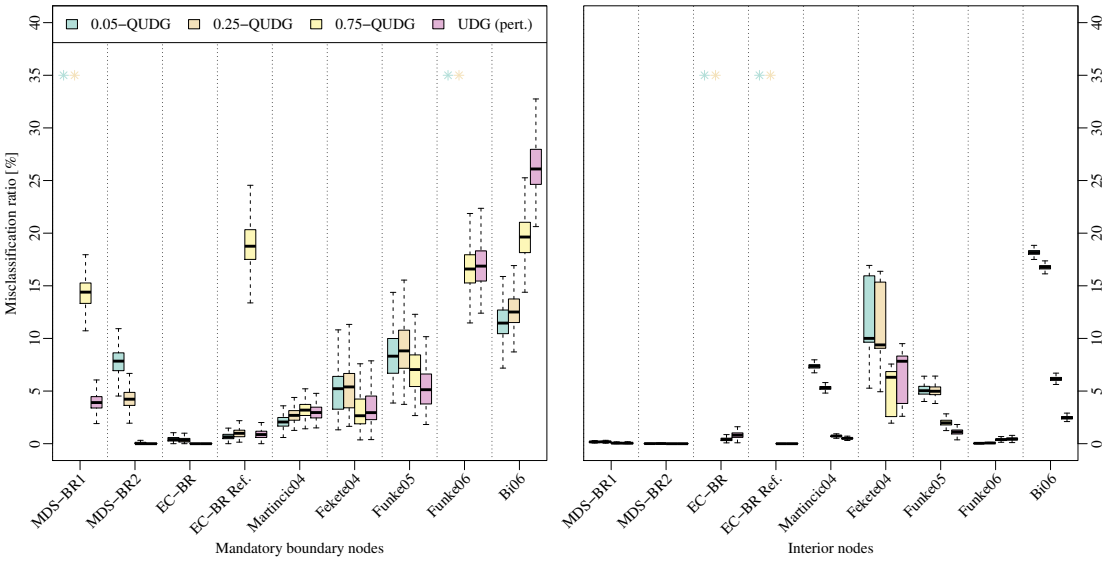


Figure B.4: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree* 18 *and uncertainty levels between* 0.05 *and* 1.00*.*

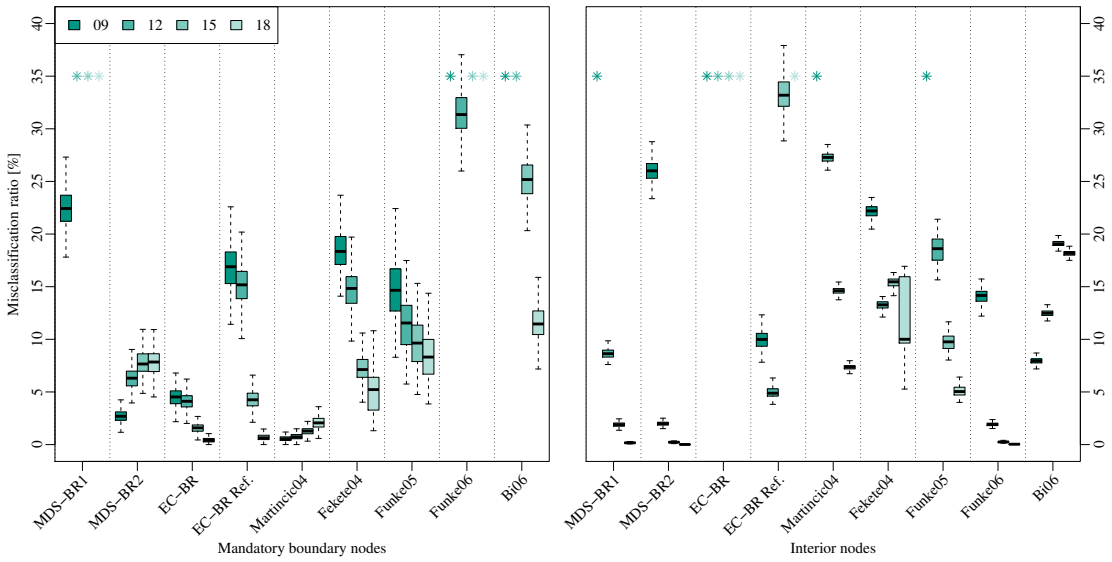


Figure B.5: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with* $d = 0.05$ *and average node degrees between* 9 *and* 18*.*

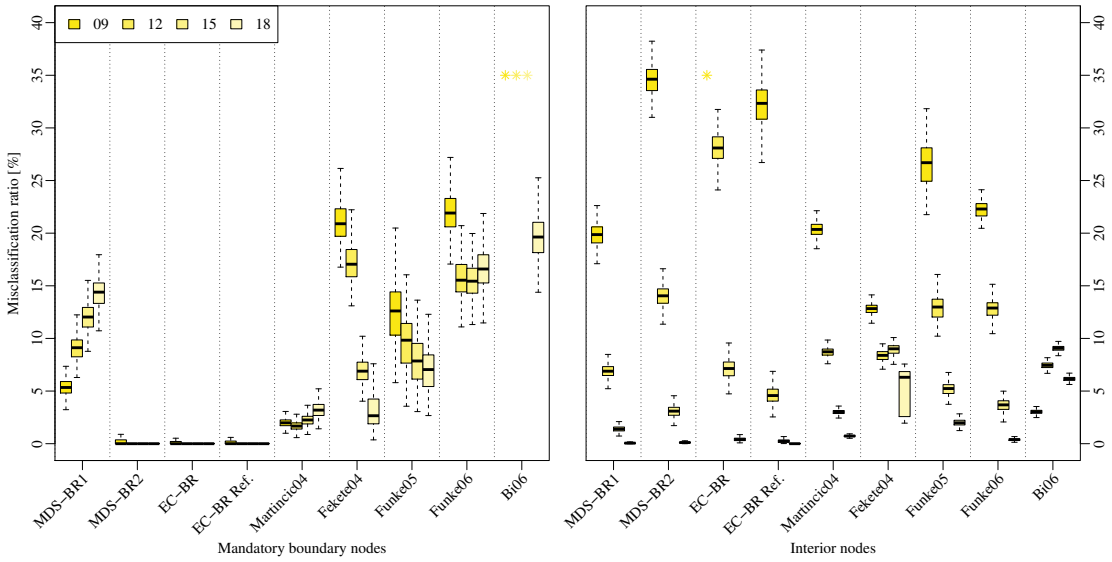


Figure B.6: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with* $d = 0.75$ *and average node degrees between* 9 *and* 18*.*

Table B.2: *Misclassification ratios (false negatives) in percent for average node degrees between* 9 *and* 21*. The table corresponds to Figure 4.17.*

	Mandatory					**Interior**					
algorithm	09	12	15	18	21	09	12	15	18	21	rating
MDS-BR1	**1.8**	2.9	3.6	3.9	4.0	16.7	0.6	0.2	0.1	**0.0**	3.5
MDS-BR2	5.7	**0.0**	**0.0**	**0.0**	**0.0**	3.8	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**
EC-BR	2.2	**0.0**	**0.0**	**0.0**	**0.0**	52.7	6.6	2.7	0.8	0.2	8.4
EC-BR Ref.	4.4	0.4	0.6	0.9	1.3	6.2	**0.0**	**0.0**	**0.0**	**0.0**	1.4
Martincic04	6.2	1.6	2.2	3.0	4.1	15.5	2.5	1.2	0.5	0.2	3.8
Fekete04	34.6	14.2	6.7	3.5	1.9	10.2	3.4	7.2	6.7	2.5	9.6
Funke05	17.5	6.1	5.7	5.4	4.9	20.9	3.4	1.9	1.1	0.8	7.0
Funke06	38.6	11.9	14.6	16.9	19.0	12.8	3.3	1.3	0.5	0.2	12.7
Bi06	76.4	54.4	41.4	26.4	15.8	**1.4**	3.7	4.2	2.5	2.5	28.7

Table B.3: *Classification ratios of optional boundary nodes as interior nodes in percent for d_{avg} between* 9 *and* 21*. The table corresponds to Figure 4.18.*

	Optional					
algorithm	09	12	15	18	21	rating
MDS-BR1	70.7	81.7	81.4	81.4	81.6	55.0
MDS-BR2	53.4	33.0	30.7	29.1	27.9	19.8
EC-BR	**1.2**	**2.3**	**2.5**	**2.7**	**2.8**	**1.2**
EC-BR Ref.	53.1	82.4	83.5	84.9	86.4	55.9
Martincic04	82.4	92.3	93.1	93.9	94.5	71.8
Fekete04	83.5	81.5	70.9	66.1	67.3	50.0
Funke05	63.3	62.0	58.6	56.3	54.4	36.0
Funke06	80.9	71.6	72.7	73.6	74.2	49.8
Bi06	96.0	90.9	81.9	77.8	76.6	64.2

Table B.4: *Misclassification ratios (false negatives) in percent for random node placement and d_{avg} between* 9 *and* 27*. The table corresponds to Figure 4.23.*

	Mandatory				**Interior**				
algorithm	09	15	21	27	09	15	21	27	rating
MDS-BR1	2.4	4.0	5.7	6.2	28.9	11.8	3.8	1.0	8.4
MDS-BR2	2.1	4.5	1.8	0.3	8.9	3.5	1.2	0.3	**2.9**
EC-BR	**0.2**	**1.9**	**1.4**	**0.2**	67.0	44.0	19.6	6.1	22.2
EC-BR Ref.	0.6	4.0	2.7	1.9	28.4	**2.6**	**0.2**	**0.0**	5.5
Martincic04	4.8	6.5	5.3	6.5	23.6	8.6	2.8	0.7	7.6
Fekete04	47.8	25.7	13.8	7.1	7.1	13.5	12.0	10.8	18.4
Funke05	20.3	15.9	8.8	6.0	28.7	15.3	6.4	2.3	13.4
Funke06	85.0	44.9	26.6	24.6	**2.8**	7.7	5.2	1.7	33.5
Bi06	79.0	62.2	33.7	17.9	3.5	6.6	5.5	5.5	34.3

Table B.5: *Misclassification ratios (false negatives) in percent for perturbed grid and random node placement with $d_{avg} = 12$. The table corresponds to Figure 4.22.*

	Mandatory		Interior		
algorithm	perturbed grid	random placement	perturbed grid	random placement	rating
MDS-BR1	2.9	3.1	0.6	19.0	6.7
MDS-BR2	**0.0**	4.1	**0.0**	**5.5**	**2.4**
EC-BR	**0.0**	**1.0**	6.6	56.6	20.4
EC-BR Ref.	0.4	2.4	**0.0**	9.8	3.2
Martincic04	1.6	6.3	2.5	14.2	6.3
Fekete04	14.2	42.5	3.4	7.7	18.6
Funke05	6.1	18.6	3.4	21.5	12.7
Funke06	11.9	67.8	3.3	5.8	28.7
Bi06	54.4	74.1	3.7	6.0	42.8

Table B.6: *Misclassification ratios (false negatives) in percent for perturbed grid and random node placement with $d_{avg} = 18$. The table corresponds to Figure B.1.*

	Mandatory		Interior		
algorithm	perturbed grid	random placement	perturbed grid	random placement	rating
MDS-BR1	3.9	5.0	0.1	6.9	4.0
MDS-BR2	**0.0**	3.4	**0.0**	2.1	1.4
EC-BR	**0.0**	**2.1**	0.8	30.7	9.4
EC-BR Ref.	0.9	3.8	**0.0**	**0.7**	**1.4**
Martincic04	3.0	5.8	0.5	5.0	3.6
Fekete04	3.5	19.8	6.7	12.7	10.9
Funke05	5.4	12.5	1.1	10.1	7.4
Funke06	16.9	31.6	0.5	7.1	14.9
Bi06	26.4	47.3	2.5	5.6	22.7

Table B.7: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree 9 and uncertainty levels between 0.05 and 1.00 (i.e. for unit disk graphs). The table corresponds to Figure B.2.*

	Mandatory				Interior				
algorithm	0.05	0.25	0.75	1.00	0.05	0.25	0.75	1.00	rating
MDS-BR1	22.5	21.3	5.4	**1.8**	35.1	35.7	19.8	16.7	20.6
MDS-BR2	2.7	1.6	0.4	5.7	26.0	30.4	6.4	3.8	10.4
EC-BR	4.5	3.2	**0.1**	2.2	49.5	51.9	60.1	52.7	32.8
EC-BR Ref.	16.9	11.3	2.0	4.4	10.0	14.1	6.7	6.2	**9.1**
Martincic04	**0.6**	**0.7**	2.0	6.2	52.0	48.9	20.4	15.5	21.1
Fekete04	18.6	19.3	21.1	34.6	22.1	21.9	12.8	10.2	20.4
Funke05	14.6	15.6	12.3	17.5	39.6	39.2	26.6	20.9	24.0
Funke06	42.5	39.6	22.0	38.6	13.8	16.3	21.6	12.8	26.8
Bi06	52.1	52.9	60.8	76.4	**7.9**	**8.0**	**3.0**	**1.4**	39.9

Table B.8: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree* 12 *and uncertainty levels between* 0.05 *and* 1.00 *(i.e. for unit disk graphs). The table corresponds to Figure 4.27.*

	Mandatory				**Interior**				
algorithm	0.05	0.25	0.75	1.00	0.05	0.25	0.75	1.00	rating
MDS-BR1	35.5	35.3	9.1	2.9	8.6	8.2	6.9	0.6	14.5
MDS-BR2	6.3	3.8	0.2	**0.0**	2.0	2.4	1.8	**0.0**	**2.1**
EC-BR	4.1	3.0	**0.0**	**0.0**	50.1	41.3	28.1	6.6	19.1
EC-BR Ref.	15.2	12.6	5.8	0.4	5.0	**1.5**	**0.5**	**0.0**	5.3
Martincic04	**0.7**	**1.0**	1.7	1.6	27.3	23.5	8.7	2.5	9.0
Fekete04	14.5	15.0	16.9	14.2	14.1	13.0	8.9	3.4	12.6
Funke05	11.4	12.4	9.6	6.1	18.5	17.6	12.9	3.4	11.6
Funke06	31.4	24.3	15.7	11.9	**1.9**	2.6	12.4	3.3	13.5
Bi06	40.6	41.4	51.3	54.4	12.5	12.1	7.4	3.7	30.7

Table B.9: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree* 15 *and uncertainty levels between* 0.05 *and* 1.00 *(i.e. for unit disk graphs). The table corresponds to Figure B.3.*

	Mandatory				**Interior**				overall
algorithm	0.05	0.25	0.75	1.00	0.05	0.25	0.75	1.00	rating
MDS-BR1	42.7	43.5	12.1	3.6	1.9	1.9	1.4	0.2	15.5
MDS-BR2	7.8	4.5	0.1	**0.0**	**0.2**	**0.3**	0.2	**0.0**	**1.7**
EC-BR	1.6	**1.2**	**0.0**	**0.0**	69.6	57.7	7.1	2.7	23.9
EC-BR Ref.	4.3	4.7	12.0	0.6	33.5	10.7	**0.0**	**0.0**	8.9
Martincic04	**1.3**	1.7	2.3	2.2	14.6	11.7	3.0	1.2	4.9
Fekete04	7.3	7.4	7.0	6.7	15.3	14.9	8.9	7.2	9.4
Funke05	9.8	10.5	7.8	5.7	9.7	9.6	5.2	1.9	7.6
Funke06	52.6	37.0	15.5	14.6	0.3	0.4	3.6	1.3	18.0
Bi06	25.3	26.5	36.7	41.4	19.1	18.2	9.1	4.2	23.5

Table B.10: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with average node degree* 18 *and uncertainty levels between* 0.05 *and* 1.00 *(i.e. for unit disk graphs). The table corresponds to Figure B.4.*

	Mandatory				**Interior**				
algorithm	0.05	0.25	0.75	1.00	0.05	0.25	0.75	1.00	rating
MDS-BR1	46.7	49.0	14.4	3.9	0.2	0.2	0.1	0.1	17.1
MDS-BR2	7.8	4.3	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.6**
EC-BR	**0.4**	**0.4**	**0.0**	**0.0**	83.6	72.3	0.4	0.8	32.2
EC-BR Ref.	0.7	1.0	19.0	0.9	75.3	41.0	**0.0**	**0.0**	23.7
Martincic04	2.1	2.7	3.2	3.0	7.3	5.3	0.7	0.5	3.1
Fekete04	5.0	5.2	3.1	3.5	12.3	11.5	5.2	6.7	6.6
Funke05	8.9	9.0	7.0	5.4	5.1	5.0	2.0	1.1	5.5
Funke06	81.5	61.7	16.7	16.9	**0.0**	0.1	0.4	0.5	31.5
Bi06	11.6	12.7	19.7	26.4	18.2	16.8	6.2	2.5	14.6

Table B.11: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with $d = 0.05$ and average node degrees between 9 and 18. The table corresponds to Figure B.5.*

	Mandatory				Interior				
algorithm	09	12	15	18	09	12	15	18	rating
MDS-BR1	22.5	35.5	42.7	46.7	35.1	8.6	1.9	0.2	26.1
MDS-BR2	2.7	6.3	7.8	7.8	26.0	2.0	**0.2**	**0.0**	**7.0**
EC-BR	4.5	4.1	1.6	**0.4**	49.5	50.1	69.6	83.6	42.9
EC-BR Ref.	16.9	15.2	4.3	0.7	10.0	5.0	33.5	75.3	25.6
Martincic04	**0.6**	**0.7**	**1.3**	2.1	52.0	27.3	14.6	7.3	15.4
Fekete04	18.6	14.5	7.3	5.0	22.1	14.1	15.3	12.3	13.8
Funke05	14.6	11.4	9.8	8.9	39.6	18.5	9.7	5.1	15.4
Funke06	42.5	31.4	52.6	81.5	13.8	**1.9**	0.3	**0.0**	35.7
Bi06	52.1	40.6	25.3	11.6	**7.9**	12.5	19.1	18.2	25.0

Table B.12: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with $d = 0.25$ and average node degrees between 9 and 18. The table corresponds to Figure 4.26.*

	Mandatory				Interior				
algorithm	09	12	15	18	09	12	15	18	rating
MDS-BR1	21.3	35.3	43.5	49.0	35.7	8.2	1.9	0.2	26.5
MDS-BR2	1.6	3.8	4.5	4.3	30.4	2.4	**0.3**	**0.0**	**6.5**
EC-BR	3.2	3.0	**1.2**	**0.4**	51.9	41.3	57.7	72.3	35.3
EC-BR Ref.	11.3	12.6	4.7	1.0	14.1	**1.5**	10.7	41.0	13.1
Martincic04	**0.7**	**1.0**	1.7	2.7	48.9	23.5	11.7	5.3	13.7
Fekete04	19.3	15.0	7.4	5.2	21.9	13.0	14.9	11.5	13.7
Funke05	15.6	12.4	10.5	9.0	39.2	17.6	9.6	5.0	15.5
Funke06	39.6	24.3	37.0	61.7	16.3	2.6	0.4	0.1	26.0
Bi06	52.9	41.4	26.5	12.7	**8.0**	12.1	18.2	16.8	25.3

Table B.13: *Misclassification ratios (false negatives) in percent for quasi unit disk graphs with $d = 0.75$ and average node degrees between 9 and 18. The table corresponds to Figure B.6.*

	Mandatory				Interior				
algorithm	09	12	15	18	09	12	15	18	rating
MDS-BR1	5.4	9.1	12.1	14.4	19.8	6.9	1.4	0.1	8.9
MDS-BR2	0.4	0.2	0.1	**0.0**	6.4	1.8	0.2	**0.0**	**1.2**
EC-BR	**0.1**	**0.0**	**0.0**	**0.0**	60.1	28.1	7.1	0.4	15.3
EC-BR Ref.	2.0	5.8	12.0	19.0	6.7	**0.5**	**0.0**	**0.0**	6.0
Martincic04	2.0	1.7	2.3	3.2	20.4	8.7	3.0	0.7	5.5
Fekete04	21.1	16.9	7.0	3.1	12.8	8.9	8.9	5.2	10.7
Funke05	12.3	9.6	7.8	7.0	26.6	12.9	5.2	2.0	10.7
Funke06	22.0	15.7	15.5	16.7	21.6	12.4	3.6	0.4	13.8
Bi06	60.8	51.3	36.7	19.7	**3.0**	7.4	9.1	6.2	27.7

Graph Embedding Strategies

We give additional results for networks with $d_{avg} = 12$, 15, when using MDS-BR1 and MDS-BR2. This is followed by tables with numerical values of all considered settings.

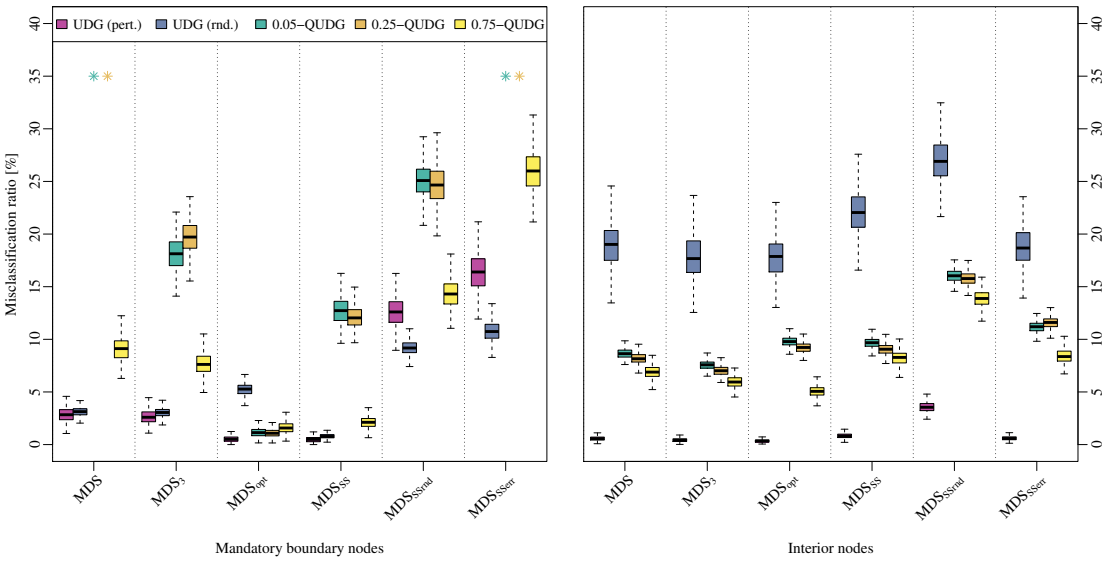


Figure B.7: *Misclassification ratios (false negatives) in percent for MDS-BR1 on different network settings with $d_{avg} = 12$ and multiple graph embedding strategies.*

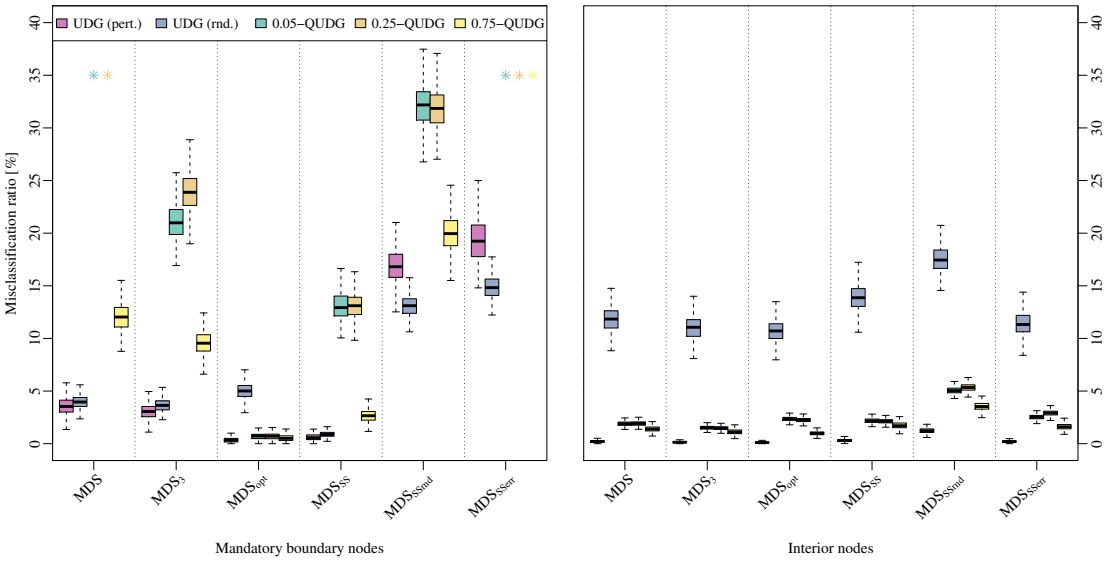


Figure B.8: *Misclassification ratios (false negatives) in percent for MDS-BR1 on different network settings with $d_{avg} = 15$ and multiple graph embedding strategies.*

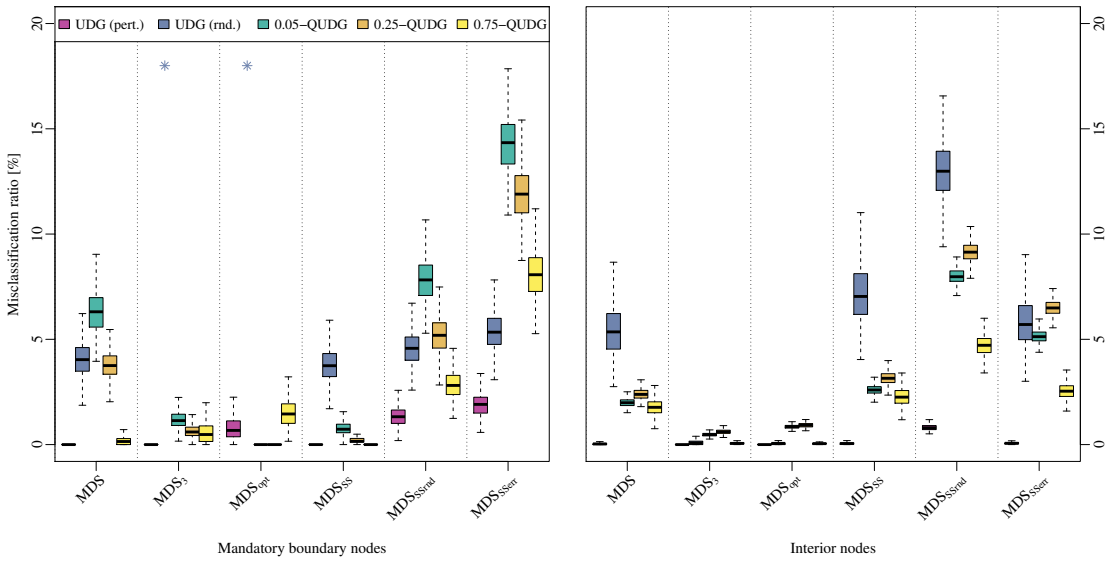


Figure B.9: *Misclassification ratios (false negatives) in percent for MDS-BR2 on different network settings with $d_{avg} = 12$ and multiple graph embedding strategies.*

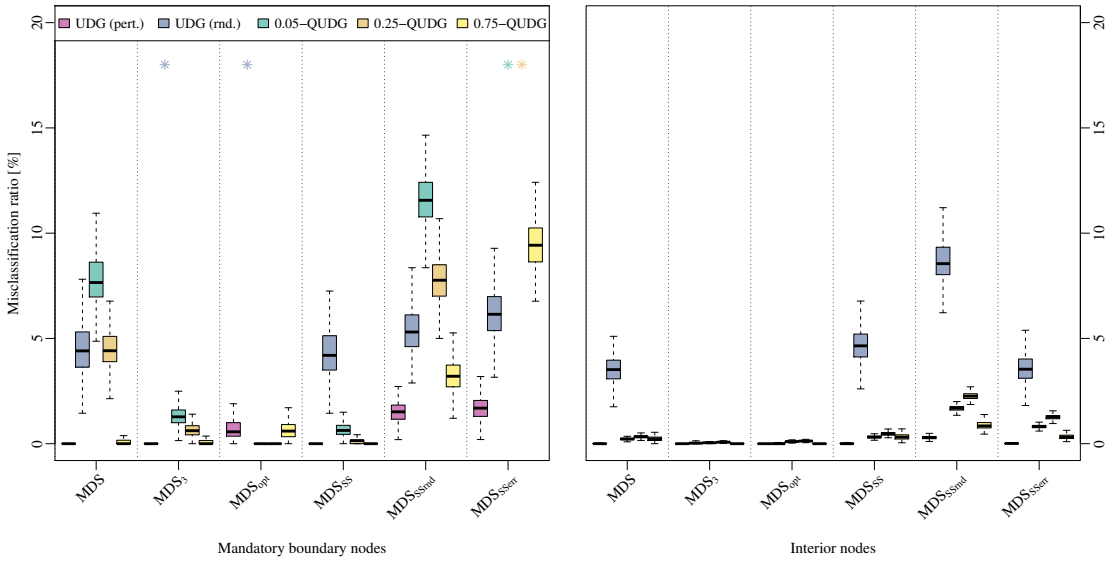


Figure B.10: *Misclassification ratios (false negatives) in percent for MDS-BR2 on different network settings with $d_{avg} = 15$ and multiple graph embedding strategies.*

Table B.14: *Results for MDS-BR1 with $d_{avg} = 12$ and multiple graph embedding strategies. The table corresponds to Figure B.7.*

	Mandatory					Interior					
algorithm	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	rating
MDS	2.9	3.1	35.5	35.3	9.1	0.6	19.0	8.6	8.2	6.9	13.9
MDS_3	2.7	3.1	18.1	19.8	7.7	0.4	17.9	**7.6**	**7.0**	5.9	9.3
MDS_{opt}	0.6	5.3	**1.2**	**1.1**	**1.6**	**0.3**	**17.8**	9.8	9.2	**5.1**	**5.4**
MDS_{SS}	**0.5**	**0.8**	12.7	12.1	2.1	0.8	22.1	9.7	9.0	8.2	8.0
MDS_{SSrnd}	12.6	9.2	25.1	24.7	14.4	3.6	27.0	16.0	15.8	13.9	16.5
MDS_{SSerr}	16.4	10.8	36.3	36.2	25.9	0.6	18.7	11.2	11.6	8.4	18.4

Table B.15: *Results for MDS-BR1 with $d_{avg} = 15$ and multiple graph embedding strategies. The table corresponds to Figure B.8.*

	Mandatory					Interior					
algorithm	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	rating
MDS	3.6	4.0	42.7	43.5	12.1	0.2	11.8	1.9	1.9	1.4	14.1
MDS_3	3.1	3.7	21.0	23.9	9.6	0.2	11.0	**1.5**	**1.5**	1.1	8.0
MDS_{opt}	**0.3**	5.0	**0.7**	**0.7**	**0.6**	**0.1**	**10.7**	2.3	2.3	**1.0**	**2.4**
MDS_{SS}	0.6	**0.9**	13.1	13.1	2.7	0.3	13.9	2.2	2.1	1.7	5.2
MDS_{SSrnd}	16.9	13.1	32.2	31.8	20.0	1.2	17.5	5.1	5.3	3.5	15.4
MDS_{SSerr}	19.3	14.9	47.8	47.8	35.4	0.2	11.4	2.5	2.9	1.6	20.6

Table B.16: *Results for MDS-BR2 with $d_{avg} = 12$ and multiple graph embedding strategies. The table corresponds to Figure 4.30.*

	Mandatory					Interior					
algorithm	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	rating
MDS	**0.0**	4.1	6.3	3.8	0.2	**0.0**	5.5	2.0	2.4	1.8	2.6
MDS_3	**0.0**	21.2	1.2	0.6	0.6	**0.0**	**0.1**	**0.5**	**0.6**	0.1	2.7
MDS_{opt}	0.8	34.7	**0.0**	**0.0**	1.5	**0.0**	**0.1**	0.8	0.9	**0.0**	4.6
MDS_{SS}	**0.0**	**3.8**	0.8	0.2	**0.0**	0.1	7.2	2.6	3.1	2.3	**2.0**
MDS_{SSrnd}	1.3	4.6	7.8	5.2	2.8	0.8	13.0	8.0	9.1	4.7	5.8
MDS_{SSerr}	1.9	5.4	14.3	11.9	8.1	0.1	5.8	5.1	6.5	2.5	6.3

Table B.17: *Results for MDS-BR2 with $d_{avg} = 15$ and multiple graph embedding strategies. The table corresponds to Figure B.10.*

	Mandatory					Interior					
algorithm	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	rating
MDS	**0.0**	4.5	7.8	4.5	0.1	**0.0**	3.5	0.2	0.3	0.2	2.1
MDS_3	**0.0**	22.5	1.3	0.6	0.1	**0.0**	**0.0**	**0.0**	**0.1**	**0.0**	2.7
MDS_{opt}	0.7	30.9	**0.0**	**0.0**	0.7	**0.0**	**0.0**	0.1	**0.1**	**0.0**	3.8
MDS_{SS}	**0.0**	**4.3**	0.7	0.1	**0.0**	**0.0**	4.7	0.3	0.5	0.3	**1.1**
MDS_{SSrnd}	1.5	5.4	11.6	7.8	3.2	0.3	8.6	1.7	2.3	0.9	4.4
MDS_{SSerr}	1.7	6.2	21.6	18.4	9.4	**0.0**	3.6	0.8	1.3	0.3	6.6

Angular Distributions

We present additional angular distributions for all considered embedding strategies.

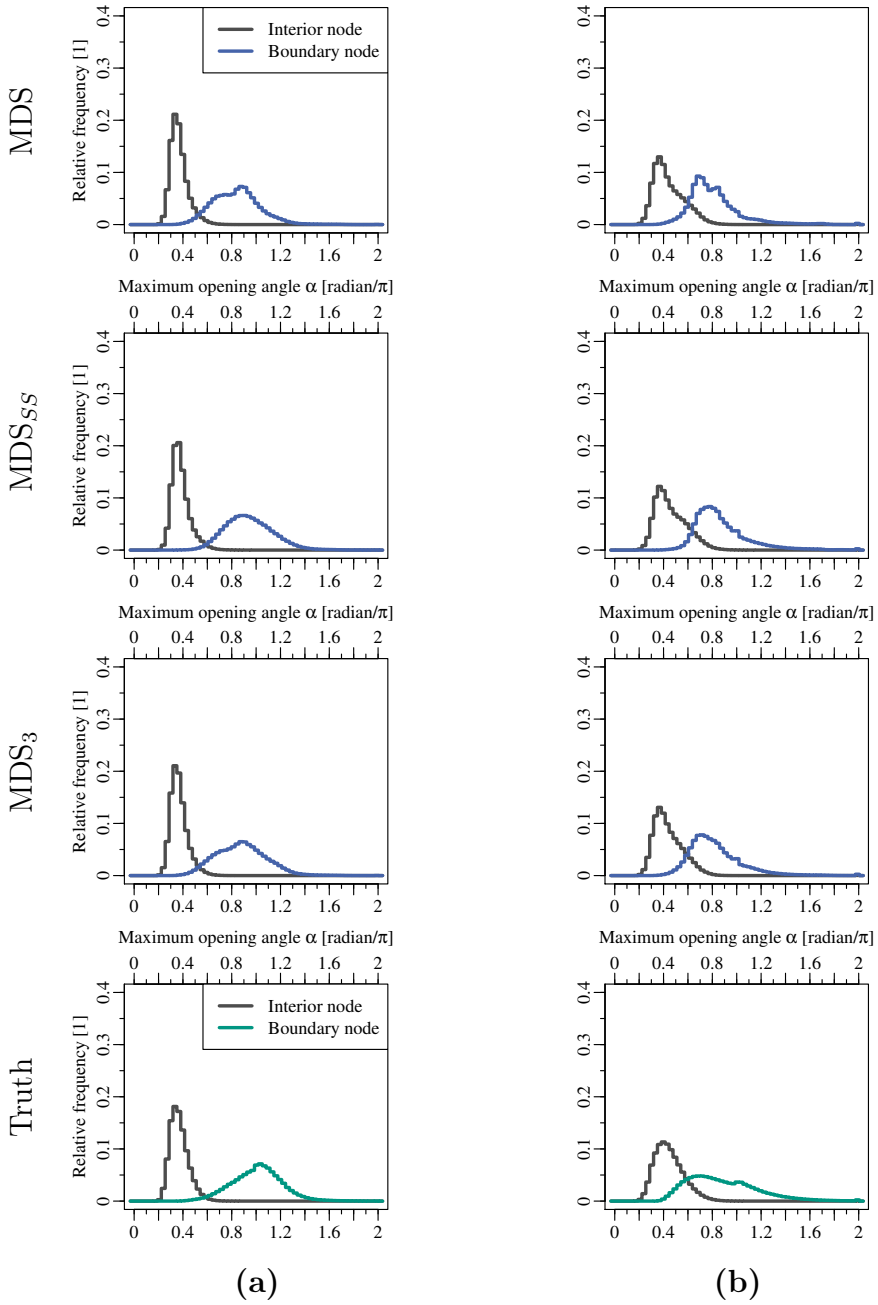


Figure B.11: *Distributions of the maximum opening angle for MDS-BR1 in the UDG model with (a) perturbed grid, and (b) random node placement.*

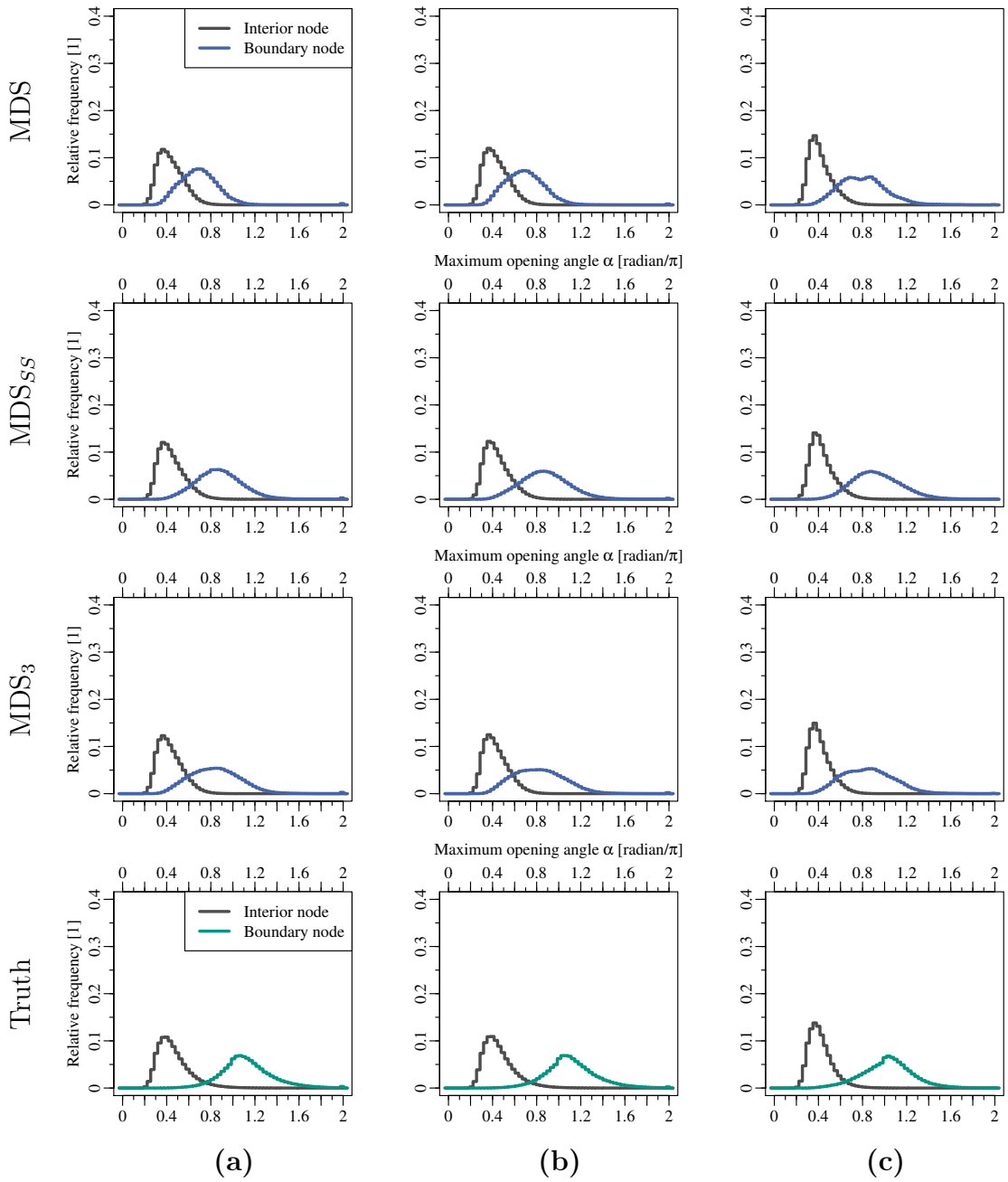


Figure B.12: *Distributions of the maximum opening angle for MDS-BR1 in the (a)* 0.05*-QUDG, (b)* 0.25*-QUDG, and (c)* 0.75*-QUDG model.*

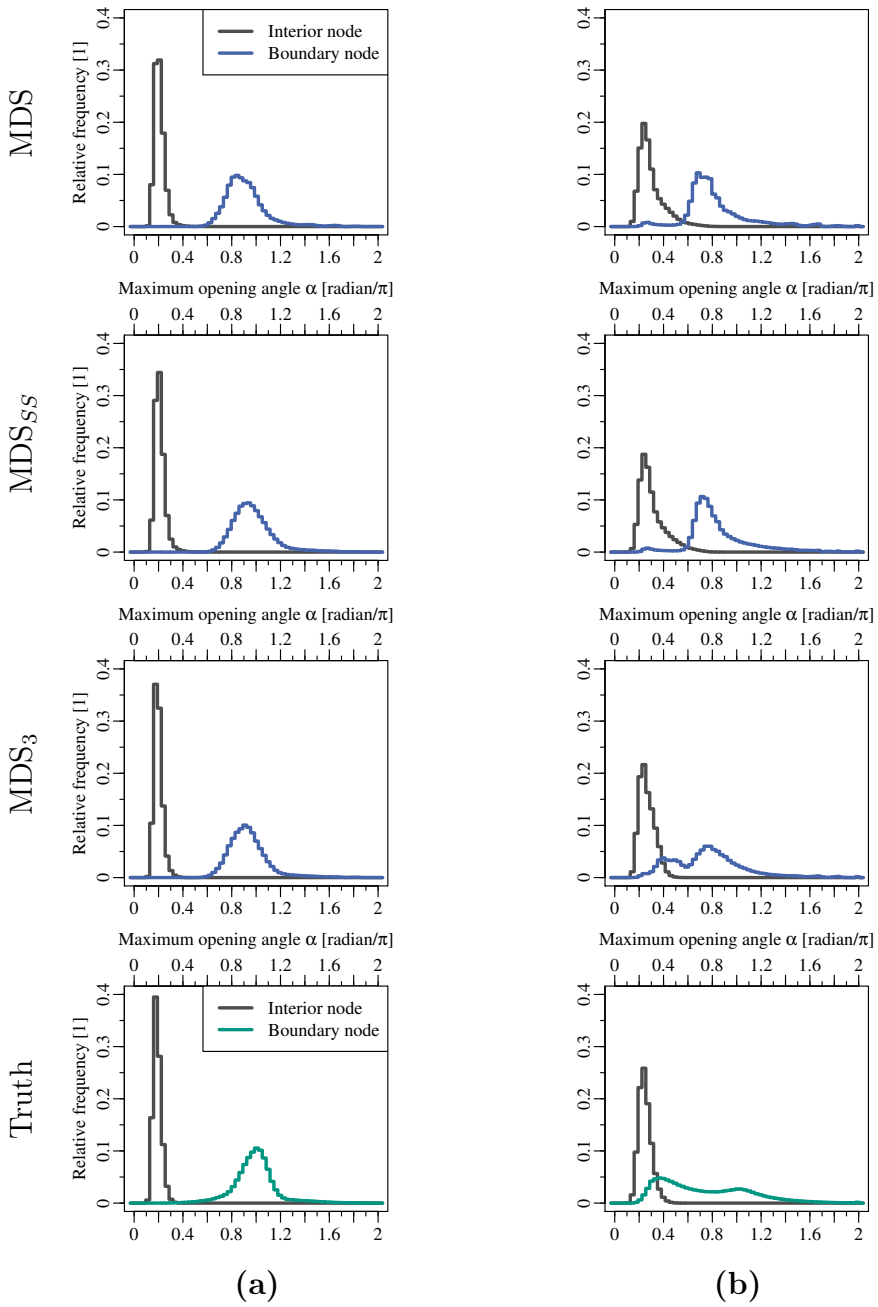


Figure B.13: *Distributions of the maximum opening angle for MDS-BR2 in the UDG model with (a) perturbed grid and (b) random node placement.*

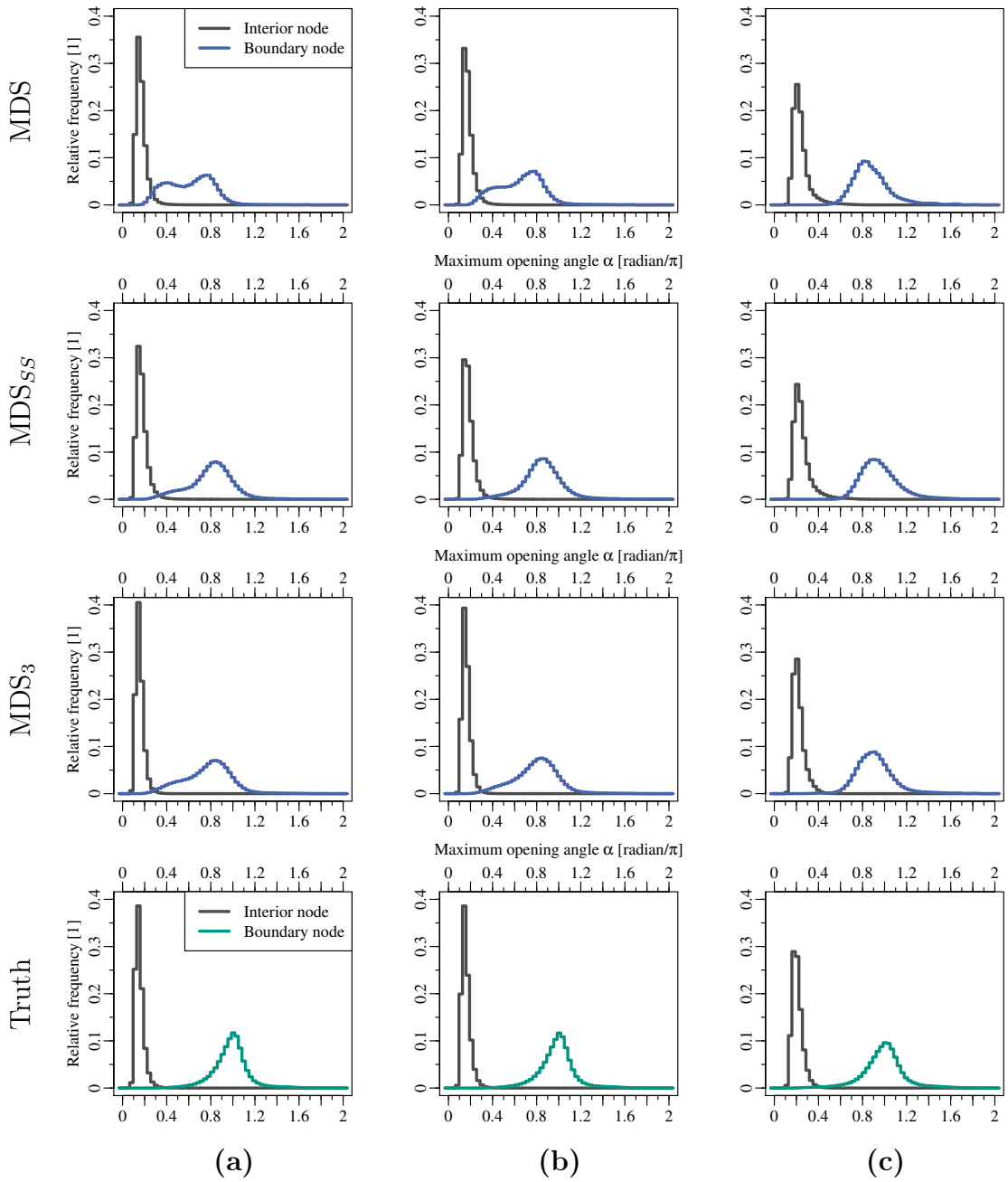


Figure B.14: *Distributions of the maximum opening angle for MDS-BR2 in the (a)* 0.05*-QUDG, (b)* 0.25*-QUDG, and (c)* 0.75*-QUDG model.*

Embedding Quality

We consider maximum opening angles to neighboring nodes and to nodes in 2-hop distance. We list absolute and relative angular errors of our embedding strategies when compared to the true values given by MDS_{opt}.

Table B.18: *Absolute angular errors in multiples of π radians for our embedding strategies on multiple network settings. Values for angles to nodes in 1-hop and 2-hop distance are given.*

		Mandatory					**Interior**				
	embedding strategy	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG
1-hop	MDS	0.63	0.53	1.37	1.34	0.81	0.16	0.21	0.27	0.27	0.22
	MDS_{SS}	0.40	0.42	0.85	0.82	0.50	0.14	0.19	0.22	0.23	0.19
	MDS_3	0.58	0.52	0.98	1.01	0.73	0.16	0.21	0.25	0.26	0.22
2-hop	MDS	0.35	0.58	1.20	1.07	0.46	0.10	0.19	0.11	0.12	0.15
	MDS_{SS}	0.24	0.56	0.66	0.54	0.31	0.09	0.21	0.11	0.12	0.16
	MDS_3	0.31	0.40	0.75	0.68	0.39	0.08	0.11	0.09	0.09	0.10

Table B.19: *Relative angular errors in percentage for our embedding strategies on multiple network settings. Values for angles to nodes in 1-hop and 2-hop distance are given.*

		Mandatory					**Interior**				
	embedding strategy	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG
1-hop	MDS	19.18	18.84	37.14	36.64	24.11	13.46	15.24	17.86	18.54	16.94
	MDS_{SS}	12.71	16.48	22.99	22.74	15.68	12.28	13.45	15.05	15.48	14.95
	MDS_3	17.78	18.51	26.61	27.76	22.03	13.57	15.21	16.64	17.51	16.76
2-hop	MDS	11.74	38.00	37.74	33.83	15.45	16.47	24.31	24.36	24.30	22.56
	MDS_{SS}	8.46	38.70	21.22	17.35	11.01	15.79	26.04	24.33	24.08	23.98
	MDS_3	10.26	22.56	24.17	21.80	12.88	13.31	14.29	18.97	18.76	16.01

Parameter Selection

We present additional heatmaps of the classification results of MDS-BR1. This is followed by tables giving numerical values of all considered settings. Each table entry denotes (1 − the geometric mean of the average correct classification ratios of mandatory boundary nodes and interior nodes). The best result over all parameter values is highlighted in bold.

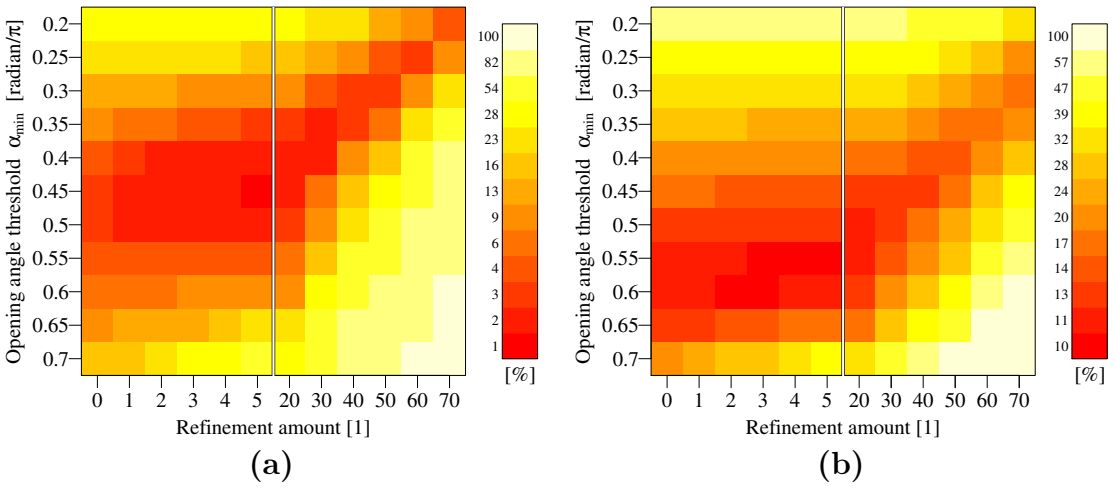


Figure B.15: *Heatmap of classification results for different sets of parameter values for MDS-BR1 in the UDG model: (a) Perturbed grid placement, (b) random placement.*

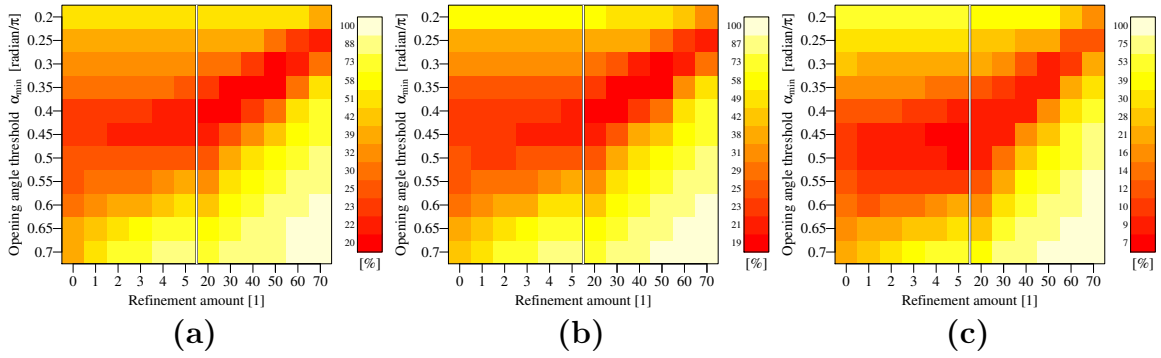


Figure B.16: *Heatmap of classification results for different sets of parameter values for MDS-BR1 on d-QUDGs: (a) $d = 0.05$, (b) $d = 0.25$, (c) $d = 0.75$.*

Table B.20: *Geometric means of the correct classification ratios for mandatory boundary nodes and interior nodes. Listed are* (1 − *geometric mean*) *values in percentage for MDS-BR1 in the UDG model with perturbed grid placement. The table corresponds to Figure B.15(a).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	26.4	26.4	26.3	26.3	26.3	26.4	25.3	22.6	17.7	12.5	6.4	3.7
	0.25	17.3	17.0	16.6	16.3	16.1	16.0	13.9	10.1	6.2	3.6	2.9	6.8
	0.30	11.0	10.2	9.2	8.5	7.9	7.5	6.4	3.8	2.2	2.2	7.5	18.8
	0.35	6.6	5.4	4.3	3.6	3.1	2.7	2.6	1.6	2.4	6.0	19.6	36.8
	0.40	3.8	2.7	2.0	1.6	1.3	1.1	1.3	2.0	6.4	14.8	35.3	54.3
	0.45	2.5	1.8	1.4	1.2	**1.0**	**1.0**	1.2	4.4	13.3	25.7	49.1	67.1
	0.50	2.4	2.0	1.8	1.7	1.6	1.6	2.2	8.7	22.0	36.9	60.0	75.9
	0.55	3.4	3.2	3.1	3.2	3.3	3.4	4.4	15.6	32.2	48.0	68.9	82.0
	0.60	5.5	5.5	5.9	6.3	6.9	7.6	8.7	25.0	43.3	58.8	76.3	86.5
	0.65	8.8	9.5	11.0	12.5	14.5	16.8	16.0	36.6	54.6	68.4	81.8	89.6
	0.70	13.2	15.3	19.3	23.3	27.9	32.7	26.4	48.8	64.6	76.1	86.1	91.8

Table B.21: *Respective results for MDS-BR1 in the UDG model with random node distribution. The table corresponds to Figure B.15(b).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	48.1	48.1	48.1	48.1	48.1	48.2	48.1	47.8	47.0	44.9	39.7	31.5
	0.25	38.9	38.9	38.9	38.9	38.9	39.0	38.6	37.6	35.4	32.0	25.9	19.8
	0.30	31.2	31.2	31.1	31.1	31.1	31.1	30.2	28.4	25.4	21.8	17.3	15.3
	0.35	24.3	24.2	24.0	24.0	24.0	24.0	22.5	20.2	17.4	15.0	14.4	17.3
	0.40	18.6	18.3	18.0	18.0	17.9	17.9	16.3	14.4	13.1	13.4	17.6	24.7
	0.45	14.7	14.2	13.9	13.8	13.6	13.6	12.5	11.7	12.9	16.1	24.2	33.7
	0.50	12.2	11.7	11.4	11.3	11.1	11.1	10.6	11.6	15.5	21.0	31.6	42.3
	0.55	10.8	10.3	10.1	10.0	**9.9**	**9.9**	10.0	13.4	19.8	27.1	39.2	50.1
	0.60	10.5	10.1	10.0	10.0	10.0	10.2	11.2	17.6	26.3	34.7	47.3	58.0
	0.65	12.6	12.5	13.1	13.8	14.6	15.7	16.9	26.7	36.7	45.3	57.3	67.0
	0.70	18.7	20.3	24.2	27.6	31.0	34.7	29.5	41.2	50.8	58.4	68.5	76.3

Table B.22: *Respective results for MDS-BR1 in the* 0.05*-QUDG model. The table corresponds to Figure B.16(a).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	50.5	50.5	50.5	50.5	50.5	50.5	50.5	50.4	49.9	48.3	42.1	32.0
	0.25	39.0	39.0	39.0	39.0	39.0	39.0	38.7	37.7	34.7	29.8	22.9	20.9
	0.30	30.6	30.6	30.6	30.5	30.5	30.5	29.3	26.4	22.0	18.7	20.1	28.5
	0.35	25.4	25.2	25.1	25.0	25.0	25.0	22.5	19.6	**18.0**	19.7	29.6	43.5
	0.40	22.9	22.5	22.3	22.1	22.0	21.8	19.6	19.4	22.4	28.6	43.4	58.2
	0.45	22.5	22.0	21.7	21.5	21.3	21.2	20.5	24.3	31.3	40.4	56.2	69.8
	0.50	23.2	23.3	23.2	23.3	23.7	24.4	24.4	32.3	42.2	52.2	67.1	78.5
	0.55	24.9	26.1	27.0	28.2	30.1	32.7	30.7	42.3	53.5	63.1	76.0	85.1
	0.60	27.7	31.2	34.0	37.4	41.8	46.9	39.6	53.8	64.8	73.1	83.3	89.8
	0.65	32.1	39.3	45.1	51.6	58.6	65.4	51.0	66.0	75.3	81.5	88.9	93.2
	0.70	38.2	50.2	59.5	68.4	76.4	82.7	63.5	76.9	83.8	87.6	92.6	95.3

Table B.23: *Respective results for MDS-BR1 in the* 0.25*-QUDG model. The table corresponds to Figure B.16(b).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	49.1	49.1	49.1	49.1	49.1	49.1	49.1	49.0	48.3	46.4	39.6	29.5
	0.25	37.6	37.6	37.6	37.6	37.6	37.6	37.3	36.1	32.8	27.7	21.0	19.8
	0.30	29.5	29.4	29.4	29.4	29.4	29.4	28.0	24.9	20.5	17.4	19.3	28.2
	0.35	24.5	24.3	24.2	24.1	24.1	24.0	21.5	18.6	**17.1**	19.0	29.3	43.5
	0.40	22.3	21.8	21.5	21.3	21.1	21.0	18.9	18.6	21.8	28.3	43.4	58.4
	0.45	22.1	21.6	21.2	20.9	20.8	20.7	20.0	23.8	31.1	40.3	56.5	70.0
	0.50	23.1	22.9	22.8	23.0	23.6	24.4	24.0	32.1	42.1	52.2	67.3	78.7
	0.55	25.0	25.8	26.7	28.2	30.4	33.2	30.4	42.1	53.4	63.1	76.0	85.0
	0.60	27.9	30.9	33.8	37.6	42.2	47.5	39.3	53.5	64.6	73.0	83.2	89.7
	0.65	32.2	38.5	44.4	51.3	58.5	65.4	50.2	65.3	74.9	81.1	88.6	93.1
	0.70	38.0	48.9	58.3	67.6	75.7	82.1	62.4	75.9	83.1	87.2	92.4	95.1

Table B.24: *Respective results for MDS-BR1 in the* 0.75*-QUDG model. The table corresponds to Figure B.16(c).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	39.2	39.2	39.2	39.2	39.2	39.2	39.0	38.1	35.7	31.3	23.0	14.9
	0.25	28.8	28.8	28.7	28.7	28.7	28.7	27.7	25.1	20.8	16.2	11.0	10.1
	0.30	21.2	21.0	20.7	20.5	20.4	20.3	18.5	15.2	11.5	8.9	9.6	16.4
	0.35	15.6	15.0	14.4	14.0	13.7	13.3	11.9	9.3	7.7	8.5	16.6	29.5
	0.40	11.9	11.0	10.2	9.6	9.0	8.4	8.2	7.2	8.9	14.0	28.6	44.4
	0.45	9.9	9.0	8.2	7.7	6.9	**6.4**	7.1	8.5	14.4	23.3	41.3	57.3
	0.50	9.4	8.5	8.0	7.6	7.1	6.8	7.9	12.8	22.7	34.2	52.8	67.4
	0.55	10.2	9.5	9.3	9.4	9.4	9.8	10.6	19.9	32.9	45.4	62.8	75.3
	0.60	12.1	11.9	12.6	13.6	14.9	16.6	15.7	29.7	44.2	56.5	71.5	81.5
	0.65	15.3	16.3	18.9	21.9	25.5	29.5	23.9	41.6	55.7	66.8	78.8	86.6
	0.70	19.7	22.7	28.8	34.9	41.3	47.7	34.6	53.7	66.0	75.1	84.4	90.3

Table B.25: *Geometric means of the correct classification ratios for mandatory boundary nodes and interior nodes. Listed are (1 − geometric mean) values in percentage for MDS-BR2 in the UDG model with perturbed grid placement. The table corresponds to Figure 4.33(a).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	29.6	29.5	29.4	29.4	29.3	29.4	28.5	26.7	23.4	19.5	13.5	8.2
	0.25	6.5	5.8	4.8	4.3	3.9	3.9	3.9	2.8	1.9	1.2	0.6	0.3
	0.30	1.3	1.0	0.7	0.6	0.5	0.6	0.5	0.3	0.2	0.1	0.1	0.2
	0.35	0.4	0.3	0.2	0.1	0.1	0.3	0.1	0.1	**0.0**	**0.0**	0.1	0.6
	0.40	0.1	0.1	0.1	0.1	0.1	0.2	**0.0**	**0.0**	**0.0**	0.1	0.3	1.2
	0.45	0.1	**0.0**	**0.0**	**0.0**	**0.0**	0.1	**0.0**	**0.0**	**0.0**	0.2	0.8	2.4
	0.50	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	0.1	**0.0**	**0.0**	0.2	0.5	1.8	4.7
	0.55	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	0.1	**0.0**	0.1	0.5	1.3	3.9	8.7
	0.60	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.5	1.3	2.9	7.7	15.4
	0.65	0.8	0.8	0.8	0.8	0.8	1.0	0.9	1.8	3.6	6.5	14.8	25.4
	0.70	2.4	2.5	2.7	3.0	3.3	3.7	3.1	4.9	8.4	13.4	25.4	37.9

Table B.26: *Respective results for MDS-BR2 in the UDG model with random node distribution. The table corresponds to Figure 4.33(b).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	66.7	66.6	66.6	66.6	66.6	66.7	66.6	66.6	66.5	66.2	65.1	62.2
	0.25	37.4	37.4	37.4	37.4	37.4	37.4	37.2	36.8	35.9	34.3	30.9	25.8
	0.30	20.7	20.6	20.5	20.5	20.5	20.5	20.0	19.0	17.4	15.3	12.0	8.8
	0.35	13.5	13.3	13.1	13.1	13.0	13.1	12.4	11.2	9.6	7.9	6.1	5.2
	0.40	9.3	9.1	8.8	8.7	8.7	8.7	8.0	6.9	5.7	4.9	4.8	6.0
	0.45	6.5	6.3	5.9	5.8	5.8	5.8	5.2	4.5	4.2	4.6	6.4	9.6
	0.50	4.8	4.5	4.2	4.1	4.1	4.3	3.8	3.8	4.7	6.4	10.2	15.0
	0.55	4.0	3.7	**3.5**	**3.5**	3.6	3.9	**3.5**	4.6	6.9	9.9	15.3	21.1
	0.60	4.5	4.3	4.3	4.4	4.6	5.0	5.0	7.5	11.1	15.0	21.4	28.0
	0.65	8.4	8.7	9.5	10.2	10.8	11.5	11.2	15.0	19.3	23.5	30.4	37.2
	0.70	16.9	18.5	21.7	24.5	26.8	29.0	22.5	26.7	30.9	34.9	41.6	48.4

Table B.27: *Respective results for MDS-BR2 in the* 0.05*-QUDG model. The table corresponds to Figure 4.34(a).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	9.8	8.8	8.4	8.0	7.6	7.4	5.9	3.9	2.3	1.4	**0.9**	1.4
	0.25	3.8	2.8	2.5	2.2	2.1	2.0	1.8	1.5	1.3	1.5	3.0	7.3
	0.30	4.2	3.7	3.6	3.5	3.5	3.5	3.5	3.6	4.1	5.3	10.4	19.6
	0.35	6.9	6.8	6.7	6.7	6.7	6.7	6.8	7.5	9.2	12.5	22.1	34.8
	0.40	10.7	10.7	10.7	10.8	10.8	10.9	11.1	13.0	16.7	22.4	35.5	49.4
	0.45	14.8	15.0	15.1	15.3	15.6	15.9	16.1	19.7	25.8	33.5	48.2	61.4
	0.50	18.7	19.2	19.5	20.1	21.0	22.0	21.4	27.1	35.4	44.4	59.0	70.8
	0.55	22.7	23.5	24.3	25.6	27.5	29.6	27.2	35.3	45.1	54.4	68.0	77.9
	0.60	26.8	28.2	29.9	32.6	35.8	39.4	34.0	44.4	55.0	63.8	75.4	83.4
	0.65	31.7	34.0	37.6	42.5	47.7	53.1	42.7	54.9	65.0	72.6	81.6	87.6
	0.70	38.1	42.5	49.2	56.8	64.0	70.3	53.9	66.4	74.6	80.2	86.5	90.6

Table B.28: *Respective results for MDS-BR2 in the* 0.25*-QUDG model. The table corresponds to Figure 4.34(b).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	11.9	11.1	10.7	10.3	10.0	9.8	8.1	5.6	3.5	2.1	1.1	0.9
	0.25	3.9	2.9	2.5	2.2	2.0	1.8	1.7	1.1	0.9	**0.8**	1.5	4.2
	0.30	3.1	2.5	2.3	2.2	2.2	2.1	2.1	2.1	2.2	2.7	5.8	12.6
	0.35	4.6	4.4	4.3	4.3	4.3	4.2	4.3	4.5	5.2	7.1	13.8	24.7
	0.40	7.3	7.2	7.2	7.2	7.2	7.2	7.3	8.1	10.1	13.9	24.5	37.8
	0.45	10.4	10.4	10.4	10.5	10.6	10.7	10.8	12.6	16.5	22.4	35.9	49.9
	0.50	13.7	13.8	13.9	14.1	14.4	14.8	14.7	18.1	24.1	31.9	46.9	60.4
	0.55	17.2	17.5	17.8	18.4	19.2	20.2	19.3	24.8	32.9	42.0	57.3	69.4
	0.60	21.1	21.8	22.6	24.1	25.8	28.0	25.2	33.1	43.0	52.6	66.6	76.9
	0.65	26.0	27.3	29.5	32.8	36.4	40.5	33.3	43.8	54.4	63.1	74.7	82.8
	0.70	32.4	35.3	40.3	46.6	52.9	59.1	44.4	56.5	66.0	73.1	81.6	87.2

Table B.29: *Respective results for MDS-BR2 in the* 0.75*-QUDG model. The table corresponds to Figure 4.34(c).*

		r_{min} (MDS-BR refinement)						γ (EC-BR refinement)					
		0	1	2	3	4	5	20%	30%	40%	50%	60%	70%
threshold α_{min} [radian/π]	0.20	42.1	42.1	42.1	42.1	42.1	42.1	41.8	40.9	39.0	36.0	29.9	22.4
	0.25	16.9	16.5	16.0	15.6	15.4	15.2	14.5	12.4	10.0	7.7	4.9	2.9
	0.30	7.2	6.7	6.0	5.6	5.3	5.0	5.1	3.9	2.8	2.0	1.2	1.0
	0.35	3.9	3.5	3.0	2.8	2.6	2.3	2.5	1.9	1.3	1.0	0.9	1.5
	0.40	2.4	2.0	1.8	1.6	1.4	1.2	1.4	1.1	0.8	0.7	1.1	2.7
	0.45	1.5	1.3	1.1	1.0	0.8	0.7	0.9	0.7	0.6	0.8	1.9	4.7
	0.50	1.0	0.8	0.7	0.7	**0.5**	**0.5**	0.6	0.6	0.7	1.3	3.6	8.1
	0.55	0.9	0.8	0.7	0.7	0.6	0.6	0.7	0.8	1.3	2.5	6.6	13.2
	0.60	1.2	1.1	1.1	1.1	1.1	1.1	1.1	1.6	2.8	5.1	11.8	20.9
	0.65	2.5	2.5	2.5	2.6	2.7	2.8	2.7	3.9	6.4	10.4	20.1	31.4
	0.70	5.3	5.4	5.9	6.5	7.1	7.9	6.3	8.9	13.4	19.3	31.6	43.8

Refinement Costs

We present additional figures in case of using MDS-BR2. This is followed by tables giving numerical values of all considered settings. Each table entry shows the average neighborhood size considered by our refinement step in the respective setting.

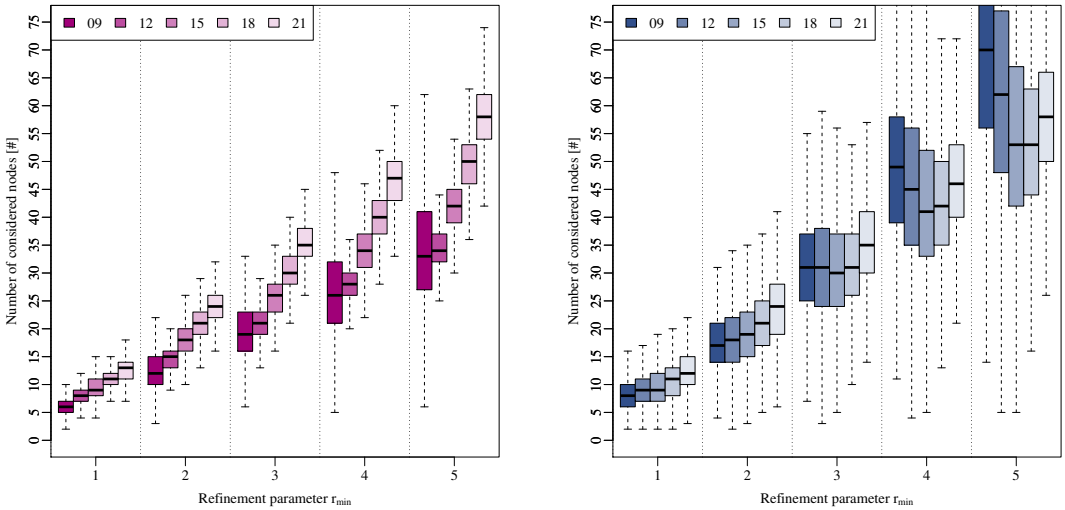


Figure B.17: *Neighborhood size that our refinement strategy considers for MDS-BR2 on UDG networks with average node degrees between 9 and 21 and (a) perturbed grid or (b) random node placement.*

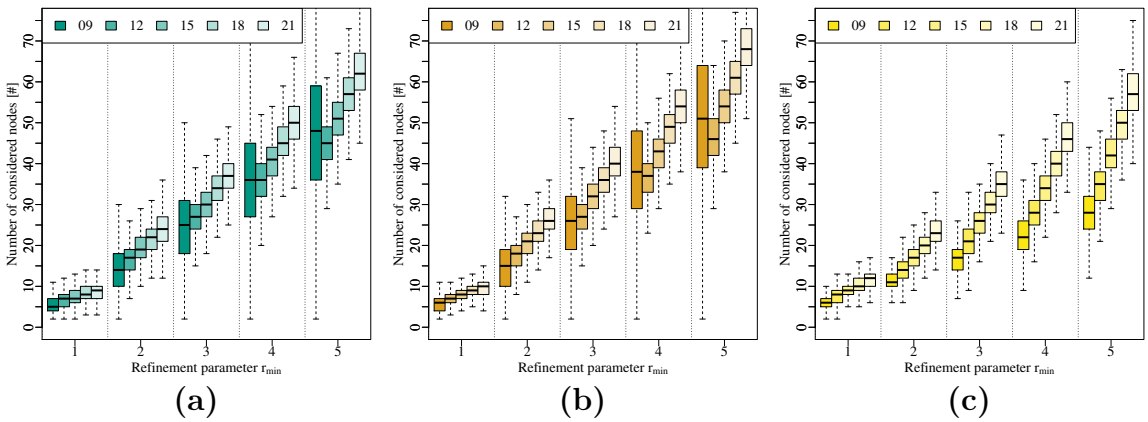


Figure B.18: *Neighborhood size that our refinement strategy considers for MDS-BR2 on d-QUDG networks with average node degrees between 9 and 21 and (a) $d = 0.05$, (b) $d = 0.25$, or (c) $d = 0.75$.*

Table B.30: *Average neighborhood sizes considered by our refinement strategy for MDS-BR1 in the UDG model with average node degrees between* 9 *and* 21*. The table corresponds to Figure 4.35.*

	Perturbed Grid Placement					**Random Placement**				
r_{min}	09	12	15	18	21	09	12	15	18	21
1	4.88	4.40	4.86	5.25	5.65	6.36	6.26	6.05	5.88	5.90
2	9.95	7.47	8.30	9.01	9.78	14.12	13.47	12.23	11.12	10.66
3	15.88	10.51	11.74	12.78	13.92	25.42	23.28	19.69	16.66	15.32
4	21.97	13.49	15.12	16.48	18.03	39.86	34.80	27.49	21.82	19.57
5	28.25	16.45	18.46	20.15	22.11	57.37	47.95	35.62	26.83	23.69

Table B.31: *Respective results for MDS-BR1 in the d-QUDG model with average node degrees between* 9 *and* 21*. The table corresponds to Figure 4.36.*

	0.05-QUDG					**0.25-QUDG**					**0.75-QUDG**				
r_{min}	09	12	15	18	21	09	12	15	18	21	09	12	15	18	21
1	7.43	6.64	5.89	5.34	5.48	7.46	6.56	5.84	5.34	5.49	4.43	4.36	4.57	4.85	5.06
2	26.24	21.48	17.06	13.98	14.13	25.24	20.01	15.92	13.09	13.07	8.66	7.79	7.85	8.31	8.68
3	56.88	43.78	31.84	23.52	22.80	53.88	39.76	28.96	21.45	20.57	13.10	11.01	10.98	11.66	12.21
4	98.38	73.12	49.82	33.71	31.47	92.75	65.53	44.69	30.31	28.07	17.32	13.93	13.92	14.89	15.60
5	150.01	109.69	71.20	44.64	40.31	141.19	97.50	63.19	39.72	35.67	21.46	16.72	16.76	18.01	18.87

Table B.32: *Respective results for MDS-BR2 in the UDG model with average node degrees between* 9 *and* 21*. The table corresponds to Figure B.17.*

	Perturbed Grid Placement					**Random Placement**				
r_{min}	09	12	15	18	21	09	12	15	18	21
1	6.33	7.92	9.52	11.06	12.60	8.10	8.80	9.66	10.92	12.55
2	12.45	14.64	17.76	20.88	24.10	17.68	18.38	19.13	20.80	23.56
3	19.61	21.40	26.01	30.70	35.57	31.24	31.18	30.52	31.64	35.03
4	26.83	28.12	34.25	40.49	47.01	48.77	46.34	42.60	42.38	46.35
5	34.04	34.85	42.50	50.31	58.47	69.95	63.24	54.92	53.05	57.60

Table B.33: *Respective results for MDS-BR2 in the d-QUDG model with average node degrees between* 9 *and* 21*. The table corresponds to Figure B.18.*

	0.05-QUDG					**0.25-QUDG**					**0.75-QUDG**				
r_{min}	09	12	15	18	21	09	12	15	18	21	09	12	15	18	21
1	5.40	6.58	7.50	8.23	8.86	5.70	6.94	8.15	9.13	10.02	5.99	7.53	9.09	10.48	11.90
2	13.98	16.71	19.56	21.95	24.12	14.68	17.16	20.64	23.67	26.41	11.33	14.21	17.43	20.51	23.53
3	24.18	26.32	30.53	34.20	37.50	25.61	26.87	32.08	36.73	40.93	16.92	20.94	25.77	30.47	35.04
4	35.23	35.65	41.13	46.01	50.41	37.81	36.36	43.24	49.45	55.06	22.21	27.58	34.08	40.39	46.50
5	46.84	44.94	51.68	57.78	63.26	50.88	45.81	54.35	62.11	69.13	27.42	34.21	42.38	50.31	57.97

Linear Time Implementation

We show numerical values of all considered settings with MDS-BR2 while using random filtering, contraction-based filtering, and without using filtering.

Table B.34: *Misclassification ratios (false negatives) in percent for different MDS filtering techniques, using MDS-BR2 on multiple network settings with* $d_{avg} = 27$. *The table corresponds to Figure 4.37.*

	Mandatory					**Interior**					
filtering technique	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	UDG (pert.)	UDG (rnd.)	0.05-QUDG	0.25-QUDG	0.75-QUDG	rating
none	**0.0**	0.3	**3.1**	3.1	**0.0**	**0.0**	**0.3**	**0.0**	**0.0**	**0.0**	**0.7**
random	**0.0**	**0.2**	7.3	3.2	**0.0**	3.5	15.5	2.2	2.5	2.2	3.8
contr.-based	**0.0**	0.3	6.6	**3.0**	**0.0**	1.0	5.3	0.7	1.0	0.3	1.8

Appendix C

C Determining Efficient Paths in Large-Scale Sensor Networks

This appendix complements our simulations in Section 5.5. We detail several basic data structures that are used by our algorithms for preprocessing and query. A description of how to generate the sensor network instances that we consider in this thesis follows. We further show how to efficiently compute the quality measures of alternative paths and conclude with an overview of our network instances and further simulational results.

Data Structures

Our algorithms apply several basic data structures whose efficiency is crucial for the performance of our techniques. We therefore provide our own implementations of them and do not rely on available libraries. In the following, we provide a short overview.

Priority Queue. A priority queue is an abstract data structure that maintains a set of elements. Each element is associated with a key that determines its importance. The key data type is required to have a total order on its values. A basic priority queue data structure supports the four operations

`insert`	inserts a new element with an associated key,
`min`	returns the minimum element,
`deleteMin`	removes the element with minimum key,
`size`	returns the number of stored elements.

An *addressable priority queue* allows for the additional operation

`decreaseKey`	decreases the key of a stored element.

Other definitions may merge `min` and `deleteMin` into a single operation or introduce further operations such as merging priority queues, removing arbitrary elements from the priority queue, or an `update` operation that combines `insert` and `decreaseKey`.

This abstract concept is usually implemented with a *heap* data structure that keeps the elements partially ordered according to their associated keys. In our case, we use a *binary heap* [WW64], implemented with an array. It allows for `min` operations in $\mathcal{O}(1)$ time while the other operations have an amortized running time of $\mathcal{O}(\log n)$, with n the number of stored elements. Other implementations such as bucket heaps [DF79], radix heaps [AMOT90], or Fibonacci heaps [FT87] may allow for faster practical runtimes or time complexities, depending on the application. As elements can be large, we store them separately from the heap for performance reasons. The heap only holds keys and references to the elements. The elements are stored in an array for runtime efficiency or in a hash map for memory efficiency. [LS12b] details how we can have both at the same time, runtime efficiency and memory efficiency.

Our query algorithms require an addressable priority queue. We further require persistent data storage to access elements and keys once they have been removed from the queue. As the priority queue is accessed and modified frequently during a query, the respective operations have to be fast. However, the performance of the priority queue becomes less important, the more efficient a speed-up technique is, i.e. the smaller its search space is. Other algorithms such as the preprocessing of Contraction Hierarchies or alternative path queries with X-BDV and its variants also apply priority queues, but their performance is not crucial. Moreover, a simple priority queue without the need for addressability or persistence is sufficient in these cases.

Graph Representation. Our graph data structure is based on an *adjacency array* [MS08]. This graph representation is also known as *forward* or *backward star* representation [AMO93]. The graph is stored as an array of directed edges. The edges are sorted in ascending order by their source, keeping all outgoing edges of a node in succession. An auxiliary array represents nodes and stores indices to the adjacency array that indicate the beginning of the outgoing edges of each node. The end is given implicitly by the beginning of the outgoing edges of the next node or by a sentinel at the end of the auxiliary array. As all outgoing edges of a node are grouped together, edges do not store their source. In case of *dynamic graphs*, we keep empty slots in the adjacency array that can be filled by new edges. The auxiliary array now also has to hold indices that specify the end of the outgoing edges of each node.

This representation comes with the disadvantage of not offering easy access to the incoming edges of a node. We could use a second adjacency array to hold the reverse graph, but we opt to keep the data in a single structure for cache-efficiency. Thus, we store each edge and its reverse edge, differentiating between them by a flag. In case of an undirected edge, we may condense normal and reverse edges to a single edge at their respective source and target nodes.

Our graph data structure is optimized for quick access to the neighbors of each node. It offers access to an array of incoming and outgoing edges for each node that can be scanned cache-efficiently. Figure C.1 provides an example of our graph representation.

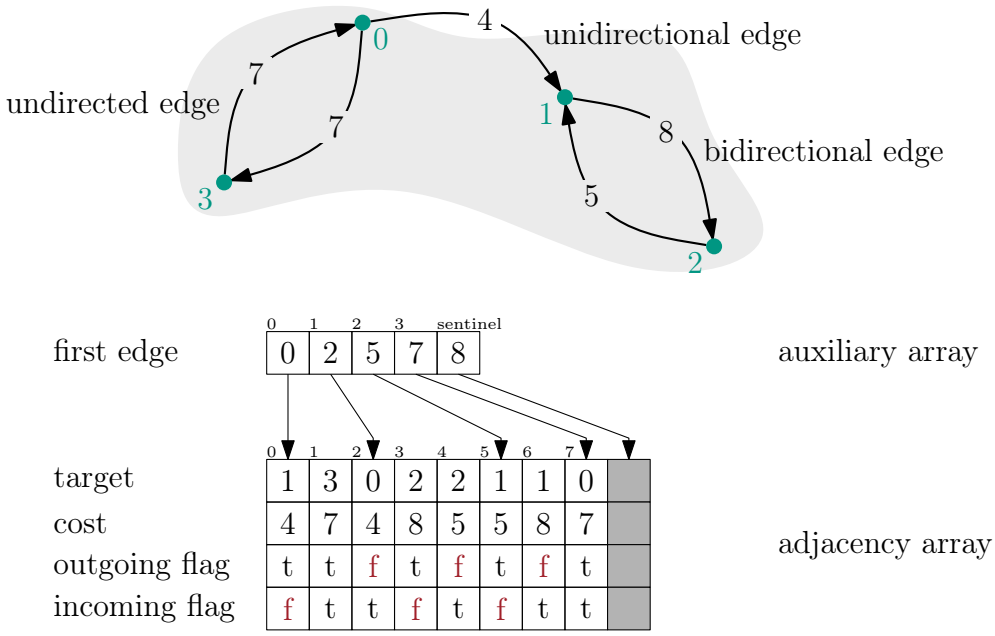


Figure C.1: *Representation of a small sample graph that contains all edge types. Array elements are arranged vertically. Outgoing and incoming flags indicate normal and reverse edges, respectively. This example is borrowed from [Sch08a].*

Sensor Network Modelling

In order to generate a sensor network that reflects roads and population densities for modelling an infrastructural network, we resort to the highly detailed *OpenStreetMap*[1] data that is freely available. OpenStreetMap is a collaborative project to collect geospatial data. The project enjoyed strong growth over the last years, and by now, the collected data is very detailed, sometimes even better than official sources. It is available under the Open Database License (ODbL). One can download a so-called *full history dump* and reconstruct the database at any given timestamp. We apply the OpenStreetMap data of *Regierungsbezirk Karlsruhe*. The respective bounding box and the daily extract are available from Geofabrik[2]. The timestamp of our data revision is May 15th, 2014, 20:55.

To render the probability map shown in Figure 5.17(a), we consider all way data in our OpenStreetMap extract with one of the *highway tag* values listed in Table C.1. We draw the ways on a black canvas of 4000 × 5245 pixels in Mercator projection using a rounded, white brush. The brush size in pixels is listed in Table C.1 for each highway tag value. Next, we shrink the canvas by a factor of 20 using cubic interpolation. Finally, we stretch the picture again to a quadratic size.

[1] `http://www.openstreetmap.org/`. Accessed: 2014-08-06
[2] `http://www.geofabrik.de/`. Accessed: 2014-08-06

The reasoning for our procedure is that population density is correlated with road density. This includes footways and other paths that are not useable by cars. Drawing all roads in a resolution that has them clearly distinct and scaling the resulting picture allows us to quickly approximate a density map with shades of grey for different densities. We put more emphasis on important roads by drawing them wider as we expect additional infrastructure along them.

Table C.1: *Highway tag values that are considered when extracting OpenStreetMap way data. Stroke width denotes the brush size in pixel that is used to draw the way.*

highway tag	stroke width	highway tag	stroke width
`motorway`	20	`motorway_link`	20
`trunk`	20	`trunk_link`	20
`primary`	20	`primary_link`	20
`secondary`	1	`secondary_link`	1
`tertiary`	1	`tertiary_link`	1
`unclassified`	1	`residential`	1
`living_street`	1	`service`	1
`services`	1	`bridleway`	1
`road`	1	`track`	1
`raceway`	1	`path`	1
`cycleway`	1	`steps`	1
`footway`	1	`pedestrian`	1

Alternative Path Quality

In the literature, the quality of an alternative path $P_{s,v,t}$ is measured by three values, uniformly bounded stretch, sharing the with shortest path $P_{s,t}$, and detour-based local optimality. Abraham et al. [ADGW13] who introduced these values do not describe how to obtain them efficiently, and Bader et al. [BDGS11] report that computing them is very time-consuming. Here, we briefly introduce an efficient method.

First, we determine the sets $X = P_{s',v}$ and $Y = P_{v,t'}$ of nodes on $P_{s,v,t}$ with s' and t' the nodes at which the shortest path $P_{s,t}$ and the alternative path $P_{s,v,t}$ diverge and converge again. Due to the nature of via paths, this happens only once. Next, we run two many-to-many queries, one from all nodes in X to v and one from v to all nodes in Y to obtain shortest path distances. We apply the many-to-many technique of [KSS+07] with CHASE for a considerable speed-up. Given these distances, we scan over all pairs $\{x, y\}$, $x \in X$, $y \in Y$, to determine the maximum stretch of all paths $R = \{P_{x,v,y} \mid x \in X, y \in Y\}$. We find the largest distance for which the paths in R remain locally optimal, i.e. the largest value with $c(P_{x,y}) = c(P_{x,v}) + c(P_{v,y})$, during the same scanning operation. Sharing is then obtained trivially as $c(P_{s,s'}) + c(P_{t',t})$. We can scan the shortest path distances in an order that in practice only requires us to consider 25% of all pairs by exploiting subpath optimality. Moreover, this approach is cache-efficient. In our simulations, computing statistics for a single alternative path takes about 150 ms on average.

Network Instances

Our simulations in Chapter 5 consider four distinct network instances with three different edge cost models each. Table C.2 lists basic information for them and for the applied partitionings. These values are obviously independent of the underlying edge cost model. We state the number of sensor nodes and communication links, i.e. edges, in each graph. For each partitioning, we give the edge cut, the number of boundary nodes, and the number of neighboring pairs of regions. Pairs are ordered, i.e. we count $\{1, 2\}$ and $\{2, 1\}$ separately. We further state the number of fine regions that overlap each coarse region at least partially. Recall that our fine partitioning does not respect the coarse partitioning.

Partitioning is done in parallel on Machine B using KaFFPaE [SS12b]. We allow for a maximum imbalance of 3%, apply the economical preconfiguration setting, and set a time limit of 2 hours. Using faster settings for KaFPPaE or plain METIS [KK99] results in a roughly 10–20% higher edge cut value, but partitioning times become negligible at mere seconds. As changes in the edge cut value translate only linearly to the required preprocessing time for computing arc flags and via node candidate sets, we would experience only a slight increase. Thus, we opt to entirely ignore partitioning times in our statistics.

Table C.2: *Network sizes of all considered settings. Numbers of sensor nodes and communication links are listed. All communication links are undirected. Statistics of the coarse (*128 *regions) and fine partitioning (*1 024 *regions) are listed as well.*

		Graph Data		**Coarse Partitioning**			**Fine Partitioning**			
type	degree	avg. nodes	links	edge cut	border nodes	neighbor pairs	edge cut	border nodes	neighbor pairs	avg. overlap
box	10	999 807	9 987 756	21 681	26 597	686	73 785	85 591	6 006	13.9
	20	1 000 000	19 954 602	110 688	66 433	680	355 975	199 537	5 934	14.3
road	10	854 491	9 440 060	10 753	11 253	574	65 760	58 785	4 878	12.8
	20	991 714	19 922 762	53 358	33 953	716	266 647	136 089	6 054	13.8

Preliminary Simulations

We examine basic properties of three shortest path algorithms that are prominently featured in our simulations on both of our machines. Table C.3 compares their performance in our default network setting (network type "road", $d_{avg} = 10$, and $p = 1$). CH-based algorithms use exact queries. We report on their query times on both of our machines to better assess their respective single-core performance. We further state the average number of settled nodes for target queries and for exploration queries, the latter with ($x = 3$) and without using relaxation ($x = 0$) for CH-based algorithms. The exploration query visits the entire search space for CH-based techniques, while the variant using a bidirectional Dijkstra's algorithm only explores the network up to a distance of $(1 + \epsilon)$ times the respective shortest path distance.

Table C.3: *Performance of our main shortest path algorithms on both machines in terms of query times. Average search space sizes of target queries and of exploration queries with $x \in \{0, 3\}$ are also listed for the respective algorithms. Values are reported for our default network setting.*

	Time [ms]		Settled Nodes [#]		
algorithm	machine A	machine B	target query	exploration query	with relaxation
Bidir. Dijkstra	183.352	220.461	307 228	875 135	–
CH	0.507	0.569	872	3 722	9 461
CHASE	0.020	0.023	36	39	937

We observe that machine A offers about 10–20% more single-core query performance than machine B. The difference is more pronounced for the bidirectional variant of Dijkstra's algorithm than for the CH-based methods as more memory accesses are required. The number of settled nodes in a CHASE exploration query is virtually the same as in a target query. Even when using relaxation, it settles only about the same amount of nodes as a target query with Contraction Hierarchies. As the latter approach settles up to 100 times as many nodes during exploration, it is obviously much more likely to find appropriate via nodes. Overall, CHASE is not suitable to be used for exploration as already argued in Section 5.4.1 since it considers only few nodes that are not already on the shortest path.

Approximate Queries

In the following, Tables C.4–C.9 provide performance values of all considered algorithms for all considered network settings. The results of Table C.5 correspond to Table 5.1 in Section 5.5.2. We repeat them here for the sake of completeness.

The general results that have been discussed in the main body of this thesis hold true for all considered network settings. The latency cost model ($p = 1$) is the most demanding edge cost model, for instance due to many similar but not equivalent paths that have to be explored by the queries. Network type "road" is easier to process than network type "box", probably due to it offering at least a slight hierarchical structure, whereas the networks of type "box" possess a relatively uniform distribution of nodes. Moreover, the network of type "road" has about 15% less nodes in the $d_{avg} = 10$ setting. Finally, networks with a high average node degree ($d_{avg} = 20$) obviously take longer to process than networks with a small average node degree ($d_{avg} = 10$). The reason for this is twofold. First, the respective graphs have twice as many edges, but more importantly, each node has twice as many neighbors to consider. Thus, considering just the 2-hop neighborhood of a node can be four times as costly in the dense settings than in the sparse ones. Our techniques, especially CHASE, still fare fairly well, though, even for the most demanding network settings.

Table C.4: *Performance of all considered (exact and approximate) algorithms using the network type "box" and the hop count cost model ($p = 0$).*

		Prepro.		**Query**		**Prepro.**		**Query**	
		time [s]	overhead [B/n]	time [ms]	error [%]	time [s]	overhead [B/n]	time [ms]	error [%]
	algorithm	node degree $d_{avg} = 10$				node degree $d_{avg} = 20$			
	Bidir. Dijkstra	0	0	129.224	–	0	0	178.440	–
	Arc Flags	1 134	94	5.917	–	5 139	202	21.693	–
	Core-ALT	202	164	2.085	–	752	431	7.368	–
ALT	exact	193	512	4.370	–	234	512	7.918	–
ALT	apx ($\epsilon = 0.01$)	193	512	3.491	0.2	234	512	6.032	0.2
ALT	apx ($\epsilon = 0.10$)	193	512	2.238	1.4	234	512	3.416	1.6
ALT	apx ($\epsilon = 0.21$)	193	512	1.941	2.3	234	512	2.891	2.5
CH	exact (nlu)	161	−28	0.575	–	1 029	−61	1.728	–
CH	exact	86	−26	0.648	–	462	−60	1.804	–
CH	apx ($\epsilon = 0.01$)	86	−26	0.666	0.0	472	−60	1.903	0.0
CH	apx ($\epsilon = 0.10$)	83	−26	0.852	0.6	482	−60	2.487	0.2
CHASE	exact	2 449	42	0.025	–	8 596	75	0.101	–
CHASE	apx ($\epsilon = 0.01$)	2 448	43	0.025	0.0	8 564	80	0.102	0.0
CHASE	apx ($\epsilon = 0.10$)	2 363	49	0.024	0.6	8 332	93	0.092	0.2
CHALT	exact	103	−0	0.153	–	491	−34	0.513	–
CHALT	apx ($\epsilon = 0.01$)	103	−0	0.152	0.0	501	−34	0.517	0.0
CHALT	apx ($\epsilon = 0.10$)	100	−1	0.174	0.6	511	−34	0.544	0.2
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	100	−1	0.096	1.9	511	−34	0.383	1.9

Table C.5: *Performance of all considered (exact and approximate) algorithms using the network type "box" and the latency cost model ($p = 1$).*

		Prepro.		**Query**		**Prepro.**		**Query**	
		time [s]	overhead [B/n]	time [ms]	error [%]	time [s]	overhead [B/n]	time [ms]	error [%]
	algorithm	node degree $d_{avg} = 10$				node degree $d_{avg} = 20$			
	Bidir. Dijkstra	0	0	156.804	–	0	0	231.287	–
	Arc Flags	1 363	95	2.373	–	9 279	182	4.658	–
	Core-ALT	205	164	1.474	–	778	432	4.850	–
ALT	exact	215	512	2.985	–	268	512	4.756	–
ALT	apx ($\epsilon = 0.01$)	215	512	1.714	0.1	268	512	1.947	0.2
ALT	apx ($\epsilon = 0.10$)	215	512	1.310	1.0	268	512	1.441	1.2
ALT	apx ($\epsilon = 0.21$)	215	512	1.262	1.9	268	512	1.408	1.9
CH	exact (nlu)	5 415	−2	2.325	–	266 749	29	14.643	–
CH	exact	895	1	2.477	–	30 492	30	14.831	–
CH	apx ($\epsilon = 0.01$)	400	−5	2.238	0.2	8 967	−8	12.615	0.2
CH	apx ($\epsilon = 0.10$)	177	−19	2.182	2.2	1 277	−51	7.223	1.8
CHASE	exact	6 216	97	0.042	–	62 761	243	0.225	–
CHASE	apx ($\epsilon = 0.01$)	4 878	86	0.038	0.2	27 863	169	0.167	0.2
CHASE	apx ($\epsilon = 0.10$)	3 120	61	0.028	2.2	9 540	84	0.073	1.8
CHALT	exact	927	26	0.415	–	30 586	56	2.522	–
CHALT	apx ($\epsilon = 0.01$)	427	20	0.351	0.2	9 028	17	1.798	0.2
CHALT	apx ($\epsilon = 0.10$)	198	7	0.292	2.2	1 312	−25	0.843	1.8
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	198	7	0.124	3.6	1 312	−25	0.379	3.4

Table C.6: *Performance of all considered (exact and approximate) algorithms using the network type "box" and the energy consumption cost model ($p = 2$).*

		Prepro.		**Query**		**Prepro.**		**Query**	
		time [s]	overhead [B/n]	time [ms]	error [%]	time [s]	overhead [B/n]	time [ms]	error [%]
	algorithm	node degree $d_{avg} = 10$				node degree $d_{avg} = 20$			
	Bidir. Dijkstra	0	0	170.320	–	0	0	258.244	–
	Arc Flags	1 107	88	2.588	–	4 572	169	3.496	–
	Core-ALT	212	169	1.631	–	774	435	4.558	–
ALT	exact	230	512	3.141	–	301	512	4.552	–
ALT	apx ($\epsilon = 0.01$)	230	512	2.443	0.1	301	512	3.551	0.1
ALT	apx ($\epsilon = 0.10$)	230	512	1.972	1.1	301	512	2.866	1.2
ALT	apx ($\epsilon = 0.21$)	230	512	1.898	3.0	301	512	2.746	3.4
CH	exact (nlu)	277	−21	0.523	–	965	−61	0.562	–
CH	exact	115	−19	0.560	–	465	−61	0.599	–
CH	apx ($\epsilon = 0.01$)	103	−20	0.545	0.1	451	−62	0.583	0.1
CH	apx ($\epsilon = 0.10$)	88	−25	0.646	1.2	446	−67	0.692	1.0
CHASE	exact	3 035	52	0.015	–	6 675	49	0.016	–
CHASE	apx ($\epsilon = 0.01$)	2 902	50	0.014	0.1	6 392	46	0.015	0.1
CHASE	apx ($\epsilon = 0.10$)	2 408	41	0.012	1.2	5 611	37	0.014	1.0
CHALT	exact	134	7	0.143	–	492	−35	0.163	–
CHALT	apx ($\epsilon = 0.01$)	122	6	0.133	0.1	477	−36	0.155	0.1
CHALT	apx ($\epsilon = 0.10$)	105	1	0.128	1.2	470	−41	0.145	1.0
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	105	1	0.072	2.4	470	−41	0.089	2.2

Table C.7: *Performance of all considered (exact and approximate) algorithms using the network type "road" and the hop count cost model ($p = 0$).*

		Prepro.		**Query**		**Prepro.**		**Query**	
		time [s]	overhead [B/n]	time [ms]	error [%]	time [s]	overhead [B/n]	time [ms]	error [%]
	algorithm	node degree $d_{avg} = 10$				node degree $d_{avg} = 20$			
	Bidir. Dijkstra	0	0	118.149	–	0	0	197.202	–
	Arc Flags	500	92	2.720	–	2 620	176	11.859	–
	Core-ALT	237	170	5.249	–	513	328	10.219	–
ALT	exact	164	512	7.153	–	228	512	10.325	–
ALT	apx ($\epsilon = 0.01$)	164	512	6.523	0.2	228	512	8.592	0.2
ALT	apx ($\epsilon = 0.10$)	164	512	3.872	1.3	228	512	4.518	1.6
ALT	apx ($\epsilon = 0.21$)	164	512	3.218	2.5	228	512	3.790	3.0
CH	exact (nlu)	144	−36	0.170	–	1 416	−66	0.774	–
CH	exact	92	−34	0.177	–	696	−65	0.822	–
CH	apx ($\epsilon = 0.01$)	92	−34	0.183	0.0	696	−65	0.848	0.0
CH	apx ($\epsilon = 0.10$)	94	−34	0.202	0.7	710	−65	1.016	0.4
CHASE	exact	1 025	25	0.012	–	5 289	45	0.059	–
CHASE	apx ($\epsilon = 0.01$)	1 026	25	0.012	0.0	5 289	46	0.059	0.0
CHASE	apx ($\epsilon = 0.10$)	997	25	0.012	0.7	5 178	50	0.057	0.4
CHALT	exact	106	−8	0.120	–	723	−39	0.392	–
CHALT	apx ($\epsilon = 0.01$)	106	−8	0.121	0.0	723	−39	0.396	0.0
CHALT	apx ($\epsilon = 0.10$)	107	−8	0.138	0.7	737	−39	0.416	0.4
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	107	−8	0.100	1.2	737	−39	0.283	1.6

Table C.8: *Performance of all considered (exact and approximate) algorithms using the network type "road" and the latency cost model ($p = 1$).*

		Prepro.		**Query**		**Prepro.**		**Query**	
		time [s]	overhead [B/n]	time [ms]	error [%]	time [s]	overhead [B/n]	time [ms]	error [%]
	algorithm	node degree $d_{avg} = 10$				node degree $d_{avg} = 20$			
	Bidir. Dijkstra	0	0	141.151	–	0	0	249.750	–
	Arc Flags	522	92	0.835	–	3 219	169	3.301	–
	Core-ALT	239	170	2.702	–	527	328	4.904	–
ALT	exact	178	512	3.666	–	255	512	5.670	–
ALT	apx ($\epsilon = 0.01$)	178	512	3.047	0.0	255	512	3.280	0.1
ALT	apx ($\epsilon = 0.10$)	178	512	2.354	1.1	255	512	2.127	1.6
ALT	apx ($\epsilon = 0.21$)	178	512	2.342	2.5	255	512	2.106	2.8
CH	exact (nlu)	1 071	−14	0.482	–	30 118	−3	3.006	–
CH	exact	254	−10	0.440	–	4 942	−0	3.001	–
CH	apx ($\epsilon = 0.01$)	166	−17	0.381	0.2	1 923	−30	2.635	0.2
CH	apx ($\epsilon = 0.10$)	116	−29	0.322	2.1	934	−60	1.972	2.0
CHASE	exact	2 086	72	0.013	–	17 521	169	0.066	–
CHASE	apx ($\epsilon = 0.01$)	1 680	59	0.011	0.2	10 161	110	0.055	0.2
CHASE	apx ($\epsilon = 0.10$)	1 147	34	0.009	2.1	5 510	51	0.032	2.0
CHALT	exact	275	16	0.244	–	5 004	26	1.019	–
CHALT	apx ($\epsilon = 0.01$)	185	9	0.213	0.2	1 968	−4	0.764	0.2
CHALT	apx ($\epsilon = 0.10$)	131	−3	0.171	2.1	964	−34	0.467	2.0
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	131	−3	0.109	2.8	964	−34	0.233	3.3

Table C.9: *Performance of all considered (exact and approximate) algorithms using the network type "road" and the energy consumption cost model ($p = 2$).*

		Prepro.		**Query**		**Prepro.**		**Query**	
		time [s]	overhead [B/n]	time [ms]	error [%]	time [s]	overhead [B/n]	time [ms]	error [%]
	algorithm	node degree $d_{avg} = 10$				node degree $d_{avg} = 20$			
	Bidir. Dijkstra	0	0	148.387	–	0	0	264.966	–
	Arc Flags	477	91	1.022	–	2 376	164	2.070	–
	Core-ALT	239	175	4.089	–	526	332	8.196	–
ALT	exact	191	512	5.155	–	286	512	8.290	–
ALT	apx ($\epsilon = 0.01$)	191	512	4.578	0.0	286	512	7.525	0.0
ALT	apx ($\epsilon = 0.10$)	191	512	4.186	0.7	286	512	6.691	0.8
ALT	apx ($\epsilon = 0.21$)	191	512	4.174	2.5	286	512	6.615	3.0
CH	exact (nlu)	204	−33	0.109	–	1 724	−67	0.188	–
CH	exact	113	−30	0.103	–	812	−65	0.198	–
CH	apx ($\epsilon = 0.01$)	111	−31	0.102	0.1	809	−66	0.195	0.1
CH	apx ($\epsilon = 0.10$)	111	−34	0.099	1.2	847	−70	0.207	1.1
CHASE	exact	1 116	31	0.006	–	4 707	34	0.009	–
CHASE	apx ($\epsilon = 0.01$)	1 096	29	0.006	0.1	4 560	32	0.009	0.1
CHASE	apx ($\epsilon = 0.10$)	950	23	0.006	1.2	4 132	25	0.008	1.1
CHALT	exact	126	−5	0.077	–	836	−40	0.118	–
CHALT	apx ($\epsilon = 0.01$)	124	−5	0.076	0.1	832	−40	0.116	0.1
CHALT	apx ($\epsilon = 0.10$)	123	−8	0.074	1.2	869	−44	0.115	1.1
CHALT	apx ($\epsilon, \epsilon' = 0.10$)	123	−8	0.061	1.5	869	−44	0.096	1.5

Alternative Connections

Tables C.10–C.17 compile all query and preprocessing results on computing alternative paths in our default network setting (network type "road", average node degree $d_{avg} = 10$) under all edge cost models ($p \in \{0, 1, 2\}$). Plots of the respective success rates follow.

Table C.10: *Query performance for computing alternative paths under the hop count cost model ($p = 0$). Relaxation is not used ($x = 0$). Exact and approximate algorithms ($\epsilon \in \{0, 0.1, 0.01\}$) are considered.*

		Performance		Path Quality			Candidate Sets		
		time	success	UBS	sharing	locality	via.cand.	fallback	tested
a	algorithm	[ms]	rate [%]	[%]	[%]	[%]	rate [%]	rate [%]	[#]
1	X-BDV	769.008	100.0	–	33.2	44.0	–	–	–
	X-CHV	1.682	89.5	3.5	41.0	69.7	–	–	–
	X-CHASEV	0.862	89.4	3.2	40.9	69.8	–	–	–
	single-level	0.195	91.7	2.8	43.5	69.6	93.7	4.7	1.6
	multi-level	0.178	91.9	2.8	43.6	69.6	97.7	0.5	1.6
	X-CHASEV ($\epsilon = 0.01$)	0.829	87.4	2.6	39.8	70.2	–	–	–
	single-level ($\epsilon = 0.01$)	0.189	90.2	2.6	42.3	70.0	92.5	4.7	1.6
	multi-level ($\epsilon = 0.01$)	0.181	87.6	3.0	43.0	69.0	96.2	0.5	1.7
	X-CHASEV ($\epsilon = 0.10$)	0.780	84.1	3.2	39.7	69.1	–	–	–
	single-level ($\epsilon = 0.10$)	0.189	90.2	2.6	42.3	70.0	92.5	4.7	1.6
	multi-level ($\epsilon = 0.10$)	0.181	87.6	3.0	43.0	69.0	96.2	0.5	1.7
2	X-BDV	915.741	99.9	–	51.6	34.3	–	–	–
	X-CHV	2.958	80.3	4.8	54.5	64.8	–	–	–
	X-CHASEV	1.333	80.4	4.5	54.5	64.9	–	–	–
	single-level	0.383	84.2	4.7	54.6	64.7	93.6	4.1	2.2
	multi-level	0.367	84.4	4.5	54.6	64.7	97.2	0.4	2.2
	X-CHASEV ($\epsilon = 0.01$)	1.241	76.8	5.6	53.5	64.0	–	–	–
	single-level ($\epsilon = 0.01$)	0.372	81.8	3.8	54.2	64.7	91.4	4.2	2.2
	multi-level ($\epsilon = 0.01$)	0.376	80.2	3.4	54.3	63.6	93.8	0.5	2.4
	X-CHASEV ($\epsilon = 0.10$)	1.221	74.7	4.3	54.2	63.1	–	–	–
	single-level ($\epsilon = 0.10$)	0.372	81.8	3.8	54.2	64.7	91.4	4.2	2.2
	multi-level ($\epsilon = 0.10$)	0.376	80.2	3.4	54.3	63.6	93.8	0.5	2.4
3	X-BDV	1 041.846	99.3	–	65.2	28.9	–	–	–
	X-CHV	4.520	73.3	5.4	60.4	60.6	–	–	–
	X-CHASEV	1.906	73.1	5.0	60.4	60.8	–	–	–
	single-level	0.630	79.0	5.0	60.0	60.9	93.8	3.9	3.1
	multi-level	0.616	79.2	4.6	59.9	61.2	97.3	0.4	3.1
	X-CHASEV ($\epsilon = 0.01$)	1.792	71.5	3.7	61.3	62.0	–	–	–
	single-level ($\epsilon = 0.01$)	0.625	76.9	4.2	60.2	61.7	90.8	4.0	3.2
	multi-level ($\epsilon = 0.01$)	0.666	75.2	4.2	58.7	59.9	96.2	0.4	3.8
	X-CHASEV ($\epsilon = 0.10$)	1.816	66.8	4.9	59.9	59.6	–	–	–
	single-level ($\epsilon = 0.10$)	0.625	76.9	4.2	60.2	61.7	90.8	4.0	3.2
	multi-level ($\epsilon = 0.10$)	0.666	75.2	4.2	58.7	59.9	96.2	0.4	3.8

Table C.11: *Query performance for computing alternative paths under the latency cost model ($p = 1$). Relaxation is not used ($x = 0$). Exact and approximate algorithms ($\epsilon \in \{0, 0.1, 0.01\}$) are considered.*

		Performance		**Path Quality**			**Candidate Sets**		
		time	success	UBS	sharing	locality	via.cand.	fallback	tested
a	algorithm	[ms]	rate [%]	[%]	[%]	[%]	rate [%]	rate [%]	[#]
1	X-BDV	871.229	98.5	7.7	29.2	63.5	–	–	–
	X-CHV	7.354	91.7	5.3	40.1	67.1	–	–	–
	X-CHASEV	3.936	91.7	5.3	40.1	67.1	–	–	–
	single-level	0.424	93.6	5.8	42.1	65.7	94.2	4.7	1.8
	multi-level	0.353	93.8	5.8	42.2	65.7	98.2	0.5	1.9
	X-CHASEV ($\epsilon = 0.01$)	2.853	89.9	5.3	39.8	63.9	–	–	–
	single-level ($\epsilon = 0.01$)	0.396	91.3	5.6	43.3	63.6	94.2	4.7	2.1
	multi-level ($\epsilon = 0.01$)	0.266	89.9	5.9	44.6	60.7	98.2	0.5	2.3
	X-CHASEV ($\epsilon = 0.10$)	1.485	86.9	5.9	40.7	61.0	–	–	–
	single-level ($\epsilon = 0.10$)	0.396	91.3	5.6	43.3	63.6	94.2	4.7	2.1
	multi-level ($\epsilon = 0.10$)	0.266	89.9	5.9	44.6	60.7	98.2	0.5	2.3
2	X-BDV	925.930	96.3	7.7	48.8	57.8	–	–	–
	X-CHV	17.994	86.8	5.6	52.8	60.6	–	–	–
	X-CHASEV	6.552	86.8	5.6	52.8	60.6	–	–	–
	single-level	0.928	88.7	6.0	52.4	61.4	94.3	4.3	2.6
	multi-level	0.817	88.9	6.0	52.4	61.5	98.1	0.4	2.7
	X-CHASEV ($\epsilon = 0.01$)	4.894	82.5	5.0	52.9	59.3	–	–	–
	single-level ($\epsilon = 0.01$)	0.873	85.6	5.1	53.3	58.8	94.2	4.2	3.1
	multi-level ($\epsilon = 0.01$)	0.630	79.5	6.0	53.6	57.8	96.5	0.4	3.8
	X-CHASEV ($\epsilon = 0.10$)	2.635	73.4	5.5	52.5	57.0	–	–	–
	single-level ($\epsilon = 0.10$)	0.873	85.6	5.1	53.3	58.8	94.2	4.2	3.1
	multi-level ($\epsilon = 0.10$)	0.630	79.5	6.0	53.6	57.8	96.5	0.4	3.8
3	X-BDV	981.484	92.0	7.4	60.3	54.7	–	–	–
	X-CHV	35.115	81.8	5.8	57.6	58.3	–	–	–
	X-CHASEV	10.574	81.8	5.8	57.6	58.3	–	–	–
	single-level	1.775	84.2	6.1	58.2	58.3	94.9	3.7	4.1
	multi-level	1.585	84.3	6.0	58.2	58.3	98.3	0.3	4.2
	X-CHASEV ($\epsilon = 0.01$)	7.838	78.1	5.1	58.6	56.5	–	–	–
	single-level ($\epsilon = 0.01$)	1.667	80.9	5.1	57.9	56.8	94.7	3.7	4.9
	multi-level ($\epsilon = 0.01$)	1.249	73.4	5.6	57.1	54.9	97.0	0.3	6.4
	X-CHASEV ($\epsilon = 0.10$)	4.247	67.7	5.6	57.9	55.1	–	–	–
	single-level ($\epsilon = 0.10$)	1.667	80.9	5.1	57.9	56.8	94.7	3.7	4.9
	multi-level ($\epsilon = 0.10$)	1.249	73.4	5.6	57.1	54.9	97.0	0.3	6.4

Table C.12: *Query performance for computing alternative paths under the energy consumption cost model ($p = 2$). Relaxation is not used ($x = 0$). Exact and approximate algorithms ($\epsilon \in \{0, 0.1, 0.01\}$) are considered.*

			Performance		**Path Quality**			**Candidate Sets**		
			time	success	UBS	sharing	locality	via.cand.	fallback	tested
a	algorithm		[ms]	rate [%]	[%]	[%]	[%]	rate [%]	rate [%]	[#]
1	X-BDV		1 018.658	94.8	9.6	38.8	71.0	–	–	–
	X-CHV		1.918	72.3	7.9	38.9	73.5	–	–	–
	X-CHASEV		0.895	72.3	7.9	38.9	73.5	–	–	–
	single-level		0.298	76.8	8.3	43.5	72.9	88.2	4.7	1.7
	multi-level		0.282	77.1	8.3	43.7	72.8	92.1	0.5	1.8
	X-CHASEV	($\epsilon = 0.01$)	0.866	71.0	7.4	40.1	71.2	–	–	–
	single-level	($\epsilon = 0.01$)	0.307	76.8	7.8	44.5	71.0	88.4	4.7	1.9
	multi-level	($\epsilon = 0.01$)	0.310	73.9	8.0	46.7	71.3	93.5	0.5	2.1
	X-CHASEV	($\epsilon = 0.10$)	0.631	65.6	7.7	42.2	72.2	–	–	–
	single-level	($\epsilon = 0.10$)	0.307	76.8	7.8	44.5	71.0	88.4	4.7	1.9
	multi-level	($\epsilon = 0.10$)	0.310	73.9	8.0	46.7	71.3	93.5	0.5	2.1
2	X-BDV		1 124.868	85.0	11.8	57.6	64.3	–	–	–
	X-CHV		3.098	65.6	9.8	54.4	67.0	–	–	–
	X-CHASEV		1.406	65.6	9.8	54.4	67.0	–	–	–
	single-level		0.607	72.8	10.3	55.0	66.2	90.5	4.3	2.4
	multi-level		0.592	73.3	10.3	55.0	66.2	94.2	0.4	2.5
	X-CHASEV	($\epsilon = 0.01$)	1.468	57.9	8.6	56.4	64.7	–	–	–
	single-level	($\epsilon = 0.01$)	0.642	67.5	9.4	56.4	64.3	89.0	4.1	2.7
	multi-level	($\epsilon = 0.01$)	0.622	55.4	9.8	54.2	63.3	86.3	0.4	3.2
	X-CHASEV	($\epsilon = 0.10$)	1.053	41.4	9.2	53.7	64.2	–	–	–
	single-level	($\epsilon = 0.10$)	0.642	67.5	9.4	56.4	64.3	89.0	4.1	2.7
	multi-level	($\epsilon = 0.10$)	0.622	55.4	9.8	54.2	63.3	86.3	0.4	3.2
3	X-BDV		1 206.628	77.7	12.9	66.1	58.5	–	–	–
	X-CHV		4.717	55.0	9.8	62.0	63.7	–	–	–
	X-CHASEV		2.106	55.0	9.8	62.0	63.7	–	–	–
	single-level		0.951	65.1	10.7	60.6	63.4	90.0	3.7	3.3
	multi-level		0.945	65.3	10.7	60.5	63.5	93.3	0.3	3.5
	X-CHASEV	($\epsilon = 0.01$)	2.189	42.2	9.6	60.8	61.6	–	–	–
	single-level	($\epsilon = 0.01$)	1.023	51.7	10.3	58.1	61.3	86.2	3.4	4.1
	multi-level	($\epsilon = 0.01$)	1.010	53.5	10.0	57.5	61.0	88.9	0.3	4.9
	X-CHASEV	($\epsilon = 0.10$)	1.573	38.7	9.7	59.3	62.6	–	–	–
	single-level	($\epsilon = 0.10$)	1.023	51.7	10.3	58.1	61.3	86.2	3.4	4.1
	multi-level	($\epsilon = 0.10$)	1.010	53.5	10.0	57.5	61.0	88.9	0.3	4.9

Table C.13: *Query performance for computing alternative paths under the hop count cost model ($p = 0$). Relaxation ($x = 3$) is applied. Exact and approximate algorithms ($\epsilon \in \{0, 0.1, 0.01\}$) are considered.*

		Performance		**Path Quality**			**Candidate Sets**		
		time	success	UBS	sharing	locality	via.cand.	fallback	tested
a	algorithm	[ms]	rate [%]	[%]	[%]	[%]	rate [%]	rate [%]	[#]
1	X-BDV	769.008	100.0	–	33.2	44.0	–	–	–
	X-CHV	4.441	95.4	3.4	36.4	70.0	–	–	–
	X-CHASEV	3.463	95.4	3.0	36.1	68.9	–	–	–
	single-level	0.289	95.9	3.0	41.4	69.3	94.9	4.7	1.6
	multi-level	0.187	95.9	3.1	41.6	69.3	99.1	0.5	1.6
	X-CHASEV ($\epsilon = 0.01$)	3.466	96.0	3.3	36.4	68.5	–	–	–
	single-level ($\epsilon = 0.01$)	0.289	96.4	2.1	41.8	68.9	94.9	4.7	1.6
	multi-level ($\epsilon = 0.01$)	0.193	95.7	3.4	42.9	68.1	99.1	0.5	1.7
	X-CHASEV ($\epsilon = 0.10$)	3.220	95.1	3.2	37.7	68.0	–	–	–
	single-level ($\epsilon = 0.10$)	0.289	96.4	2.1	41.8	68.9	94.9	4.7	1.6
	multi-level ($\epsilon = 0.10$)	0.193	95.7	3.4	42.9	68.1	99.1	0.5	1.7
2	X-BDV	915.741	99.9	–	51.6	34.3	–	–	–
	X-CHV	6.383	89.8	5.4	53.1	62.4	–	–	–
	X-CHASEV	4.350	89.7	5.1	53.0	62.6	–	–	–
	single-level	0.495	92.1	4.1	55.0	63.9	94.7	4.7	2.2
	multi-level	0.377	92.2	4.0	55.0	64.0	98.9	0.5	2.2
	X-CHASEV ($\epsilon = 0.01$)	4.366	90.0	6.4	53.7	61.7	–	–	–
	single-level ($\epsilon = 0.01$)	0.496	92.3	4.0	55.7	63.3	94.8	4.7	2.2
	multi-level ($\epsilon = 0.01$)	0.390	90.9	4.0	55.9	62.6	98.8	0.5	2.4
	X-CHASEV ($\epsilon = 0.10$)	4.111	87.0	5.4	54.5	62.4	–	–	–
	single-level ($\epsilon = 0.10$)	0.496	92.3	4.0	55.7	63.3	94.8	4.7	2.2
	multi-level ($\epsilon = 0.10$)	0.390	90.9	4.0	55.9	62.6	98.8	0.5	2.4
3	X-BDV	1 041.846	99.3	–	65.2	28.9	–	–	–
	X-CHV	8.941	85.3	6.8	61.3	58.6	–	–	–
	X-CHASEV	5.436	85.2	6.8	61.3	58.7	–	–	–
	single-level	0.751	89.3	5.2	62.1	59.9	94.8	4.6	3.0
	multi-level	0.619	89.5	5.2	62.0	60.1	98.9	0.5	3.0
	X-CHASEV ($\epsilon = 0.01$)	5.441	83.3	7.5	61.8	59.6	–	–	–
	single-level ($\epsilon = 0.01$)	0.762	86.7	4.4	62.0	61.1	94.3	4.6	3.1
	multi-level ($\epsilon = 0.01$)	0.650	84.8	5.1	61.2	58.7	98.0	0.5	3.4
	X-CHASEV ($\epsilon = 0.10$)	5.220	78.7	6.5	62.0	58.0	–	–	–
	single-level ($\epsilon = 0.10$)	0.762	86.7	4.4	62.0	61.1	94.3	4.6	3.1
	multi-level ($\epsilon = 0.10$)	0.650	84.8	5.1	61.2	58.7	98.0	0.5	3.4

Table C.14: *Query performance for computing alternative paths under the latency cost model ($p = 1$). Relaxation ($x = 3$) is applied. Exact and approximate algorithms ($\epsilon \in \{0, 0.1, 0.01\}$) are considered.*

			Performance		**Path Quality**			**Candidate Sets**		
			time	success	UBS	sharing	locality	via.cand.	fallback	tested
a	algorithm		[ms]	rate [%]	[%]	[%]	[%]	rate [%]	rate [%]	[#]
1	X-BDV		871.229	98.5	7.7	29.2	63.5	–	–	–
	X-CHV		16.151	96.7	6.7	32.3	63.6	–	–	–
	X-CHASEV		10.553	96.7	6.7	32.3	63.6	–	–	–
	single-level		0.659	96.8	6.6	38.1	63.8	95.0	4.7	1.8
	multi-level		0.361	96.8	6.6	38.6	63.9	99.2	0.5	1.9
	X-CHASEV	($\epsilon = 0.01$)	8.015	96.1	5.4	35.5	62.5	–	–	–
	single-level	($\epsilon = 0.01$)	0.569	96.0	5.7	40.8	61.6	95.0	4.7	2.1
	multi-level	($\epsilon = 0.01$)	0.272	95.3	5.9	41.9	60.0	99.1	0.5	2.3
	X-CHASEV	($\epsilon = 0.10$)	5.127	94.6	6.1	37.3	61.4	–	–	–
	single-level	($\epsilon = 0.10$)	0.569	96.0	5.7	40.8	61.6	95.0	4.7	2.1
	multi-level	($\epsilon = 0.10$)	0.272	95.3	5.9	41.9	60.0	99.1	0.5	2.3
2	X-BDV		925.930	96.3	7.7	48.8	57.8	–	–	–
	X-CHV		32.375	92.1	6.7	49.8	58.8	–	–	–
	X-CHASEV		14.654	92.1	6.7	49.8	58.8	–	–	–
	single-level		1.431	93.5	6.6	53.0	59.9	94.9	4.7	2.6
	multi-level		0.820	93.7	6.5	53.2	60.1	99.1	0.5	2.6
	X-CHASEV	($\epsilon = 0.01$)	11.419	89.8	6.0	51.3	57.9	–	–	–
	single-level	($\epsilon = 0.01$)	1.201	92.2	6.0	53.5	57.5	95.0	4.6	3.0
	multi-level	($\epsilon = 0.01$)	0.607	91.0	5.9	55.0	56.9	99.3	0.5	3.4
	X-CHASEV	($\epsilon = 0.10$)	7.535	86.9	5.7	53.6	57.2	–	–	–
	single-level	($\epsilon = 0.10$)	1.201	92.2	6.0	53.5	57.5	95.0	4.6	3.0
	multi-level	($\epsilon = 0.10$)	0.607	91.0	5.9	55.0	56.9	99.3	0.5	3.4
3	X-BDV		981.484	92.0	7.4	60.3	54.7	–	–	–
	X-CHV		57.828	86.4	6.7	59.0	56.4	–	–	–
	X-CHASEV		20.874	86.4	6.7	59.0	56.4	–	–	–
	single-level		2.637	88.8	6.7	60.0	57.1	95.3	4.5	3.9
	multi-level		1.531	89.0	6.7	59.9	57.0	99.3	0.4	4.0
	X-CHASEV	($\epsilon = 0.01$)	16.533	81.3	6.6	60.4	56.4	–	–	–
	single-level	($\epsilon = 0.01$)	2.125	86.9	6.1	60.0	56.2	95.0	4.4	4.5
	multi-level	($\epsilon = 0.01$)	1.140	86.3	6.0	61.4	55.1	99.1	0.5	5.4
	X-CHASEV	($\epsilon = 0.10$)	10.648	75.5	6.2	61.6	55.5	–	–	–
	single-level	($\epsilon = 0.10$)	2.125	86.9	6.1	60.0	56.2	95.0	4.4	4.5
	multi-level	($\epsilon = 0.10$)	1.140	86.3	6.0	61.4	55.1	99.1	0.5	5.4

Table C.15: *Query performance for computing alternative paths under the energy consumption cost model ($p = 2$). Relaxation ($x = 3$) is applied. Exact and approximate algorithms ($\epsilon \in \{0, 0.1, 0.01\}$) are considered.*

		Performance		Path Quality			Candidate Sets		
		time	success	UBS	sharing	locality	via.cand.	fallback	tested
a	algorithm	[ms]	rate [%]	[%]	[%]	[%]	rate [%]	rate [%]	[#]
1	X-BDV	1 018.658	94.8	9.6	38.8	71.0	–	–	–
	X-CHV	4.246	90.7	9.3	40.1	70.2	–	–	–
	X-CHASEV	2.874	90.7	9.3	40.1	70.2	–	–	–
	single-level	0.415	91.2	9.2	45.4	70.3	94.8	4.7	1.8
	multi-level	0.329	91.3	9.2	45.8	70.4	98.9	0.5	1.8
	X-CHASEV ($\epsilon = 0.01$)	2.778	88.8	7.7	41.9	69.4	–	–	–
	single-level ($\epsilon = 0.01$)	0.416	90.8	7.9	46.9	70.3	94.7	4.7	1.9
	multi-level ($\epsilon = 0.01$)	0.352	88.4	7.7	47.7	70.3	98.7	0.5	2.2
	X-CHASEV ($\epsilon = 0.10$)	2.370	88.6	75.5	42.9	69.6	–	–	–
	single-level ($\epsilon = 0.10$)	0.416	90.8	7.9	46.9	70.3	94.7	4.7	1.9
	multi-level ($\epsilon = 0.10$)	0.352	88.4	7.7	47.7	70.3	98.7	0.5	2.2
2	X-BDV	1 124.868	85.0	11.8	57.6	64.3	–	–	–
	X-CHV	6.321	74.6	10.7	56.4	64.4	–	–	–
	X-CHASEV	3.894	74.6	10.7	56.4	64.4	–	–	–
	single-level	0.773	79.3	11.2	57.7	63.9	92.3	4.6	2.7
	multi-level	0.667	79.6	11.2	57.9	63.9	96.3	0.5	2.7
	X-CHASEV ($\epsilon = 0.01$)	3.789	74.7	52.8	57.9	63.5	–	–	–
	single-level ($\epsilon = 0.01$)	0.797	77.9	10.7	59.0	63.0	92.2	4.7	3.0
	multi-level ($\epsilon = 0.01$)	0.720	76.5	10.9	59.4	62.0	96.8	0.5	3.3
	X-CHASEV ($\epsilon = 0.10$)	3.287	70.3	147.3	58.3	63.1	–	–	–
	single-level ($\epsilon = 0.10$)	0.797	77.9	10.7	59.0	63.0	92.2	4.7	3.0
	multi-level ($\epsilon = 0.10$)	0.720	76.5	10.9	59.4	62.0	96.8	0.5	3.3
3	X-BDV	1 206.628	77.7	12.9	66.1	58.5	–	–	–
	X-CHV	8.974	68.6	11.1	64.1	61.7	–	–	–
	X-CHASEV	5.162	68.6	11.1	64.1	61.7	–	–	–
	single-level	1.206	73.8	11.7	63.2	61.3	93.2	4.6	3.9
	multi-level	1.084	74.2	11.7	63.2	61.3	97.3	0.5	4.0
	X-CHASEV ($\epsilon = 0.01$)	5.033	62.7	12.7	64.5	60.9	–	–	–
	single-level ($\epsilon = 0.01$)	1.236	71.4	11.7	62.6	59.3	92.6	4.5	4.2
	multi-level ($\epsilon = 0.01$)	1.178	69.9	11.8	62.5	59.3	96.7	0.5	4.9
	X-CHASEV ($\epsilon = 0.10$)	4.404	58.1	44.6	65.3	59.8	–	–	–
	single-level ($\epsilon = 0.10$)	1.236	71.4	11.7	62.6	59.3	92.6	4.5	4.2
	multi-level ($\epsilon = 0.10$)	1.178	69.9	11.8	62.5	59.3	96.7	0.5	4.9

Table C.16: *Preprocessing performance for each edge cost model ($p \in \{0, 1, 2\}$) with exact and approximate queries ($\epsilon \in \{0, 0.01, 0.1\}$) and w/o relaxation ($x = 0$).*

			Performance		**Candidate Sets**					
					$a = 1$		$a = 2$		$a = 3$	
			time	size	empty	size	empty	size	empty	size
p	preprocessing type		[h]	[kiB]	[%]	[#]	[%]	[#]	[%]	[#]
0	single-level		1.0	1 223	1.9	4.6	6.4	6.5	12.3	8.4
	+ multi-level		2.1	6 808	3.7	6.1	8.7	8.7	13.4	11.2
	single-level	($\epsilon = 0.01$)	0.9	1 197	3.1	4.5	9.3	6.4	18.3	8.2
	+ multi-level	($\epsilon = 0.01$)	2.1	7 107	4.1	6.4	9.4	9.2	15.5	11.6
	single-level	($\epsilon = 0.10$)	0.9	1 197	3.1	4.5	9.3	6.4	18.3	8.2
	+ multi-level	($\epsilon = 0.10$)	2.1	7 107	4.1	6.4	9.4	9.2	15.5	11.6
1	single-level		6.0	1 795	1.2	6.2	4.1	9.4	8.8	13.0
	+ multi-level		15.7	11 755	2.2	9.8	5.4	14.9	9.5	20.3
	single-level	($\epsilon = 0.01$)	4.8	2 206	1.3	8.1	5.4	11.7	10.2	15.4
	+ multi-level	($\epsilon = 0.01$)	7.4	12 232	3.5	11.0	8.1	15.9	14.1	19.9
	single-level	($\epsilon = 0.10$)	4.8	2 206	1.3	8.1	5.4	11.7	10.2	15.4
	+ multi-level	($\epsilon = 0.10$)	7.4	12 232	3.5	11.0	8.1	15.9	14.1	19.9
2	single-level		1.4	883	6.9	3.9	16.6	4.9	28.5	5.3
	+ multi-level		2.4	5 891	5.9	5.9	14.0	7.8	23.1	8.8
	single-level	($\epsilon = 0.01$)	1.7	950	7.0	4.5	17.9	5.4	33.6	5.3
	+ multi-level	($\epsilon = 0.01$)	2.1	6 024	6.8	6.6	17.0	8.1	29.3	8.3
	single-level	($\epsilon = 0.10$)	1.7	950	7.0	4.5	17.9	5.4	33.6	5.3
	+ multi-level	($\epsilon = 0.10$)	2.1	6 024	6.8	6.6	17.0	8.1	29.3	8.3

Table C.17: *Preprocessing performance for each edge cost model ($p \in \{0, 1, 2\}$) with exact and approximate queries ($\epsilon \in \{0, 0.01, 0.1\}$) and with relaxation ($x = 3$).*

			Performance		**Candidate Sets**					
					$a = 1$		$a = 2$		$a = 3$	
			time	size	empty	size	empty	size	empty	size
p	preprocessing type		[h]	[kiB]	[%]	[#]	[%]	[#]	[%]	[#]
0	single-level		0.9	1 524	0.4	5.2	2.0	8.0	4.3	11.1
	+ multi-level		2.1	8 607	0.3	7.3	1.5	10.8	3.4	14.8
	single-level	($\epsilon = 0.01$)	1.0	1 552	0.4	5.4	1.6	8.1	4.7	11.3
	+ multi-level	($\epsilon = 0.01$)	2.2	8 992	0.4	7.6	1.7	11.3	4.3	15.5
	single-level	($\epsilon = 0.10$)	1.0	1 552	0.4	5.4	1.6	8.1	4.7	11.3
	+ multi-level	($\epsilon = 0.10$)	2.2	8 992	0.4	7.6	1.7	11.3	4.3	15.5
1	single-level		4.4	2 009	0.3	6.8	1.2	10.4	3.1	14.8
	+ multi-level		15.2	13 907	0.2	11.0	1.0	17.3	2.8	24.9
	single-level	($\epsilon = 0.01$)	3.7	2 473	0.3	8.9	1.3	12.9	3.7	17.7
	+ multi-level	($\epsilon = 0.01$)	6.6	15 265	0.4	12.5	1.5	19.0	3.8	27.0
	single-level	($\epsilon = 0.10$)	3.7	2 473	0.3	8.9	1.3	12.9	3.7	17.7
	+ multi-level	($\epsilon = 0.10$)	6.6	15 265	0.4	12.5	1.5	19.0	3.8	27.0
2	single-level		2.3	1 337	0.7	5.1	4.9	7.3	12.5	9.0
	+ multi-level		2.9	8 885	0.6	7.6	3.0	11.5	8.1	15.0
	single-level	($\epsilon = 0.01$)	2.3	1 479	0.7	5.8	5.2	8.1	14.0	9.7
	+ multi-level	($\epsilon = 0.01$)	2.6	9 726	1.2	8.8	4.8	12.7	11.2	15.8
	single-level	($\epsilon = 0.10$)	2.3	1 479	0.7	5.8	5.2	8.1	14.0	9.7
	+ multi-level	($\epsilon = 0.10$)	2.6	9 726	1.2	8.8	4.8	12.7	11.2	15.8

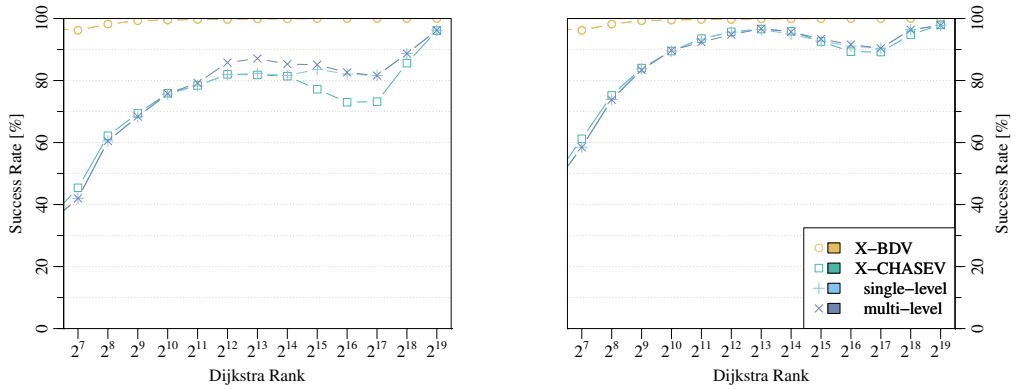


Figure C.2: *Success rates against Dijkstra rank for the hop count cost model ($p = 0$) and exact queries w/o ($x = 0$), left, and with relaxation ($x = 3$), right.*

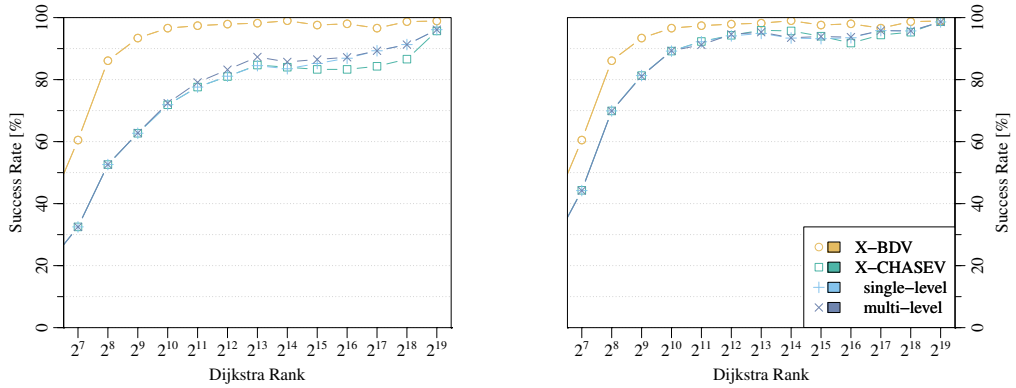


Figure C.3: *Success rates against Dijkstra rank for the latency cost model ($p = 1$) and exact queries w/o ($x = 0$), left, and with relaxation ($x = 3$), right.*

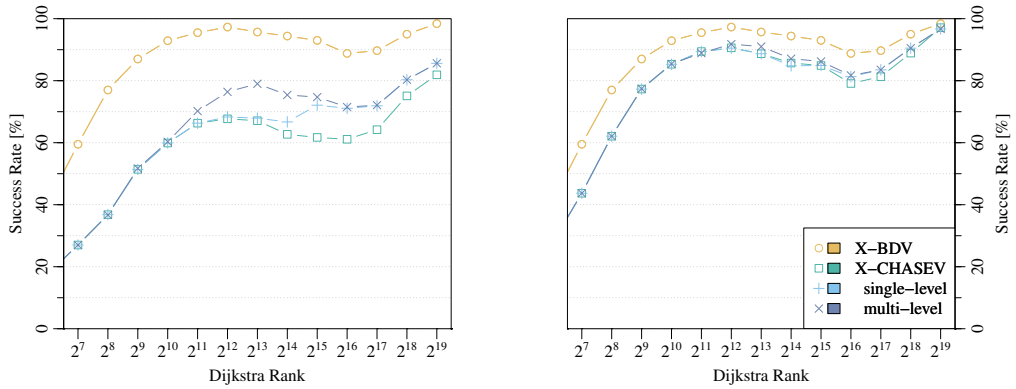
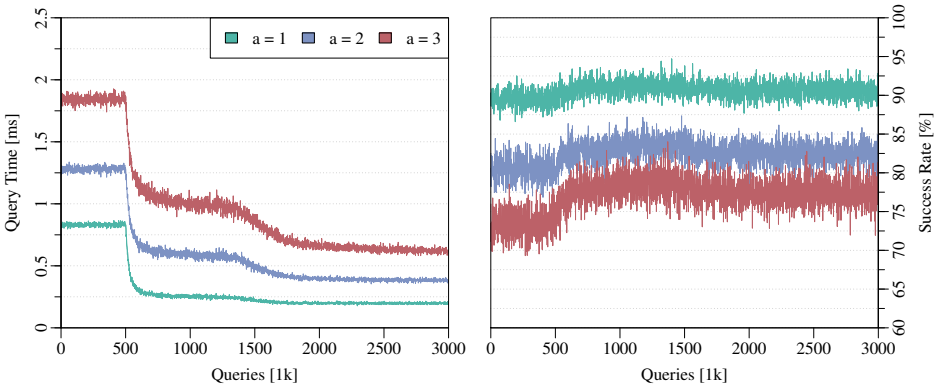


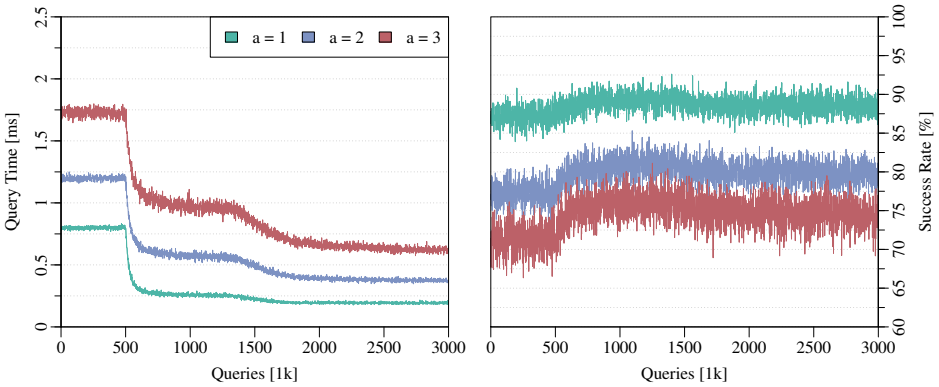
Figure C.4: *Success rates against Dijkstra rank for the energy consumption cost model ($p = 2$) and exact queries w/o ($x = 0$), left, and with relaxation ($x = 3$), right.*

Online Algorithm

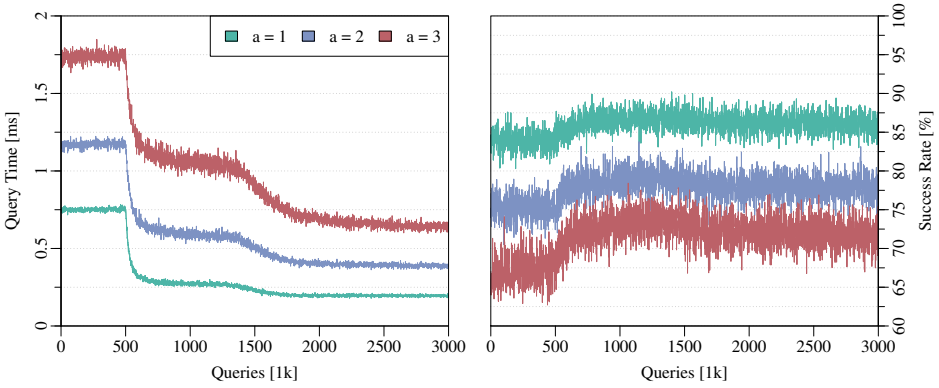
We list further results for our online setting. Some figures that have already been presented in Section 5.5.3 are repeated here for the sake of completeness.



(a) *Results for exact queries ($\epsilon = 0.0$).*

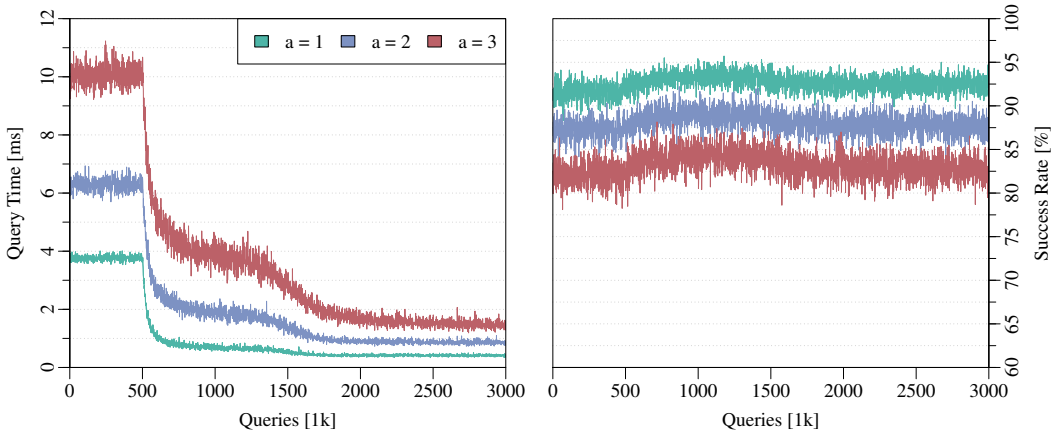


(b) *Results for approximate queries ($\epsilon = 0.01$).*

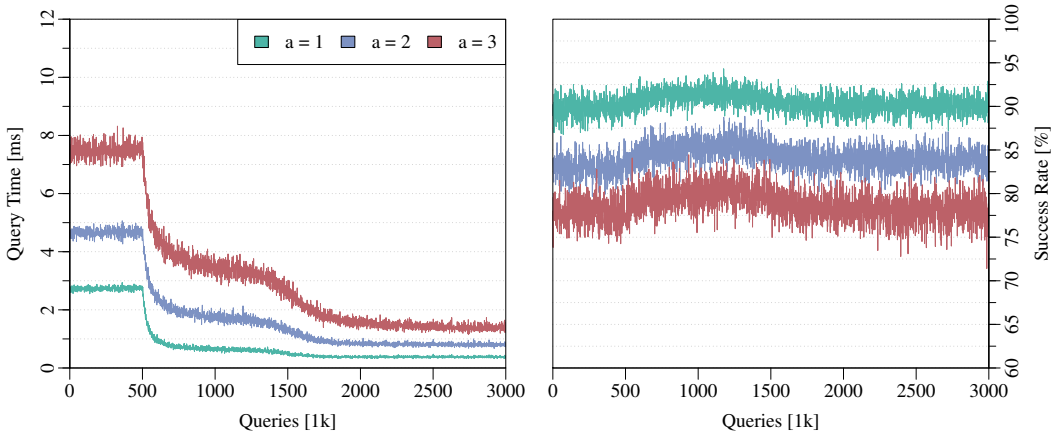


(c) *Results for approximate queries ($\epsilon = 0.1$).*

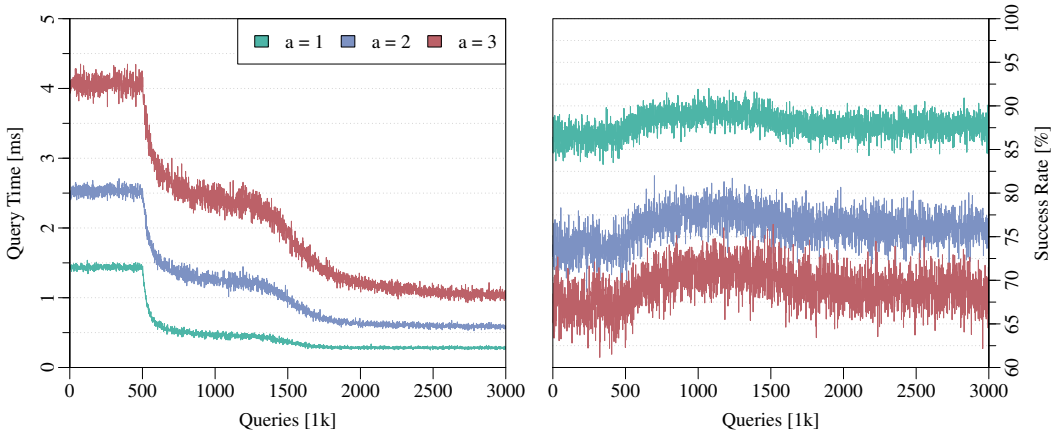
Figure C.5: *Query time and success rate of our approach in the online setting. The hop count cost model ($p = 0$) is used.*



(a) *Results for exact queries* ($\epsilon = 0.0$).

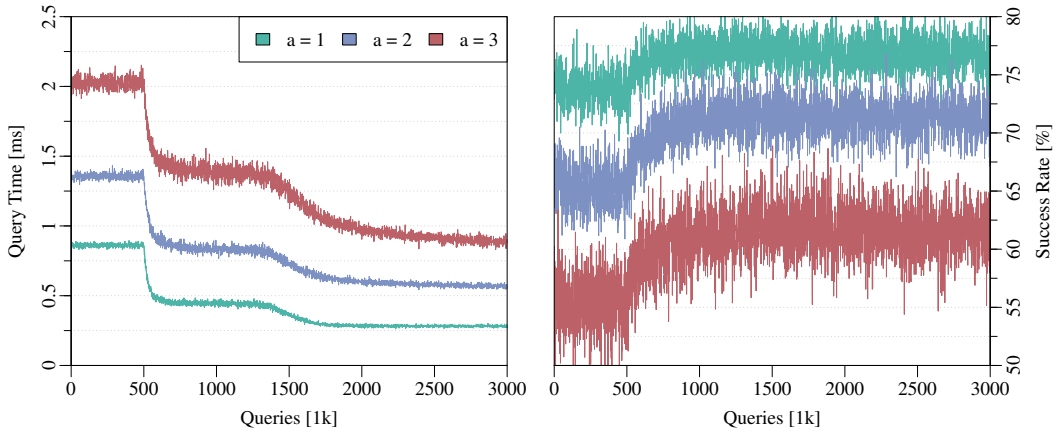


(b) *Results for approximate queries* ($\epsilon = 0.01$).

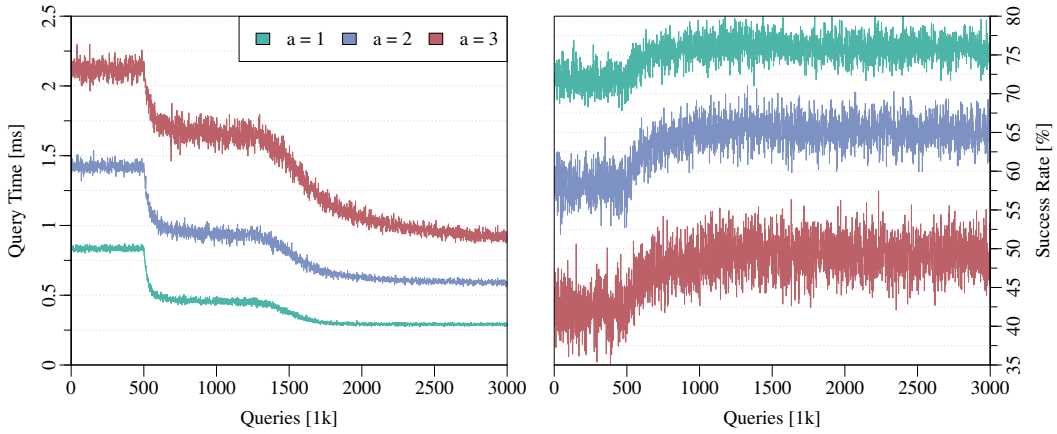


(c) *Results for approximate queries* ($\epsilon = 0.1$).

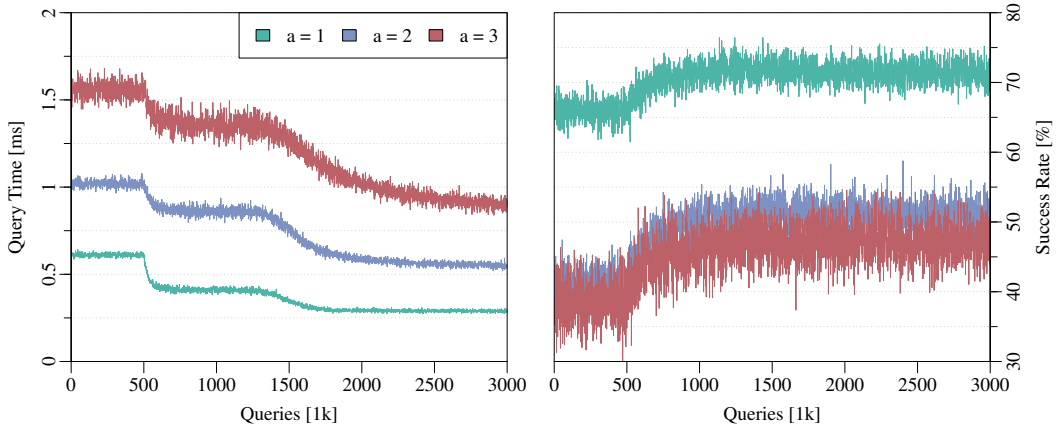
Figure C.6: *Query time and success rate of our approach in the online setting. The latency cost model* ($p = 1$) *is used.*



(a) *Results for exact queries* ($\epsilon = 0.0$).

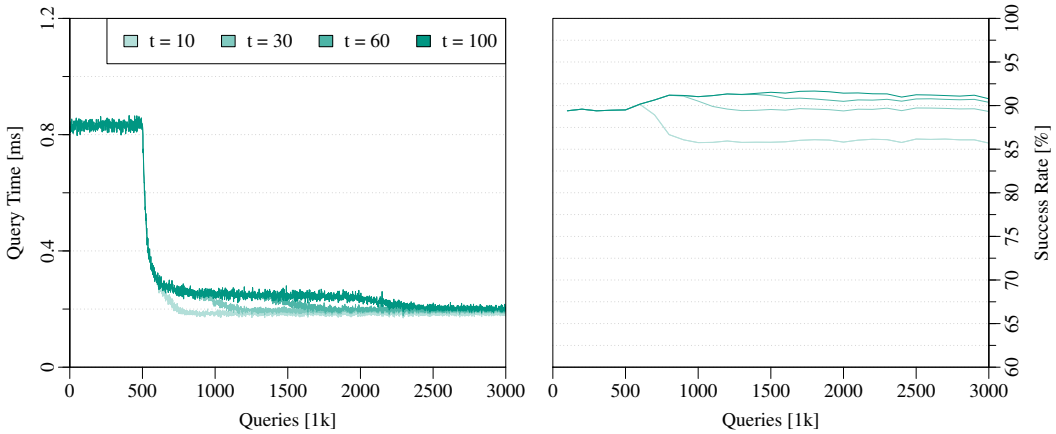


(b) *Results for approximate queries* ($\epsilon = 0.01$).

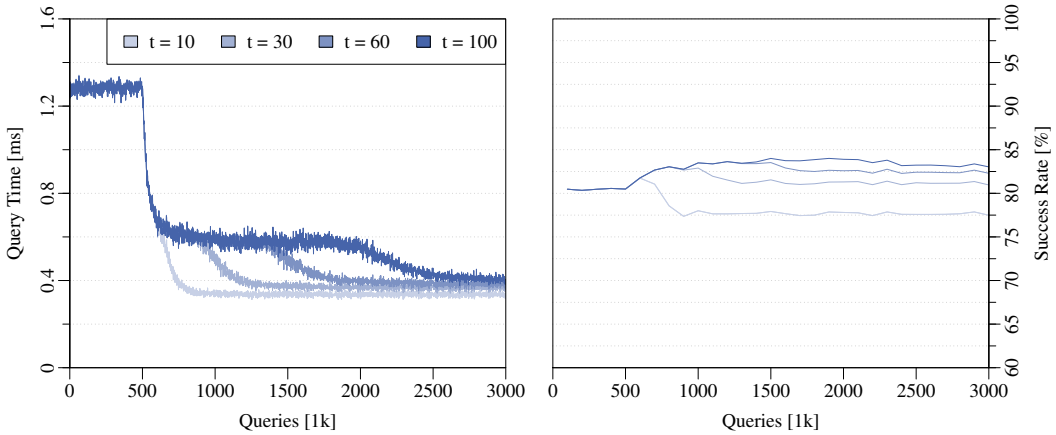


(c) *Results for approximate queries* ($\epsilon = 0.1$).

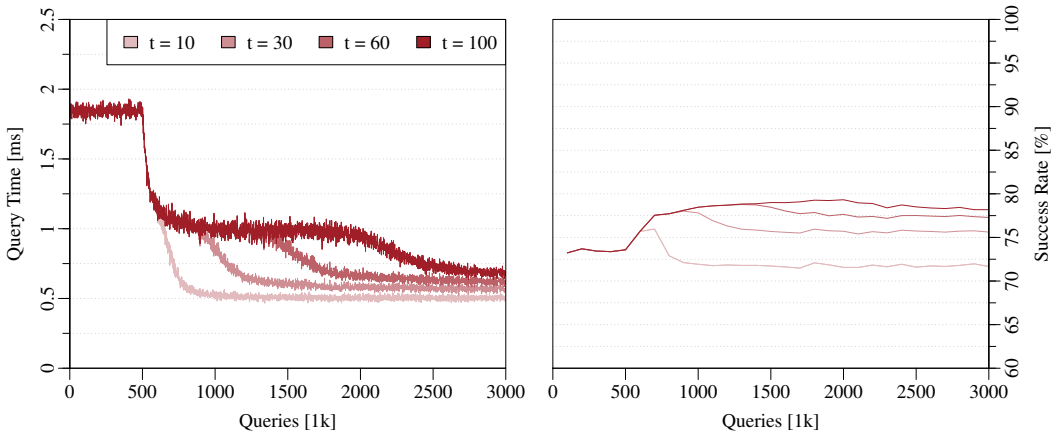
Figure C.7: *Query time and success rate of our approach in the online setting. The energy consumption cost model ($p = 2$) is used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

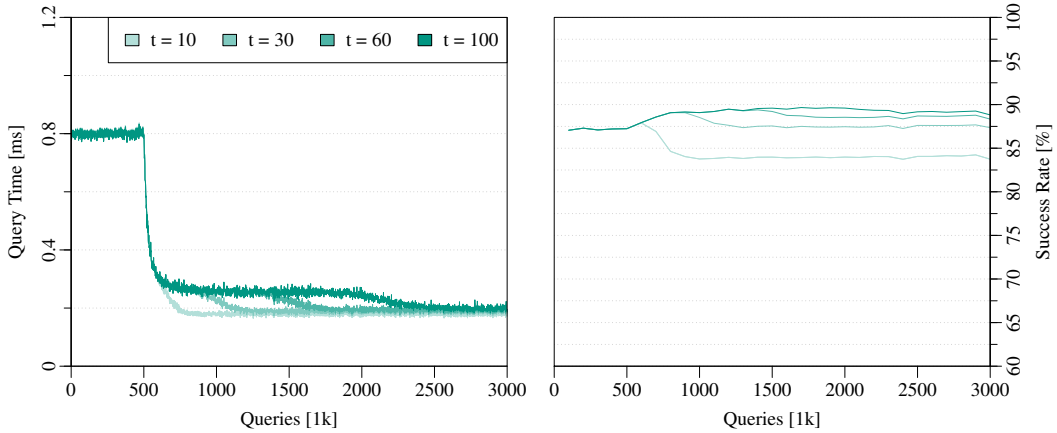


(b) *Query time and success rates for the second ($a = 2$) alternative.*

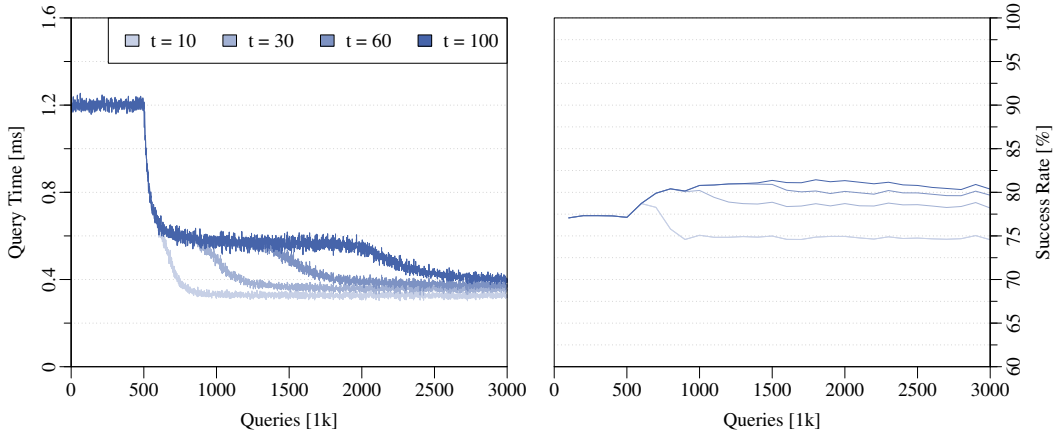


(c) *Query time and success rates for the third ($a = 3$) alternative.*

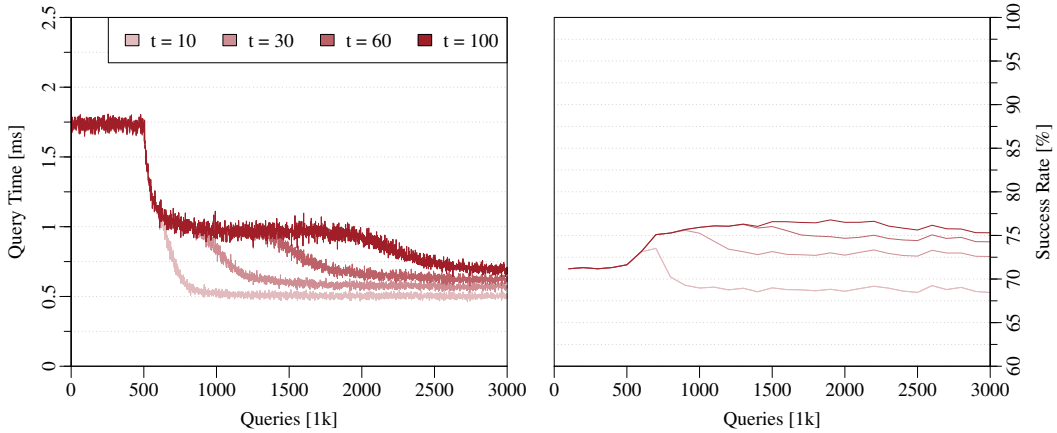
Figure C.8: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Exact queries ($\epsilon = 0.0$) and the hop count cost model ($p = 0$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

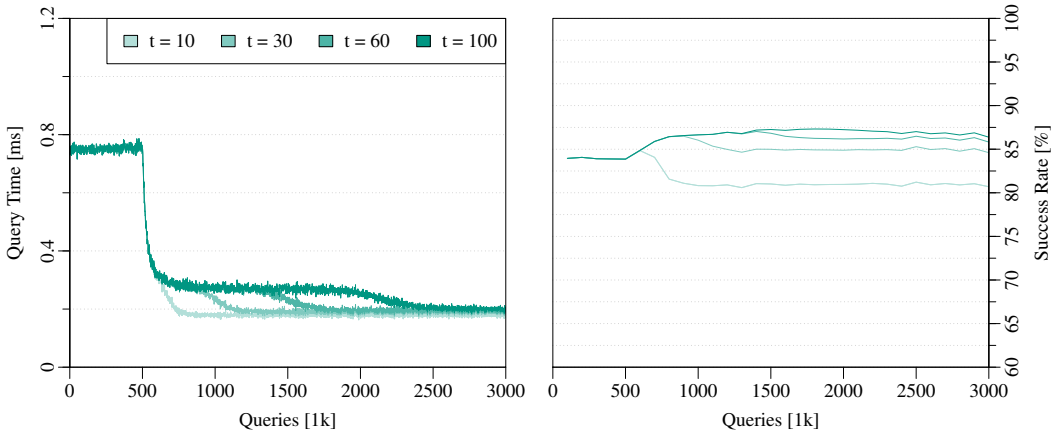


(b) *Query time and success rates for the second ($a = 2$) alternative.*

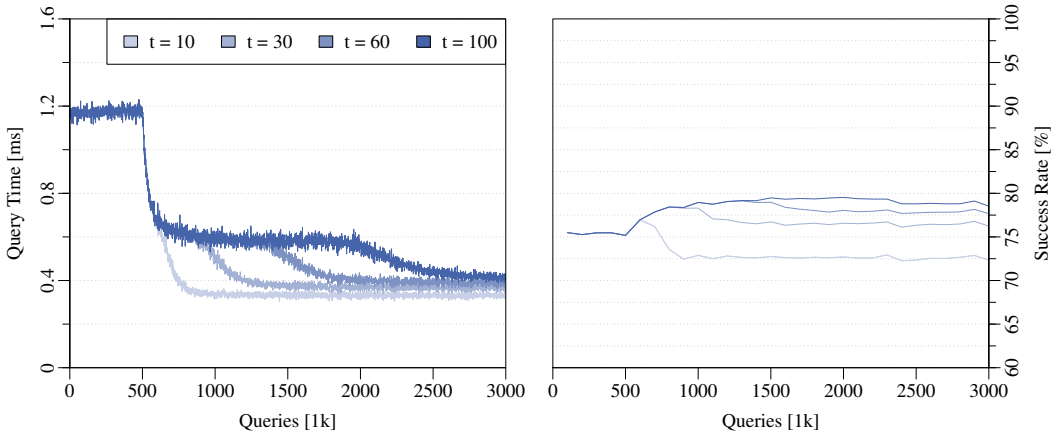


(c) *Query time and success rates for the third ($a = 3$) alternative.*

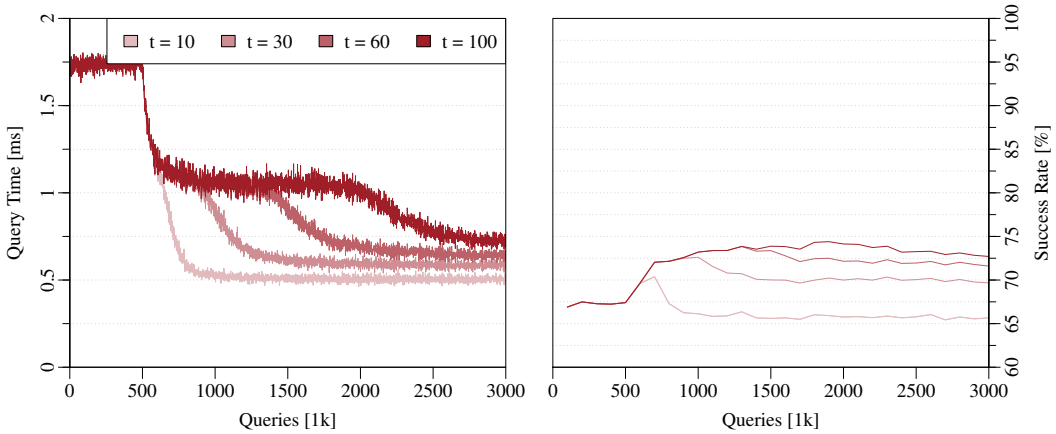
Figure C.9: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Approximate queries ($\epsilon = 0.01$) and the hop count cost model ($p = 0$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

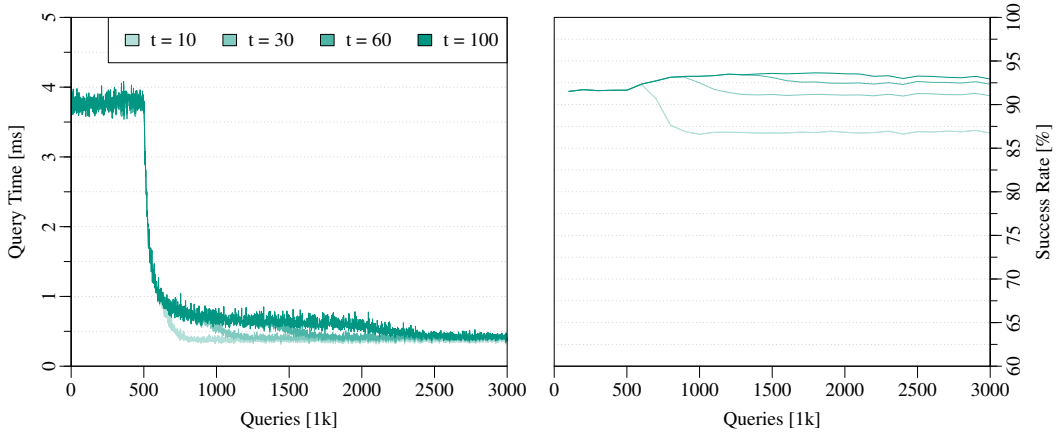


(b) *Query time and success rates for the second ($a = 2$) alternative.*

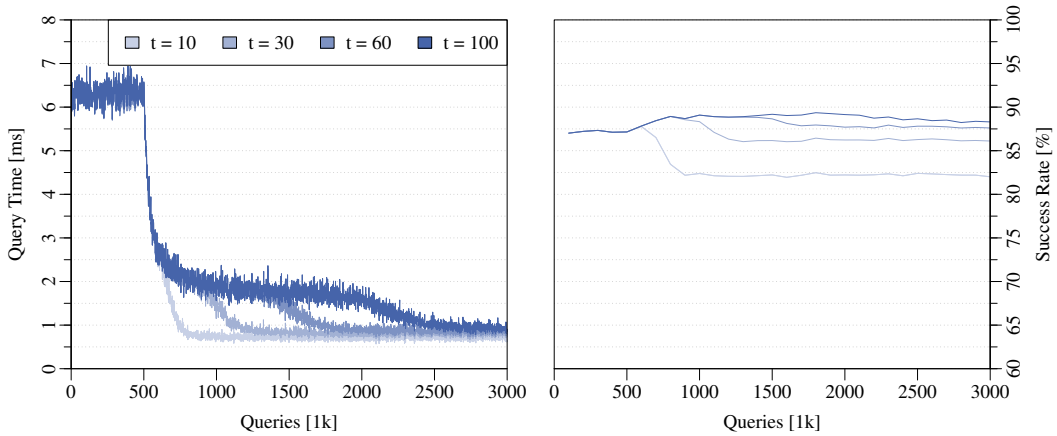


(c) *Query time and success rates for the third ($a = 3$) alternative.*

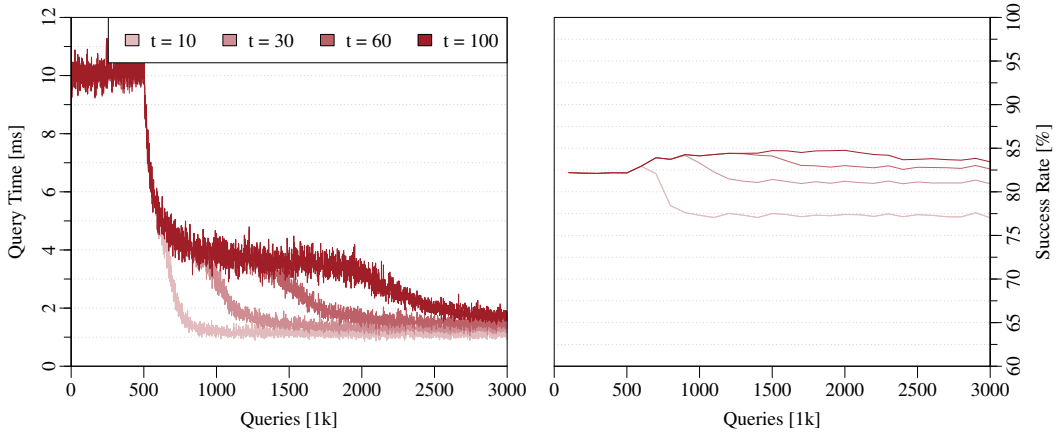
Figure C.10: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Approximate queries ($\epsilon = 0.10$) and the hop count cost model ($p = 0$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

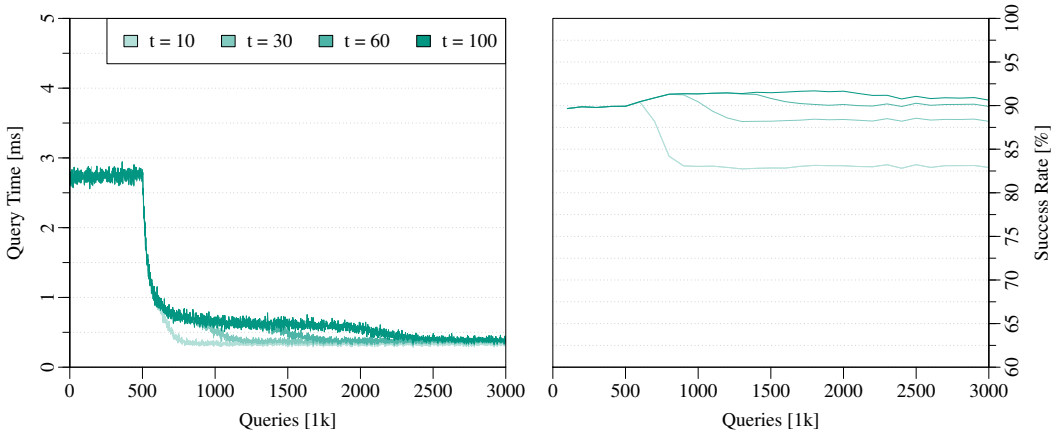


(b) *Query time and success rates for the second ($a = 2$) alternative.*

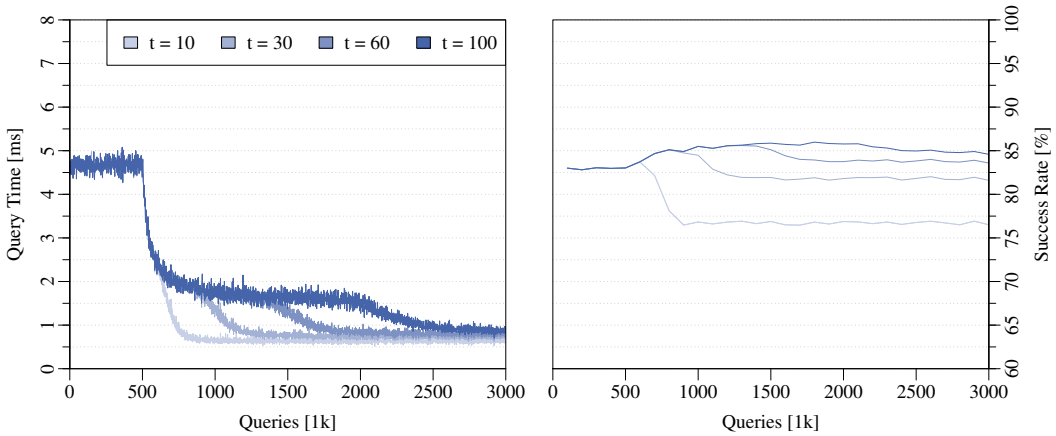


(c) *Query time and success rates for the third ($a = 3$) alternative.*

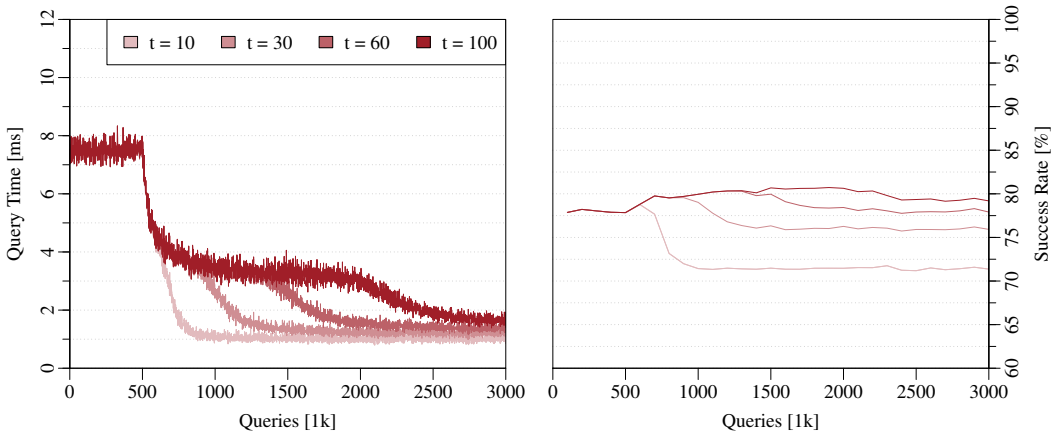
Figure C.11: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Exact queries ($\epsilon = 0.0$) and the latency cost model ($p = 1$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

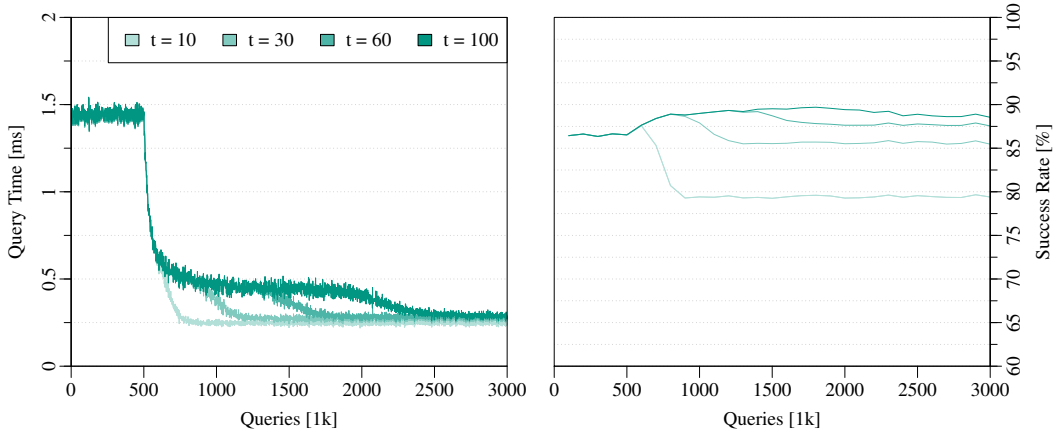


(b) *Query time and success rates for the second ($a = 2$) alternative.*

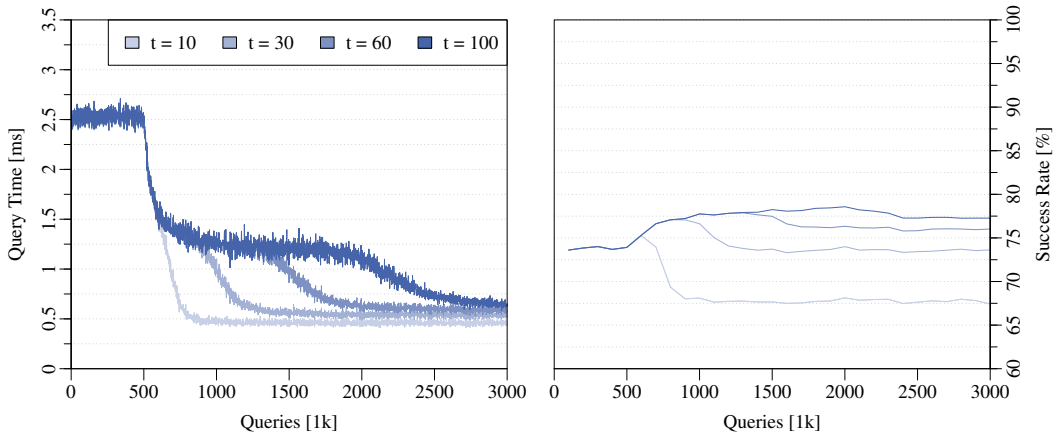


(c) *Query time and success rates for the third ($a = 3$) alternative.*

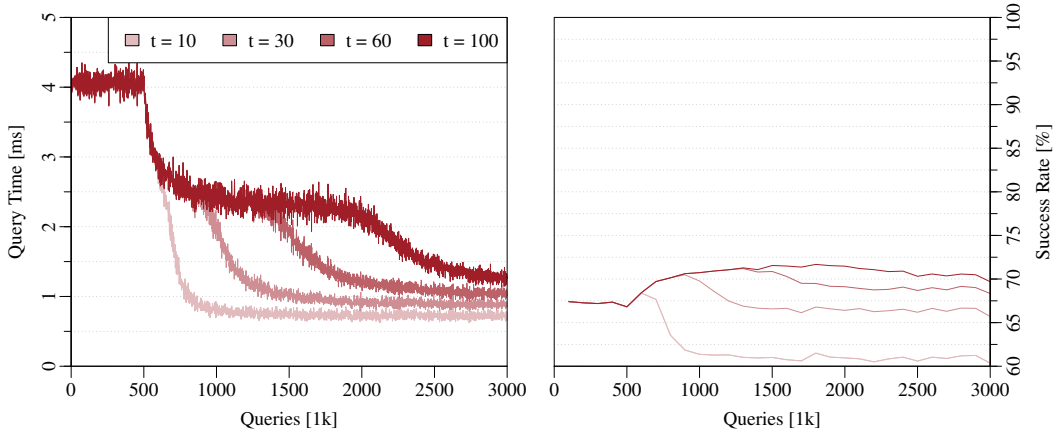
Figure C.12: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Approximate queries ($\epsilon = 0.01$) and the latency cost model ($p = 1$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

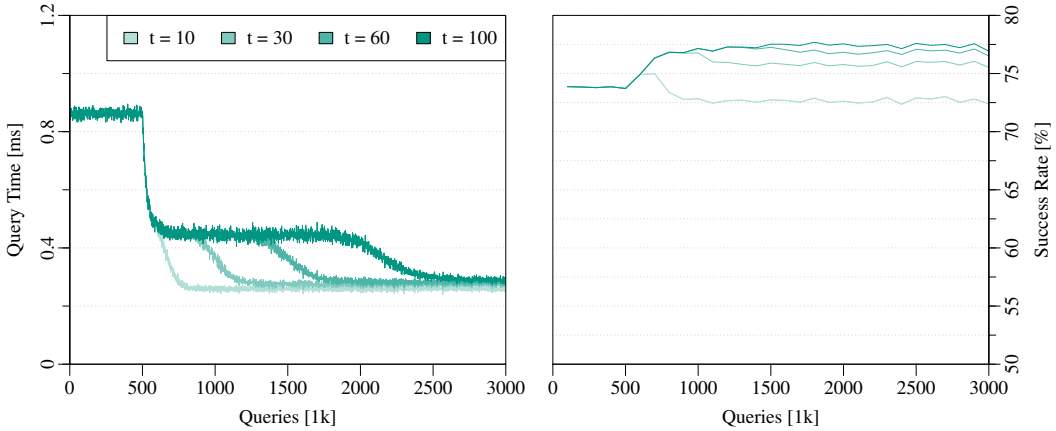


(b) *Query time and success rates for the second ($a = 2$) alternative.*

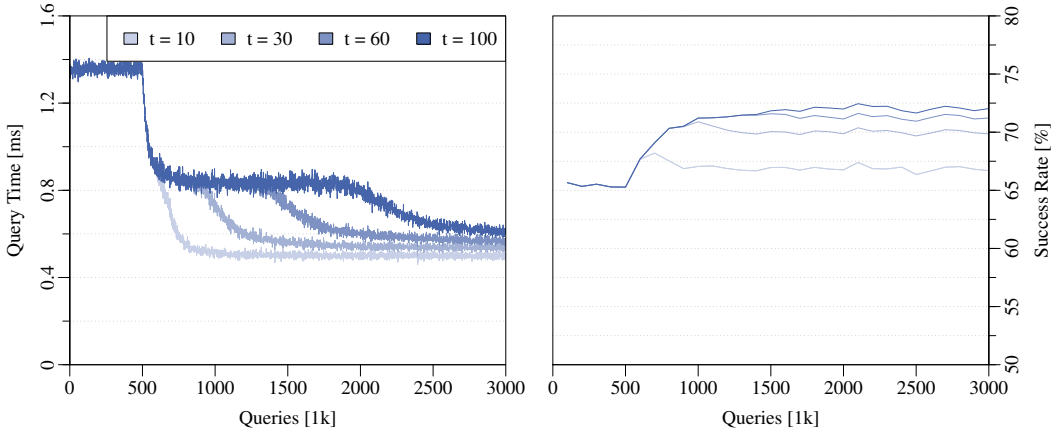


(c) *Query time and success rates for the third ($a = 3$) alternative.*

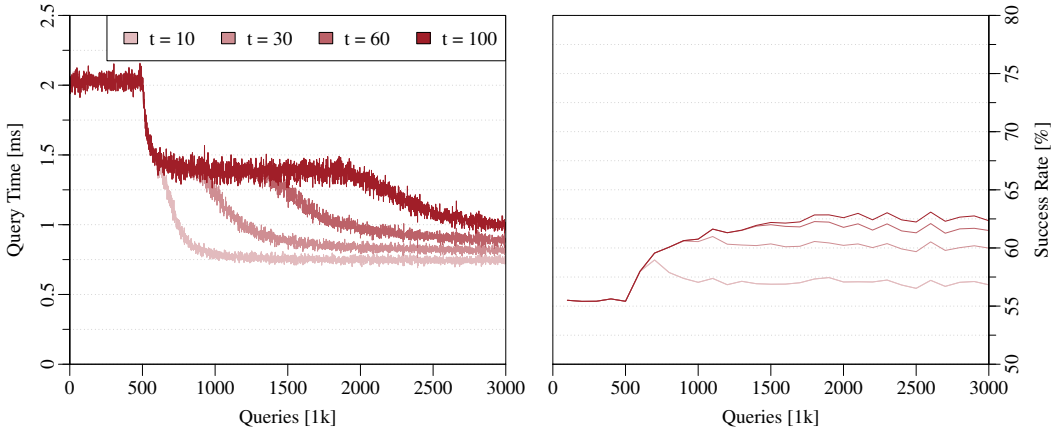
Figure C.13: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Approximate queries ($\epsilon = 0.1$) and the latency cost model ($p = 1$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

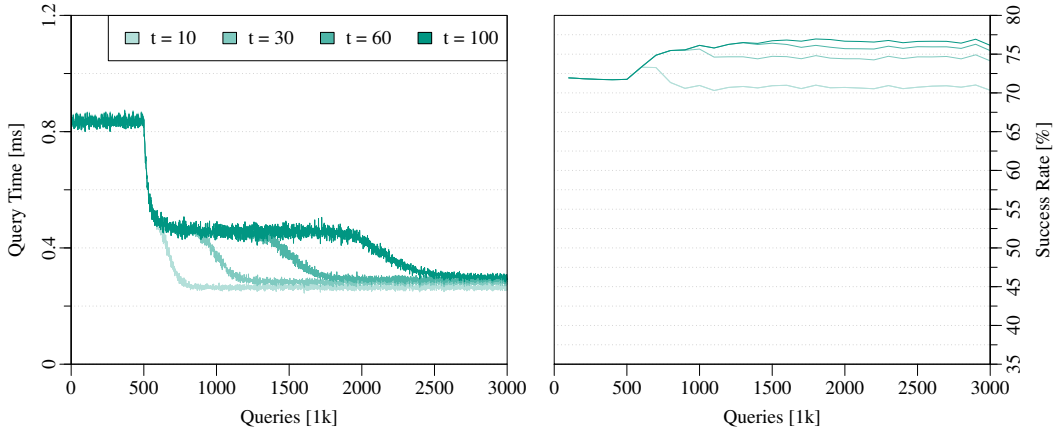


(b) *Query time and success rates for the second ($a = 2$) alternative.*

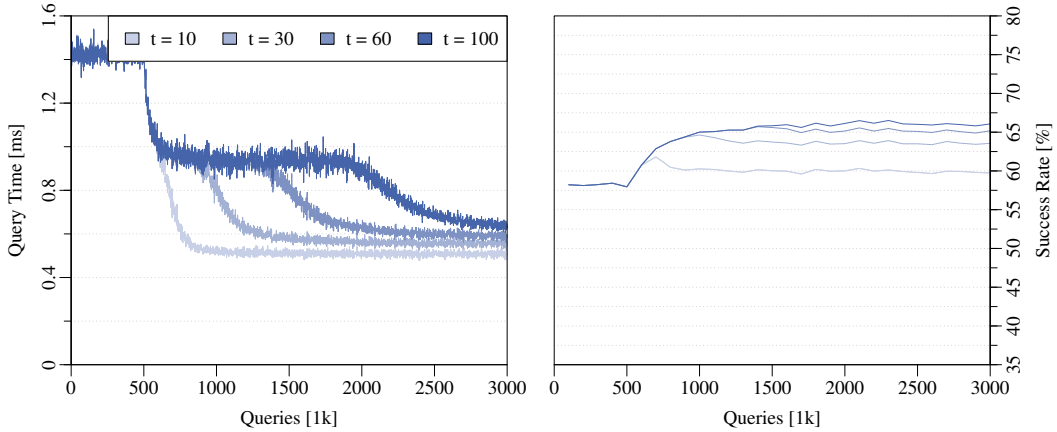


(c) *Query time and success rates for the third ($a = 3$) alternative.*

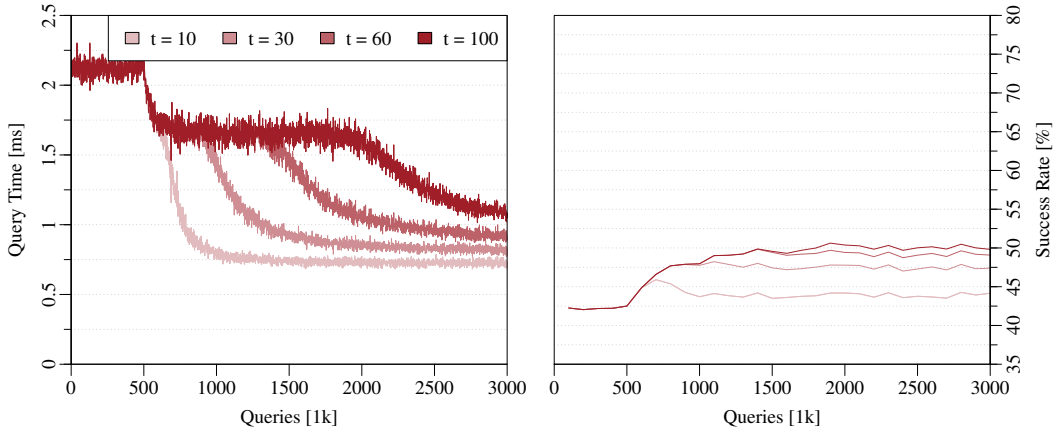
Figure C.14: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Exact queries ($\epsilon = 0.0$) and the energy consumption cost model ($p = 2$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*

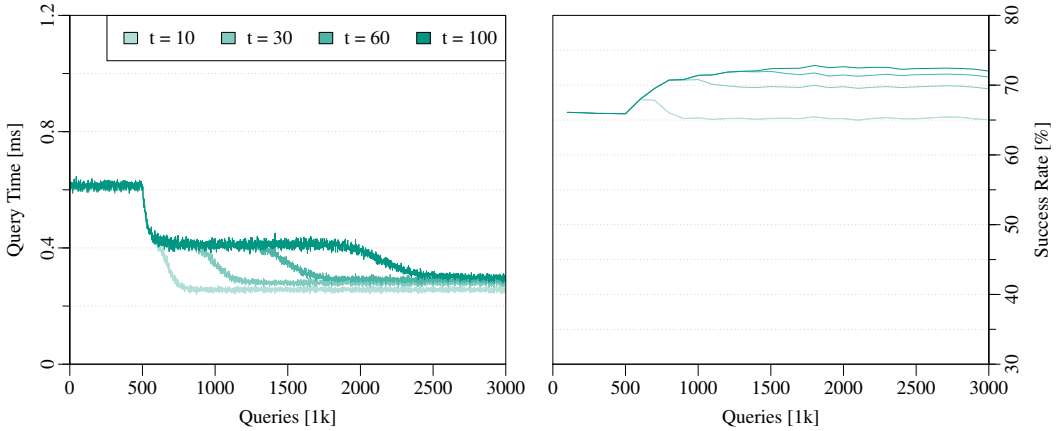


(b) *Query time and success rates for the second ($a = 2$) alternative.*

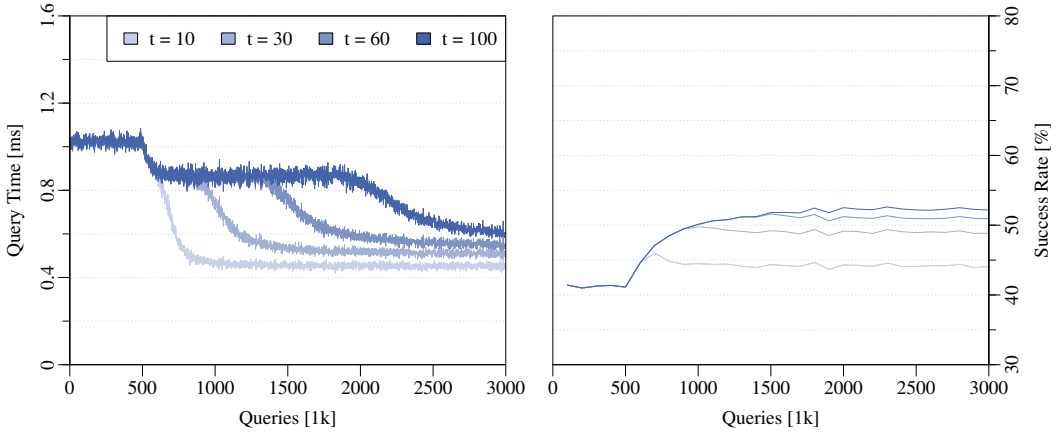


(c) *Query time and success rates for the third ($a = 3$) alternative.*

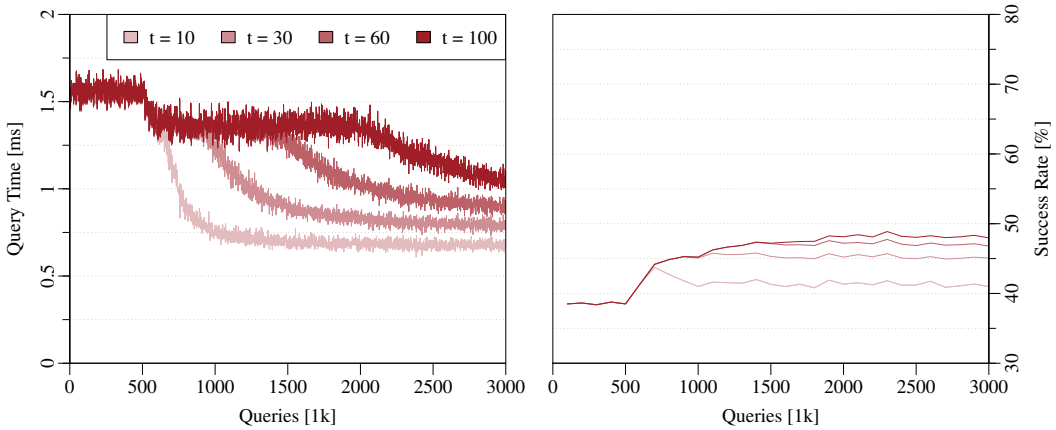
Figure C.15: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Approximate queries ($\epsilon = 0.01$) and the energy consumption cost model ($p = 2$) are used.*



(a) *Query time and success rates for the first ($a = 1$) alternative.*



(b) *Query time and success rates for the second ($a = 2$) alternative.*



(c) *Query time and success rates for the third ($a = 3$) alternative.*

Figure C.16: *Query time and success rate of our approach in the online setting with different threshold values t for saturating via node candidate sets. Approximate queries ($\epsilon = 0.1$) and the energy consumption cost model ($p = 2$) are used.*

Curriculum Vitæ

Personal Data

Full Name:	Dennis Schieferdecker
Date of Birth:	October 2^{nd}, 1978
Place of Birth:	Aalen, Germany
Nationality:	German

Work Experience

Current–Jun 2008 | Researcher at **Karlsruhe Institute of Technology**, Germany
Institute for Theoretical Informatics—Algorithms
Academic research on Algorithm Engineering with focus on algorithms for sensor networks and routing algorithms.

Mar 2005–Oct 2001 | Student Assistant at **University of Karlsruhe**, Germany
Student advisor for first year courses in Informatics (5 terms). Conception of official presentation and exercise material for first year courses in Informatics (2 terms).

Sep 1999 | Administrative Assistant at **Ostalb-Klinikum**, Aalen, Germany
Compilation of statistics and performance of analyses on operational sequences.

Aug 1999–Aug 1998 | Alternative Civilian Service at **Ostalb-Klinikum**, Aalen, Germany
Administration of other people doing alternative civilian service.

Jul 1998 | IT Assistant at **Telenot Electronic GmbH**, Aalen, Germany
Conception of a toolchain to convert product manuals for online publication.

Education

Jul 2014–
Jun 2008
Doctorate, **Karlsruhe Institute of Technology**, Germany
PhD Degree in Informatics (Dr. rer. nat.)
Grade: magna cum laude (very good)
Thesis Title: "An Algorithmic View on Sensor Networks -– Surveillance, Localization, and Communication"
Advisor: Prof. Dr. Peter Sanders

Jan 2008–
Apr 2002
Studies in Informatics, **University of Karlsruhe**, Germany
Diploma Degree in Informatics
Grade: 1.2 (very good)—top 10% of all graduates
Final Thesis: "Systematic Combination of Speed-Up Techniques for Exact Shortest-Path-Queries"
Advisor: Prof. Dr. Dorothea Wagner

Feb 2006–
Oct 1999
Studies in Physics, **University of Karlsruhe**, Germany
Diploma Degree in Physics
Grade: 1.3 (very good)
Final Thesis: "Analysis of the $t\bar{t}H$ Channel at the CMS Detector of LHC with Neural Networks"
Advisor: Prof. Dr. Günter Quast

Jun 1998–
Aug 1989
Secondary School, **Schubart Gymnasium**, Aalen, Germany
Abitur (General Qualification for University Entrance)
Grade: 1.0 (very good)—top of class
Additional merits for best degrees in Physics and Mathematics

Languages

German: Native
English: Fluent
French: Basic Knowledge

List of Publications

Journal Articles

[BDS+10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, 2010.

[LS14] Dennis Luxen and Dennis Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 2014. Accepted for publication.

Peer-reviewed Conference Papers

[BDS+08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In *International Workshop on Experimental Algorithms (WEA'08)*, *LNCS*, vol. 5038, pp. 303–318. Springer, 2008.

[FHI+14] Daniel Funke, Thomas Hauth, Vincenzo Innocente, Günter Quast, Peter Sanders, and Dennis Schieferdecker. Parallel Track Reconstruction in CMS Using the Cellular Automaton Approach. In *International Conference on Computing in High Energy and Nuclear Physics (CHEP'13)*, *JPCS*, vol. 513, pp. 1–7. IOP Publishing, 2014.

[GS10] Robert Geisberger and Dennis Schieferdecker. Heuristic Contraction Hierarchies with Approximation Guarantee. In *International Symposium on Combinatorial Search (SoCS'10)*, pp. 31–38. AAAI Press, 2010.

[KRS13] Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. Evolution and Evaluation of the Penalty Method for Alternative Graphs. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, *OASIcs*, vol. 33, pp. 94–107. Dagstuhl Publishing, 2013.

[LS12a] Dennis Luxen and Dennis Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, *LNCS*, vol. 7276, pp. 260–270. Springer, 2012.

[SH09] Dennis Schieferdecker and Marco F. Huber. Gaussian Mixture Reduction via Clustering. In *International Conference on Information Fusion (FUSION'09)*, pp. 1536–1543. IEEE, 2009.

[SS10] Peter Sanders and Dennis Schieferdecker. Lifetime Maximization of Monitoring Sensor Networks. In *International Workshop on Algorithms for Sensor Systems, Wireless Ad Hoc Networks, and Autonomous Mobile Entities (ALGOSENSORS'10)*, *LNCS*, vol. 6451, pp. 134–147. Springer, 2010.

[SVW11b] Dennis Schieferdecker, Markus Völker, and Dorothea Wagner. Efficient Algorithms for Distributed Detection of Holes and Boundaries in Wireless Networks. In *International Symposium on Experimental Algorithms (SEA'11)*, *LNCS*, vol. 6630, pp. 388–399. Springer, 2011.

Technical Reports

[KSS14] Moritz Kobitzsch, Samitha Samaranayake, and Dennis Schieferdecker. Pruning Techniques for the Stochastic on-time Arrival Problem – An Experimental Study. Preprint available at arXiv:1407.8295 [cs.DS], Karlsruhe Institute of Technology and University of California, Berkeley, 2014.

[LS12b] Dennis Luxen and Dennis Schieferdecker. Doing More for Less—Cache-Aware Parallel Contraction Hierarchies Preprocessing. Preprint available at arXiv:1208.2543 [cs.DS], Karlsruhe Institute of Technology, 2012.

[SVW11a] Dennis Schieferdecker, Markus Völker, and Dorothea Wagner. Efficient Algorithms for Distributed Detection of Holes and Boundaries in Wireless Networks. Karlsruhe Reports in Informatics 2011,8, Karlsruhe Institute of Technology, 2011.

Theses

[Sch06] Dennis Schieferdecker. Analysis of the $t\bar{t}H$ Channel at the CMS Detector of LHC with Neural Networks. Diploma thesis, University of Karlsruhe, Department of Physics, 2006.

[Sch08] Dennis Schieferdecker. Systematic Combination of Speed-Up Techniques for Exact Shortest-Path Queries. Diploma thesis, University of Karlsruhe, Department of Informatics, 2008.

Supervised Theses

[Fun13] Daniel Funke. Parallel Triplet Finding for CMS Track Reconstruction. Master thesis, Karlsruhe Institute of Technology, Department of Informatics, 2013.

[Itt09] Dominik Itte. Mehrstufiges Clustering-Verfahren zur Komponentenreduktion von Gaußmischdichten. Student research project, Karlsruhe Institute of Technology, Department of Informatics, 2009.

[Kol14] Orlin Kolev. Alternative Routes via Avoidance. Diploma thesis, Karlsruhe Institute of Technology, Department of Informatics, 2014. Ongoing thesis.

[Rad12] Marcel Radermacher. Schnelle Berechnung von Alternativgraphen. Bachelor thesis, Karlsruhe Institute of Technology, Department of Informatics, 2012.