Engineering Shortest Paths and Layout Algorithms for Large Graphs

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Thomas Willhalm

aus Villingen-Schwenningen

Tag der mündlichen Prüfung: 16.02.2005

Erste Gutachterin: Frau Prof. Dr. Dorothea Wagner

Zweiter Gutachter: Herr Prof. Dr. Rolf Möhring

Zusammenfassung

Eine der typischen Anwendungen von Kürzeste-Wege-Algorithmen sind Verkehrssysteme wie Routenplaner für Straßendaten und Auskunftsysteme für den Zug- und Busverkehr. Benutzer solcher Systeme stellen Anfragen nach der besten Reiseroute, die möglichst kurz, schnell oder kostengünstig sein soll. Abgesehen von interaktiven Systemen taucht diese Problemstellung ebenfalls als Teilproblem in der Touren- und Dispositionsplanung auf. In der Regel ändert sich in beiden Fällen der zugrunde liegende Graph wenig bis gar nicht. Des Weiteren sind durch die Anwendung geographische Koordinaten des Verkehrsnetzes gegeben, die eine Zeichnung des Graphen ermöglichen. Das algorithmische Kernproblem besteht daher darin, wiederholt kürzeste Wege in einem gerichteten Graphen mit nichtnegativen Kantengewichten zu bestimmen, für den ein Layout gegeben ist.

Wenn nun sehr viele und immer wieder die selben kürzesten Wege bestimmt werden müssen, so lohnt sich eine Vorabberechnung der kürzesten Wege, um so Anfragen schneller beantworten zu können. Verkehrsgraphen sind jedoch in der Regel sehr groß, aber auch sehr dünn, d.h. die Anzahl der Kanten im Graphen ist linear in der Anzahl der Knoten. Deshalb wächst der Speicherplatzbedarf für alle kürzesten Wege quadratisch mit der Größe des Graphen. Der nötige Speicherplatzbedarf verbietet daher das Abspeichern aller kürzesten Wege für diese Graphen mit hunderttausenden von Knoten.

Das Thema des ersten Teils meiner Arbeit ist nun die Entwicklung von *Beschleunigungtechniken für Kürzeste-Wege-Algorithmen*, die unter Benutzung der geographischen Informationen Teilergebnisse der Vorabberechnung verwenden und dabei aber nur linearen Speicherplatzbedarf haben. Die Evaluation der Algorithmen geschieht dabei in der Regel durch deren Implementierung und statistischer Auswertung mit realen und generierten Daten, wie es im relativ jungen Gebiet des *Algorithmen Engineering* üblich ist.

In meiner Arbeit wurde insbesondere der Ansatz untersucht, für jede Kante des Graphen die geographische Region in Form eines geometrischen "Container" abzuspeichern, in der diejenigen Knoten liegen, zu denen ein kürzester Weg mit dieser Kante beginnt. Bildlich gesprochen stellt man also an jeder Kante einen Wegweiser auf, der mit einer Art *Kürzeste-Wege-Container* anzeigt, in welche Region man mit dieser Kante auf einem kürzesten Weg kommen kann. In einer experimentellen Studien wurde untersucht, welche Art von geometrischen Objekten dafür besonders gut geeignet sind. Es zeigte sich, dass bereits mit einfachen Objekten wie kleinsten umschließenden, achsenparallelen Rechtecken (Bounding Boxes) eine beeindruckende Verkleinerung des Suchraums möglich ist (siehe Abb. 1).

In einer weiteren Studie wurden verschiedene Algorithmen entwickelt und verglichen,

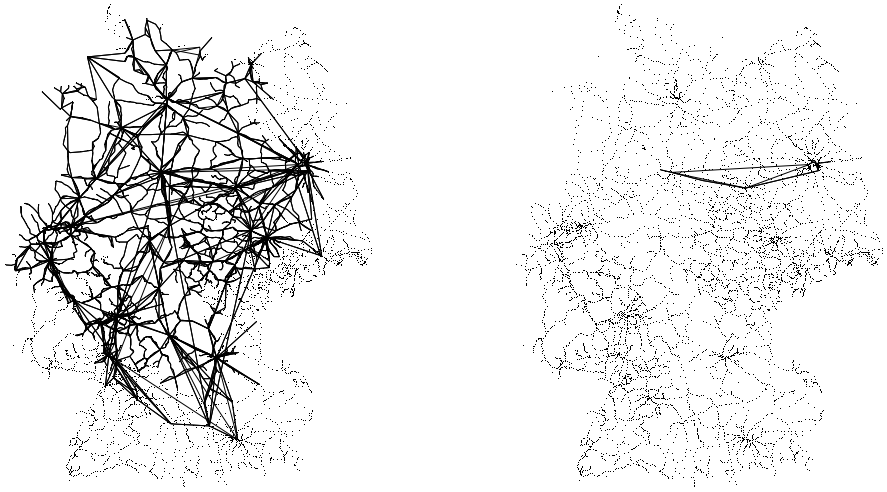


Abbildung 1: Suchraum für einen kürzesten Weg von Hannover nach Berlin ohne Beschleunigungtechnik (links) und unter Verwendung von Kürzeste-Wege-Containern (rechts).

die dieses Verfahren auf dynamische Graphen erweitern, wenn z.B. durch Baustellen oder Staus Kanten des Graphen ungültig werden bzw. durch Baumaßnahmen oder nach Beenden einer Störung neue Kanten verfügbar werden. Kürzeste-Wege-Container werden durch solche Veränderungen ungültig, wodurch Wege ausgegeben würden, die nicht die kürzesten sind. Eine komplett neue Vorverarbeitung kann jedoch vermieden werden, wenn sich nur wenige Kanten des Graphen verändern. Konkret haben wir festgestellt, dass die Vorverarbeitung in der Praxis um den Faktor 2-3 beschleunigt werden kann ohne Abstriche an die Qualität der Kürzeste-Wege-Container machen zu müssen.

Eine relativ neue Technik zur Beschleunigung von Kürzeste-Wege-Suchen verfolgt einen ziemlich ähnlichen Ansatz wie Kürzeste-Wege-Container. Sie partitioniert dazu die Knotenmenge des Graphen und speichert an jeder Kante einen *Bitvektor*, der angibt, in welchen Partitionen des Graphen Knoten liegen, die auf einem kürzesten Weg mit dieser Kante erreicht werden können. Eine experimentelle Studie geht der Frage nach, welche geometrischen Partitionierungen für diese Technik am besten geeignet sind. Des Weiteren wird eine Methode vorgestellt und getestet, den Speicherplatzbedarf der Bitvektoren durch eine geeignete Komprimierung zu reduzieren.

Schließlich untersucht eine vierte Studie, wie sich Kürzeste-Wege-Container mit anderen Beschleunigungstechniken kombinieren lassen und wie schnell diese Kombinationen sind. Konkret wurden alle 16 Kombinationen von vier Beschleunigungstechniken implementiert und miteinander unter verschiedenen Gesichtspunkten verglichen. Als besonders effektiv stellte sich dabei die Kombination von Kürzeste-Wege-Containern mit bidirektionaler Suche heraus.

Der zweite Teil meiner Arbeit beschäftigt sich mit der Frage, ob man geometrische Beschleunigungstechniken für Kürzeste-Wege-Algorithmen nutzbar machen kann, wenn die geographische Information aus irgendwelchen Gründen nicht einfach zu Verfügung gestellt

werden kann oder wenn man in Anwendungen kürzeste Wege in Graphen sucht, für die kein natürliches Layout gegeben ist. Für das Generieren dieser Layouts werden Methoden des *Graphenzeichens* verwendet, welche bei derart großen Graphen vor eine eigene Herausforderung gestellt werden.

In einer ersten Studie betrachten wir den Spezialfall eines Auskunftssystems für Bus und Bahn. (Gerade beim Busverkehr sind die Verkehrsbetriebe häufig nicht bereit, eine kostenintensive Datenaufbereitung der geographischen Koordinaten "nur" für eine Fahrplanauskunft durchzuführen.) Es war möglich, die spezielle Struktur eines Fahrplangraphen nicht nur im zur Evaluation verwendeten Auskunftssystem zu benutzen, sondern auch bei der Modellierung und Implementierung des Algorithmus' zum Zeichnen der Graphen zu verwenden. Für den Fall, dass bereits einige wenige Koordinaten vorgegeben werden, stellte sich heraus, dass generierte Koordinaten eine ähnliche Qualität bzgl. der Beschleunigungstechniken erreichen wie echte geographische Koordinaten.

Durch das positive Ergebnis für Fahrplanauskunft motiviert, führten wir eine Studie für das Kürzeste-Wege-Problem auf allgemeinen Graphen durch. Wir implementierten drei aktuelle Algorithmen zum Zeichnen großer Graphen und verglichen deren Ausgaben für Graphen aus unterschiedlichen Anwendungen in Bezug auf drei geometrische Beschleunigungstechniken und deren Kombinationen. Insbesondere die Ergebnisse für Kürzeste-Wege-Containern übertrafen mit Beschleunigungsfaktoren im dreistelligen Bereich für einige Arten von Graphen unsere Erwartungen bei weitem. Es stellte sich auch heraus, dass Zeichnungen, die optisch gleich ansprechend waren, bzgl. der Beschleunigungstechniken unterschiedlich gut waren, ja sogar in einigen Fällen weniger "schöne" Zeichnungen gut für die Suche nach kürzesten Wegen waren.

Zusammenfassend lässt sich sagen, dass man mittels Kürzeste-Wege-Containern im Vergleich zu und erst recht in der Kombination mit anderen Beschleunigungstechniken die Suche nach kürzesten Wegen enorm verbessern kann und dass man dabei durchaus auf einfache geometrische Objekte wie achsenparallele Rechtecke zurückgreifen kann. Darüberhinaus kann der Ansatz auf die Fälle erweitert werden, dass sich der Graph verändert oder dass keine oder nur wenige Koordinaten für Knoten des Graphen gegeben sind.

Acknowledgments

First of all, I would like to thank my supervisor Dorothea Wagner for her guidance and support. Her office door was never closed for me and my problems of any kind. Furthermore, a big "thank you" goes to all the co-authors of my publications. I really enjoyed the collaboration with and advice from Uli. I'm glad to share not only the office with Frank but also many, many fruitful discussions. Martin was always very patient in discussing and realizing proposed projects. Heiko traveled a lot to discuss and work on new ideas. I'm grateful to Christos for his kind invitation to Patras and the friendly guidance and collaboration there. Furthermore, I would like to thank Greg for the warm welcome and all his tips on how to survive in Greece. Apart from co-authors, there are quite a few colleagues with whom I could clarify and discuss many problems. They always had time to listen to my crazy ideas or answer my silly questions. These people are Annegret, Sabine, Leon, Jan-Willem, Teggy, Jens, and Jürgen in Constance and Marco, Sascha, Steffen, Michael, and Silke in Karlsruhe. My work was also supported by the help of students to implement and test the algorithms. In particular, I would like to thank Jasper, Volker, Birk, and Sebastian. Since life is not restricted to university and a dissertation must also be supported outside of it, I would like to thank Christina for her support and understanding. Last, but not least, I am thankful that the referees Dorothea Wagner and Rolf Möhring offered their time to write their reports.

Contents

Part I

Shortest Paths

Chapter 1

Introduction

Engineering algorithms has attracted increasing attention during the last years. This thesis focuses on the engineering of algorithms for large graphs for two types of problems: to find shortest paths and to draw the graph nicely. Of particular interest is the connection of the two, namely to draw a graph in order to find shortest paths faster.

Shortest Paths

Computing shortest paths is a base operation for many problems in traffic applications. The most prominent are certainly route planning systems for cars, bikes and hikers [1, 2, 3], or scheduled vehicles like trains and buses [4, 5, 6, 7, 8]. If such a system is realized as a central server, it has to answer a huge number of customer queries asking for their best itineraries. Users of such a system continuously enter their requests for finding their "best" connections. Furthermore, similar queries appear as sub-problems in line planning, timetable generation, tour planning, logistics, and traffic simulations. The main goal is to reduce the (average) response time for answering a query.

The algorithmic core problem that underlies the above applications is the single-source shortest-path problem on a given directed graph with non-negative edge lengths related to a layout of the graph which is also provided. The particular graph is quite large (though sparse), and hence only space requirements are acceptable that are linear in the number of nodes. One of the features of travel planning is the fact that the network hardly changes for a certain period of time while there are many queries for shortest paths. This justifies a heavy preprocessing of the network to speed up the queries (see e.g., [7, 9]). Although pre-computing and storing the shortest paths for all pairs of nodes would give us "constant-time" shortest-path queries, the quadratic space requirement for traffic networks with more than 10^5 nodes makes it prohibitive. The most commonly used approach for answering shortest path queries concerns variants of Dijkstra's algorithm [1], targeting at reducing its *search-space* (the number of nodes visited by the algorithm).

In the first part of this thesis, we explore the possibility to reduce the search space of Dijkstra's algorithm by using precomputed information that can be stored in $\mathcal{O}(n + m)$ space. We can use a given layout of the graph to extract geometric information to answer

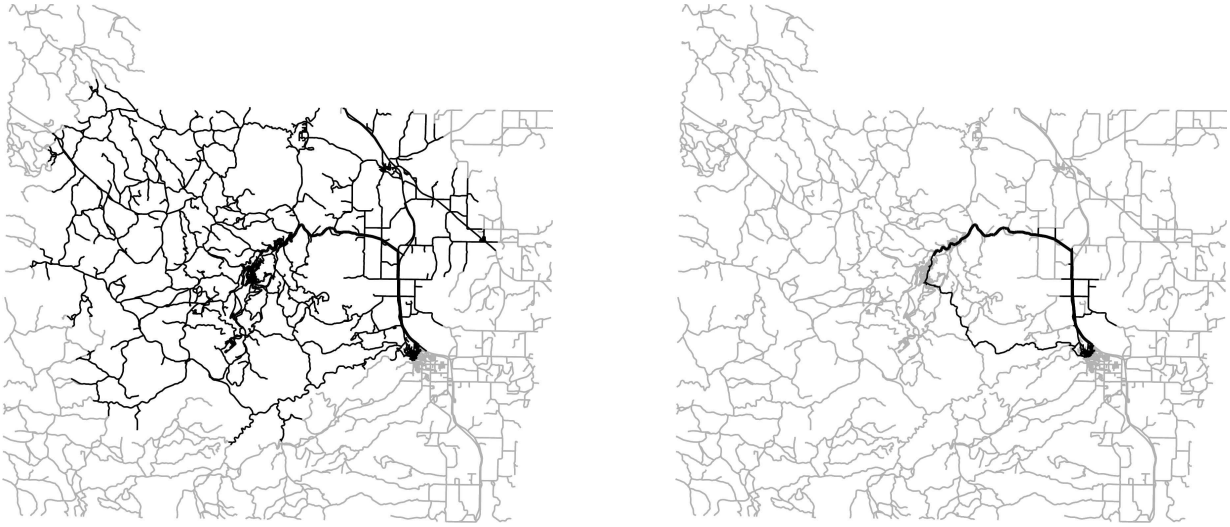


Figure 1.1: The search space of DIJKSTRA'S ALGORITHM (left) and DIJKSTRA'S ALGORITHM WITH PRUNING using bounding boxes (right).

the on-line queries fast. (Figure 1.1 gives an illustrative example for a street network.) We use a very fundamental observation on shortest paths. In general, an edge that is not the first edge on a shortest path to the target can be safely ignored in any shortest path computation to this target. More precisely, we apply the following concept. In the preprocessing, for each edge e a set of nodes $S(e)$ is computed which are the nodes that can be reached by a shortest path starting with e. While running Dijkstra's algorithm, those edges e for which the target is not in $S(e)$ are ignored.

As storing all sets $S(e)$ would need $\mathcal{O}(mn)$ space, we relax the condition by storing a geometric object for each edge that contains *at least* the nodes in $S(e)$. We will call such an object a *shortest-path container* as it contains (at least) the targets of shortest paths. The shortest-path queries are then answered by Dijkstra's algorithm restricted to those edges for which the target node is inside their associated geometric object. Note that this method does in fact still lead to a correct result, but may increase the number of visited nodes to more than the strict minimum (i.e., the number of nodes in the shortest path). In order to generate the geometric objects, a layout $L : V \to \mathbb{R}^2$ is used. For the application of travel information systems, such a layout is given by the geographic locations of the nodes. It is however not required that the edge lengths are derived from the layout. In fact, for some of our experimental data this is not even the case.

We would like to mention that a particular type of geometric objects, the angular sectors, has been introduced in [7] for the special case of a time table information system. Our results, however, are more general in two respects: (a) we examine the impact of various different geometric objects; and (b) we consider Dijkstra's algorithm for general embedded graphs. In Section 2.3, we present an extensive experimental study comparing the impact of these objects using real-world test data from railway and street networks. It

turns out that a significant improvement can be achieved by using other geometric objects than angular sectors.

The following section concerns the *dynamic version* of the above mentioned scenario; namely, the case where the graph may dynamically change over time as streets may be blocked, built, or destroyed, and trains may be added or canceled. In this work, we present new algorithms that dynamically maintain geometric containers when the weight of an edge is increased or decreased (note that these cases cover also edge deletions and insertions). We also report on an experimental study with real-world railway data. Our experiments show that the new algorithms are 2-3 times faster than the naive approach of recomputing the geometric containers from scratch.

Our dynamic algorithms are perhaps the first results towards an efficient algorithm for the dynamic single-source shortest-path problem without using the output complexity model – introduced in [10, 11] and extended in [12, 13] – under which algorithms for the dynamic single-source shortest-path problem are usually analyzed. We would also like to mention that existing approaches for the dynamic all-pairs shortest paths problem (see e.g., [14, 15, 16, 17, 18], and [19] for a recent overview) are not applicable to maintain geometric containers, because of their inherent quadratic space requirements.

In [20, 21], a speed-up technique is introduced that is very similar to the geometric containers in Section 2.3 but uses—at least theoretically—more than linear space. The graph is partitioned into regions and the shortest-path container for an edge is not a geometric object but a *bit-vector* where each bit corresponds to one region. A bit is set for an edge if a shortest path to a node in the corresponding region exists that starts with this edge. Also this method does in fact still lead to a correct shortest path, and if a node t is inside a marked region for an edge e but no shortest path to this node starts with the edge e, then shortest-path queries to t may unnecessarily consider the edge e.

Apart from their impressing speed-up factors, bit-vectors as shortest-path containers have the advantage that the preprocessing can be realized without the computation of all-pairs shortest paths. In fact, it is sufficient in practice to perform a single-source shortest-path computation for all nodes that are on the boundary of a region. This insight substantially reduces the preprocessing step.

In Section 2.5, we compare the impact of different geometric partitioning methods like grids, quadtrees or kd-trees. Furthermore, we examine a two-level approach for the bit-vectors which reduces the space requirement but maintains the speed improvement. The two-level bit-vectors compress the bits for nodes that are far apart and therefore exact information is less important for the shortest-path search.

In Chapter 3, the *shortest-path containers are compared* with other known speed-up techniques. The further techniques that we consider are the classic goal-directed and bidirectional search, as well as a multi-level approach for shortest-path computations. In a goal-directed search [22], a potential function is added to the priorities in the priority queues. If the potential function is chosen carefully, the search is pushed towards the target while a correct shortest-path can be guaranteed. A bidirectional search starts two searches simultaneously at the source and at the target node towards the source. If the two search horizons meet, a shortest path from the source to the target can be composed of

two sub-paths in the two search spaces. The multi-level approach [23] enriches the graph by additional edges which represent shortest path. The graph is decomposed by a node separator and the additional edges connect the separator nodes among each other and with the nodes in the adjacent region.

Even more important than the comparison of the 4 speed-up techniques is the evaluation of all 16 *combinations*. We examine and discuss which techniques combine well and for which combinations the speed-up does not scale.

The last chapter of the first part of this thesis concerns methodological issues regarding our *implementation*, which have been carried out in `C++`. Implementing and supporting that many variations of Dijkstra's algorithm in `C++` is a tedious task if it is not planned carefully. We employ several techniques to maintain a common code base that is at the same time small, flexible and efficient. We use a blend of the design pattern *template method* [24], parameterized inheritance [25, 26], and template meta-programming [27, 28].

Adding functionality to graph algorithms can be achieved by the design pattern *template method* [24] or an extension of the design pattern *visitor*, the approach of the BOOST graph library [29]. Our work deviates from the latter and is closer to the former, which it actually enhances to grasp parts of *aspect-oriented programming*. Aspect-oriented programming tries to provide a modular way to overcome the single dimension of functional decomposition by the design pattern *template method* (see e.g., [30]). More precisely, it is necessary to change the inheritance hierarchy to create arbitrary combinations of aspects. To support aspect-oriented programming in `C++`, an extension of the `C++` language is proposed in [30]. However, such an extension can be avoided through the use of parameterized inheritance [26] (also known as mix-in classes [25]) and template meta-programming [27, 28], which provides the base to a solution with standard `C++` compilers.

Graph Drawing

The second part of this thesis is dedicated to graph drawing of large sparse graphs. In the first chapter, we provide an overview over known algorithms that are currently known to perform well even on large graphs. Algorithms in this area are often more a product of algorithmic engineering than a clear algorithm with exact solution. Therefore, it is more precise to say that we present *groups* of algorithms that share a common idea. The three methods that are presented are the following:

Force-Directed Layout. The idea to embed a graph according to forces of springs is due to Eades [31]. We will introduce it as an extension of a barycentric layout [32]. It has been extended later by [33, 34] with forces that reflect the structure of the graph. Recently, three variations of a multi-level spring-embedder [35, 36, 37] have been developed to use this approach for large graphs.

Spectral Layout. A spectral layout algorithm draws the graph with eigenvectors of a matrix that is derived from the graph. Although this method is fairly old [38], it has not drawn much attention by the graph drawing community until recently, where, again, a multi-level variant made it feasible for large graphs [39].

High-Dimensional Embedding. This very recent method constructs a drawing in a very high-dimensional space first, and then, this drawing is projected to the plane using principal component analysis [40]. The high-dimensional drawing is determined with shortest-path trees, which fits nicely into the picture of this thesis.

After the presentation of the graph drawing techniques in Chapter 5, we describe an application for bibliographic networks in Chapter 6, which combines methods of two of them. It illustrates nicely how a concrete application can be engineered using and combining ideas from the mainstream graph drawing algorithms. The bibliographic network is visualized as a landscape where the text entities are represented by houses. The location of the houses are determined with a spectral layout whereas the elevation is given by a centrality measure. The surface of the landscape is computed with a variant of the barycentric layout. However, the algorithm for the surface prevents edges and nodes of the main graph to cross the surface. Therefore, the large and distracting graph is hidden below the surface and only the most prominent papers are on the surface.

After this illustrative example, Chapter 7 presents an application of graph drawing where the visualization itself is not the goal. The application stems from the scenario of a timetable information system that answers customer queries for fast connections. As we already mentioned in the first part of the thesis, these queries can be modeled as a special variant of a shortest-path problem in a specific timetable graph [7]. Moreover, many of speed-up techniques can be adopted to this application. For geometric speed-up techniques, a layout of the graph is provided by the geographic location of the stations and bus stops. Surprisingly, some providers for public transport do not maintain (accurate) coordinates for their transportation network. This is mainly the case for local bus companies that can plan and operate without much technical support. However, their timetables are integrated in larger travel planning systems, which results in timetable graphs with a partially unknown layout. Chapter 7 shows how to model the (re-)construction of a layout which can be used for geometric speed-up techniques. A specific layout algorithm is presented that makes use of the special structure of the graph. The results of the layout algorithm are evaluated experimentally with the (simplified) timetable information system from [7].

The last chapter before the conclusion extends these results to general weighted undirected graphs. Without any special structure of graphs, all three approaches of Chapter 5 can be used. We examine experimentally layouts created by these algorithms with three geometric speed-up techniques (goal-directed search, shortest-path containers, and reach-based routing). The geometric speed-up techniques are combined and compared with bidirectional search, because bidirectional search can be applied without any preprocessing. Where a layout is provided for the test graphs, it is included in the comparison.

1.1 Definitions and Problem Description

1.1.1 Graphs

An *(undirected) graph* G is a pair (V, E), where V is a finite set of *nodes* and E is a set of *edges*, where an edge is an unordered pair $\{u, v\}$ of nodes $u, v \in V$. If the edges are ordered pairs (u, v) of nodes $u, v \in V$, we call the graph *directed*. Throughout this thesis, the number of nodes $|V|$ is denoted by n and the number of edges $|E|$ is denoted by m. For a node $u \in V$, the number of outgoing edges $|\{(u, v) \in E\}|$ is called the *degree* of the node. Given edge weights $l : E \to \mathbb{R}_0^+$, the *weighted degree* of u is $\sum_{(u,v)\in E} l(u, v)$.

A graph (without multiple edges) can have up to $\mathcal{O}(n^2)$ edges. We call a graph *sparse*, if $m \in \mathcal{O}(n)$, and we call a graph *large*, if one can only afford a memory consumption in $\mathcal{O}(n)$. In particular, for large sparse graphs $\mathcal{O}(n^2)$ space is not affordable.

A *path* in G is a sequence of nodes $(u_1, \ldots, u_k)$ such that $\{u_i, u_{i+1}\} \in E$ for all $1 \leq i < k$. If G is directed, a path must respect the direction of the edges, i.e. $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. A path with $u_1 = u_k$ is called a *cycle*.

Given edge weights $l : E \to \mathbb{R}_0^+$ ("lengths"), the *length of a path* $P = (u_1, \ldots, u_k)$ is the sum of the lengths of its edges $l(P) := \sum_{1\leq i<k} l(u_i, u_{i+1})$. For two nodes $s, t \in V$, a *shortest s-t path* is a path of minimal length with $u_1 = s$ and $u_k = t$. The *(graph-theoretic) distance* $d(s, t)$ of s and t is the length of a shortest s-t path.

We define the *reverse graph* G_{rev} of a directed graph $G = (V, E)$ as the graph $G_{\text{rev}} = (V, E_{\text{rev}})$ with $E_{\text{rev}} = \{(u, v)|(v, u) \in E\}$. If the graph is weighted by $l : E \to \mathbb{R}$, the reverse graph is weighted by $l_{\text{rev}}(u, v) = l(v, u)$. In words, the reverse-graph is the graph G with all edges reversed. It is easy to see that $(s, \ldots, t)$ is a shortest path from s to t in G iff $(t, \ldots, s$ is a shortest path in G_{rev} with the same edges reversed.

A *layout* of a graph $G = (V, E)$ is a function $L : V \to \mathbb{R}^d$ that assigns each node a position in $\mathbb{R}^d$ with $d \in \mathbb{N}$. A layout $L : V \to \mathbb{R}^d$ of a graph is called *balanced*, iff the center of gravity is the origin or, more formally, $\frac{1}{n}\sum_{v\in V} L(v) = 0$. Most layouts will be in the Euclidean plane, i.e. $d = 2$. For ease of notation, we will sometimes identify a node $v \in V$ with its location $L(v) \in \mathbb{R}^2$ in the plane, if the layout is fixed. The Euclidean distance between two nodes $u, v \in V$ is then denoted by $\|u - v\|$.

1.1.2 Shortest Path Problem

Let $G = (V, E)$ be a directed graph whose edges are *weighted* by a function $l : E \to \mathbb{R}$. We interpret the weights as edge lengths in the sense that the *length of a path* is the sum of the weights of its edges. The *(single-source single-target) shortest-path problem* consists in finding a path of minimum length from a given source $s \in V$ to a given target $t \in V$. Note that the problem is only well defined for all pairs, if G does not contain negative cycles (cycles with negative length). In the presence of negative weights but not negative cycles, it is possible, using Johnson's algorithm [41], to convert in $\mathcal{O}(nm + n^2 \log n)$ time the original edge weights $l : E \to \mathbb{R}$ to non-negative edge weights $l' : E \to \mathbb{R}_0^+$ that result in the same shortest paths. Hence, in the rest of this thesis, we can safely assume that

```
1  for all nodes u ∈ V set dist(u) := ∞
2  initialize priority queue Q with source s and set dist(s) := 0
3  while priority queue Q is not empty
4      get node u with smallest tentative distance dist(u) in Q
5      for all neighbor nodes v of u
7          set new-dist := dist(u) + l(u, v)
8          if new-dist < dist(v)
9              if dist(v) = ∞
10                 insert neighbor node v in Q with priority new-dist
11             else
12                 set priority of neighbor node v in Q to new-dist
13             set dist(v) := new-dist
```

Algorithm 1: DIJKSTRA'S ALGORITHM.

edge weights are non-negative. Throughout the thesis we also assume that for all pairs $(s,t) \in V \times V$, the shortest path from s to t is unique. (This can be achieved by adding a small fraction to the edge weights, if necessary.)

The classical algorithm for computing shortest paths in a directed graph with non-negative edge weights is that of Dijkstra [42] (Algorithm 1). The algorithm maintains, for each node $v \in V$, a label $\texttt{dist}(v)$ with the current tentative distance. The algorithm uses a priority queue Q containing the nodes that build the current search horizon around s. Nodes are either *unvisited* (i.e. $\texttt{dist}(u) = \infty$), in the priority queue, or *finished* (already removed from the priority queue). It is easy to verify that nodes are inserted at most once in the priority queue if the extracted node u in line 4 is the node with the smallest tentative distance in the priority queue and all edge weights are non-negative. Thus, the labels are updated while the algorithm visits the nodes of the graph with non-decreasing distance from the source s.

In order to compute a shortest path tree, one has to remember that u is the predecessor of v if a shorter path to v has been found (i.e. between line 8 and 9). DIJKSTRA'S ALGORITHM computes the shortest paths to all nodes in the graph. If only one shortest path is needed to a target node $t \in V$, the algorithm can stop if the target t is removed from the priority queue in line 4. If DIJKSTRA'S ALGORITHM is executed more than once, the initialization of `dist` in line 1 for each run can be omitted by introducing a global integer variable `time` and replacing the test $\texttt{dist}(v) = \infty$ by a comparison of the `time` with a time stamp for every node. (See e.g., [7] for a detailed description.)

The asymptotic time complexity of DIJKSTRA'S ALGORITHM depends on the choice of the priority queue. For general graphs, Fibonacci heaps [43] still provide the best theoretical worst-case time of $\mathcal{O}(m + n \log n)$. For sparse graphs, binary heaps result in the same asymptotic time complexity. Even more, binary heaps are (1) easier to implement and (2) perform better for many instances in practice [44].

For special cases of edge weights, better algorithms are known. If edge weight are

integral and bounded by a small constant, Dial's implementation [45] with an array of lists ("buckets") provides a priority queue where all operations take constant time. An extension with average linear complexity for uniformly distributed edge weights is presented in [46, 47]. Note however, that the better a speed-up technique works, the smaller the search front is, and the less important the priority queue is.

1.2 Speed-up Techniques

In this section, we present known speed-up techniques with which we will compare and combine shortest path containers (Chapter 2.3). In all cases, our base algorithm is Dijkstra's algorithm as the speed-up techniques rely on the fact that Dijkstra's algorithm does not need to visit all nodes in the graph.

1.2.1 Bidirectional Search

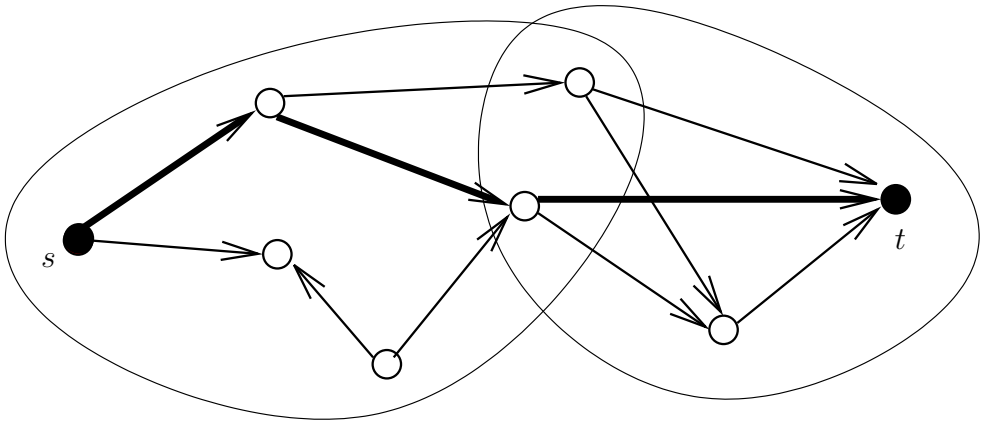


Figure 1.2: Bidirectional search performs two searches "in parallel," a forward and a backward search. The algorithm stops when a node has been removed from both priority queues. A shortest path consists of a path from the source s to a node in the intersection of the search horizons and continues to the target t.

As depicted in Fig. 1.2, the bidirectional search simultaneously performs two searches: a "normal," or forward, variant of the algorithm, starting at the source node; and a so-called reverse, or backward, variant of Dijkstra's algorithm, starting at the destination node. With the reverse variant, the algorithm is applied to the reverse graph, i.e., a graph with the same node set V as that of the original graph, and the reverse edge set $\overline{E} = \{(u, v) \mid (v, u) \in E\}$.

Let $\texttt{dist}_f(u)$ be the distance labels of the forward search and $\texttt{dist}_b(u)$ the labels of the backward search, respectively. The algorithm can be terminated when one node has been designated to be permanent by both the forward and the backward algorithm. Then, the

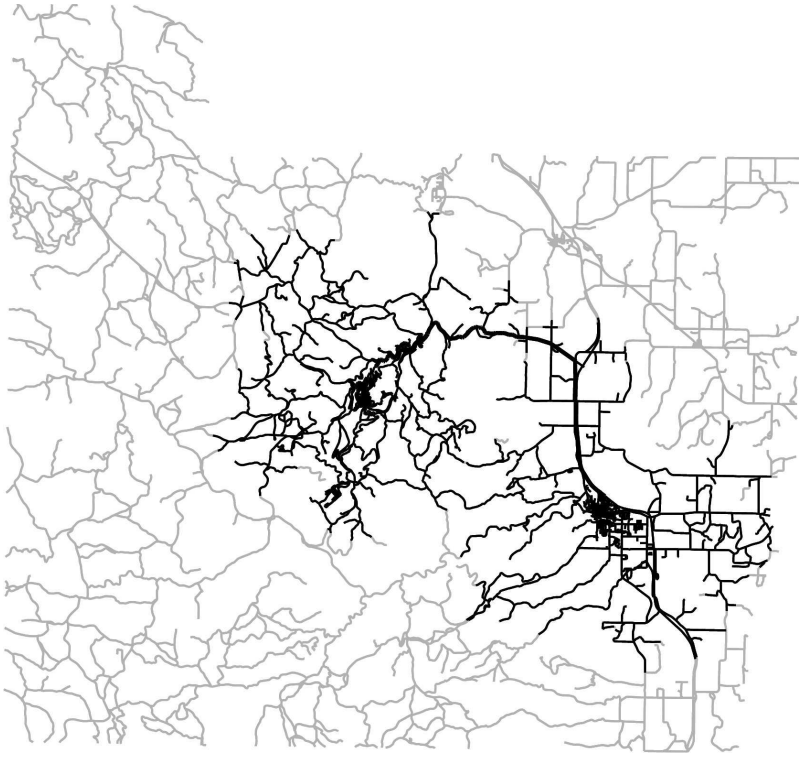


Figure 1.3: The search space of a bidirectional search for the same query as in Fig. 1.1.

shortest path is determined by the node u with minimum value $\texttt{dist}_f(u) + \texttt{dist}_b(u)$ and it can be composed of the shortest path from the start node s to u, (found by the forward search), and the shortest path from u to the destination t (found by the reverse search). Note that the node u itself is not necessarily marked as permanent by both searches.

Intuitively, the visited nodes in a unidirectional algorithm form a ball around the source of the search. For grid-like graphs such as transportation networks, one can expect that the search space is roughly r^2 where r denotes the number of nodes on the shortest path. In the bidirectional search, however, two searches with half of the radius are performed (Fig. 1.3). The number of visited nodes is therefore approximately $2 \cdot (\frac{r}{2})^2 = \frac{1}{2}r^2$. This estimation is not completely achieved in practice, because the search horizon may be limited by the size of the graph. However, bidirectional search scales fairly well in combination with other speed-up techniques.

One degree of freedom in bidirectional search is the choice whether a forward or backward step is executed. Common strategies are to choose the direction with the smaller priority queue, to select the direction with the smaller minimal distance in the priority queue, or to simply alternate the directions. As our goal is to reduce the total number of visited nodes, the first strategy usually results in the fastest algorithm.

1.2.2 Goal-Directed Search or A^*

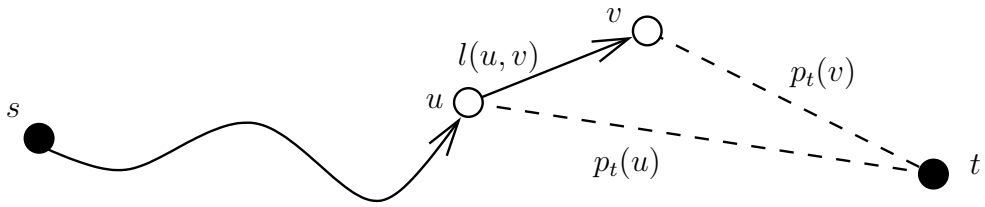


Figure 1.4: Goal-directed search uses modified edge lengths $l'(u,v) := l(u,v) - p_t(v) + p_t(u)$ where $p_t(v)$ provides a lower bound for the length of a shortest v-t paths. A path from s to t is a shortest s-t path according to l', if and only if it is a shortest s-t path according to l.

This technique, originating from AI, modifies the priority of active nodes to change the order in which the nodes are processed. More precisely, a goal-directed search adds to the priority $\texttt{dist}(u)$ a *potential* $p_t : V \to \mathbb{R}_0^+$ (often called *heuristic*) depending on the target t of the search [22]. The modified priority of a node $v \in V$ is therefore $\texttt{dist}(v) + p_t(v)$. With a suited potential, the search can be pushed towards the target thereby reducing the running time while the algorithm provably returns a shortest path. (See Fig. 1.5.)

Intuitively speaking, one can compare a path in traffic network with a walk in a landscape. If you add a potential, the affected region is raised. If the added potential is small next to the target, you create a valley around the target. As walking downhill is easier than uphill, you are likely to hit the target sooner than without the potential added.

We will now use an alternative formulation of goal-directed search to discuss its correctness. Equivalently to modifying the priority, one can change the edge lengths such that the search is driven towards the target t (illustrated in Fig. 1.4). In this case, the weight of an edge $(u,v) \in E$ is replaced by $l'(u,v) := l(u,v) - p_t(u) + p_t(v)$. The length of a s-v path $P = (s = v_1, v_2, \ldots, v_{k+1} = v)$ is then

$$\begin{aligned} l'(P) = \sum_{i=1}^{k} l'(v_i, v_{i+1}) &= \sum_{i=1}^{k} l(v_i, v_{i+1}) - p_t(v_i) + p_t(v_{i+1}) \\ &= -p_t(s) + p_t(v) + \sum_{i=1}^{k} l(v_i, v_{i+1}) \\ &= -p_t(s) + p_t(v) + l(P). \end{aligned}$$

Up to the constant $-p_t(s)$, the length of P with modified edge lengths is therefore $p_t(u)$. In particular, the length of an s-t path with modified edge lengths is the same up to the constant $-p_t(s) + p_t(t)$. Therefore, a path from s to t is a shortest s-t path according to l', if and only if it is a shortest s-t path according to l.

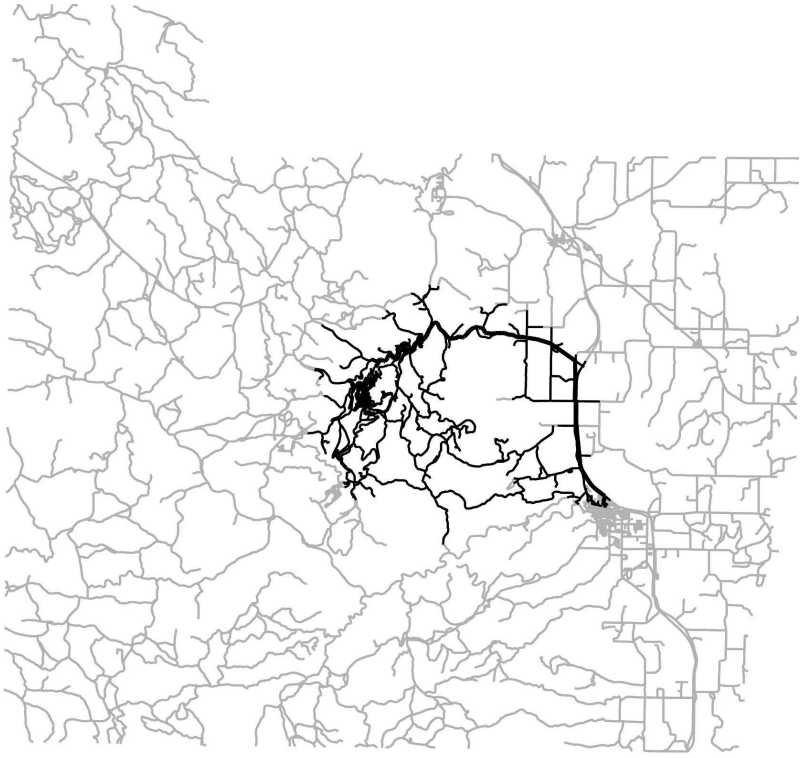


Figure 1.5: The search space of a goal-directed search for the same query as in Fig. 1.1.

If all modified edge lengths $l'(u, v)$ are non-negative, we can apply Dijkstra's algorithm to the graph with modified edge lengths l' and get a shortest s-t path according to l. This leads to the following definition:

Definition 1 *Given a weighted graph $G = (V, E), l : V \to \mathbb{R}_0^+$, a potential $p : V \to \mathbb{R}$ is called* feasible*, if $l(u, v) - p(u) + p(v) \geq 0$ for all edges $e \in E$.*

Usually, potentials are used that estimate the distance to the target. In fact, it can be shown that a feasible potential p is a lower bound of the distance to the target t if $p(t) \leq 0$. Note that every feasible potential p can be transposed into an equivalent potential $p'(v) = p(v) - p(t)$, which is a lower bound of the distance to the target. We can, therefore, assume without loss of generality that the potential is indeed a lower bound. The tighter the bound is, the more the search is attracted to the target. In particular, a goal-directed search visits only nodes on the shortest path, if the potential is the distance to the target.

In an actual implementation of goal-directed search, you will most likely use the first formulation, namely to modify the priority with which nodes are inserted in the priority queue. This has the advantage that p is called (at most) once per edge instead of two calls. Furthermore, the distance labels of the nodes are unmodified. This improves the numerical stability and simplifies the handling of the labels (in particular in combinations with other speed-up techniques).

We will now present three scenarios and how to obtain feasible potentials in these cases:

Given a layout $L : V \to \mathbb{R}^2$ of the graph where the length of an edge is correlated with the distance of its end nodes. A feasible potential for a node v can be obtained using the Euclidean distance (the "flight distance") $\|L(v) - L(t)\|$ to the target t.

In case the edge lengths are in fact the Euclidean distances, the Euclidean distance $\|L(v) - L(t)\|$ itself is already a feasible potential, due to the triangular inequality. Using this potential, an edge that points directly towards the destination has a modified edge length of zero while the modified length of an edge that points in the opposite direction is twice the distance.

If the edge lengths are *not* the Euclidean distances of the end nodes, a feasible potential can be defined as follows: let v_{max} denote the maximum "edge-speed" $\|L(u) - L(v)\|/l(u, v)$, over all edges $(u, v) \in E$. The potential of a node u can now be defined as $p(u) = \|L(u) - L(t)\|/v_{max}$. The maximum velocity can be computed in a preprocessing step by a linear scan over all edges. Numerical problems can be reduced if the maximum velocity is multiplied by $1 + \varepsilon$ for a small $\varepsilon > 0$.

This approach can be extended in a straight forward manner to other metric spaces than $(\mathbb{R}^2, \|\cdot\|)$. In particular, it is possible to use more than two dimensions or other metrics like the Manhattan metric. Finally, the expensive square root function to compute the Euclidean distance can be replaced by an approximation.

Usually, the speed-up for goal-directed search is around 2 if the distance is estimated with a metric.

With a preprocessing, it is possible to gather information about the graph that can be used to obtain improved lower bounds. In [48], a small fixed-sized subset $L \subset V$ of "landmarks" is chosen. Then, for all nodes $v \in V$, the distance $d(v, l)$ to all nodes $l \in L$ is precomputed and stored. These distances can be used to determine a feasible potential. For each landmark $l \in L$, we define the potential $p_t^{(l)}(v) := d(v, l) - d(t, l)$. Due to the triangle inequality $d(v, l) \leq d(v, t) + d(t, v)$, the potential $p_t^{(l)}$ is feasible and indeed a lower bound for the distance to t. The potential is then defined as the maximum over all potentials: $p_t(v) := \max\{p_t^{(l)}(v); l \in L\}$. It is easy to show that the maximum of feasible potentials is a feasible potential, too.

For landmarks that are situated next to or "behind" the target t, the lower bound $p_t^{(l)}(u)$ should be fairly tight, as shortest paths to t and l most likely share a common sub-path. Landmarks in other regions of the graph, however, may attract the search to themselves. This insight justifies to consider, in a specific search from s to t, only those landmarks with the highest potential $p_t^{(l)}(u)$. The restriction of the landmarks in use has the advantage that the calculation of the potential is faster while its quality is improved.

Using landmarks, goal-directed search is usually up to 10 times faster than plain Dijkstra.

For restricted shortest-path problems, performing a single run of an unrestricted Dijkstra's algorithm is a relatively inexpensive operation. Examples are travel planning systems for scheduled vehicles like buses or trains. The complexity of the problem is much higher if you take connections, vehicle types, transfer times, or traffic days into account.

It is therefore feasible to perform a shortest-path computation to find tighter lower bounds [8]. More precisely, you run Dijkstra's algorithm on a condensed graph of the full, time-expanded graph: The nodes of this graph are the stations (or stops) and an edge between two stations exists iff there is a non-stop connection. The edges are weighted by the minimal travel time. The distances of all $v \in V$ to the target t can be obtained by a single run of Dijkstra's algorithm from the target t with reversed edges. They provide a feasible potential for the time-expanded graph, since the distances are a feasible potential in the condensed graph and an edge between two stations in the time-expanded graph is at least as long as the corresponding edge in the condensed graph.

1.2.3 Multi-Level Approach

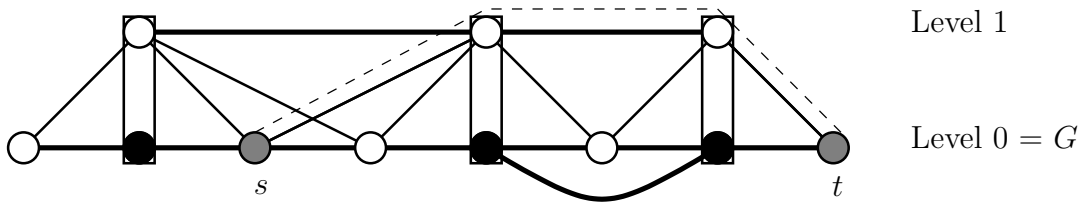


Figure 1.6: The multi-level approach enhances the graph G by adding a coarsened graph. The new level is induced by a separator S_1 (black nodes). An edge in the coarsened graph (level 1) represents a shortest path in the original graph (level 0). Inter-level edges connect all nodes in a connected component of $G \backslash S_1$ to the separator nodes on the border of the connected component.

This speed-up technique requires a preprocessing step at which the input graph $G = (V, E)$ is decomposed into $l+1$ ($l \geq 1$) levels and enhanced with additional edges representing shortest paths between certain nodes. The additional edges can be seen as "bridges" or "short-cuts" for Dijkstra's algorithm. (Of course, the "short-cuts" are not shorter concerning the length of the path but only the number of visited edges.) An illustrative example for $l = 1$ is shown in Fig. 1.6. The decomposition of the graph depends on separators $S_i \subset V$ for each level, called selected nodes at level i: $S_0 := V \supseteq S_1 \supseteq \ldots \supseteq S_l$. These node sets can be determined on diverse criteria. In a simple, but practical implementation, they consist of the desired numbers of nodes with highest degree in the graph. (However,

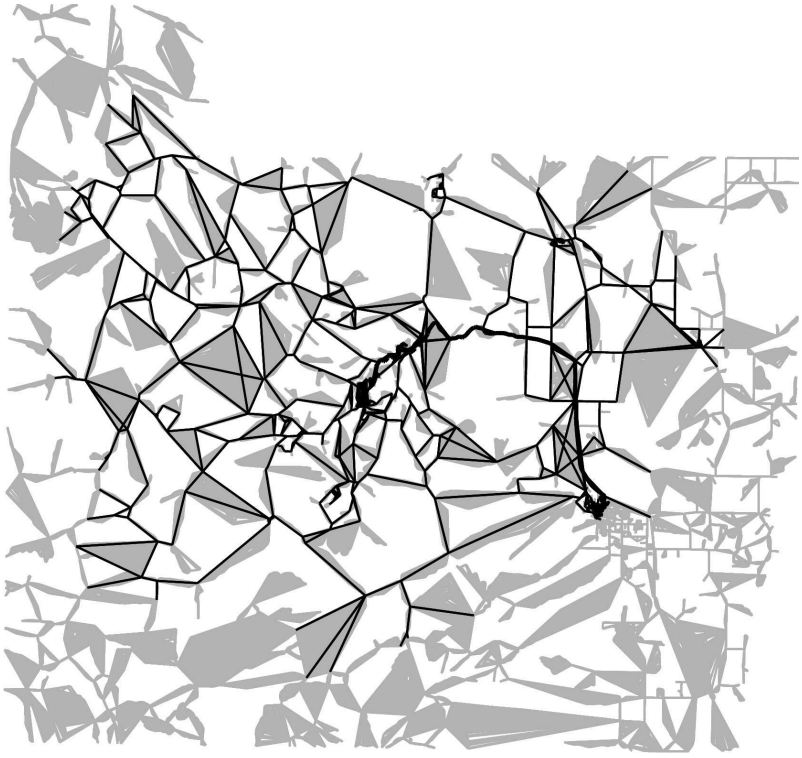


Figure 1.7: The search space using a multi-level approach for the same query as in Fig. 1.1.

with domain-specific knowledge about the central nodes in the graph, better separators can be found [23].)

There are three different types of edges being added to the graph: *upward edges*, going from a node that is not selected at one level to a node selected at that level; *downward edges*, going from selected to non-selected nodes; and *level edges*, passing between selected nodes at one level. The weight of such an edge is assigned the length of a shortest path between the end-nodes.

To find a shortest path between two nodes s and t, then, it suffices for Dijkstra's algorithm to consider a relatively small subgraph of the multi-level graph (Fig. 1.7). The hierarchical structure of the multi-level graph entails that a shortest path from s to t can be represented by a certain set of upward and of downward edges and a set of level edges passing at a maximal level that has to be taken into account.

1.2.4 Reach-Based Routing

This fairly recent approach prunes the search space based on a centrality measure called "reach" [49]. Intuitively, a node in the graph is important for shortest paths, if it is situated in the middle of long shortest paths. Nodes that are only at the beginning or the end of

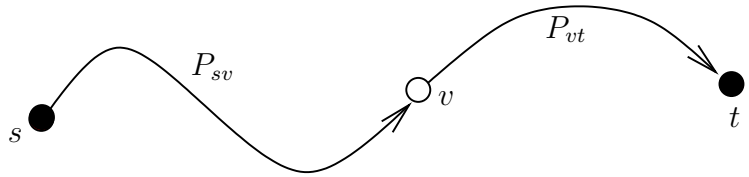


Figure 1.8: The *reach on the path* P of a node v is defined as the minimum of the length of P_{sv} and the length of P_{vt}. The *reach* $r(v)$ is defined as the maximum reach for all shortest s-t paths in G containing v.

long shortest paths are less central. This leads to the following formal definition:

Definition 2 (Reach) *Given a weighted graph* $G = (V, E), l : E \to \mathbb{R}_0^+$ *and a shortest* s-t *path* P*, the* reach on the path P *of a node* $v \in P$ *is defined as* $r(v, P) := \min\{l(P_{sv}), l(P_{vt})\}$ *where* P_{sv} *and* P_{vt} *denote the sub-paths of* P *from* s *to* v *and from* v *to* t*, respectively (see Fig. 1.8). The* reach $r(v)$ *of* $v \in V$ *is defined as the maximum reach for all shortest* s-t *paths in* G *containing* v*.*

In a search for a shortest s-t path P_{st}, a node $v \in V$ can be ignored, if (1) the distance $l(P_{sv})$ from s to v is larger than the reach of v and (2) the distance $l(P_{vt})$ from v to t is larger than the reach of v. While performing Dijkstra's algorithm, the first condition is easy to check, since $l(P_{sv})$ is already known. The second condition is fulfilled if the reach is smaller than a lower bound of the distance from v to t. (Suited lower bounds for the distance of a node to the target are already described for goal-directed search in Section 1.2.2.) Lines 7-13 of algorithm 1 are therefore not performed if conditions (1) and (2) are certainly fulfilled.

To compute the reach for all nodes, we perform a single-source all-target shortest-path computation for every node. With a modified depth first search on the shortest-path trees, it is easy to compute the reach of all nodes using the following insight: For two shortest paths P_{sx} and P_{sy} with a common node $v \in P_{sx}$ and $v \in P_{sy}$, we have $\max\{r(v, P_{sx}), r(v, P_{sy})\} = \min\{l(P_{sv}), \max\{l(P_{vx}), l(P_{vy})\}\}$. The preprocessing for sparse graphs needs, therefore, $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space. (In case such a heavy preprocessing is not acceptable, [49] also describes how to compute upper bounds for the reach.)

Using reach-based routing, a speed-up factor up to 20 has been reported in [49]. However, this effect decreases if reach-based routing is combined with bidirectional search.

Chapter 2

Shortest-Path Containers

This chapter introduces the abstract concept of shortest-path containers. The basic idea behind a shortest-path container is to store, for each edge, some information to decide for which targets of shortest-path queries the edge might be necessary. We seek for methods that can guarantee that *at least* the target nodes for shortest-path that start with this edge are inside the container. In this case, a modified version of Dijkstra's algorithm can provably find a shortest path, but usually visits much fewer nodes. A special variant of this method has been presented in [7] where the containers are angular sectors in the plane and the graph is a timetable graph for railways.

The next section defines shortest-path containers and shows how to use them. Then, in Sect. 2.2, we discuss how to generate shortest-path containers. Sect. 2.3 contains an experimental study about constant-sized geometric objects that can be used as shortest path containers. The following section presents first methods to maintain these geometric objects in case the graphs (and therefore its shortest paths) change. In the last section of this chapter, we discuss an approach without the condition that a container must have constant size.

2.1 Definition of Shortest-Path Containers

We will now formally define the shortest-path containers which help to reduce the search space of Dijkstra's algorithm. Containers are used to keep the nodes which are potentially useful for shortest path computations. This idea gives rise to DIJKSTRA'S ALGORITHM WITH PRUNING (Algorithm 2), which reduces the search space by examining at each iteration only a subset of the neighbors of a node (line 5a); the differences to DIJKSTRA'S ALGORITHM (Algorithm 1) are shown in boldface. The idea is illustrated in Fig. 2.1. The condition in line 5a is formalized by the notion of a consistent container.

Definition 3 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph. We call a set of nodes $C \subseteq V$ a* container*. A container C associated with an edge (u, v) is called* consistent*, if for all shortest paths from u to t that start with the edge (u, v), the target t is in C.*

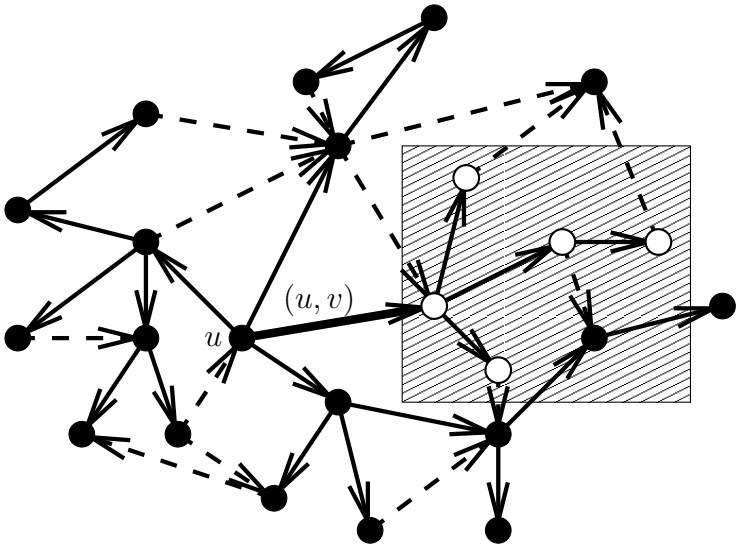


Figure 2.1: DIJKSTRA'S ALGORITHM is run for a node $u \in V$. Let the white nodes be those nodes that can be reached on a shortest path using the edge (u, v). A geometric object is constructed that contains these nodes. It may contain other nodes, but this does only affect the running time and not the correctness of DIJKSTRA'S ALGORITHM WITH PRUNING.

In other words, $C(u, v)$ is consistent, if $S(u, v) \subseteq C(u, v)$, where $S(u, v)$ represents the set of nodes x for which the shortest u-x path starts with the edge (u, v). Note that further nodes may be part of a consistent container. However, at least the nodes that can be reached by a shortest path starting with (u, v) must be in $C(u, v)$. We will refer to the additional nodes as *wrong nodes*, since they lead us the wrong way.

Theorem 4 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph and for each edge e let $C(e)$ be a consistent container. Then* DIJKSTRA'S ALGORITHM WITH PRUNING *finds a shortest path from s to t.*

Proof: Consider the shortest path P from s to t that is found by DIJKSTRA'S ALGORITHM. If for all edges $e \in P$ the target node t is in $C(e)$, the path P is found by DIJKSTRA'S ALGORITHM WITH PRUNING, because the pruning does not change the order in which the edges are processed. A sub-path of a shortest path is again a shortest path, so for all $(u, v) \in P$, the sub-path of P from u to t is a shortest u-t path. Then by definition of consistent container, $t \in C(u, v)$. □

This idea of pruning can be extended to bidirectional search (see Sect. 1.2.1). A second set of containers is determined by running the preprocessing a second time on the reversed graph G_{rev} (see Sect. 1.1). We will refer to the geometric objects of this graph with reversed edges as *reverse containers*. A forward step in the bidirectional search checks the normal containers whereas a backward step uses reverse containers.

```
 1   for all nodes u ∈ V set dist(u) := ∞
 2   initialize priority queue Q with source s and set dist(s) := 0
 3   while priority queue Q is not empty
3a       if u = t return
 4       get node u with smallest tentative distance dist(u) in Q
 5       for all neighbor nodes v of u
5a           if t ∈ C(u, v)
 7               set new-dist := dist(u) + l(u, v)
 8               if new-dist < dist(v)
 9                   if dist(v) = ∞
10                       insert neighbor node v in Q with priority new-dist
11                   else
12                       set priority of neighbor node v in Q to new-dist
13                   set dist(v) := new-dist
```

Algorithm 2: DIJKSTRA'S ALGORITHM WITH PRUNING. Neighbors are only visited, if the edge (u, v) is in the consistent container $C(u, v)$. (Differences to DIJKSTRA'S ALGORITHM (Algorithm 1) are printed in bold face.)

The quality of a set of containers was evaluated according to the following criterion.

Definition 5 *Let C denote a set of containers and for each edge $e \in E$ let $S(e) \subseteq V$ denote the set of nodes that can be reached by a shortest path starting with e. For both sets, we count the number of nodes inside all containers: $\sum_{e \in E} |\{t \in C(e)\}|$ and $\sum_{e \in E} |\{t \in S(e)\}|$. Both sums are bounded by $n \cdot m$. We therefore define the* quality *of C as:*

$$\frac{n \cdot m - \sum_{e \in E} |\{t \in C(e)\}|}{n \cdot m - \sum_{e \in E} |\{t \in S(e)\}|}$$

This fraction is biased by the number of correct nodes. It equals 1, if the number of wrong nodes inside containers is zero, while it becomes 0, if all containers in C contain the entire graph.

2.2 Creating Consistent Containers

We now describe in detail how to compute $C(s, x)$ for all edges $(s, x) \in E$. The complete algorithm is shown as Algorithm 3. (The differences to DIJKSTRA'S ALGORITHM (Algorithm 1) are printed in bold face.)

Recall that $S(s, x)$ is the set of all nodes t with the property that there is the (unique) shortest s-t path that starts with the edge (s, x). To determine $S(s, x)$ for every edge $(s, x) \in E$, Dijkstra's algorithm is run for each node $s \in V$. We keep a node array *SourceEdge* where the entry $SourceEdge[v]$, $v \in V$, stores the first edge (s, x) in a shortest s-v path in G. This can be constructed in a way similar to that of a shortest path tree: Every time the distance label of a node v is adjusted via (u, v), we set $SourceEdge[v]$ to

```
 0  for all s ∈ V do
 1      for all nodes u ∈ V set dist(u) := ∞
 2      initialize priority queue Q with source s and set dist(s) := 0
 3      while priority queue Q is not empty
 4          get node u with smallest tentative distance dist(u) in Q
4a          if u ≠ s enlarge C(SourceEdge[u]) to contain u
 5          for all neighbor nodes v of u
 7                  set new-dist := dist(u) + l(u, v)
 8                  if new-dist < dist(v)
 9                      if dist(v) = ∞
10                          insert neighbor node v in Q with priority new-dist
11                      else
12                          set priority of neighbor node v in Q to new-dist
13                      set dist(v) := new-dist
14                      if u = s
15                          set SourceEdge[v] := (s, v)
16                      else
17                          set SourceEdge[v] := SourceEdge[u]
```

Algorithm 3: CREATE-CONTAINERS. Running a modification of DIJKSTRA'S ALGORITHM (Algorithm 1) for all nodes $s \in V$ to create consistent containers. (Differences to DIJKSTRA'S ALGORITHM are printed in bold face.)

(u, v), if $u = s$, and to $SourceEdge[u]$, otherwise (lines 14–17). When a node u is removed from the priority queue, $SourceEdge[u]$ holds the outgoing edge of s with which a shortest path from s to u starts. Enlarging $C(SourceEdge[u])$ to contain u in line 4a therefore constructs consistent containers $C(s, x)$ for all neighbors x of s.

Since Dijkstra's algorithm runs in $\mathcal{O}(n \log n)$ time for sparse graphs, the overall running time is $\mathcal{O}(n^2 \log n)$ plus the time to construct the containers. The storage requirement is $\mathcal{O}(n)$ plus the space required to store the containers.

Some types of containers are not possible to be constructed *on-line* by enlarging them when a new node is inserted. In other words, no efficient method exists to update a container $C(s, x)$ with a new node u that has turned out to be in $S(s, x)$, and it is necessary to actually create the sets $S(s, x)$ in memory in line 4a. The sets $S(s, x)$ can then be used to construct the containers $C(s, x)$ *off-line*, after DIJKSTRA'S ALGORITHM has finished for s. Remark that the storage requirement is still $\mathcal{O}(n)$, because $\sum_{(s,x)\in E} |S(s, x)| \leq n$.

For very large graphs, it may be necessary to parallelize CREATE-CONTAINERS. In fact, the results for each node $v \in V$ in line 0 are indeed independent and can be distributed to different machines. Furthermore, it may be acceptable to compute the shortest-path containers only for a subset of the edges, thereby avoiding the all-pairs shortest-path computation. More precisely, line 0 loops only over a subset of the nodes in this case. Figure 2.2 shows the average search space of DIJKSTRA'S ALGORITHM WITH PRUNING with an in-

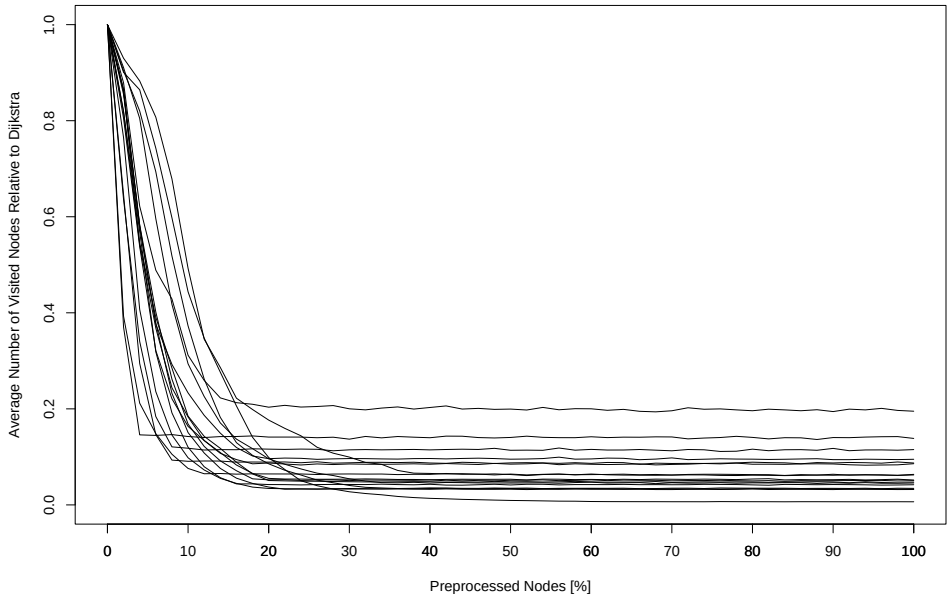


Figure 2.2: Average search space relative to DIJKSTRA'S ALGORITHM if the preprocessing is only performed for a subset of the nodes.

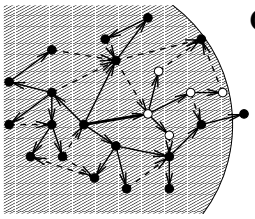
creasing subset of nodes. For a given size of the subset, the nodes of highest degree are selected. Among nodes of same degree, the selected nodes are chosen at random. Figure 2.2 depicts for 17 street networks the average search space relative the search space of DIJKSTRA'S ALGORITHM. Remark that for most of the graphs, the reduction for 20% preprocessed nodes is already similar to a full preprocessing (100%).

2.3 Geometric Containers of Constant Size

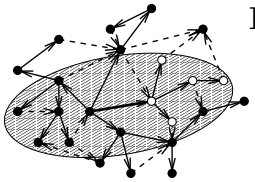
The shortest-path containers that we are using are geometric objects. Recall from Section 1.1 that a *layout* $L : V \to \mathbb{R}^2$ of the graph in the Euclidean plane is given. In Section 2.1, we explained how consistent containers are used to prune the search space of DIJKSTRA'S ALGORITHM. In particular, the correctness of the result does not depend on the layout of the graph that is used to construct the containers (in contrast to the impact of the container for speeding up Dijkstra's Algorithm). To use the containers for speeding up Dijkstra's algorithm and thus be able to answer on-line queries fast, we require that a geometric container has a description of constant size and that its containment test takes constant time. We will first provide a list of the geometric objects and describe how to construct them. Then, in the subsequent subsection, we describe in detail the computational statistics that we performed. The corresponding results and conclusions are presented in the last subsection. Preliminary portions of this work appeared in [50].

2.3.1 Geometric Objects

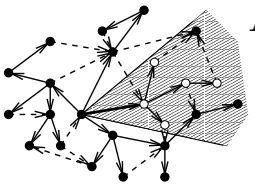
This section describes the geometric containers that we used in our tests. Except for the convex hull, all of them have constant size and therefore the containment test takes constant time. In particular, we have considered the following types of containers for an edge (u, v):



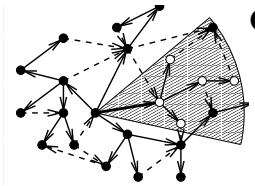
Circle Centered at Tail (Circle) For each edge (u, v), the circle with center at u and minimum radius that contains $S(u, v)$ is computed. This is the same as finding the maximal distance of all nodes in $S(u, v)$ from u. The size of such an object is constant, because the only value that needs to be stored is the radius[1] which leads to a space consumption that is linear in the number of edges. The radius can be determined on-line by increasing it if necessary.



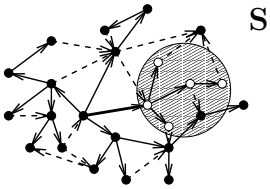
Ellipse. An extension of the circle is the ellipse with foci u and v and minimum radius needed to contain $S(u, v)$. It suffices to remember the radius, which can be found on-line similarly as in the circle case.



Angular Sector (AngularSect). Angular sectors are the objects that were used in [7]. For each edge (u, v) a node p left of (u, v) and a node q right of (u, v) are determined such that all nodes in $S(u, v)$ lie within the angular sector $\angle(p, u, q)$. The nodes p and q are chosen in a way that minimizes the angle $\angle(p, u, q)$. They can be determined in an on-line fashion: If a new node w is outside the angular sector $\angle(p, u, q)$, we set $p := w$ if w is to the left of (u, v) and $q := w$ if w is to the right of it. (Note that this is not necessarily the minimum angle at u that contains all points in $S(u, v)$.)

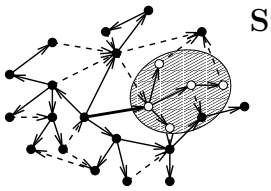


Circular Sector (CircularSect). By intersecting an angular sector with a circle at the tail of the edge, we get a circular sector. Obviously the minimal circular sector can be found on-line and needs only constant space (two points and the radius).

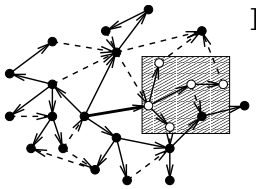


Smallest Enclosing Circle (MinCircle). The smallest enclosing circle is the unique circle with smallest area that includes all points. We use the implementation in CGAL [51] of Welzl's algorithm [52] with expected linear running time. The algorithm works off-line and storage requirement is at most three points.

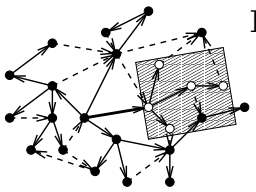
[1] In practice, the squared radius is stored to avoid the computationally expensive square root function.



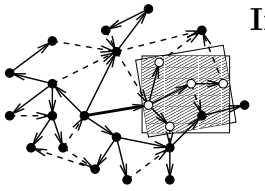
Smallest Enclosing Ellipse (MinEllipse). The smallest enclosing ellipse is a generalization of the smallest enclosing circle. Therefore, the search space using this container will be at most as large as for smallest enclosing circles (although the actual running time might be larger since the inclusion test is more expensive). Again, Welzl's algorithm is used. The space requirement is constant.



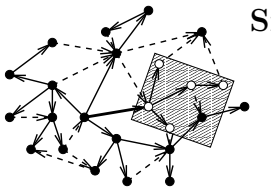
Bounding Box (BBox). This is the simplest object in our collection. It suffices to store four numbers for each object, which are the lower, upper, left and right boundary of the box. The bounding boxes can easily be computed on-line while the shortest paths are computed in the preprocessing.



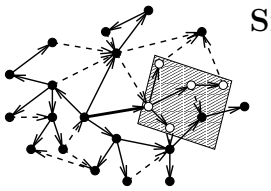
Edge-Parallel Rectangle (Rect || Edge). Such a rectangle is not parallel to an axis, but to the edge to which it belongs. Thus, for each edge, the coordinate system is rotated and then the bounding box is determined in this rotated coordinate system. Our motivation to implement this container was the insight that the target nodes for an edge are usually situated in the direction of the edge. A rectangle that targets in this direction might therefore be a better model for the geometric region than one that is parallel to the axes. Note that storage requirements are actually the same as for a bounding box, but additional computations are required to rotate the coordinate system.



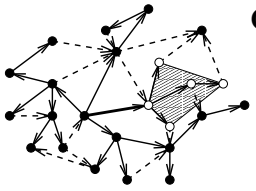
Intersection of Rectangles (Intersection) The rectangle parallel to the axes and the rectangle parallel to the edge are intersected, which should lead to a smaller object. The space consumption of the two objects sums up, but is still constant, and as both objects can be computed on-line the intersection can be as well.



Smallest Enclosing Rectangle (MinRect). We allow the rectangle to be oriented in any direction and search for one with smallest area containing all points. The algorithm from [53] finds such a rectangle in linear time. However, due to numerical inconsistencies we had to incorporate additional tests to ensure that all points are in fact inside the rectangle. As for the minimal enclosing circle, this container has to be calculated off-line, but needs only constant space for its orientation and dimensions.



Smallest Enclosing Parallelogram (MinPara). A parallelogram is a generalization of a rectangle, and surprisingly Toussaint's idea to use rotating calipers can been extended to find the smallest enclosing parallelogram [54]. Space consumption is constant and the algorithm is off-line, too.



Convex Hull. As the convex hull is the smallest enclosing convex polygon of the points, it does not fulfill our requirement that containers must be of constant size. It is included here, because it provides a lower bound for all convex objects. If there is a best convex container, it cannot exclude more points than the convex hull.

For some types of containers, it is obvious that they are at least as good as others. In particular, if a container is a subset of another container, using the first container in Dijkstra's algorithm with pruning excludes at least as many nodes as the second container. If we assume that the distribution of the nodes is uniformly at random, the expected number of nodes inside a geometric container is proportional to its area. The larger the container, the larger the average number of wrong nodes should be.

2.3.2 Experimental Setup

We implemented the algorithm in `C++` using `GCC 2.95.3`. We used the graph data structure from LEDA 4.3 (see [44]) as well as the priority queue and the convex hull algorithm provided. I/O was done by the LEDA extension package for GraphML with Xerces 2.1. For the minimal circles, ellipses and parallelograms, we used CGAL 2.4 [51]. In order to perform efficient containment tests for minimal circles, we converted the result from arbitrary precision to built-in doubles. To overcome numerical inaccuracies, the radius was increased if necessary to guarantee that all points are in fact inside the container. For minimal ellipses, we used arbitrary precision which affects the running time but not the search space. Instead of calculating the minimal circle (or ellipse) of a point set, we determine the minimal circle (or ellipse) of the convex hull. This speeds up the preprocessing for these containers considerably, because the convex hull could be computed with built-in doubles. Although CGAL also provides an algorithm for minimal rectangles, we decided to implement one ourselves, because one cannot simply increase a radius in this case. Due to numeric instabilities, our implementation does not guarantee to find the minimal container, but asserts that all points are inside the container. The convex hulls were computed with LEDA [44]. The experiments were performed on an Intel Xeon with 2.4 GHz on the Linux 2.4 platform.

It is crucial to this problem to do the statistics with data that stem from real applications. We are using two types of data:

Street Graphs. We have gathered street maps from various public Internet servers. They cover some American cities and their surroundings. Unfortunately the maps did not contain more information than the mere location of the streets. In particular, streets are not distinguished from freeways, and one-way streets are not marked as such, which makes these graphs bidirected with approximately Euclidean edge length. The street networks are typically very sparse with an average degree hardly above 2. The size of these networks varies from 1444 to 20466 nodes.

Railway Graphs. The railway networks of different European countries were derived from the winter 1996/1997 time table. The nodes of such a graph are the stations and an edge between two stations exists iff there is an non-stop connection. The edges are weighted by the average travel time. In particular, here the weights do *not* directly correspond to the layout. They have between 409 nodes (Netherlands) and 6884 nodes (Germany) but are not as sparse as the street graphs.

All test sets were converted to the XML-based GraphML file format [55] to allow a unified processing.

We sampled random single-source single-target queries to determine the average number of nodes that are visited by the algorithm. For an exact description, we need to define the mean and variance of n samples.

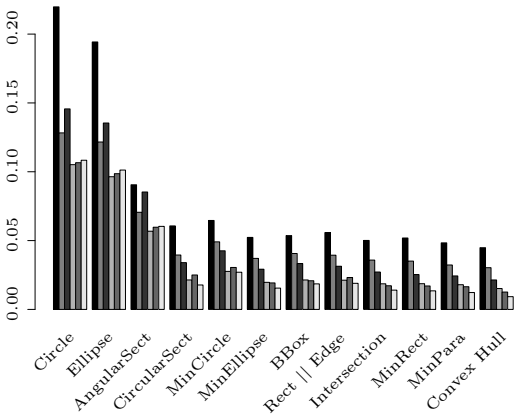
Definition 6 (Mean and Variance) *Given n samples $x = (x_1, \ldots, x_n)^t \in \mathbb{R}^n$, the* mean value *of x is defined as*

$$\overline{x} := \frac{1}{n} \sum_{k=1}^{n} x_k.$$

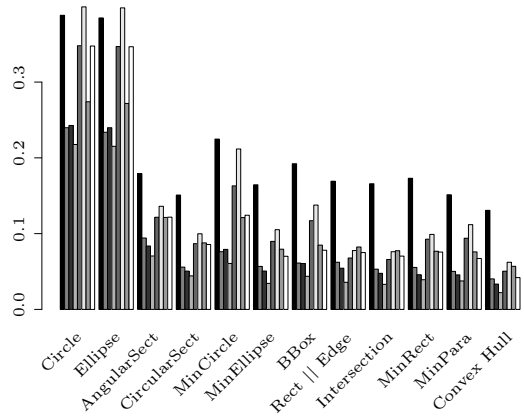
The (empirical) variance *of x is defined as*

$$s^2 := \frac{1}{n-1} \sum_{k=1}^{n} (x_k - \overline{x})^2.$$

Sampling is based on the *length of the* $(1 - \frac{1}{\alpha})$*-confidence interval* on this average which is $2t_{n-1,1-\frac{\alpha}{2}} s n^{-\frac{1}{2}}$ (see, e.g., [56]), where $t_{n-1,1-\frac{\alpha}{2}}$ denotes the $1 - \frac{\alpha}{2}$-quantile of Student's t-distribution with $n-1$ degrees of freedom (where n is the number of samples, s the standard error, and α our chosen error probability). The sampling was done until the length of the 95%-confidence interval was smaller than 5% of the average search space. For $n > 100$ we approximated the t-distribution by the normal distribution. Note that the sample mean $\overline{x}$ and the standard error s can be calculated recursively with $\overline{x}_{(n)} = \frac{1}{n}\left(\overline{x}_{(n-1)}(n-1) + x_n\right)$ and $s^2_{(n)} = \frac{1}{n-1}\left[(n-2)s^2_{(n-1)} + (n-1)\overline{x}^2_{(n-1)} + x_n^2 - n\overline{x}^2_{(n)}\right]$, where the subscripts in brackets mark the sample size. Using these formulas, it is possible to run



(a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884



(b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk 51510

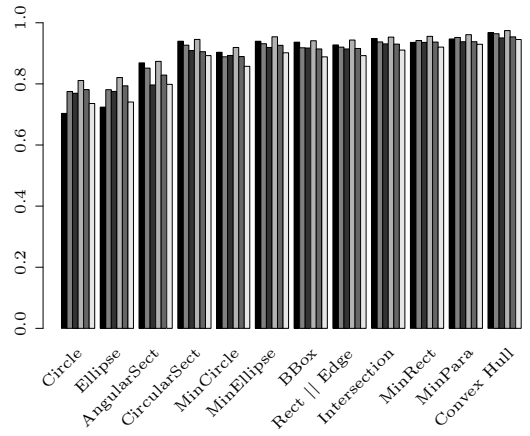
Figure 2.3: Average number of visited nodes relative to Dijkstra's algorithm for all graphs and geometric objects. The graphs are ordered according to the number of nodes.

random single-source single-target shortest-path queries until the length of the confidence interval is small enough.

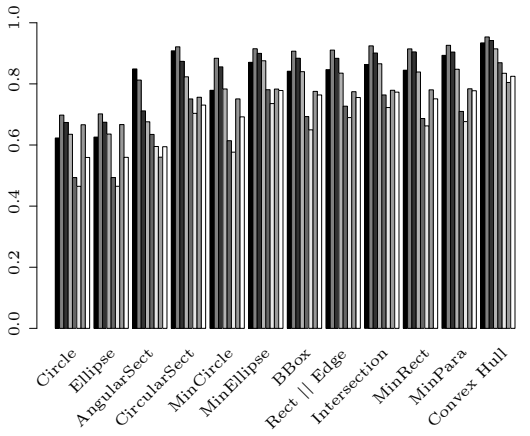
We are interested in two results: What is the average size of the search space and what is the actual CPU time spent per search on average? The first number is independent from implementation, compiler, CPU, and operating system, whereas the latter includes the overhead that is introduced by a speed-up technique. As measurement for the search space, we used the number of nodes that are put into the priority queue during the search.

2.3.3 Computational Results

Figure 2.3 depicts the results for railway and street networks. The average number of nodes that the algorithm visited are shown. To enable the comparison of the result for different graphs, the numbers are relative to the average search space of Dijkstra's algorithm (without pruning). As expected, the pruning for circles around the tail is by far not as good as the other methods. Note, however, that the average search space is still reduced to about 10% (25% for street networks). The only type of objects studied previously [7], the angular sectors, result in a reduction to about 6% (10%), but if both are intersected, we get only 3.5% (7%). Surprisingly, the result for the simplest container (bounding boxes) is about the same as for the better tailored containers (circular sectors, edge-parallel and smallest enclosing rectangles, or parallelograms). Of course the results of more general containers (e.g., smallest enclosing rectangle vs. bounding box) are better, but the differences are not

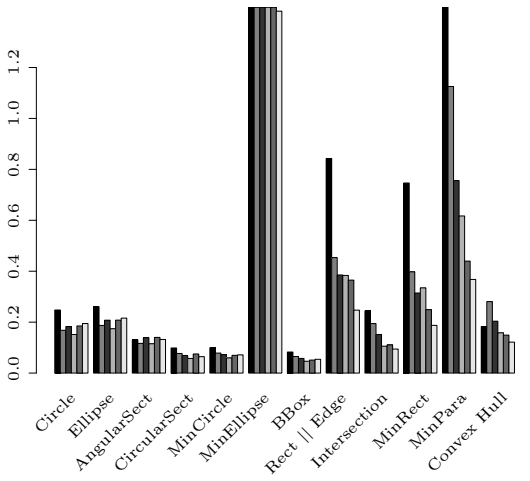


(a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884

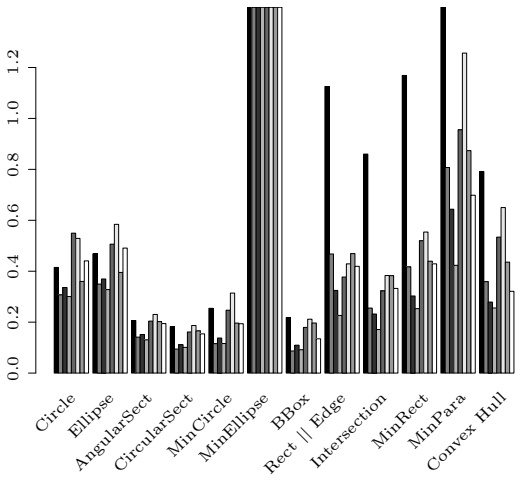


(b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk 51510

Figure 2.4: Quality according to Definition 5 for all graphs and geometric objects. The graphs are ordered according to the number of nodes.



(a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884



(b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk 51510

Figure 2.5: Average query running time relative to DIJKSTRA'S ALGORITHM for all data sets and geometric objects. The values for minimal ellipse and minimal parallelogram are clipped. They use arbitrary precision and therefore their containment tests are much slower.

very big. Furthermore, the difference to our lower bound for convex objects (convex hull) is comparatively small. The data sets are ordered according to their size. In most cases the speed-up is better the larger the graph. This can be explained by the observation that the search space of a lot of queries is already limited by the size of the graph.

In Figure 2.4, the quality according to Definition 5 is shown. Comparing Figures 2.3 and 2.4 confirms that the quality reflects the search space. Containers that result in few visited nodes have a higher quality. Furthermore the quality is not so dependent on the size of the graph, because the quality measure is normalized.

Finally, we examined the average running time. We depict them in Fig. 2.5, again relative to the running time of the unmodified Dijkstra. It is obvious that the slightly smaller search space for the more complicated containers does not pay off. In fact the simplest container, the axis-parallel bounding box, results in the fastest algorithm.

2.4 Updating Containers

If a weight of an edge is changed, some containers must be updated to stay consistent. Generally speaking, for every new shortest path $(u_0, u_1, \ldots, u_{k-1}, u_k)$ in the graph, $C(u_0, u_1)$ has to be updated to include u_k. If we maintain containers C^{rev} for reversed edges (e.g., to perform a bidirectional search), u_0 must be added to $C^{\text{rev}}(u_{k-1}, u_k)$. (We will refer to the containers for this graph with reversed edges as *reverse containers* and mark them with the superscript "rev".)

As running Create-Containers after each update is not desirable, we look for faster methods to maintain consistent containers (but possibly with worse quality). If containers are too large, then their quality is decreased but their consistency is preserved. The first helpful observation is the fact that only a part of the containers may be too small.

In this section, we will present necessary conditions for new shortest paths when edge weights increase or decrease. (Preliminary portions of this work appeared in [57].) They enable us to maintain consistent containers without running completely from scratch Create-Containers (Algorithm 3). Throughout the section, we will mark variables before the update with the subscript "old" and updated values with the subscript "new".

2.4.1 Increasing an edge weight

Let us first consider the case of increasing the weight of an edge $(x, y) \in E$. The following lemma is suited to restrict the set of containers that we have to update.

Lemma 7 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph. Assume that the increase of the weight of an edge $(x, y) \in E$ creates a new shortest s-t path P_{new} in G. Then, before the weight change, (x, y) is the last edge of a shortest s-y path and the first edge of a shortest x-t path.*

Proof: Let P_{old} denote the old shortest path from s to t as illustrated in Fig. 2.6. Since the weight $l(x, y)$ is increased, $(x, y) \in P_{\text{old}}$. Let P_{sy} denote the first part of this path P_{old}

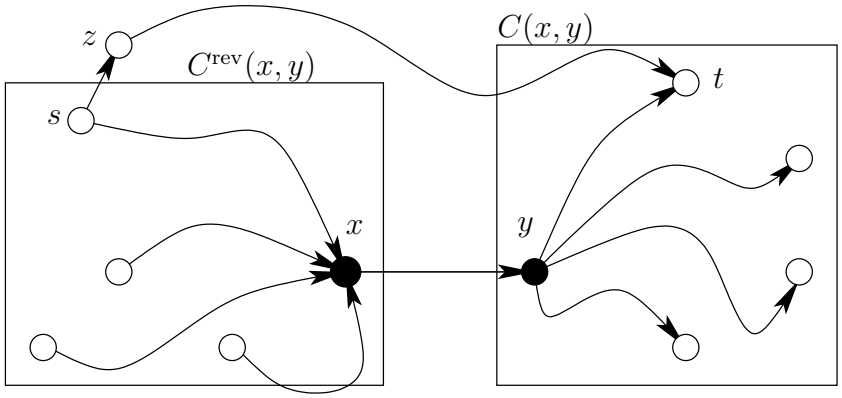


Figure 2.6: When the weight of the edge (x, y) is increased, the source s of a new shortest path from s to t must be inside $S_{\text{old}}(x, y)$.

from s to y. Since a sub-path of a shortest path is again a shortest path, P_{sy} was the shortest path from s to y. For symmetric reasons, the first edge of a shortest x-t path is (x, y). □

By the definition of (reverse) containers, the first and last nodes of new shortest paths can be described as follows.

Corollary 8 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph. Assume that the increase of the weight of an edge $(x, y) \in E$ creates a new shortest s-t path P_{new} in G. Then*

$$s \in C^{\text{rev}}_{\text{old}}(x, y) \qquad \textit{and} \qquad t \in C_{\text{old}}(x, y).$$

It is therefore sufficient to run a (modified) Dijkstra for all nodes in $C^{\text{rev}}_{\text{old}}(x, y)$ to update (normal) containers and in $C_{\text{old}}(x, y)$ to update reverse containers.

Either Lemma 7 or Corollary 8 can be used to restrict the set of nodes for which Dijkstra's algorithm must be performed. The following lemma is suited to restrict Dijkstra's algorithm itself.

Lemma 9 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph and let P_{new} be a path from a node s to a node t that has become a shortest path because of an increase of the weight of an edge (x, y). Then, for all nodes $u \in P_{\text{new}}$:*

$$d_{\text{new}}(s, u) < d_{\text{new}}(s, x) + l_{\text{new}}(x, y) + d_{\text{new}}(y, u)$$

Proof: The new shortest path P_{new} does not contain the edge (x, y), and the sub-path of P_{new} from s to u is also a shortest path that does not contain the edge (x, y). The right hand side of the inequality is the length of some path from s to u containing (x, y). Since shortest paths are assumed to be unique, the lemma follows immediately. □

2.4.2 Decreasing an edge weight

Similar to the case of a weight increase, we can prove a lemma about start and end nodes of new shortest paths for the case of a weight decrease.

Lemma 10 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph. Assume that the decrease of the weight of an edge $(x, y) \in E$ creates a new shortest s-t path P_{new} in G. Then, after the weight change, (x, y) is the last edge of a shortest s-y path and the first edge of a shortest x-t path.*

Proof: Obviously, the edge (x, y) must be part of this path P_{new}. Let P_{sy} denote the sub-path of P_{new} from s to v. As a sub-path of a shortest path is also a shortest path, P_{sy} is a shortest that ends with the edge (x, y). For symmetric reasons, the first edge of a shortest x-t path is (x, y). □

Again the condition provides a simple test using the containers $C(x, y)$ and $C^{\text{rev}}(x, y)$. This time however, both containers must already be updated.

Corollary 11 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph. Assume that the decrease of the weight of an edge $(x, y) \in E$ creates a new shortest s-t path P_{new} in G. Then*

$$s \in C^{\text{rev}}_{\text{new}}(x, y) \qquad \textit{and} \qquad t \in C_{\text{new}}(x, y).$$

In order to run a (modified) Dijkstra for all nodes in $C_{\text{new}}(x, y)$, it is necessary to compute $C_{\text{new}}(x, y)$, i.e., to enlarge it if necessary. This can be done similarly to its creation in Create-Containers (Algorithm 3). In contrast to Create-Containers, the loop in line 0 is replaced by a single run for $s := x$. Furthermore in line 4a, only the container $C(x, y)$ must be enlarged (i.e., $A[u] = (x, y)$). Finally, Dijkstra's algorithm can be truncated to the part of the graph, where distance labels change. This can be achieved by executing lines 8–17 only if `new-dist` $< l_{\text{old}}(x, y) + d_{\text{old}}(y, u)$. If a node v is excluded because `new-dist` $\geq l_{\text{old}}(x, y) + d_{\text{old}}(y, u)$, we distinguish between two cases. If `new-dist` $= l_{\text{new}}(x, y) + d_{\text{old}}(y, v)$, the distance of v has not changed. Furthermore, the distance has not changed for all nodes a where the shortest x-a path contains v. Ignoring nodes $v \in V$ with `new-dist` $= l_{\text{new}}(x, y) + d_{\text{old}}(y, v)$ therefore does not change the result of the algorithm. If `new-dist` $> l_{\text{new}}(x, y) + d_{\text{old}}(y, v)$, there exists a shorter path from x to v that does not contain (u, v). The node v can therefore be ignored in this case, too.

Similarly to Lemma 9 the next lemma reduces the search space of Create-Containers for the updating of the rest of the containers, but this time the old weight of the edge (x, y) is used in the comparison.

Lemma 12 *Let $G = (V, E)$, $l : E \to \mathbb{R}$ be a weighted graph and let P_{new} be a path from a node s to a node t that has become a shortest path because of a decrease of the weight of an edge (x, y). Then, for all nodes $u \in P_{\text{new}}$:*

$$d_{\text{new}}(s, u) < d_{\text{new}}(s, x) + l_{\text{old}}(x, y) + d_{\text{new}}(y, u)$$

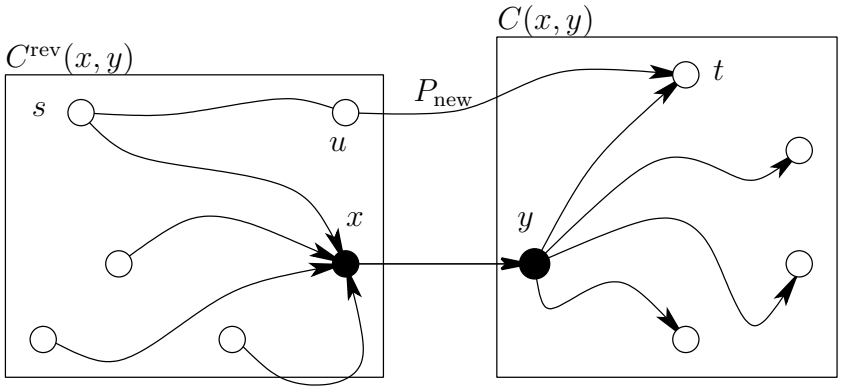


Figure 2.7: When the weight of the edge (x, y) is decreased, for all nodes u on a new shortest path from s to t, $d_{\text{new}}(s, u) < d_{\text{new}}(s, x) + l_{\text{old}}(x, y) + d_{\text{new}}(y, u)$.

Proof: Since $l_{\text{new}}(x, y) < l_{\text{old}}(x, y)$, the new distance $d_{\text{new}}(s, t)$ must be shorter than the old distance $d_{\text{old}}(s, t)$. The new shortest path P_{new} does contain the edge (x, y) in contrast to the old shortest path from s to t. Therefore $d_{\text{new}}(s, t) = d_{\text{new}}(s, x)+l_{\text{new}}(x, y)+d_{\text{new}}(y, t) < d_{\text{new}}(s, x)+l_{\text{old}}(x, y)+d_{\text{new}}(y, t)$. Consider now some node $u \in P_{\text{new}}$ (see Figure 2.7). Let $P_{s,u}$ denote the sub-path of P_{new} from s to u. If $P_{s,u}$ does not contain (x, y), i.e. if the edge (x, y) appears in P_{new} after u, then $d_{\text{new}}(s, u) < d_{\text{new}}(s, x) < d_{\text{new}}(s, x)+l_{\text{old}}(x, y)+d_{\text{new}}(y, t)$, since $l_{\text{old}}(x, y) > 0$. If $P_{s,u}$ contains (x, y), then $d_{\text{new}}(s, u) = d_{\text{new}}(s, x) + l_{\text{new}}(x, y) + d_{\text{new}}(y, u)$. Otherwise a shorter path from s to u would exist which contradicts the fact that P_{new} is a shortest path. Since $l_{\text{old}}(x, y) > l_{\text{new}}(x, y)$ the lemma follows. □

2.4.3 Update strategies

According to Lemma 7 and 10, only those containers have to be updated that belong to an outgoing edge of a node $s \in V$, where the last edge on a shortest s-y path is (x, y). We will call such nodes s *potentially affected* and denote their number by p. The potentially affected nodes can be determined by a run of a modified Dijkstra starting at y with reversed edges.

If we maintain reverse containers, Corollary 8 and 11 provide an even simpler method to find containers that may need maintenance. When a weight of an edge (x, y) has changed, only those containers must be updated that belong to an outgoing edge of a node in $C^{\text{rev}}(x, y)$. Symmetrically, the reverse containers that belong to an incoming edge of a node in $C(x, y)$ should be checked. If the weight has been decreased, the containers $C(x, y)$ and $C^{\text{rev}}(x, y)$ must be updated as described in the previous section *before* we determine the nodes inside them.

Both methods, with and without reverse containers, find those nodes for which the containers of incident edges must be updated. For both of them, we studied three different methods to update the container of an edge:

Compute the container from scratch. The result is slightly different from recomputing all containers from scratch because not all containers are recomputed. A container that can shrink is not necessarily updated. As DIJKSTRA'S ALGORITHM is run for every potentially affected node, the overall running time is bounded by $\mathcal{O}(p \cdot n \log n)$.

Set the container to infinity (without any further computation). If the entire graph is inside the container, it is certainly consistent. However, the quality of the containers drops dramatically with this method although the running time—being linear in p—is appealing.

Enlarge the container as much as necessary. A variant of Dijkstra's algorithm truncated according to Lemma 9 and 12 can be used to enlarge on-line containers. (This method is not applicable for off-line containers.) More precisely, the lines 8–17 of CREATE-CONTAINERS are only executed for a node v that satisfies `new-dist` $< d(s,x) + l_{\text{new}}(x,y) + d(y,v)$ or `new-dist` $< d(s,x) + l_{\text{old}}(x,y) + d(y,v)$, respectively. In particular, the rest of the nodes are never inserted in the queue Q. The complexity of this update strategy is therefore $\mathcal{O}(p \cdot k \log k)$, if for all potentially affected nodes s an upper bound k exists for the size of the set of nodes fulfilling the condition in Lemma 9 and 12 for edge weight increases and decreases, respectively.

The conditions in Lemma 9 and 12 use distance values $d(u,x)$ and $d(y,u)$ for different $u \in E$, which have to be computed beforehand by running two instances of DIJKSTRA'S ALGORITHM.

The reverse containers are the counterpart to normal containers, if all edges in the graph are reversed. Therefore, the same algorithm can be applied to a graph with reversed edges to enlarge the reverse containers as necessary.

2.4.4 Experimental Setup

We performed an experimental study to evaluate the performance and quality of our algorithms. More precisely, we examined how much time is needed to update the containers (on average) and how much the containers differ from containers computed from scratch. (Remember that we do not shrink all containers in our updates.)

As the experiments in the static case showed that bounding boxes are, in practice, the fastest container type with constant size, the experiments for the dynamic version are performed with this geometric container. The construction of all containers from scratch is very time consuming. A *single* preprocessing that takes hours or even days should be feasible. However, for this experimental study, we need to construct all containers from scratch hundreds of times in order to compare them to updated containers. Therefore, we evaluate the different update strategies only for the (smaller) railway networks. After all, we assume that the results scale to the larger graphs that todays and future computers can handle.

For each graph, we increase the weight of 100 random edges to a large value (i.e., the sum of all weights in the graph). This is similar to removing the edge from the graph.

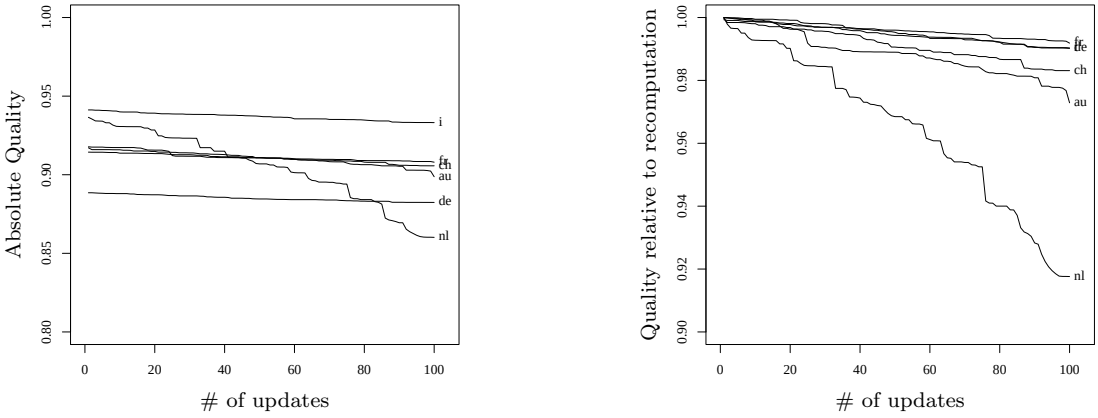


Figure 2.8: The quality of updated containers for 100 increased edge weights. The containers are enlarged by a truncated Dijkstra for all nodes s with (x, y) as the last edge of a shortest s-y path. In the right diagram the quality is divided by the quality of containers computed from scratch.

After every weight change, the containers are updated according to section 2.4.3. A second set of containers is determined from scratch to compute the quality and compare the computation time. For the evaluation of decreasing edge weights, we start with the graph where 100 random edges have been set to a large weight. The weights are then decreased to their original values. Again, the updated containers are compared to newly computed containers.

All six variants have been implemented in `C++` based on the graph structure provided by LEDA 4.4. The programs were compiled with `GCC 3.2` and run on a single Intel Xeon with 2.4 GHz performing Linux 2.4.

2.4.5 Computational Results

In order to reflect the fact that also the quality of containers that are computed from scratch varies from graph to graph and after each update, we examined the quality of updated containers relative to the quality of containers that are computed from scratch. It turns out that in both cases—with and without reverse containers—the outcome is very similar. Furthermore, the case of an edge weight increase resembles the case of an edge weight decrease.

If containers are only enlarged, their quality decreases most of the time as expected (Figure 2.8). It is interesting to note that the larger the graph, the larger its quality remains. Single "bad" containers are clearly less important if the graph contains more edges. For large graphs, the quality stays close to 1 even after 100 updates.

If the containers are simply set to infinity, the situation is dramatically different though.

	reverse containers		used			not used		
	container update		set to infinity	enlarge	from scratch	set to infinity	enlarge	from scratch
	n	m						
nl	409	1215	1331; 173	2.17;2.00	2.45;2.70	269; 228	2.52;3.43	5.02;3.02
au	1660	4327	8093; 700	1.99;1.90	2.54;2.38	1551;1429	2.07;2.10	2.13;2.10
ch	2279	6015	10211; 953	2.18;2.24	2.56;2.58	2235;2300	2.77;2.85	2.48;3.09
i	2399	8008	9552; 965	2.85;2.75	2.77;2.68	2095;2097	2.80;2.65	2.64;2.94
fr	4598	14937	17691;1932	2.09;2.21	2.87;2.66	4382;4291	2.55;3.13	4.25;3.36
de	6884	18601	33160;2974	1.72;1.71	2.04;2.21	6568;6583	2.53;2.31	2.56;2.76

Table 2.2: Average speed-up for updating the containers with increasing weights (first number) and decreasing weights (second number).

After a few updates, the containers settle in a state with a quality below 0.2 where almost all nodes are inside all containers. Such a state is clearly not desirable, because no nodes are pruned by Dijkstra's algorithm for queries.

In the case where containers of incident edges are recomputed from scratch the resulting containers coincide most of the time with the newly computed containers. In other words, the quality equals 1 after almost every update. In practice, such updates can therefore be considered as good as using freshly determined containers.

The analysis of the time measurements are shown in Table 2.2. The first two columns list the number of nodes and the number of edges in the respective graphs. The other six columns refer to the six cases that have been examined in our study. They report the speed-up factor as the ratio between the time required for computing containers from scratch and the time for updating the containers. The three types of updates (enlarge the containers to infinity, enlarge the containers according to Lemma 9 and 12, and recompute the containers from scratch) were tested with maintenance of reverse containers and without it (using a backward Dijkstra instead). Although the time improvements are huge, if the containers are enlarged to infinity, these values are more or less meaningless, because of the unacceptable quality. We report them only for the sake of completeness.

An interesting observation is the fact that the speed-up factor does not seem to be correlated with the size of the graph. Furthermore, the similarity of the algorithms for increasing and decreasing edge weights probably explains the similar behavior in terms of timings. The speed-up values with and without reverse containers are quite similar, but note that the absolute time values with reverse containers are about twice as large. Maintaining reverse containers can therefore be justified only if they are used for other purposes as well (e.g., for bidirectional search). The most interesting observation however is the fact that using a pruned Dijkstra (column "enlarge") is often slower than Dijkstra without pruning (column "from scratch"). Obviously the additional check and computing the distances to x and from y for all nodes outweigh the gain of the pruning.

2.5 Geometric Containers of Non-Constant Size

If you drop the requirement that a shortest-path container must be of constant size, a substantially higher speed-up is possible [20, 21]. Although its asymptotic space consumption is higher, in practice, a much better speed-up of 100 and more can be achieved with the same amount of space. We first present these so-called bit-vectors in Section 2.5.1 by incorporating them in the framework of shortest-path containers.

Furthermore, [20, 21] show that the preprocessing can be realized without computing all-pairs shortest paths. We discuss in Sect. 2.5.2, how the preprocessing for bit-vectors can avoid the computation of all-pairs shortest paths, and also show in detail the method to partially re-use the result of the previous run of DIJKSTRA'S ALGORITHM in the preprocessing.

The bit-vector speed-up technique is based on a partition of the set of nodes. If we consider a street network and hence a graph with a layout, intelligent partitions are those in which nodes inside the same region are geometrically adjacent. We have examined several partitioning algorithms which are well applicable and give good speed-ups. Tested on real-world street networks, we compared the different strategies concerning the resulting average search space. In Section 2.5.3, our selection of geometric partitioning algorithms is presented that we used for our analysis.

Furthermore, we develop in Sect. 2.5.4 a two-level variant of the bit-vectors. They provide some kind of (lossy) compression of bit-vectors. For the same amount of space, they result in a smaller search space and a higher speed-up. Note that the compression guarantees the correctness of a shortest-path query but may be slower than the uncompressed bit-vector.

In Sect. 2.5.5, our experiments are described whose results are discussed and presented in the subsequent subsection. Preliminary portions of this work appeared in [58].

2.5.1 Bit-Vectors as Shortest-Path Containers

In Section 2.1, convex geometric objects approximated for each edge e, the set of all target nodes of shortest paths that start with the edge e. We will now use a partition of the set of nodes V into p regions for a different approximation. Formally, we will use a function $r : V \to \{1, \ldots, p\}$ that assigns to each node the number of its region. (Given a 2D layout of the graph, a simple method to partition a graph is to use a regular grid as illustrated in Figure 2.9 and assign all nodes inside a grid cell the same number.) In contrast to Section 2.3, where we assigned a convex geometric object to each edge, we will now use a bit-vector $b_e : \{1, \ldots, p\} \to \{\texttt{true}, \texttt{false}\}$ with p bits, each of which corresponds to a region. For each edge e, we therefore set the bit $b_e(i)$ to `true` iff e is the beginning of a shortest path to a node in region $i \in \{1, \ldots, p\}$.

It is easy to see that by definition such bit-vectors provide consistent shortest-path containers, i.e. for all shortest paths from u to t that start with the edge (u, v), the target t is in a region that is marked by the bit-vector of (u, v). All possible partitions of the nodes lead to a correct solution but most of them would not lead to the desired speed-up of

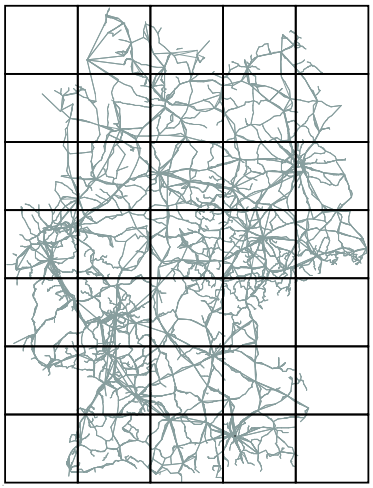


Figure 2.9: A 5×7 grid partitioning of Germany

the computation. During the shortest-path search, while scanning a node u, DIJKSTRA'S ALGORITHM WITH PRUNING (Algorithm 2) considers all outgoing edges and can now ignore those edges which have not set the bit for the region of the target node.

The space requirement for the preprocessed data is $\mathcal{O}(p \cdot m)$ for p regions because we have to store one bit for each region and edge. If $p = n$ and we assign to every node its own region number, we store in fact all-pairs shortest paths: if a node is assigned to its own, specific region, the modified shortest-path algorithm will find the direct path without regarding unnecessary edges or nodes. Note however, that in practice even for $p \ll n$ we achieved an average search space that is only 4 times the number of nodes in the shortest path. Furthermore, it is possible within the framework of the bit-vector speed-up technique to use a specific region only for the most important nodes. Storing the shortest paths to important nodes can therefore be realized without any additional implementation effort. It is common practice to cache the shortest paths to the most important nodes in the graph.

As other shortest-path containers, bit-vectors can be combined with bidirectional search (see Section 1.2.1). In principle, they can be used independently for the forward search, the backward search, or both of them. In our experiments the best results (with a fixed total number of bits per edge) achieved a forward and backward accelerated bidirectional search, which means that we applied the partition-based speed-up technique on both search directions (with half of the bits for each direction).

2.5.2 Preprocessing without All-Pairs Shortest Paths

In the preprocessing, the bit-vectors of all edges can also be filled correctly with Algorithm 3. For each node $s \in V$, we determine with a slightly modified DIJKSTRA'S ALGORITHM for all nodes $t \in V$ the first edge e of the shortest s-t path and set the set the bit of partition $r(t)$ of the bit-vector of e. Setting the bit-vectors needs $\mathcal{O}(mn)$ time for all pairs

of edges and nodes. The running time for the preprocessing is therefore dominated by the time required to compute n times DIJKSTRA'S ALGORITHM with total time complexity $\mathcal{O}(n \cdot (m + n \log n + n))$. For sparse graphs $m \in \mathcal{O}(n)$ like typical traffic networks, we get a worst-case time complexity of $\mathcal{O}(n^2 \log n)$.

Fortunately, it is not necessary to compute all-pairs shortest paths to fill the bit-vectors correctly as presented in [21]. A better approach uses the following insight: Every shortest path from any node s to a region R with the region number p_R has to enter the region R: if s is not a member of region R an edge $e = (u, v)$ with $r(u) \neq p_R$ and $r(v) = p_R$ exists. We will see that it is sufficient if the preprocessing algorithm only regards shortest paths to such nodes v. We will call such nodes *boundary nodes*.

Theorem 13 (Boundary Nodes) *Given a graph $G = (V, E)$ and a partition of V in p regions by $r : V \to \{1, \ldots, p\}$, consider the bit-vectors b_e for $e \in E$ that are computed with the set of shortest paths to boundary nodes only. If for all edges $(u, v) \in E$ the region bit of the target node of the edge $b_{(u,v)}(r(v))$ is set to* `true`*, the bit-vectors are consistent shortest-path containers.*

Proof: Let $s \in V$ and $t \in V$ be arbitrary but fixed nodes. Let $P = (s = v_1, \ldots, v_k = t)$ denote a shortest s-t path. We have to show that for the (v_1, v_2) the bit $b_{(v_1,v_2)}(r(t))$ for the target regions $r(t)$ is set. If $r(v_2) = r(t)$ (i.e. v_2 is in the target region), then the region bit $r(v_2)$ is set. If $r(v_2) \neq r(t)$, we consider the first edge $(v_i, v_{i+1}) \in P$ such that $r(v_i) \neq r(v_{i+1}) = r(t)$. The sub-path of $(v_1, \ldots, v_{i+1})$ of P is also a shortest path and v_{i+1} is a boundary node. The preprocessing restricted to shortest paths to boundary node has considered the path from v_1 to v_{i+1} and hence it has set the bit of region $r(v_{i+1}) = r(t)$. □

For street networks, it is reasonable to assume that for all edges $(u, v) \in E$, the edge itself is a shortest u-v path (i.e. $l(u, v) = d(u, v)$). Therefore, for all edges $(u, v) \in E$ the region bit of the target node of the edge $b_{(u,v)}(r(v))$ is set.

The set of shortest paths to boundary nodes can be computed efficiently in the reverse graph G_{rev}. For every boundary node b, the shortest-path tree T_b rooted at b is determined. For all edges $e \in T_b$, bit $r(b)$ must be set to `true`. If we denote the number of boundary nodes by k, the time complexity for the improved preprocessing is $\mathcal{O}(k \cdot n \log n)$.

The number k of boundary nodes is highly dependent on the partitioning of the nodes. We can minimize k by taking the minimal edge-separator for a certain number of regions. However, we will see in section 2.5.6 that this partitioning method does not always lead to the best results is respect to the search space of DIJKSTRA'S ALGORITHM WITH PRUNING. As an example for the resulting preprocessing time: This improved preprocessing algorithm needed for one of our real-world graphs with 473000 nodes and 1.1 million edges using a 10×10 grid-partition in 2.5 hours and the average search space during the single requests was reduced to less than 4% compared to the standard search.

Preprocessing with Pruned Shortest-Path Trees

We will now present a technique to avoid the computation of the full shortest-path trees of all boundary nodes in G_{rev}. Consider the shortest-path tree of a node v_1 and a node v_2

in the same region as v_1. For each node $v \in V$ of the graph, we get an upper bound of the length of the shortest path from v_2 to v: it can not be longer than the distance from v_2 to v_1 and the shortest path from v_1 to v. We can now use this upper bound to reduce our preprocessing effort.

For the first boundary node v_1 of each region, the algorithm computes its (reverse) shortest-path tree, the corresponding bit-vectors and distance from the closest boundary node v_2 of that region. During the computation of the shortest-path tree of v_2, we use the upper bounds of the prior search: if we find a shortest path to a node v which is longer or equal to the upper bound, the algorithm does not put v into the priority queue. (If a shorter path is found later, the node will be put into the priority queue and the algorithm provides correct results.) During the computation of the shortest path tree for the next boundary node v_3, upper bounds are computed using v_2, and so on.

Experiments show that this method reduces the number of nodes that are inserted in the priority queue during the preprocessing to less than 70%. The running time was improved by up to 20%.

2.5.3 Partitioning Algorithms

The bit-vector speed-up technique uses a partitioning of the graph to precompute information whether an edge may be part of a shortest path. Any possible partitioning can be used and the algorithm returns a shortest path, but most partitions do not lead to an acceleration. In this section, we will present the partitioning algorithms that we examined. Most of these algorithms need a layout of the graph. In our case, this is a 2D layout, but all partitioning algorithms can be adapted easily to higher dimensions.

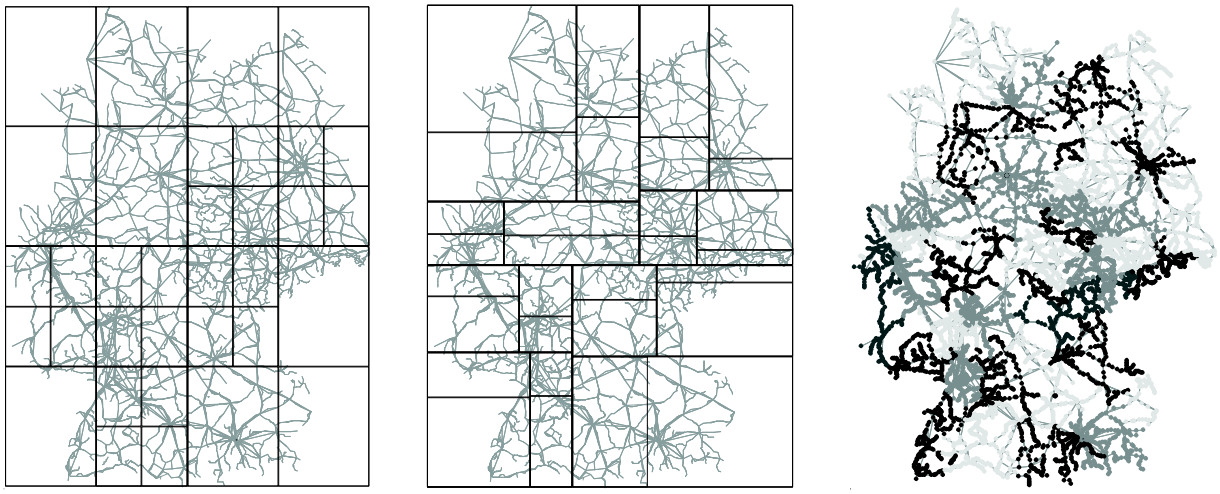
Grid

Probably the easiest way to partition a graph with a 2D layout is to use regions induced by a $n \times m$ grid of the bounding box. More precisely, we denote with $(l,t)^t$ the top-left vertex of the bounding box of the 2D layout of the graph and with $(r,b)^t$ the bottom-right vertex. The grid cell $G_{i,j}$ with $0 \le i < n, 0 \le j < m$ is now defined as the rectangle

$$\left[l + i \cdot \frac{r-l}{n}; l + (i+1) \cdot \frac{r-l}{n}\right] \quad \times \quad \left[b + j \cdot \frac{t-b}{m}; b + (j+1) \cdot \frac{t-b}{m}\right]$$

(Nodes on a grid line are assigned to an arbitrary but fixed grid cell.) Figure 2.9 shows an example of a 5×7 grid.

A bit-vector for a grid can be seen as a *raster image* of $S(u,v)$, where $S(u,v)$ represents the set of nodes x for which the shortest u-x path starts with the edge (u,v). The pixel i in the image is set, iff (u,v) is the beginning of a shortest path to a node in region $i \in \{1,\ldots,p\}$. A finer grid (i.e., an image with higher resolution) provides a better image of $S(u,v)$, but requires more memory. (On the other hand, the geometric objects in Section 2.3 approximate $S(u,v)$ by a single convex object of constant size.)



(a) Quadtree (34 regions) (b) Kd-Tree (32 regions) (c) METIS (32 regions).

Figure 2.10: Germany with three different partitions

The grid partitioning method uses only the bounding box of the graph—all other properties like the structure of the graph or the density of nodes are ignored and hence it is not surprising that the grid partitioning always has the worst results in our experiments. Since earlier works on this speed-up method used this partitioning method, we use the grid partitioning as a baseline and compare all other partitioning algorithms with it.

Quadtrees

A *quadtree* is a data structure for storing points in the plane. Quadtrees are typically used in computational geometry for range queries and have applications in computer graphics, image analysis, and geographic information systems.

Definition 14 (Quadtree) *Let P be a set of points in the plane and R_0 its bounding-box. Then, the data structure* quadtree *is a rooted tree of rectangles, where*

- *the root is the bounding region R_0, and*
- *R_0 and all other regions R_i are recursively divided into the four quadrants, while they contain more than one point of P.*

The leaves of a quadtree form a subdivision of the bounding-box R_0. Even more, the leaves of every sub-tree containing the root form such a subdivision. Since, for our application, we do not want to create a separate region for each node, we use a sub-tree of the quadtree. More precisely, we define an upper bound $b \in \mathbb{N}$ of points in a region and stop the division if a region contains less points than this bound b. This results in a partition of our graph where each region contains at most b nodes. Fig. 2.10(a) shows such a partition with 32

regions. In contrast to the grid-partition, this partitioning reflects the geometry of the graph—dense parts will be divided into more regions than sparse parts.

Kd-Trees

In the construction of a quadtree, a region is recursively divided into four equally-sized sub-regions. However, equally-sized sub-regions do not take the distribution of the points into account. This leads to the definition of a *kd-tree*. In the construction of a *kd-tree*, the plane is recursively divided, similar to a quadtree. The underlying rectangle is decomposed into *two halves* by a straight line parallel to an axis. The directions of the dividing line alternate. The positions of the dividing line can depend on the data. Frequently used positions are given by the center of the rectangle (*standard* kd-tree), the *average*, or the *median* of the points inside. (Fig. 2.10(b) shows a result for the median and 32 regions.) In applications with higher dimensions, the partition axes are not cycled but the dimension with the largest variance is used.

If the median of points in general position is used, the partitioning has always 2^l regions and at every decomposition, one node of the graph lies exactly on the boundary of two regions. For these nodes, it is worthwhile to check whether all neighbors of that node have their positions in the other region. If yes, the node can be transfered to the other region and will not become a boundary node.

The median of the nodes can be computed in linear time with the *median of medians* algorithm [59]. Since the running time of the preprocessing is dominated by the shortest-path computations after the partitioning of the graph, we decided to use a standard sorting algorithms instead. (As a concrete example, the *kd-tree* partitioning with 64 regions for one of our test graphs with one million nodes was calculated in 175s, calculating the bit-vectors took seven hours.)

METIS

A fast method to partition a graph into k almost equally-sized sets with a small cut-set is presented in [60]. An efficient implementation can be obtained free-of-charge from [61]. There are two advantages of this method for our application. The METIS partitioning does not need a layout of the graph and the preprocessing is faster because the number of edges in the cut is noticeable smaller than in the other partitioning methods. Fig. 2.10(c) shows a partitioning of a graph generated by METIS.

2.5.4 Two-Level Bit-Vectors

An analysis of the calculated bit-vectors reveals that there might exist possibilities to compress the bit-vectors. For 80% of the edges either almost none or nearly all bits of their bit-vectors are set. Table 2.3 shows an excerpt of the analysis we made. The column "= 1"shows the number of edges, which are only responsible for shortest paths inside their own region (only one bit is set). Edges with more than 95% bits set, could be important

graph	#edges	algorithm	# marked regions		
			$= 1$	$< 10\%$	$> 95\%$
street_network_1	920,000	KdTree(32)	351,255	443,600	312,021
street_network_1	920,000	KdTree(64)	334,533	470,818	294,664
street_network_1	920,000	METIS(80)	346,935	468,101	290,332
street_network_4	2,534,000	KdTree(32)	960,779	1,171,877	854,670
street_network_4	2,534,000	KdTree(64)	913,605	1,209,353	799,206

Table 2.3: Some statistics about the number of regions that are marked in bit-vectors.

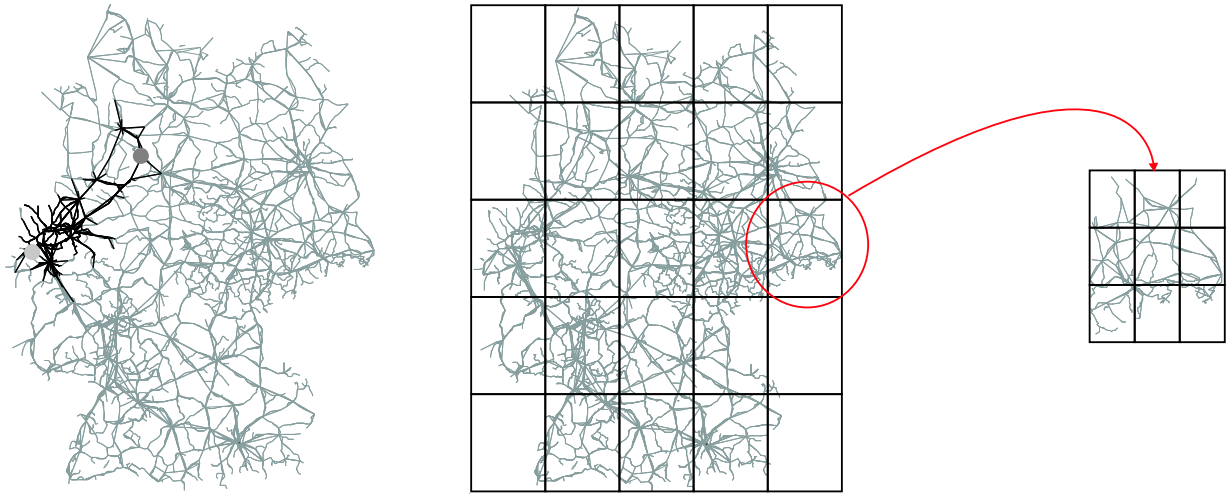
roads. This justifies ideas for (lossy) compression of the bit-vectors, but it is important that the decompression algorithm is very fast—otherwise the speed-up of time will be lost.

Let us have a closer look at a search space to get an idea of how to compress the bit-vectors. As illustrated in Figure 2.11(a) for a search from the dark grey node to the light grey node, the modified Dijkstra search reduces the search space next to the beginning of the search but once the target region has been reached, almost all nodes and edges are visited. This is not very surprising if you consider that all edges of a region have set the region-bit of their own region. We could handle this problem if we used a finer partition of the graph but this would lead to longer bit-vectors (requiring more memory). As a concrete example, if we used a 15×15 grid instead of a 5×5 grid, each region would be split in 9 additional regions but the preprocessed data increases from 25 to 225 bits per edge. However, the additional information for the fine grid is mainly needed for edges in the target region of the coarse grid. This leads to the idea that we could split each region of the coarse partition but store this additional data (for the fine grid) only for the edges inside the same coarse region. Therefore, each edge gets two bit-vectors: one for the coarse partition and one for the associated region of the fine partition.

The advantage of this method is that the preprocessed data is smaller than for a fine one-level partitioning, because the second bit-vector exists only for the target region (34 bits per edge instead of 225). It is clear that the 15×15 grid would lead to better results. However, the difference for the search spaces is small because we expect that entries in bit-vectors of neighboring regions are similar for regions far away. Thus, we could see this two-level method as a (lossy) compression of the first-level bit-vectors: We summarize the bits for remote regions. If one bit is set for a fine region, the bit is set for the whole group.

Only a slight modification of the search algorithm is required. Until the target region is reached, everything will remain unaffected, unnecessary edges will be ignored with the bit-vectors of level one. If the algorithm has entered the target region, the second-level bit-vector provides further information on whether an edge can be ignored for the search of a shortest path to the target-node.

Experiments showed (Section 2.5.6) that this method leads to the best results concerning the reduction of the search space, but an increased preprocessing effort is needed. Note however, that it is not necessary in the preprocessing to compute the complete shortest-path trees for all boundary nodes of the fine partitioning. The computation can be stopped if all nodes in the same coarse region are finished.



(a) Without a two-level bit-vector, a search visits almost all edges in the target region.

(b) For each edge, a bit-vector is stored for the coarse 5×5 grid and a bit-vector for a fine 3×3 grid *in the same coarse region as the edge*.

Figure 2.11: Illustrations for two-level vectors

2.5.5 Experimental Setup

The main goal of this section is to compare the different partitioning algorithms with regard to their resulting search space and speed-up of time during the accelerated DIJKSTRA search. We tested the algorithms on German street networks, which are directed and have a 2D layout and positive, integral edge weights. Table 2.4 shows some characteristics of the graphs. The column "shortest path" is the average number of nodes on a shortest path in the graph. For the unmodified DIJKSTRA'S ALGORITHM, the average running time and number of nodes touched by the algorithm is given for 5000 runs.

All experiments are performed with an implementation of the algorithms in C++ using the GCC compiler 3.3. We used the graph data structure from LEDA 4.4 [44]. In order

Graph	#nodes	#edges	shortest path	DIJKSTRA'S ALGORITHM time [s]	#touched nodes
street_network_1	362,000	920,000	250	0.26	183,509
street_network_2	474,000	1,169,000	440	0.27	240,421
street_network_3	609,000	1,534,000	580	0.30	306,607
street_network_4	1,046,000	2,534,000	490	0.78	522,850

Table 2.4: Characteristics of tested street networks. The columns "shortest paths" provides the average number of nodes on a shortest path.

to measure the unaffected speed-ups of time, we additionally implemented the algorithms without the framework for code re-use presented in Section 4. Experiments showed, that even the most modern C++ compilers do not optimize away virtual functions calls when possible (even with aggressive optimization turned on). Removing unnecessary function calls "by hand" accelerated a single shortest-path request up to a factor of ten for complex inheritance hierarchies.

For each graph, we generated a demand file with 5000 random shortest-path requests so that all algorithms use the same shortest-path demands. All runtime measurements were made on a single AMD Opteron Processor with 2.2 GHz and 8 GB RAM.

From an abstract point of view, our speed-up method reduces the complete graph for each search to a smaller sub-graph and this leads to a smaller search space. We sampled the average size of the search space by counting the number of visited nodes and measured the average CPU time per query. DIJKSTRA'S ALGORITHM is used as a reference algorithm to compare search space and CPU time. Fortunately, DIJKSTRA'S ALGORITHM WITH PRUNING only tests additionally a bit of a bit-vector and this does not lead to a significant overhead. In graphs we tested, there is a strong linear correlation between the search space and the CPU time. This justifies that in the analysis it is sufficient to consider the search space only.

2.5.6 Computational Results

Quadtrees and Kd-Trees

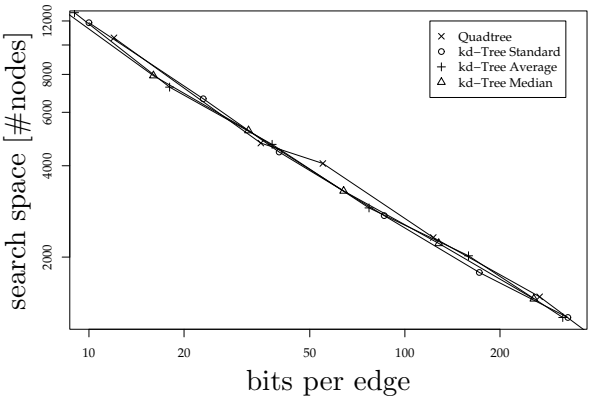
We first compared the four geometric partitioning methods quadtrees and kd-trees for the center (standard), average, and median. Figure 2.12(a) shows the average search space for a street network for an increasing number of bits per edge. As the differences are indeed very small, we will use only kd-trees with median in the rest of this section as a representative for this partitioning class.

We now compare the average search space for different graphs. For an easy comparison we consider the search space *relative* to the average search space of DIJKSTRA'S ALGORITHM in this graph. Figure 2.12(b) provides the relative average search space for an increasing number of bits per edge. It is remarkable that for bit-vectors in this range of size all curves follow a power law.

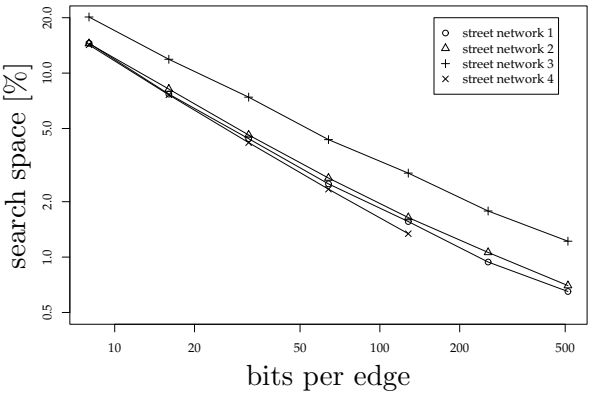
Two-Level Partitionings

The main reason for the introduction of the second-level partitions was that no edge is excluded from the shortest-path search inside the region of the target node t. Therefore, the second-level bit-vectors reduce the shortest-path search mainly if the search already approaches the target. Figure 2.13 compares the search spaces of the one-level and two-level accelerated searches. Although only very few bits are added, the average search space is reduced to about half of its size.

Using a bidirectional search, the two-level strategy becomes less important, because



(a) Partitioning with quadtree and three kd-tree partitions (standard, average, and median). The difference for the resulting search space is marginal.



(b) Partitioning with median kd-tree for street network 1-4. The search space is plotted relative to the search space of Dijkstra's algorithm.

Figure 2.12: Average search space for different sizes of bit-vectors. With an increasing number of bits per edge, the search space gets smaller.

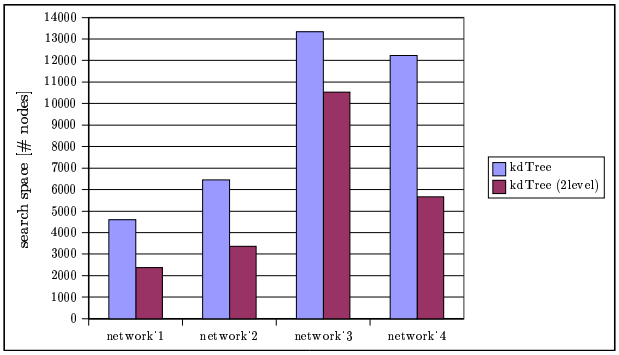


Figure 2.13: Comparison of one-level (64 regions) and two-level (64 first-level regions, 8 second-level regions) bit-vectors with kd-trees.

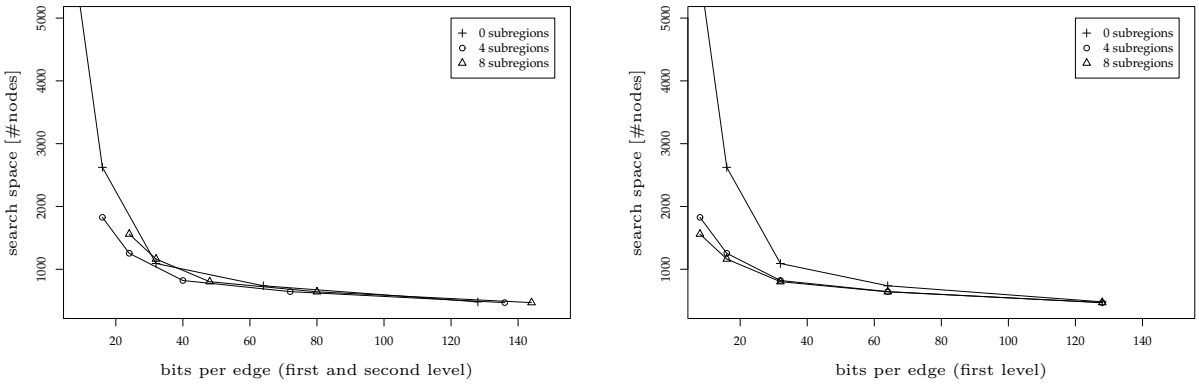


Figure 2.14: Average search space for a bidirectional search using bit-vectors by kd-trees. The two-level strategy becomes irrelevant for the bidirectional search. If more than 50 regions are used for the first-level, the two-level acceleration provides no noticeable improvement.

the second-level bit-vectors will not be used in most of the shortest-path searches: the second-level bit-vectors are only used, if the search enters the region of the target. During a bidirectional search the probability is high that the two search horizons meet in a different region than the source or target region. Therefore, the second-level bit-vectors are only used, if both nodes are lying in the same region. Figure 2.14 confirms this estimation. Only for large partitions in the first level is a speed-up recognizable with two-level bit-vectors. If more than 50 bits for the first level are used, the difference is very small. We conclude that the second-level strategy does not seem to be useful in a bidirectional search.

Comparison of the Partitioning Methods

Finally, we want to compare the different algorithms directly. We have four orthogonal dimensions in our algorithm tool-box:

1. The base partitioning method: Grid, KdTree or METIS
2. The number of partitions
3. Usage of one-level partitions or two-level partitions
4. Unidirectional or bidirectional search

Since computing all possible combinations on all graphs takes way too much time, we selected the algorithms that are listed in Table 2.5. (We refrained from implementing Bi2Metis, because usually the two-level bit-vectors in a bidirectional search hardly performed better than the one-level variant.) Furthermore, we fix the size of the preprocessed data to nearly the same number for all algorithms. (The same size can not be realized

Name of partitioning	forward		backward		bits per edge
	1^{st} level	2^{nd} level	1^{st} level	2^{nd} level	
Grid	9×9	-	-	-	81
KdTree	64	-	-	-	64
METIS	80	-	-	-	80
2LevelGrid	8×8	4×4	-	-	80
2LevelKd	64	16	-	-	80
2LevelMETIS	72	8	-	-	80
BiGrid	7×7	-	6×6	-	85
BiKd	32	-	32	-	64
BiMETIS	40	-	40	-	80
Bi2LevelGrid	6×6	2×2	6×6	2×2	80
Bi2LevelKd	32	8	32	8	80

Table 2.5: Partitionings with nearly the same preprocessed data size of 80 bit

due to the restrictions by the construction of the partitioning algorithms.) Figure 2.15 compares the results of our partitioning methods on the four street networks.

For the unidirectional searches, the two-level strategies yield the best results (a factor of 2 better than for their corresponding one-level partitioning). For the bidirectional search, we can see some kind of saturation: the differences between the partitioning techniques are very small.

The best partition-based speed-up method we tested is a bidirectional search, accelerated in both directions with kd-tree or METIS partitions. We can measure speed-ups of more than 500. In general, the speed-up increases with the size of the graph. Figure 2.16 shows the search space for the four street networks. Note that in case of a bidirectional search, a large number of bits per edge already shows some effects of saturation as the curves are bent. In contrast, in Fig. 2.12(b) the curves follow a power-law. Even with the smallest preprocessed data (16 bit per edge), we get a speed-up of more than 50. The accelerated search on network_4 is 545 times faster than plain DIJKSTRA'S ALGORITHM using 128 bits per edge preprocessed data ($1.3ms$ per search).

Of the tested partitioning methods, we can recommend the kd-tree used for forward and backward acceleration. The partitioning with kd-trees and METIS yield the highest speed-up factors, but kd-trees easier to implement.

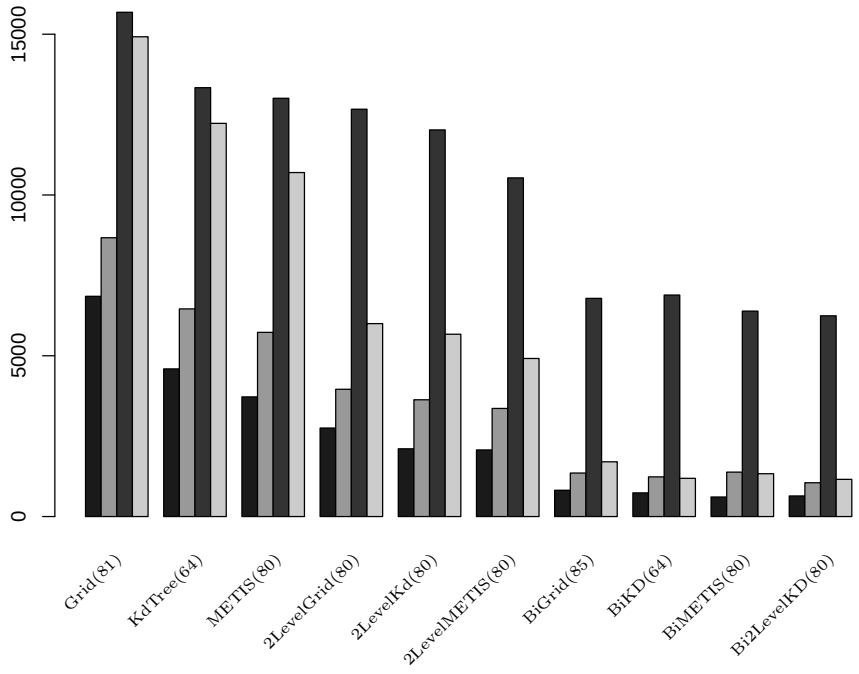


Figure 2.15: Average search space for most of the implemented algorithms in street networks 1-4. The number of bits are noted in brackets.

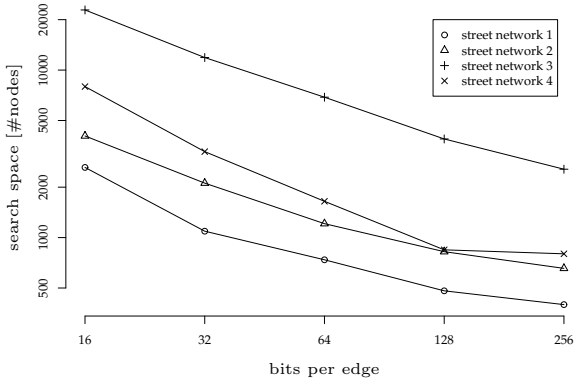


Figure 2.16: Search space for street networks 1-4 with a bidirectional accelerated search using kd-tree partitions.

Chapter 3

Combining Speed-up Techniques

In this chapter, the impact of using shortest-path containers is compared with the impact of other previously known speed-up techniques. Even more important, we examine their combinations and how well a speed-up technique scales if it is added to another combination. In particular, we consider all 16 combinations of the following four speed-up techniques from Section 1.2:

Goal-Directed Search (`go`) modifies the given edge weights to favor edges leading towards the target node [22, 62]. With graphs from timetable information, a speed-up in running time of a factor of roughly 1.5 is reported in [7].

Bidirectional Search (`bi`) starts a second search backwards, from the target to the source (see [63], Section 4.5). Both searches stop when their search horizons meet. Experiments in [64] showed that the search space can be reduced by a factor of 2, and in [65] it was shown that combinations with the goal-directed search can be beneficial.

Multi-Level Approach (`ml`) takes advantage of hierarchical coarsenings of the given graph, where additional edges have to be computed. They can be regarded as distributed to multiple levels. Depending on the given query, only a small fraction of these edges has to be considered to find a shortest path. Using this technique, speed-up factors of more than 3.5 have been observed for road map and public transport graphs [23]. Timetable information queries could be improved by a factor of 11 (see [9]), and in [66], good improvements for road maps are reported for a similar approach.

Shortest-Path Container (`sc`). For this computational study, we selected bounding boxes as container type, since they are easy to implement, and—among constant-sized objects—the resulting algorithm is the fastest in terms of CPU cycles and still very fast with respect to the number of visited nodes.

Goal-directed search and shortest-path containers are only applicable if a layout of the graph is provided. Multi-level approach and shortest-path containers both require a preprocessing, calculating additional edges and bounding boxes, respectively. All these four

techniques are tailored to Dijkstra's algorithm. They crucially depend on the fact that Dijkstra's algorithm is label-setting and that it can be terminated when the destination node is settled. (Therefore, the algorithm does not necessarily search the whole graph.)

We first show that, with more or less effort, all $2^4 = 16$ combinations can be implemented. Then, an extensive experimental study of their performance is provided. Benchmarks were run on several real-world and generated graphs, where operation counts as well as CPU time were measured.

The next section provides details how to combine the speed-up techniques. Section 3.2 presents the experimental setup and data sets for our statistics, and the belonging results are given in Section 3.3.

3.1 Speed-up Techniques and their Combinations

In this section, we enlist for every pair of speed-up techniques how we combined them. The extension to a combination of three or four techniques is straight forward, once the problem of combining two of them is solved.

3.1.1 Goal-Directed Search and Bidirectional Search

Combining goal-directed and bidirectional search is not as obvious as it may seem at first glance. [64] provides a counter-example for the fact that simple application of a goal-directed search forward and backward yields a wrong termination condition. However, the alternative condition proposed there has been shown in [65] to be quite inefficient, as the search in each direction almost reaches the source of the other direction. This often results in a slower algorithm.

To overcome these deficiencies, we simply use the very same edge weights $l'(v, w) := l(v, w) - p_t(v) + p_t(w)$ for both the forward and the backward search. With these weights, the forward search is directed to the target t and the backward search has no preferred direction, but favors edges that are directed towards t. This should be (and indeed is) faster than each of the two speed-up techniques. This combination computes a shortest path, because a shortest s-t path is the same for given edge weights l and edge weights modified according to goal-directed search, l'.

3.1.2 Goal-Directed Search and Multi-Level Approach

As described in Section 1.2.3, the multi-level approach basically determines for each query a subgraph of the multi-level graph, on which Dijkstra's algorithm is run to compute a shortest path. The computation of this subgraph does not involve edge lengths and thus goal-directed search can be simply performed on it.

3.1.3 Goal-Directed Search and Shortest-Path Containers

Similar to the multi-level approach, the shortest-path containers approach determines for a given query a subgraph of the original graph. Again, edge lengths are irrelevant for the computation of the subgraph and goal-directed search can be applied offhand.

3.1.4 Bidirectional Search and Multi-Level Approach

Basically, bidirectional search can be applied to the subgraph defined by the multi-level approach. In our implementation, that subgraph is computed on the fly during Dijkstra's algorithm: for each node considered, the set of necessary outgoing edges is determined. If a bidirectional search is applied to the multi-level subgraph, a symmetric, backward version of the subgraph computation has to be implemented: for each node considered in the backward search, the incoming edges that are part of the subgraph have to be determined.

3.1.5 Bidirectional Search and Shortest-Path Containers

In order to take advantage of shortest-path containers in both directions of a bidirectional search, a second set of containers is needed. For each edge $e \in E$, we compute the set $S_{\text{rev}}(e)$ of those nodes from which a shortest path ending with e exists. We store for each edge $e \in E$ the bounding box of $S_{\text{rev}}(e)$ in an associative array C_{rev} with index set E. The forward search checks whether the target is contained in $C(e)$. The backward search checks whether the source is contained in $C_{\text{rev}}(e)$.

3.1.6 Multi-Level Approach and Shortest-Path Containers

The multi-level approach enriches a given graph with additional edges. Each new edge (u_1, u_k) represents a shortest path $(u_1, u_2, \ldots, u_k)$ in G. We annotate such a new edge (u_1, u_k) with $C(u_1, u_2)$, the associated bounding box of the first edge on this path.

3.2 Experimental Setup

In this section, we provide details on the input data used, consisting of real-world and randomly generated graphs, and on the execution of the experiments.

3.2.1 Data

Real-World Graphs

In our experiments, we included a set of graphs that stem from real applications. As in other experimental work, it turned out that using realistic data is quite important as the performance of the algorithms strongly depends on the characteristics of the data.

	street									
n	1444	3045	16471	20466	25982	38823	45852	45073	51510	79456
m	3060	7310	34530	42288	57620	79988	98098	91314	110676	172374
	public transport									
n	409	705	1660	2279	2399	4598	6884	10815	12070	14335
m	1215	1681	4327	6015	8008	14937	18601	29351	33728	39887
	planar									
n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
m	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
	waxman									
n	938	1974	2951	3938	4949	5946	6943	7917	8882	9906
m	4070	9504	14506	19658	24474	29648	34764	39138	44208	48730

Table 3.1: Number of nodes and edges for all test graphs

Street Graphs. Our street graphs are street networks of US cities and their surroundings. These graphs are bidirectional, and edge lengths are Euclidean distances. The graphs are fairly large and very sparse because bends are represented by polygonal lines. (With such a representation of a street network, it is possible to efficiently find the nearest point in a street by a point-to-point search.)

Public Transport Graphs. A public transport graph represents a network of trains, buses, and other scheduled vehicles. The nodes of such a graph correspond to stations or stops, and an edge between two nodes exists if there is a non-stop connection between the respective stations. The weight of an edge is the average travel time of all vehicles that contribute to this edge. In particular, the edge lengths are not Euclidean distances in this set of graphs.

Random Graphs

We generated two sets of random graphs that have an estimated average degree of 2.5 (which corresponds to the average degree in the real-world graphs). Each set consists of ten connected, bidirectional graphs with (approximately) $1000 \cdot i$ nodes ($i = 1, \ldots, 10$).

Random Planar Graphs. For the construction of random planar graphs, we used a generator provided by LEDA [44]. A given number of n nodes are uniformly distributed in a square with a lateral length of 1, and a triangulation of the nodes is computed. This yields a complete undirected planar graph. Finally, edges are deleted at random until the graph contains $2.5 \cdot n$ edges, and each of these is replaced by two directed edges, one in either direction.

Random Waxman Graphs. The construction of these graphs is based on a random graph model introduced by Waxman [67]. Input parameters are the number of nodes

n and two positive rational numbers α and β. The nodes are again uniformly distributed in a square of a lateral length of 1, and the probability that an edge (u, v) exists is $\beta \cdot \exp(-d(u, v)/(\sqrt{2}\alpha))$. Higher β values increase the edge density, while smaller α values increase the density of short edges in relation to long edges. To ensure connectedness and bidirectionalness of the graphs, all nodes that do not belong to the largest connected component are deleted (thus, slightly less than n nodes remain) and the graph is bidirectional by insertion of missing reverse edges. We set $\alpha = 0.01$ and empirically determined that setting $\beta = 2.5 \cdot 1620/n$ yields an average degree of 2.5, as wished.

3.2.2 Experiments

We have implemented all combinations of speed-up techniques as described in Sections 1.2 and 3.1 in C++, using the graph and binary heap data structures of the LEDA library [44] (version 4.4). The code was compiled with the GNU compiler (version 3.3), and experiments were run on an Intel Xeon machine with 2.6 GHz and 2 GB of memory, running Linux (kernel version 2.4).

For each graph and combination, we computed for a set of queries shortest paths, measuring two types of *performance*: the mean values of the *running times* (CPU time in seconds) and the *number of nodes* inserted in the priority queue. The queries were chosen at random and the amount of them was determined such that statistical relevance can be guaranteed (see also Sect. 2.3.2).

3.3 Experimental Results

The outcome of the experimental study is shown in Figures 3.1 and 3.2. Further diagrams that we used for our analysis are depicted in Figures 3.3–3.11. Each combination is referred to by a 4-tuple of shortcuts: `go` (goal-directed), `bi` (bidirectional), `ml` (multi-level), `sc` (shortest-path container), and `--` if the respective technique is not used (e.g., `go bi -- sc`).

We calculated two different values denoting relative *speed-up*: on the one hand, Figures 3.1 and 3.2 show the speed-up that we achieved compared to plain Dijkstra, i.e., for each combination of techniques the ratio of the performance of plain Dijkstra and the performance of Dijkstra with the specific combination of techniques applied. There are separate figures for the number of nodes and running time, respectively.

On the other hand, for each of the Figures 3.3–3.6, we focus on one technique $\mathcal{T}$ and show for each combination containing $\mathcal{T}$ the speed-up that can be achieved compared to the combination without $\mathcal{T}$. For example, when focusing on bidirectional search and considering the combination `go bi -- sc`, say, we investigate by which factor the performance gets better when the combination `go bi -- sc` is used instead of `go -- -- sc` only.

In the following, we discuss, for each technique separately, how combinations with the specific technique behave, and then turn to the relation of the two performance parameters measured, the number of visited nodes and running time: we define the *overhead* of a

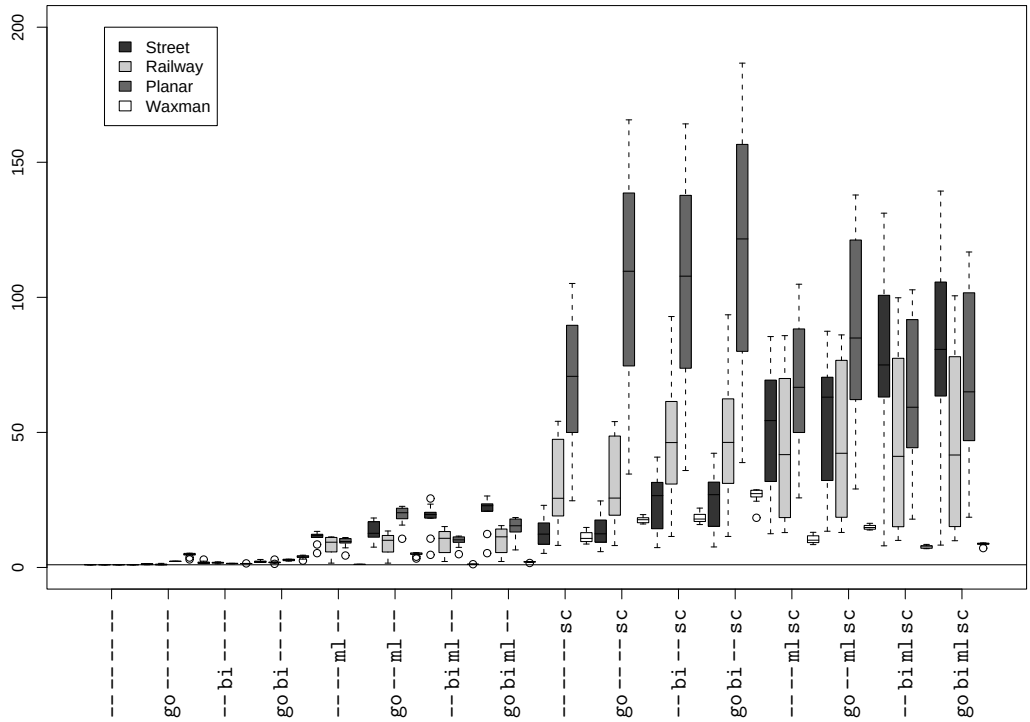


Figure 3.1: Speed-up relative to Dijkstra's algorithm in terms of visited nodes

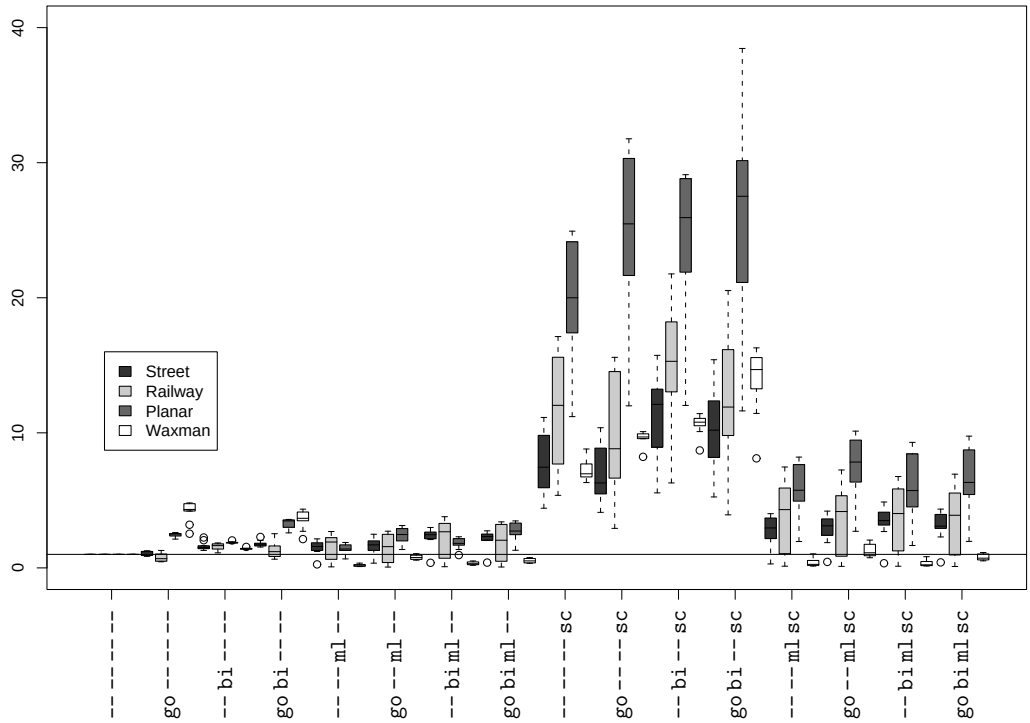


Figure 3.2: Speed-up relative to Dijkstra's algorithm in terms of running time

combination of techniques to be the ratio of running time and the number of visited nodes. In other words, the overhead reflects the time spent per node.

3.3.1 Speed-up of the Combinations

Goal-Directed Search

Individually comparing goal-directed search with plain Dijkstra (Figure 3.3), speed-up varies greatly between the different types of graphs: Considering the random graphs, we obtain a speed-up of about 2 for planar graphs but of up to 5 for the Waxman graphs, which is quite surprising. Only little speed-up, of less than 2, can be observed for the real-world graphs.

Concerning the number of visited nodes, adding goal-directed search to the multi-level approach is slightly worse than adding it to plain Dijkstra and with bidirectional search, we get another slight deterioration. Adding it to shortest-path containers (and combinations including them) is hardly beneficial.

For real-world graphs, adding goal-directed search to any combination does not improve the running time. For generated graphs, however, running time decreases. In particular, it is advantageous to add it to a combination containing multi-level approach. We conclude that combining goal-directed search with the multi-level approach generally seems to be a good idea.

Bidirectional Search

Bidirectional search individually gives a speed-up of about 1.5 for the number of visited nodes (see Figure 3.4) and for the running time, for all types of graphs. For combinations of bidirectional search with other speed-up techniques, the situation is different: For the generated graphs, neither the number of visited nodes nor the running time improves when bidirectional search is applied additionally to goal-directed search. However, running time improves with the combination containing the multi-level approach, and also combining bidirectional search with shortest-path containers works very well. In the latter case, the speed-up is about 1.5 (as good as the speed-up of individual bidirectional search) for all types of graphs.

Multi-Level Approach

The multi-level approach crucially depends on the decomposition of the graph. The Waxman graphs could not be decomposed properly by the multi-level approach, and therefore all combinations containing the latter yield speed-up factors of less than 1, which means a slowing down. Thus we consider only the remaining graph classes.

Adding multi-levels to goal-directed and bidirectional search and their combination gives a good improvement in the range between 5 and 12 for the number of nodes (see Figure 3.5). Caused by the big overhead of the multi-level approach, however, we get a considerable improvement in running time only for the real-world graphs. In combination

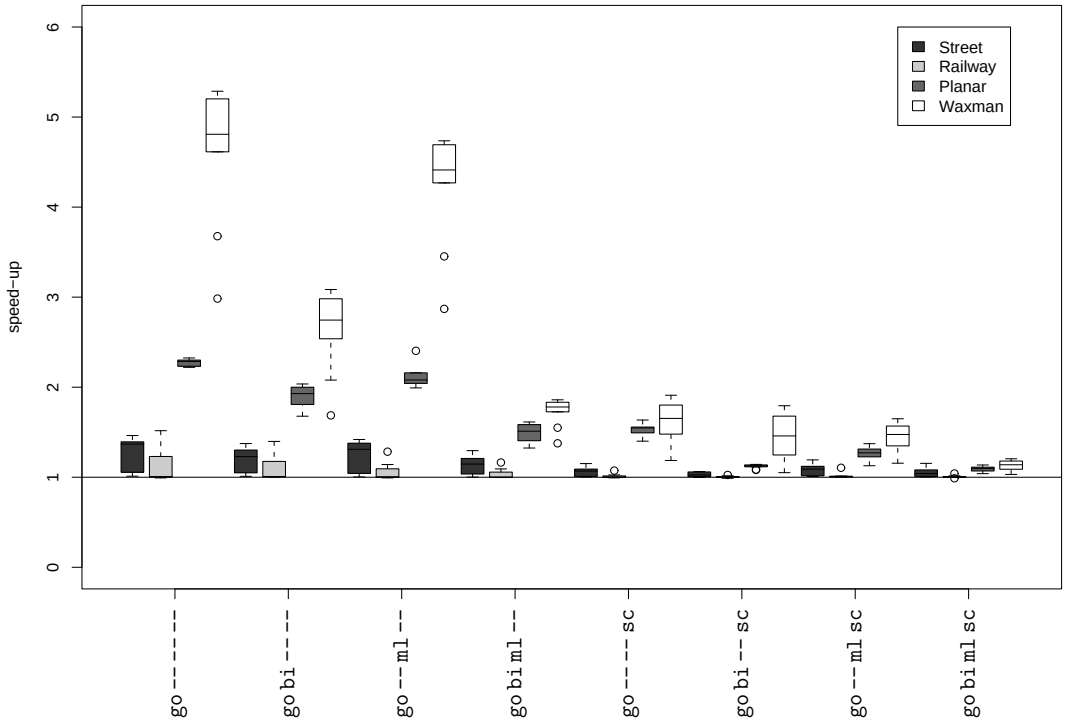


Figure 3.3: Speed-up relative to the combination without goal-directed search in terms of visited nodes

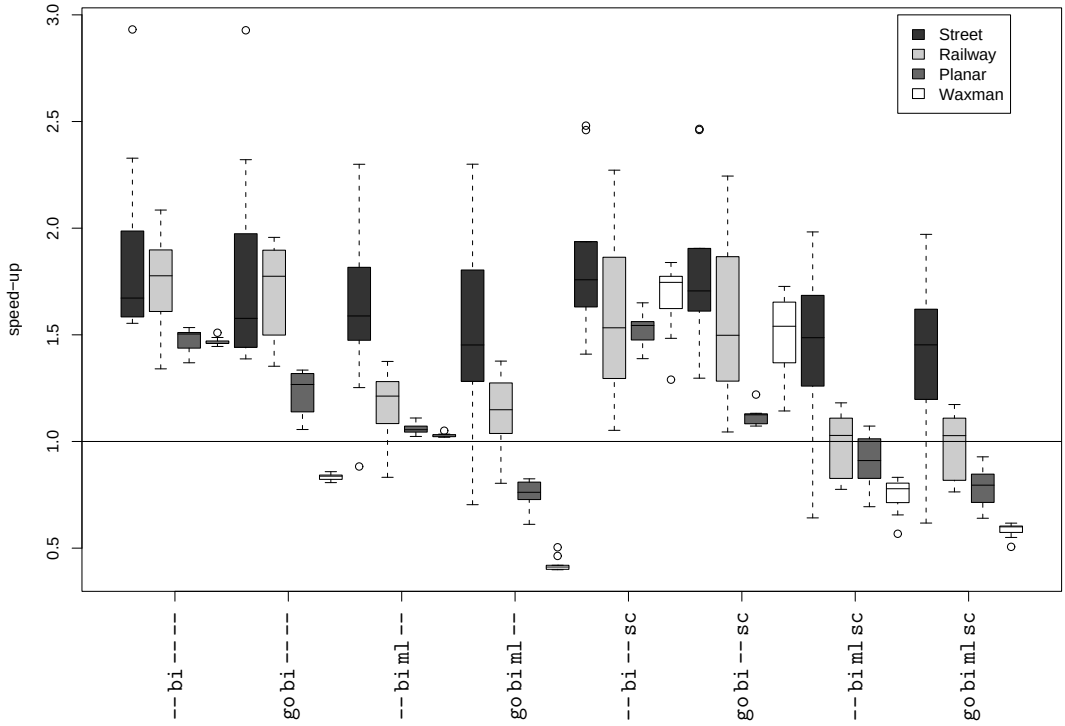


Figure 3.4: Speed-up relative to the combination without bidirectional search in terms of visited nodes

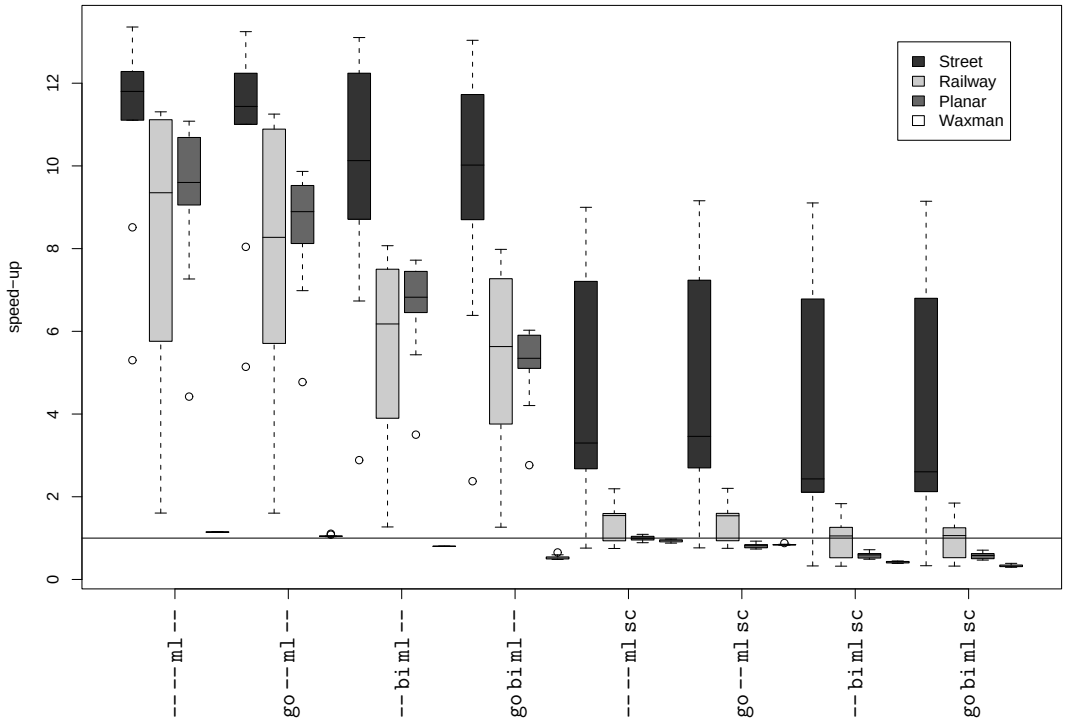


Figure 3.5: Speed-up relative to the combination without multi-level approach in terms of visited nodes

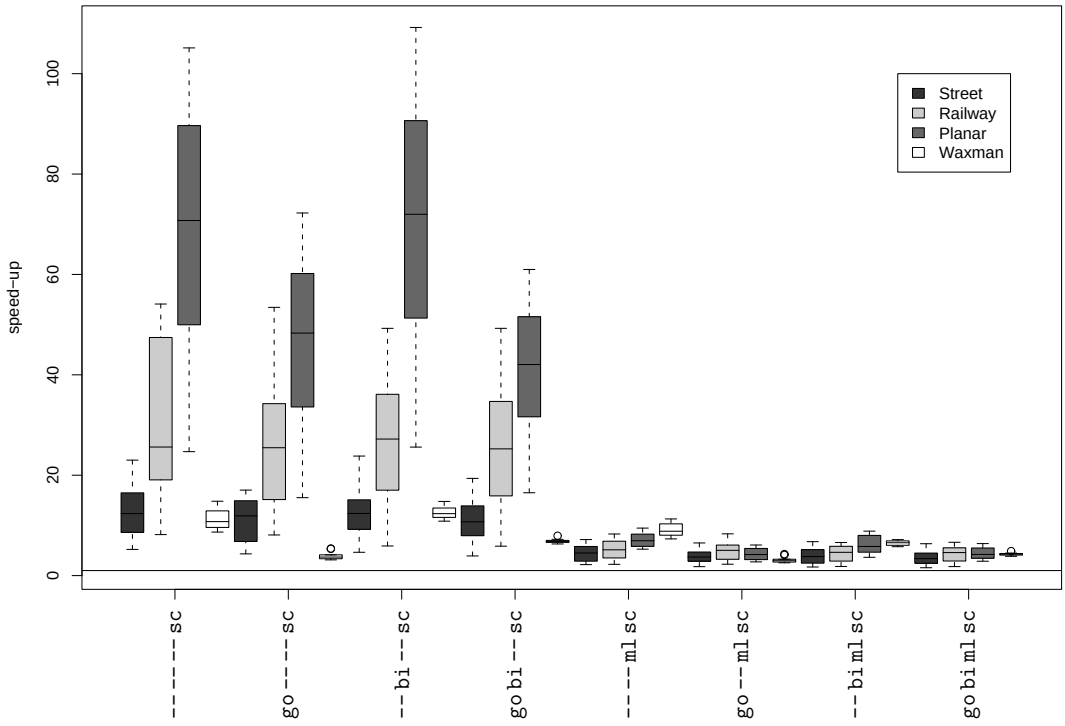


Figure 3.6: Speed-up relative to the combination without shortest-path containers in terms of visited nodes

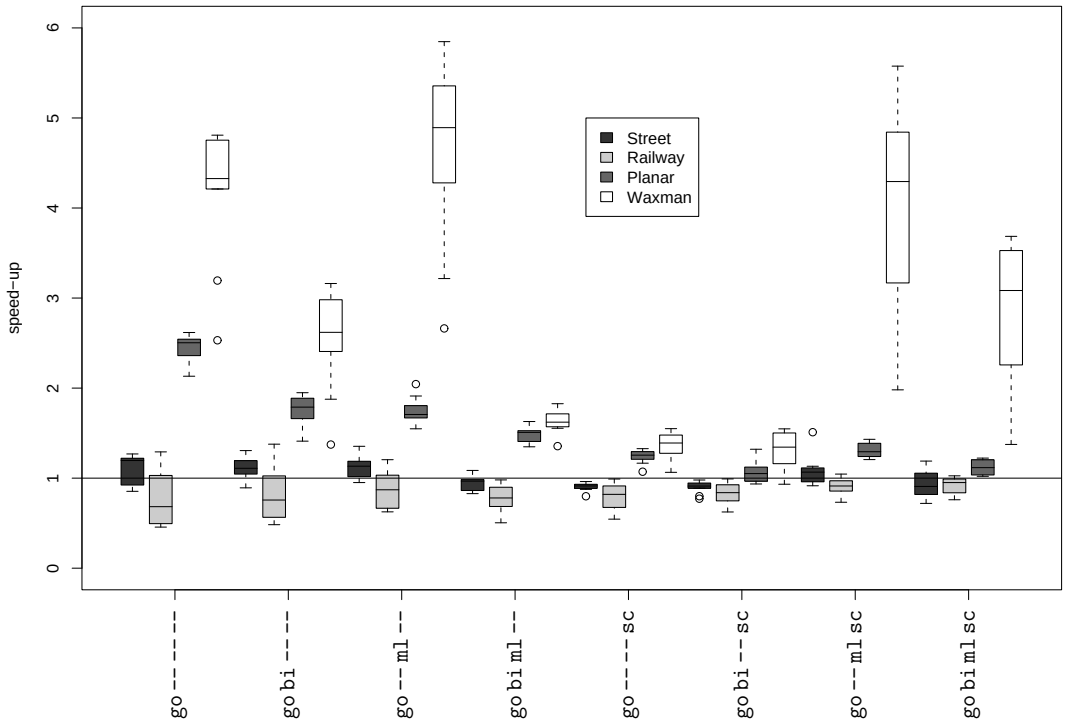


Figure 3.7: Speed-up relative to the combination without goal-directed search in terms of running time

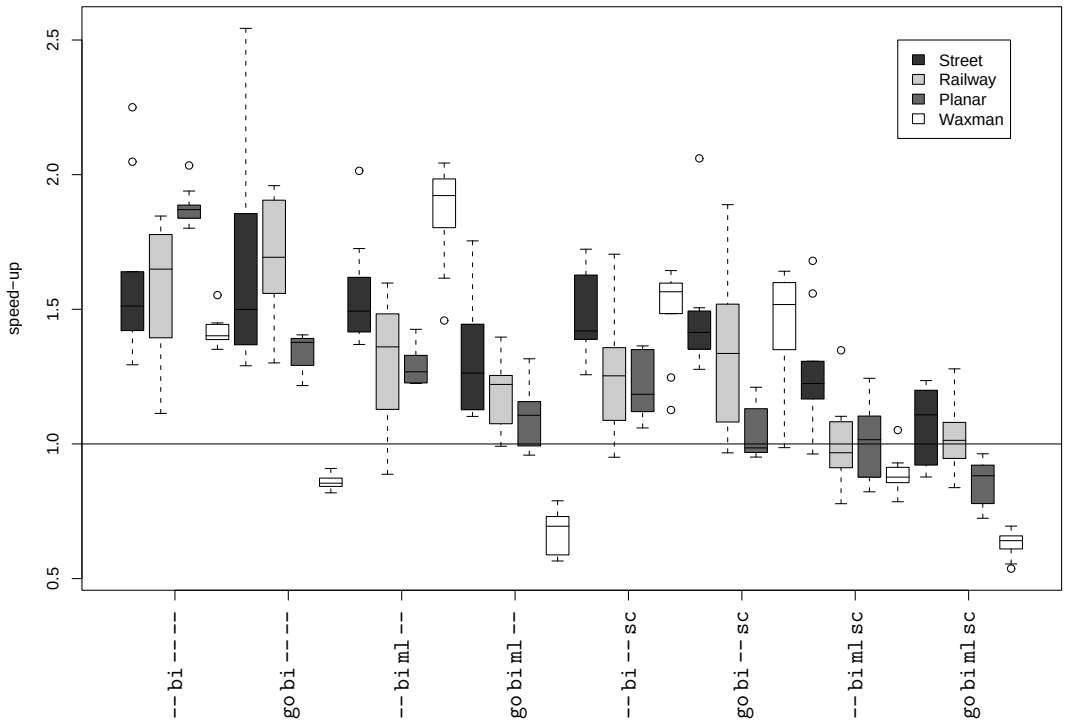


Figure 3.8: Speed-up relative to the combination without bidirectional search in terms of running time

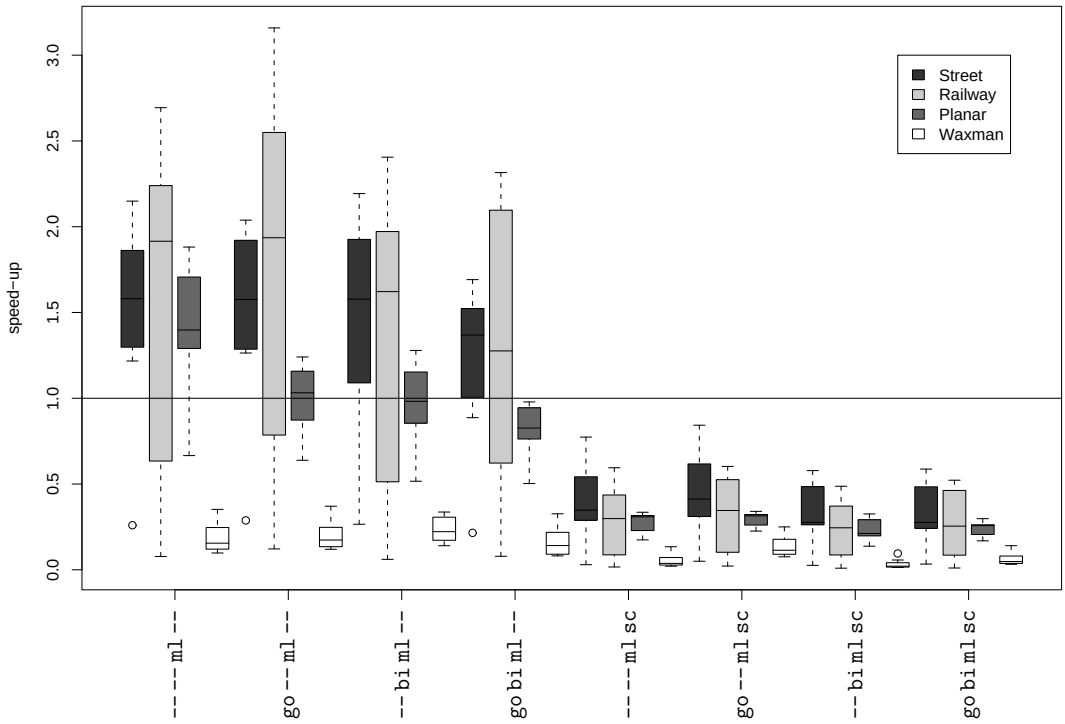


Figure 3.9: Speed-up relative to the combination without multi-level approach in terms of running time

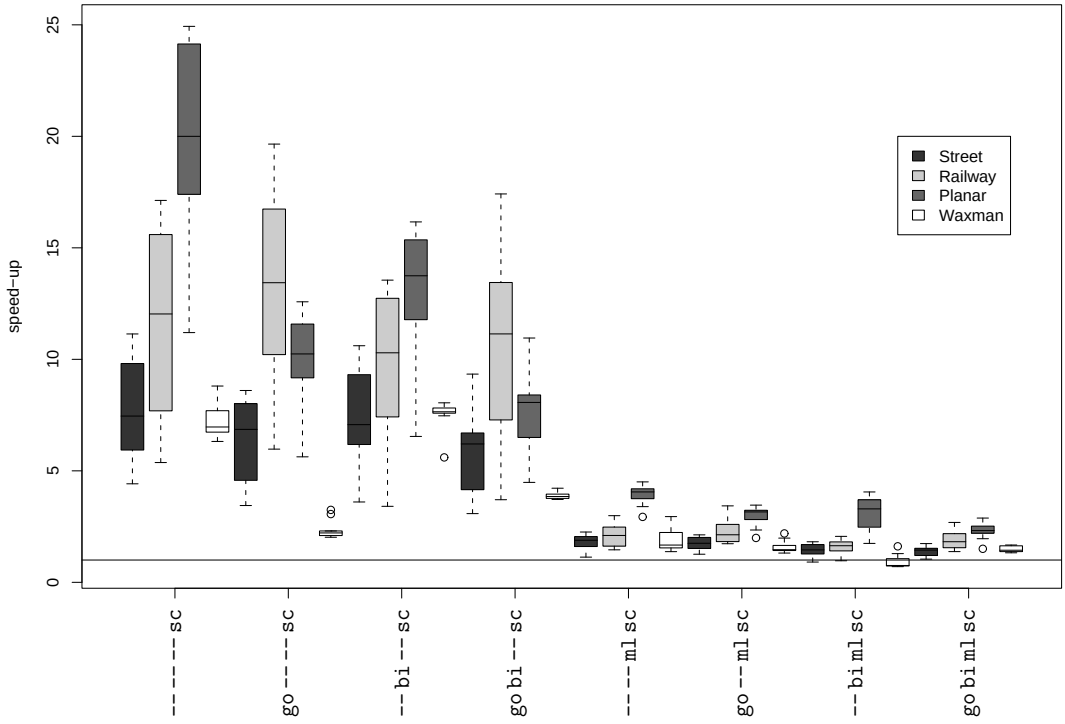


Figure 3.10: Speed-up relative to the combination without shortest-path containers in terms of running time

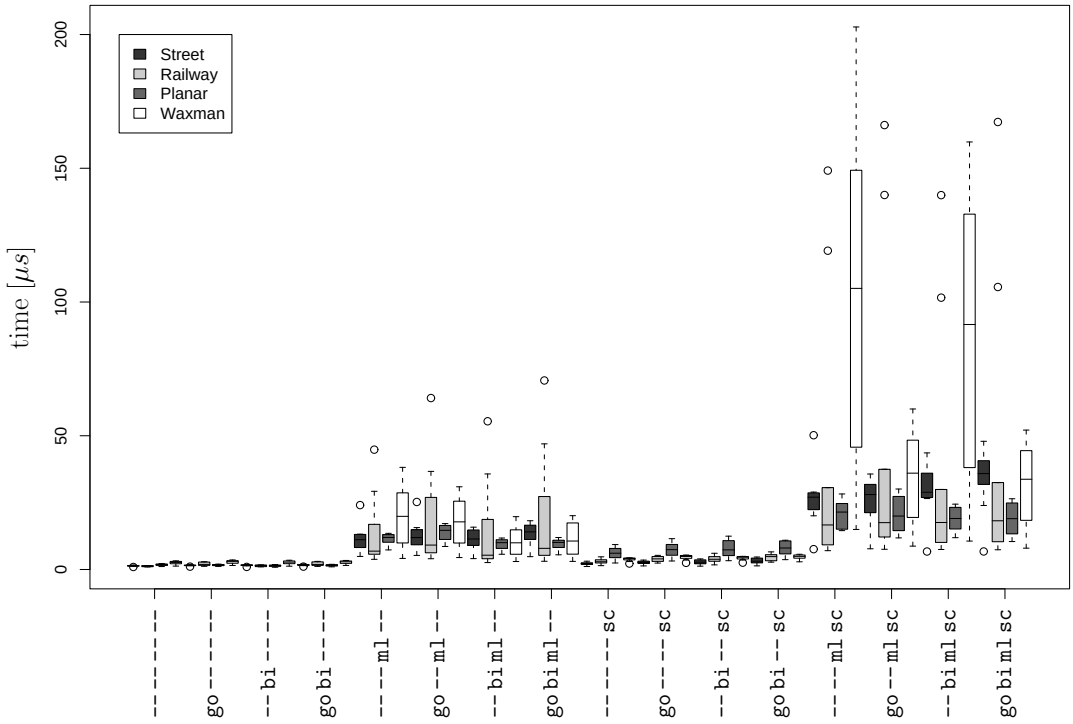


Figure 3.11: Average running time per visited node

with shortest-path containers, the multi-level approach is beneficial only for the number of visited nodes in the case of street graphs.

The multi-level approach allows tuning of several parameters, such as the number of levels and the choice of the selected nodes. The tuning crucially depends on the input graph [23]. Hence, we believe that considerable improvements of the presented results are possible if specific parameters are chosen for every single graph.

Shortest-Path Containers

Shortest-path containers work especially well when applied to planar graphs, i n fact, the speed-up even increases with the size of the graph (see Figure 3.6). For Waxman graphs, the situation is completely different: with the graph size, the speed-up becomes smaller. This can be explained by the fact that large Waxman graphs have, due to construction, more long-distance edges than small ones. Because of this, shortest paths become more tortuous and the bounding boxes contain more "wrong" nodes.

Throughout the different types of graphs, shortest-path containers individually as well as in combination with goal-directed and bidirectional search yield exceptionally high speed-ups. Only the combinations that include the multi-level approach cannot be improved that much.

3.3.2 Overhead

For goal-directed and bidirectional search, the overhead (time per visited node) is quite small, while for shortest-path containers it is a factor of about 2 compared to plain Dijkstra (see Figures 3.11. The overhead caused by the multi-level approach is generally high and quite different, depending on the type of graph. As Waxman graphs do not decompose well, the overhead for the multi-level approach is large and becomes even larger when the size of the graph increases. For very large street graphs, the multi-level approach overhead increases dramatically. We assume that it would be necessary to add a third level for graphs of this size. It is also interesting to note that the relative overhead of the combination goal-directed, bidirectional, and multi-level is smaller than just multi-level —especially for the generated graphs.

3.3.3 Conclusion

To summarize, we conclude that there are speed-up techniques that combine well and others where speed-up does not scale. Our result is that goal-directed search and multi-level approach is a good combination and bidirectional search with shortest-path containers complement each other.

For real-world graphs, a combination including bidirectional search, multi-level, and shortest-path containers is the best choice as to the number of visited nodes. In terms of running time, the winner is bidirectional search in combination with shortest-path containers. For generated graphs, the best combination is goal-directed, bidirectional, and shortest-path containers for both the number of nodes and running time.

Without an expensive preprocessing, the combination of goal-directed and bidirectional search is generally the fastest algorithm with smallest search space—except for Waxman graphs. For these graphs, pure goal-directed is better than the combination with bidirectional search. Actually, goal-directed search is the only speed-up technique that works comparatively well for Waxman graphs. Because of this different behaviour, we conclude that planar graphs are a better approximation of the real-world graphs than Waxman graphs (although the public transport graphs are not planar).

Chapter 4

Implementation

In the implementation of experimental algorithms, switching flexibly between different variants is a major issue apart from the efficiency of the algorithm. In our case, a basic algorithm (Dijkstra's algorithm) is refined in different ways in order to compare their performance and output. As a concrete example, the shortest path computation with Dijkstra's algorithm can be improved by goal-directed search or geometric pruning. Apart from such refinements, other so-called *aspects* [30] can be added to the algorithm: operation counting, time measurement or debugging output.

4.1 Adding Aspects by Parameterized Inheritance

For complex algorithms, it is favorable to keep the code of every aspect separate from the basic algorithm while preserving the efficiency. The basic algorithm is defined as a class with virtual functions at major key points as sketched below:

```
struct Algo {
  virtual void init() {...}
  virtual bool finished() {...}
  virtual void do_something() {...}

  void run() {
    init();
    while (!finished())
     { do_something(); }
  }
};
```

In our case, the algorithm is of course Dijkstra's algorithm. The constructor function of `Algo` initializes the priority queue, `init()` inserts the source in the priority queue, `finished()` tests whether the priority queue is empty, and so on. The function `do_something()` is just a mere representative for the number of functions inside the loop.

Suppose we are interested in the number of calls to the function `do_something()`. The aspect in question is then added by deriving a subclass that modifies virtual functions of the base class. This realization has the drawback that different aspects cannot be combined freely, because the base class is fixed. We get a much more flexible system, if we turn the aspect into a mix-in class, i.e., we make the base class a template argument.

```
template<typename Base> struct AspectCount::public Base {
  int operations;
  virtual void init() { Base::init(); operations=0; }
  virtual void do_something() { Base::do_something(); ++operations; }
};
```

Note that calling the base class function is mandatory in order to preserve the correctness of the algorithm and other aspects.

We are now ready to explain how different variants of Dijkstra's algorithm can be realized using aspects. Suppose that another aspect is implemented similarly to the above, e.g., `AspectTarget` for terminating Dijkstra's algorithm when the target is finished. The four variants of Dijkstra's algorithm with operation counting, termination at target, and termination at target with operation counting, can be easily instantiated:

```
Dijkstra AlgoA;                                 AlgoA.run(source,target);
AspectCount<Dijkstra> AlgoB;                    AlgoB.run(source,target);
AspectTarget<Dijkstra> AlgoC;                   AlgoC.run(source,target);
AspectTarget<AspectCount<Dijkstra> > AlgoD; AlgoD.run(source,target);
```

Thus, it is possible with parameterized inheritance to instantiate all combinations of aspects, while it is not necessary to actually implement the (exponentially many) combinations. Since all functions are inlined, the function calls can be optimized away by the compiler which leads to a flexible and efficient set of algorithms.

Other aspects that we implemented include storing a layout of the graph, goal-directed search, geometric pruning, performing operation counts including mean value and variance, constructing on-line or off-line containers in Create-Containers, or truncating Dijkstra's algorithm according to Lemma 9 or 12 for updates after changing edge weights.

4.2 Constructor Parameter List

The initialization of the algorithm is done by the constructor of the class and necessary parameters are given as arguments to the constructor. We will now discuss the question how additional parameters that are needed by aspects can be provided. We use a technique inspired by [68], but omit the creation of a repository.

Consider again Dijkstra's algorithm. The parameters that are given to the constructor are the graph and the edge lengths in this case. A speed-up technique that uses a layout of the graph would expect the layout in addition to the graph and the edge lengths.

The constructor of the aspect has to pass along the parameters to the base class.

```
Algo::Algo(Type1 Parameter1){...}

template<typename Base>
AspectA<Base>::AspectA(Type1 Parameter1, Type2 Parameter2):
  public Base(Parameter1)
{ do something with Parameter2 }
```

If there is another `AspectB` that expects another Parameter and we want to freely combine them, we have the problem that the constructor does not know which parameters the base class (including other aspects) needs. Either we have to write constructors for all combinations of parameters, or we have to sacrifice type safety by passing a list of pointers to `void` that are then casted to the respective parameters. A third method, the one that we follow, is to pass a meta-list of types. We will now describe, how such a meta-list can be realized in a way similar to that in [28].

```
struct END {};
static END End;

template<typename T, typename NEXT=END> struct LISTITEM {
  T &value;
  NEXT next;
};
```

The template class `LISTITEM` is the list item of our parameter list. It holds a value and a pointer to the next list item. The class `END` is used as an anchor to create an empty list. Usage of the list is as follows.

```
struct Algo {
  typedef LISTITEM<Type1> ParamType;
  Algo(ParamType Parameter);
};

template<typename Base> struct AspectA::public Base {
  typedef LISTITEM<Type2,Base::ParamType> ParamType;
  AspectA(ParamType Parameter):Base(Parameter.next)
  { do something with Parameter.value }
};
```

An aspect that needs to add a parameter can do this by attaching it to the parameter type `ParamType`. It can access the value of the additional parameter as `Parameter.value` and pass the rest of the list `Parameter.next` to its base class. The list that is given to the base class could be the parameter list of `Algo`, but it can also contain further arguments that are needed by other aspects. Since the parameter list `Base::ParamList` of the template argument `Base` is used, both cases are handled by this construction.

In order to nicely construct the parameter list, we need the class `LISTITEM` to provide generic constructors for lists with different lengths:

```
template<typename T, typename NEXT=END>
struct LISTITEM {
  T &value;
  NEXT next;

  LISTITEM(T &t):value(t),next(Nil){}
  template<typename T2> LISTITEM(T2 &t2, T &t):value(t), next(t2) {}
  template<typename T2, typename T3> LISTITEM(T3 &t3, T2 &t2, T &t):
      value(t), next(t3,t2) {}
  // ... and so on
};
```

The best way to show how simple it is to use a parameter list is to provide an example. Assume our basic algorithm `Dijkstra` needs the parameters `G` and `l`. Furthermore, we would like to use an aspect `PruningAspect`, which needs the parameter `L`, and an aspect `CountingAspect` without additional parameters. To realize such an algorithm, it is simply needed to provide the three parameters as the meta-list:

```
PruningAspect<CountingAspect<Dijkstra> >::ParamType P(G, l, L);
PruningAspect<CountingAspect<Dijkstra> > Algo(P);
```

For obvious reasons, it is required to know which aspects need what additional parameters. Furthermore, the respective parameters must be given in the same order as the aspects are added.

4.3 Dependencies

Another issue of the code organization is that some aspects depend on others. To give a concrete example, goal-directed search as well as geometric pruning both depend on a layout of the graph. Template meta-programming can also be used to check dependencies of such concepts [69]. In our case, concepts coincide most of the time with the use of a mix-in class (a derived class where the base class is a template parameter). This enables us to actually *add* the mix-in class in case it is needed but has not been included yet. (The aspect "layout" must only be added once if both goal-directed search and geometric pruning are used.)

In order to check a condition at compile-time, we make use of partial specialization as presented in [28]:

```
template<bool Cond, typename A, typename B>
struct IF { typedef A RET; };

template<typename A, typename B>
struct IF<false,A,B> { typedef B RET; };
```

The template class `IF` can be used to decide at compile time, which type `A` or `B` is used

depending on the condition `Cond`. The "return value" of the meta function `IF` is the type `RET`. Consider the definition

```
IF<(sizeof(int)<4), int, long>::RET a
```

If the size of `int` is 4, the general definition of `IF` is used. Therefore `a` is of type `int`. If the size of `int` is smaller than 4, the specialization of `IF` is used and `a` is of type `long`.

Our goal is to test at compile time whether the given base class `T` provides a certain aspect `Aspect` (e.g., whether the algorithm class stores a layout of the graph). Hence, what we need is to test whether the base class `T` is of the form `Aspect<B>` for some class `B`. Furthermore the base class `T` also provides the aspect `Aspect`, if it is *derived* from `Aspect<B>` (e.g., because some other aspect has been added after `Aspect`). Testing whether a class `T` is derived from a class `B` is a little bit tricky (see Chapter 2.7 in [27] for more details).

```
template<typename T, template<typename L> class Aspect>
struct Provides {
private:
  class Yes { char a[1]; };
  class No { char a[10]; };

  static No ProvidesTest( ... );
  template<typename S> static Yes ProvidesTest( Aspect<S> const* );

public:
  static bool const RET = (sizeof(ProvidesTest(static_cast<T*>(0)))
                           == sizeof(Yes));
};
```

The aspect to test is provided as a template-template argument (a template class as template argument) of the class `Provides`. The template function `ProvidesTest(Aspect<S> const*)` accepts a pointer to `Aspect<S>` for some class `S` or a pointer to a class that is derived from `Aspect<S>`. The function `ProvidesTest(...)` can take any pointer as argument. Both functions will actually never be called, so there is no need to define them. What is important about these functions is that they differ in their return type. If we call `ProvidesTest()` with a pointer to a class derived from `Aspect<S>`, the return type would be `Yes`. Otherwise it would be `No`. Even more important is that the sizes of their return types differ. Hence, we can distinguish by `sizeof(ProvidesTest(static_cast<T*>(0))` whether `T` is derived from `Aspect<S>`. The return value of the meta function `Provides` is stored in `RET`.

Using the template classes `IF` and `Provides`, we are now able to add to a base class an aspect if and only if it is not already included:

```
template<typename Base, template<typename L> class Aspect>
struct EnsureAspect {
  typedef typename IF<Provides<Base,Aspect>::RET, Base,
```

```
                              Aspect<Base> >::RET  RET;
};
```

Usage is as follows:

```
template<typename Base>
struct A:public EnsureAspect<Base,B>::RET {
  typedef typename EnsureAspect<Base,B>::RET MyBase;
  // ... further class members
};
```

The aspect `A` (e.g., geometric pruning) needs the base class to include aspect `B` (e.g., layout). If `Base` does not provide it, `A` is not derived from `Base` but from `B<Base>`. For our convenience, the type of the actual base class is remembered as `MyBase`.

For a concrete application we return to our main example. Geometric pruning `PruningAspect` and goal-directed search `GoalDirectedAspect` both need a layout `LayoutAspect`. If they ensure the usage of `LayoutAspect` as shown above, it is not necessary to include this aspect by hand:

```
PruningAspect<LayoutAspect<Dijkstra> >::ParamType P1(Graph, Lengths,
                                                     Layout);
PruningAspect<LayoutAspect<Dijkstra> > Algo1(P1);

PruningAspect<Dijkstra>::ParamType P2(Graph, Lengths, Layout);
PruningAspect<Dijkstra> Algo2(P2);
```

Both instantiations `Algo1` and `Algo2` result in the same algorithm. Furthermore, the aspects `PruningAspect` and `GoalDirectedAspect` can be combined as

```
PruningAspect<GoalDirectedAspect<Dijkstra> >::ParamType P3(Graph,
                                                           Lengths,
                                                           Layout);
PruningAspect<GoalDirectedAspect<Dijkstra> > Algo3(P3);
```

Again, it is necessary to know which parameters are needed and in which order according to the added aspects *including* their dependencies. However, instantiating an algorithm gets much shorter and clearer, since only the main aspects need to be mentioned.

Part II

Layouts

Chapter 5

Graph Drawing

In this chapter, we present the three methods to draw graphs that have been shown to produce good results even for large graphs: multi-level force-directed layout, eigenvectors of the Laplacian, and principal component analysis of a high-dimensional embedding. The main goal of the rest of this thesis it to draw graphs (i.e. generate layouts of graphs) in order to use them for geometric speed-up techniques. Since these speed-up techniques have been shown to be very efficient, it is therefore appreciable if they can be used in case the layout of the graph is (partially) unknown or not given by the application.

In this chapter, we assume that all graphs are undirected graphs (ignoring the direction of the edges), because preliminary results suggest that a drawing of the directed graph does not improve the quality of speed-up techniques. Furthermore, we assume that graphs are connected and loop-free, i.e. for all edges $(i, j) \in E$, $i \neq j$.

5.1 Force-Directed Layout

Eades presented in [31] the idea to draw a graph by simulating the edges of the graph as springs and the nodes of the graph as rings connecting these springs. The approach has been refined in [33, 34] by adding repelling forces to avoid small distances between nodes. We will introduce this algorithm differently than history by looking at barycentric layouts at first.

Let $L : V \to \mathbb{R}^2$ be a layout of a graph, then the potential function

$$P_B(p) = \sum_{\{u,v\} \in E} \omega(\{u, v\}) \cdot \|L(u) - L(v)\|^2 \tag{5.1}$$

where $\omega(e) = \frac{1}{l(e)}$ weights the influence of an edge according to its length $l(e)$, defines a weighted *barycentric layout model* [32]. This model has an interesting physical analogy, since each of the terms in (5.1) can be interpreted as the potential energy of a spring with spring constant $\omega(e)$ and ideal length zero.

A necessary condition for a local minimum of $P_B(p)$ is that all partial derivatives vanish.

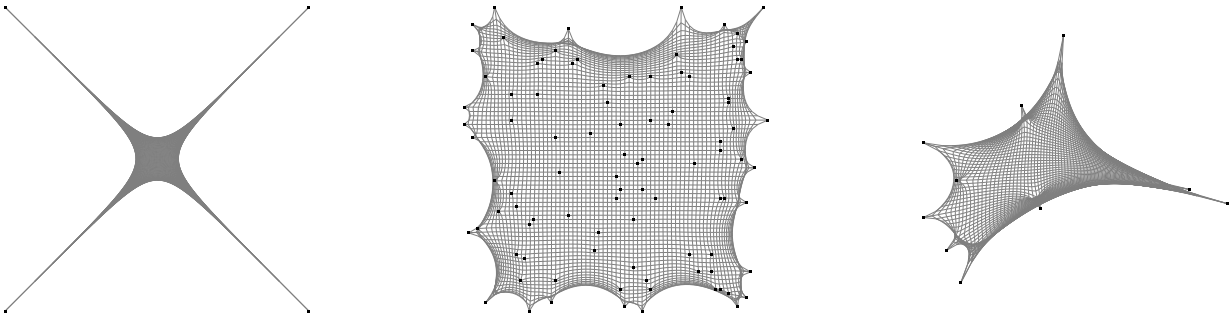


Figure 5.1: Barycentric layout of an 72×72 grid with the four corners fixed, and the same grid with 95 and with 10 randomly selected nodes fixed

That is, for all $L(v) = (x_v, y_v)$, $v \in V$, we have

$$x_v = \frac{\sum\limits_{u:\{u,v\}\in E} \omega(\{u,v\}) \cdot x_u}{\sum\limits_{u:\{u,v\}\in E} \omega(\{u,v\})} \qquad y_v = \frac{\sum\limits_{u:\{u,v\}\in E} \omega(\{u,v\}) \cdot y_u}{\sum\limits_{u:\{u,v\}\in E} \omega(\{u,v\})}.$$

In other words, each node must be positioned in the weighted barycenter of its neighbors. It is well known that this system of linear equations has a unique solution, if at least one $L(v)$ in each connected component of G is given (and the equations induced by v are omitted) [70]. Note that, in the physical analogy, this corresponds to fixing some of the points in the spring system. Moreover, the matrix corresponding to this system of equations is weakly diagonally dominant, so that iterative equation solvers can be used to approximate a solution quickly (see, e.g., [71]).

Assuming that the given set of node positions provide the cornerstones necessary to unfold the network appropriately, we can thus generate coordinates for the other nodes using, e.g., Gauss-Seidel iteration, i.e. by iteratively placing them at the weighted barycenter of their neighbors. Figure 5.1 indicates that this approach is highly dependent on the set of given positions. As is discussed in Sect. 7.4, it nevertheless has some practical merits.

The main drawbacks of the barycentric approach are that all nodes are positioned inside of the convex hull of the nodes with given positions, and that provided edge length are not preserved. We will now modify the potential (5.1) to take these edge lengths into account.

Kamada and Kawai [34] use springs of length $d(u, v)$, i.e. equal to the length of a shortest path between u and v, between every pair of nodes. The potential then becomes

$$P_{KK}(L) = \sum_{u,v\in V} \omega(u,v) \cdot \left(\|L(u) - L(v)\| - d(u,v)\right)^2, \tag{5.2}$$

the idea being that constituent edges of a shortest path in the graph should form a straight line in the drawing of the graph. To preserve local structure, spring constants are chosen as $\omega_e = \frac{1}{l(e)^2}$, so that long springs are more flexible than short ones. (The longer a path in the graph, the less likely are we able to represent it straight.) Note that this is a special

case of multidimensional scaling, where the input matrix contains all pairwise distances in the graph.

This model certainly does reflect our layout objectives more precisely. Note, however, that it is $\mathcal{NP}$-hard to determine whether a graph has an embedding with given edge lengths, even for planar graphs [72].

In contrast to the barycentric model, the necessary condition of vanishing partial derivatives leads to a system of non-linear equations, with dependencies between x- and y-coordinates. Therefore, we can no longer iteratively position nodes optimally with respect to the other temporarily fixed nodes, as in the barycentric model. As a substitute, a modified Newton-Raphson method can be used to approximate an optimal move for a single node [34, 73].

We compute a local minimum of a potential $P(L)$ by relocating one node at a time according to the forces acting on it, i.e. the negative of the gradient, $-\nabla P(L)$. For each node v (in arbitrary order) we move only this node in dependence of $P(L(v))$. The node is shifted in the opposite direction of $d := \nabla P(L(v))$. A substantial parameter of a gradient descent method is the size of each step. For small graphs, it is often sufficient to take a fixed multiple of the gradient (see the classic example of [31]), while others suggest some sort of step size reduction schedule (e.g., see [33]).

We applied a more elaborated method that is robust against change of scale, namely the method of Wolfe and Powell (see, e.g., [74]). The step size $\sigma \in (0, \infty)$ is determined by

$$\begin{aligned} \frac{\nabla P(L(v) - \sigma d)d}{\nabla P(L(v))d} &\leq \kappa \\ \frac{P(L(v)) - P(L(v) - \sigma d)}{\sigma \cdot \nabla P(L(v))d} &\geq \delta \end{aligned}$$

for given parameters $\delta \in (0, 0.5)$ and $\kappa \in (\delta, 1)$. Roughly speaking, this guarantees that the potential is reduced and that the step is not too small without any "fine-tuning" of a cooling schedule. We remember the last step length to initialize the next step length calculation, which speeds up the algorithm considerably. In our experiments, this method clearly outperformed the simpler methods both in terms of convergence and overall running time.

We also implemented the Newton-Raphson method, but it turned out to be an order of magnitude slower to achieve the same minima. This is mainly due to the fact that we worked in three dimensions, where it is necessary to invert a 3×3-matrix and to compute six instead of three second derivatives. The matrix inversion was performed by our own implementation as well as by LAPACK [75]. Both versions were not competitive with the method of Wolfe and Powell.

While these algorithms produce appealing drawings, their running time is unacceptable for large graphs. (In the graph drawing literature, similar objective functions have been subjected to simulated annealing [76, 77] and genetic algorithms [78, 79]. These methods seem to scale even worse.) Three groups discovered independently [35, 36, 37] how a force-directed layout can be generated efficiently with a multi-level approach. For our

implementation, we combined some of their methods: To create the next coarser graph in the multi-level hierarchy, edges of a maximal matching are contracted [36]. This works very well except for graphs that contain many star-like structures. (Then, only few edges are removed in the next coarser graph.) For such graphs, we therefore use an inclusion-maximal independent set as in [35].

5.2 Spectral Layout

This graph drawing method uses eigenvectors of a special matrix associated to a graph, the so-called Laplacian matrix. Spectral layouts have been introduced in [38], but can also be found in text books like, e.g., [80]. Their value for graph drawing has been renewed lately by [39], which introduces a multi-level approach to make this drawing technique feasible for large graphs. In this section, we will first provide some definitions and basic theorems concerning graphs and their matrices and then show how to use them to produce a layout for a graph. We will assume in this section that $V = \{1, \dots, n\}$ which is convenient to describe vectors and matrices associated to a graph $G = (V, E)$.

Definition 15 (Adjacency and Laplacian Matrix) *Given a graph $G = (V, E)$ with $V = \{1, \dots, n\}$, the* adjacency matrix *of G is the matrix $A \in \mathbb{R}^{n \times n}$ with $n = |V|$ and*

$$a_{ij} = \begin{cases} 1 & \text{if } \{i, j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

If G is weighted with $w : E \to \mathbb{R}_0^+$, the weighted adjacency matrix *is the matrix $A \in \mathbb{R}^{n \times n}$ with*

$$a_{ij} = \begin{cases} w(i, j) & \text{if } \{i, j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

The (weighted) Laplacian matrix *L of an undirected graph with (weighted) adjacency matrix $A \in \mathbb{R}^{n \times n}$ is defined as $D - A$, where D denotes the diagonal matrix containing the (weighted) degree of the corresponding nodes on the diagonal.*

Since a spectral layout uses eigenvectors of the Laplacian as coordinates, we first recall the definition of eigenvectors:

Definition 16 (Eigenvector and Eigenvalue) *Let $M \in \mathbb{C}^{n \times n}$ be a matrix. A vector $0 \neq v \in \mathbb{C}^n$ is called an* eigenvector *of M, if there exists an $\lambda \in \mathbb{C}$ such that*

$$Mv = \lambda v$$

The number $\lambda \in \mathbb{C}$ is called an eigenvalue *of M.*

By construction, the all 1 vector $\mathbb{1} = (1, \dots, 1)^t \in \mathbb{R}^n$ is an eigenvector of a Laplacian matrix with eigenvalue 0: The value on the diagonal is the negative sum of all other entries in this row.

The dimension of an eigenvector of L is n, the number of nodes in G. Therefore, one can interpret an eigenvector of L as one-dimensional coordinates. Similarly, two eigenvectors provide a two-dimensional drawing. Before we discuss which eigenvectors we use and how we can determine them, we will show that the eigenvectors and eigenvalues are real.

Definition 17 (Symmetric Matrix) *A matrix $M = (m_{ij}) \in \mathbb{C}^{n\times n}$ is called* symmetric*, if $m_{ij} = m_{ji}$ for all $i, j \in \{1, \ldots, n\}$, or $M^t = M$ for short.*

Obviously, the Laplacian matrix of a graph is symmetric by construction. This implies that its eigenvalues are real:

Lemma 18 *The eigenvalues of a real-valued, symmetric matrix are real, and the corresponding eigenvectors are real-valued.*

Proof: Let $M \in \mathbb{R}^{n\times n}$ be a symmetric matrix, $v \in \mathbb{C}^n$ an eigenvector for the eigenvalue $\lambda \in \mathbb{C}$. Then

$$\langle v, Mv\rangle = \langle v, \lambda v\rangle = \overline{\lambda}\langle v, v\rangle = \overline{\lambda}\|v\|^2$$

and

$$\langle v, Mv\rangle = v^tMv = v^tM^tv = (Mv)^tv = \langle Mv, v\rangle = \langle \lambda v, v\rangle = \lambda\langle v, v\rangle = \lambda\|v\|^2$$

Since eigenvectors are by definition not zero, we get $\lambda = \overline{\lambda}$. The eigenvectors of M are then in the kernel of $M - \lambda I$, which is a real subspace, since $\lambda \in \mathbb{R}$. This concludes the proof. □

We have now seen that, given a graph G, the eigenvectors of its Laplacian matrix are in $\mathbb{R}^n$. The reason for using eigenvectors $x \in \mathbb{R}^n$ of the Laplacian matrix for graph layout, in particular those associated with small eigenvalues, is the following. The *energy* or *stress* resulting from x is the value of the quadratic form

$$\varepsilon(x) := \frac{x^tLx}{x^tx} = \frac{1}{2\|x\|}\sum_{\{u,v\}\in E} w(\{i,j\})\cdot(x_i - x_j)^2$$

where $w(\{i,j\})$ is the weight of the edge $\{i,j\}$, if L is weighted, and 1, otherwise. Minimizing $\varepsilon(x)$ brings connected nodes close together, which is a common criteria for "nice" drawings of a graph. Without loss of generality, we may assume that x is normalized, so $\varepsilon(x) = x^tLx$. Vectors minimizing $\varepsilon(x)$ with $\|x\| = 1$ are eigenvectors of L as shown in the following theorem.

Theorem 19 *Given a symmetric matrix $M \in \mathbb{R}^n$, the extremal of x^tMx under the restriction $\|x\| = 1$ is an eigenvalue of M.*

Proof: The Lagrangian of the maximization problem is

$$L(x, \lambda) = x^tMx - \lambda(x^tx - 1)$$

The partial derivatives are

$$\begin{aligned}\frac{\partial}{\partial x}L(x,\lambda) &= 2Mx - 2\lambda x \\ \frac{\partial}{\partial \lambda}L(x,\lambda) &= 1 - x^t x\end{aligned}$$

The necessary conditions are therefore $Mx = \lambda x$ and $\|x\| = 1$. An extremal value is therefore an eigenvalue of M as

$$x^t M x = x^t \lambda x = \lambda x^t x = \lambda \|x\| = \lambda$$

□

The all 1 vector $\mathbb{1} \in \mathbb{R}^n$ is the eigenvector with the smallest eigenvalue 0. As this eigenvector is useless for a drawing, orthogonal eigenvectors x and y for the smallest eigenvalues greater than zero are used to create the layout.

Lemma 20 *For a symmetric matrix, eigenvectors with distinct eigenvalues are orthogonal.*

Proof: Let $M \in \mathbb{C}^{n\times n}$ be a symmetric matrix, $v_1, v_2 \in \mathbb{C}^n$ two eigenvectors with eigenvalues $\lambda_1, \lambda_2 \in \mathbb{R}$ where $\lambda_1 \neq \lambda_2$. Then

$$\lambda_1\langle v_1, v_2\rangle = \langle \lambda_1 v_1, v_2\rangle = \langle Mv_1, v_2\rangle = \langle v_1, Mv_2\rangle = \langle v_1, \lambda_2 v_2\rangle = \lambda_2\langle v_1, v_2\rangle$$

If $\langle v_1, v_2\rangle \neq 0$, then $\lambda_1 = \lambda_2$ which contradicts $\lambda_1 \neq \lambda_2$. □

The non-trivial eigenvectors of L are orthogonal to the trivial minimizer $\mathbb{1}$, and thus centered around the origin. Their resulting stress is the associated eigenvalue of L. Therefore, pairwise orthogonal eigenvectors associated with the smallest non-zero eigenvalues yield layouts of minimum stress that are balanced (i.e. centered around the origin).

The eigenvalues can be computed efficiently with power iteration of the matrix $2\Delta I - L$, where Δ denotes the maximum degree of a node in V and $I \in \mathbb{R}^{n\times n}$ is the identity matrix. The largest eigenvalue of L is bounded by twice the maximum degree Δ in G. A vector $x \in \mathbb{R}^n$ is an eigenvector of L with eigenvalue λ iff x is an eigenvector of $2\Delta I - L$ with eigenvalue $2\Delta - \lambda$. Thus, $2\Delta I - L$ has the same eigenvectors as L in reversed order of their (positive) eigenvalues. For a sparse graph, the matrices L and $2\Delta I - L$ are sparse as well, so the matrix multiplication can be implemented efficiently using adjacency lists.

To obtain the second largest eigenvalue, one applies the power method to the matrix $M_1 := M - \lambda_1 x_1 x_1^t/\|x\|^2$ as proposed in [81]. Except for x_1, the matrix M_1 has the same eigenvalues as M: Given an eigenvector x_i of M with eigenvalue $\lambda_i \neq \lambda_1$, it holds that

$$M_1 x_i = (M - \lambda_1 x_1 x_1^t/\|x\|^2)x_i = Mx_i - \lambda_1/\|x\|^2 x_1 \underbrace{x_1^t x_i}_{0} = \lambda_i x_i$$

as eigenvectors with distinct eigenvalues are orthogonal (Lemma 20).

5.3 High-Dimensional Embedding

The key idea of this method [40] is to draw the graph in a very high-dimensional space (usually $d = 50$) and then project the graph to 2D. The projection is realized using principal component analysis (PCA), which maximizes the variance of the projected data.

5.3.1 Drawing in $\mathbb{R}^d$

In order to draw the graph in $\mathbb{R}^d$, a node $s \in V$ is selected for each of the d dimensions. For this dimension, the coordinate of a node $v \in V$ is then defined as the distance of v from s, i.e. the length of a shortest s-v path in G. As our graphs are weighted, the shortest paths were calculated using these edge weights. (The breadth first search in [40] is simply replaced by Dijkstra's algorithm.) Using these distances as coordinates provides a layout in $\mathbb{R}^d$.

This method can be illustrated as if we had d different "pictures" of the graph. For each picture, we selected a different position in the graph and take a (one-dimensional) picture that sorts the other nodes according to their distances. If we have pictures from a lot of different angles, we should get a reasonable overview of the graph.

A good choice of positions should gather as many different aspects of the graph as possible. It is therefore common to select the positions in a greedy-like fashion that maximize the sum of distances to the other selected nodes.

5.3.2 Projecting to $\mathbb{R}^2$

In order to project the graph into a plane, *principal component analysis (PCA)* is used. The projected coordinates are a linear combination of d-dimensional coordinates $x_1, \ldots, x_d \in \mathbb{R}^n$. PCA determines the linear combination $\sum_{i=1}^{d} u_i x_i$ of coordinates with $\sum_{i=1}^{d} u_i^2$ that produces the largest variance. For doing so, we need the notion of *covariance*:

Definition 21 (Covariance) *Given two n-dimensional observations $x_1, x_2 \in \mathbb{R}^n$, the* (empirical) covariance *of x_1 and x_2 is defined as*

$$s_{12} := \frac{1}{n-1} \sum_{k=1}^{n} (x_{k1} - \overline{x}_1)(x_{k2} - \overline{x}_2)$$

where $\overline{x}_1$ denotes the mean value $\frac{1}{n}\sum_{k=1}^{n} x_{k1}$ and $\overline{x}_2$ denotes the mean value $\frac{1}{n}\sum_{k=1}^{n} x_{k2}$. Given d n-dimensional observations $x_1, \ldots, x_d \in \mathbb{R}^n$, the covariance matrix *is defined as*

$$S = \begin{pmatrix} s_{11} & \cdots & s_{1d} \\ \vdots & \ddots & \vdots \\ s_{d1} & \cdots & s_{dd} \end{pmatrix}$$

Observe that the covariance s_{ii} of a vector $x_i \in \mathbb{R}^n$ with itself is the variance s_i^2 (Definition 6), so the entries on the diagonal of the covariance matrix are the variances $s_1^2, \ldots, s_d^2$.

Let us look at the definition of the covariance matrix from the point of view of our d-dimensional layout. At first, the layout is balanced by a translation of the values x_i in every dimension $i = 1, \ldots, d$, such that $\sum_{k=1}^{n} x_{ki} = 0$. Then, the covariance matrix $S \in \mathbb{R}^d$ is computed as $s_{ij} = \frac{1}{n-1}\langle x_i, x_j \rangle$. For ease of notation, we will therefore assume in the rest of this section that the d-dimensional layout is balanced, i.e. $\sum_{k=1}^{n} x_{ki} = 0$ for all $i = 1, \ldots, d$. If the layout is balanced in $\mathbb{R}^d$, then every linear combinations of the coordinates is balanced, too:

$$\sum_{k=1}^{n}\sum_{i=1}^{d} u_i x_{ik} = \sum_{i=1}^{d} u_i \underbrace{\sum_{k=1}^{n} x_{ik}}_{0} = 0$$

for all $u = (u_1, \ldots, u_k)^t \in \mathbb{R}^d$. The variance of a linear combination $\sum_{i=1}^{d} u_i x_i$ can therefore be computed as

$$\frac{1}{n-1}\langle \sum_{i=1}^{d} u_i x_i, \sum_{j=1}^{d} u_j x_j \rangle = \frac{1}{n-1}\sum_{i=1}^{d}\sum_{j=1}^{d} u_i u_j \langle x_i, x_j \rangle = \frac{1}{n-1}\sum_{i=1}^{d}\sum_{j=1}^{d} u_i s_{ij} u_j = \frac{1}{n-1}\langle u, Su \rangle$$

Therefore, the coefficients $(u_1, \ldots, u_d)^t \in \mathbb{R}^d$ with $\|u\| = 0$ for a linear combination $\sum_{i=1}^{d} u_i x_i$ of coordinates with maximal variance maximize $\langle u, Su \rangle$ under the restriction that $\|u\| = 0$. According to Theorem 19, u must be an eigenvector of S for the largest eigenvalue. The eigenvector v of S for the second largest eigenvalue provides the second coordinate $\sum_{i=1}^{d} v_i x_i$. Both can be determined efficiently using power iteration of S as presented in Section 5.2.

Chapter 6

Example: Visualization of Bibliographic Networks

Bibliographic analysis [82] uses publication data to structure and summarize a scientific field. This data is often given in the form of *bibliographic networks*, with nodes representing authors, journals, or publications, and edges representing relations between these entities such as authorship, collaboration, or citation.

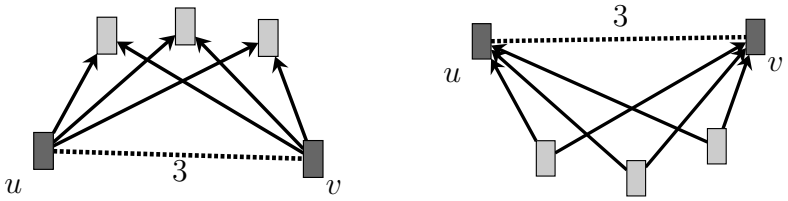
As a concrete example for the graph drawing algorithms in the last chapter, we provide a novel approach to visualize bibliographic networks that facilitates the identification of clusters (e.g., topic areas) and prominent entities (e.g., surveys or landmark papers) at the same time. More precisely, the bibliographic network is embedded in a landscape where nodes are represented by buildings. (This work has been published previously in [83].)

While employing the landscape metaphor has been proposed in several earlier publications, we introduce new means to define and compute the relevant parameters of the landscape. A noteworthy aspect of our method is that determination of prominent entities, the spatial representation of the clustering of the network, and the elevation of the landscape's surface are carried out in a surprisingly uniform and simple way. From a technical point of view, it combines a spectral layout (Sect. 5.1) of the (modified) network with a restricted force-directed layout (Sect. 5.2) of a refined graph. The effectiveness of our network visualizations is illustrated on a data set from the graph drawing literature (Sect. 6.4).

Since we propose an integrated method of analysis and visualization directed at particular aspects of bibliographic analysis, it may serve as a specialized component in more elaborate systems such as [84, 85, 86], and in particular as a communication back-end for systems that specialize in extracting and presenting network data [87, 88].

6.1 Landmark Papers

To identify prominent entities in bibliographic networks, we determine the structural importance of nodes according to their position in the graph. Many concepts formalizing this



(a) bibliographic coupling (b) co-citation

Figure 6.1: Two operators producing weighted undirected graphs that capture the essence of an analytic perspective

notion are in use, but the concept of hubs and authorities [89], though originally conceived to improve relevance ranking in Web search engines, appears to be particularly suitable for bibliographic networks. In this subsection, we present a slightly different derivation of these indices to emphasize the similarity to the spectral layout that we use to position the nodes.

A straightforward notion of prominence in an undirected graph, commonly applied in the analysis of social networks [90], is the idea that the importance of a node is determined by the importance of its neighbors. According to the following definition, the importance assigned to a node is proportional to the total importance of its neighbors.

Definition 22 (Eigenvector Centrality [91]) *Let A be the adjacency matrix of a connected undirected graph $G = (V, E)$. The* eigenvector centrality*, $c(G) = (c_v)_{v \in V}$, is the (unique) eigenvector for the largest eigenvalue of A subject to $c_v > 0$ for all $v \in V$ and $\sum_{v \in V} a_v = 1$.*

To simplify the presentation, we restrict ourselves to the analysis of (connected) citation networks with respect to landmark publications. We thus consider directed graphs $G = (V, E)$, in which each node $v \in V$ represents a publication, and directed edges $(u, v) \in E$ that represent a citation of v in u. With straightforward modifications, our methods can be applied to other types of bibliographic networks and other types of analyses (e.g., at surveys, prominent authors, or journals with high impact).

We now define two operators to transform an unweighted directed graph into a weighted undirected graph. The two operators model two different structural aspects in the directed graph that can be extracted in the transformed graphs by a centrality analysis. See Fig. 6.1 for an illustration of the operators.

Definition 23 (Bibliographic Coupling [92] and Co-Citation [93]) *Let $G = (V, E)$ be a directed graph with adjacency matrix A. The weighted undirected graphs $\mathcal{B}(G)$ and $\mathcal{C}(G)$ induced by adjacency matrices $B = AA^t$ and $C = A^tA$ are called the* bibliographic coupling *and* co-citation *graph, respectively.*

It is interesting to note that bibliographic coupling of a bipartite graph in which nodes represent authors or publications, with edges from authors to their publications, yields a collaboration graph.

The bibliographic coupling and co-citation graphs can then be used to formally define the notions of hubs and authorities. Designed to increase the effectiveness of Web search engines, hubs and authorities are formal notions of structural prominence of nodes in directed graphs. Intuitively, a Web page is considered a hub, if it links to many authorities, and a resource is an authority, if many hubs link to it. The implicit assumptions about the meaning of a link are generally the same as the ones made for citations. In fact, the Web can be considered to be the largest citation network there is.

Definition 24 (Hubs and Authorities [89]) *For a connected directed graph $G = (V, E)$, let B and C denote the adjacency matrices of $\mathcal{B}(G)$ and $\mathcal{C}(G)$, respectively. The* hub index*, $h(G) = (h_v)_{v \in V}$, and the* authority index*, $a(G) = (a_v)_{v \in V}$, are defined as the eigenvector centrality of B and C, respectively.*

Starting from $a^{(1)} \leftarrow \frac{1}{n} \cdot \mathbf{1}$, the following interleaved version of power iteration is used to compute the indices without explicitly constructing the undirected graphs.

$$\begin{aligned} h^{(k)} &\leftarrow A \cdot a^{(k)}; & h^{(k)} &\leftarrow h^{(k)} / \|h^{(k)}\| \\ a^{(k+1)} &\leftarrow A^t \cdot h^{(k)}; & a^{(k+1)} &\leftarrow a^{(k+1)} / \|a^{(k+1)}\|. \end{aligned}$$

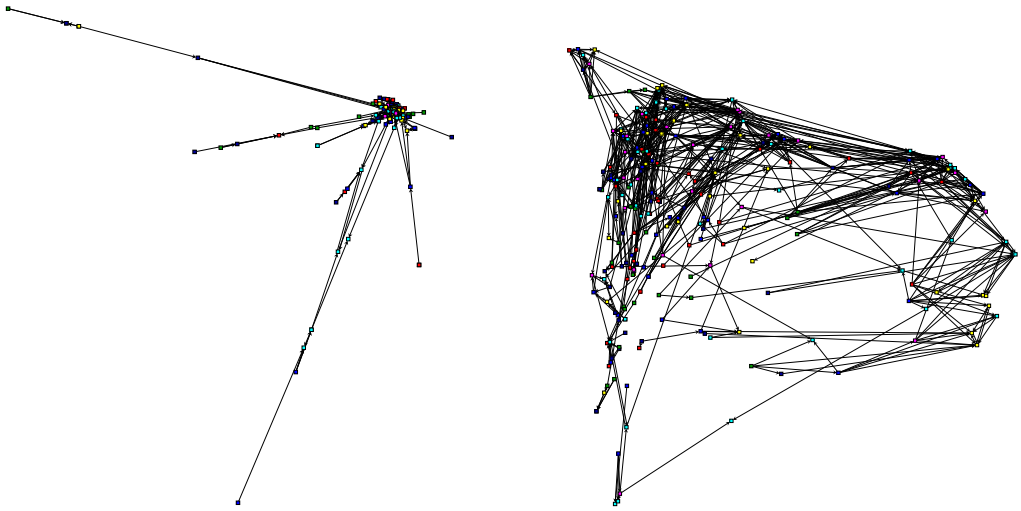
While the speed of convergence depends on the ratio between the largest and second-largest eigenvalue, convergence is usually rapid and we use stabilization of the eigenvalue approximation as our stopping criterion. Since bibliographic networks tend to be very sparse with $m \in \mathcal{O}(n)$, each iteration takes linear time.

6.2 Topics

In this section, we describe a method to compute a two-dimensional layout of a bibliographic network that represents thematic clusters geometrically. Since the layout algorithm is a spectral layout (see Sect. 5.2) of a transformed graph, it is technically very similar to the iterative computation of a prominence vector in the previous subsection.

The prominence analysis carried out in the previous subsection is based on an undirected graph in which weighted edges correspond to the extend of bibliographic coupling (hubs) or co-citation (authorities). Weights thus reflect similarity of entities with respect to the analytic perspective taken. However, if a pair of nodes in a directed graph G is connected by just a single edge, they are adjacent in neither $\mathcal{B}(G)$ nor $\mathcal{C}(G)$. To incorporate similarity implicit in directed linkages, our definition of similarity contains an additional unit weight for each directed edge.

Definition 25 (Similarity Graphs) *Let $G = (V, E)$ be a directed graph with adjacency matrix A. The weighted undirected graphs $\mathcal{S}_\mathcal{B}(G)$ and $\mathcal{S}_\mathcal{C}(G)$ induced by adjacency matri-*



(a) Laplacian (b) modified Laplacian ($\rho = 0.25$)

Figure 6.2: Co-citation similarity layout of a citation network. See Sect. 6.4 for details on the data and note that the layout is not primarily determined by the citation graph, but the similarity of citation patterns

ces $S_B = AA^t + A + A^t$ *and* $S_C = A^tA + A + A^t$ *are called the* similarity graphs *with respect to bibliographic coupling and co-citation, respectively.*

Similarity graphs may be clustered geometrically using standard methods such as multi-dimensional scaling or force-directed graph layout algorithms. However, for reasons stated more elaborately in [86], a spectral layout (see Sect. 5.2) is more appropriate for our application.

Spectral layout refers to the use of eigenvectors of the Laplacian matrix $L = D - A$, where A denotes the (weighted) adjacency matrix of the graph and Δ is the diagonal matrix containing the (weighted) degree of the corresponding nodes on the diagonal. If the underlying graph is not "round-shaped" (roughly, if the second-smallest eigenvalue is not large enough), Laplacian layouts yield in a clustering which is too dense to be useful for visualization. This defect is also noted in [86], where it is suggested to use the Laplacian layout to initialize a force-directed layout algorithm which significantly increases the running time. However, we propose to modify the Laplacian matrix by introducing a relaxation factor $0 \leq \rho \leq 1$. The matrix $L_\rho = (1 - \rho) \cdot D - A$ compromises between the Laplacian and the adjacency matrix and thus avoids excessive displacement of loosely connected nodes. Figure 6.2 illustrates the effect.

To be able to compute eigenvectors of L_ρ with the same simple power iteration used for hubs and authorities, we reverse the order of it eigenvalues and repeatedly orthogonalize with $\mathbf{1}$. Moreover, because of the potential loss of sparsity, we do not construct the

similarity graphs explicitly.

To compute a similarity clustering with respect to, say, co-citation, let A be the adjacency matrix of a directed graph G with n nodes, $D_{\mathcal{S}_\mathcal{C}(G)}$ the diagonal weighted degree matrix of $\mathcal{S}_\mathcal{C}(G)$, and Δ the maximum weighted degree of $\mathcal{S}_\mathcal{C}(G)$.

$$\begin{aligned}
x^{(k+1)} &\leftarrow A \cdot x^{(k)} \\
x^{(k+1)} &\leftarrow A^t \cdot x^{(k+1)} + (A + A^t) \cdot x^{(k)} \\
x^{(k+1)} &\leftarrow x^{(k+1)} + (2\Delta \cdot I - (1-\rho) \cdot D_{\mathcal{S}_\mathcal{C}(G)}) \cdot x^{(k)} \\
x^{(k+1)} &\leftarrow x^{(k+1)} - \frac{1}{n} \sum_{v \in V} x_v^{(k+1)} \\
x^{(k+1)} &\leftarrow x^{(k+1)} / \|x^{(k+1)}\|
\end{aligned}$$

A second dimension, y, is computed in much the same way, except that we orthogonalize with the first dimension by computing

$$y^{(k+1)} \leftarrow y^{(k+1)} - \frac{x^t \cdot y^{(k+1)}}{x^t \cdot x} x$$

at the end of each iteration. Again, we are requiring only sparse matrix-vector and vector-vector multiplication, so that each iteration needs linear time and space.

6.3 Scientific Landscapes

The landscape metaphor is popular for visualizing bibliographic networks [85, 86, 94], but in general the landscape is produced simply by overlaying a triangulated grid, where grid points are elevated according to the density of data points in their vicinity. The shape of the landscape thus conveys only one aspect in the network's analysis, i.e. clustering.

We define the shape of the landscape so as to display both clustering and prominence in the same visualization and to represent the underlying network layout more accurately. Intuitively, we simplify a three-dimensional drawing of the network in which two dimensions represent similarity between entities, while the third is determined by a prominence index by placing a table cloth over it. It almost goes without saying that this table-cloth is positioned with yet another variation of a graph-drawing algorithm.

Assume we are given a connected undirected graph $G = (V, E)$ with n nodes and m edges together with a three-dimensional layout (x, y, z), in which each $v \in V$ is associated with a point $(x_v, y_v, z_v) \in \mathbb{R}^3$. In our application, x- and y-coordinates are the entries of eigenvectors of the modified Laplacian matrix of G, and z-coordinates are eigenvector centralities in G, i.e. $z = c(G)$, but a landscape could be generated in much the same way from any other three-dimensional layout as well.

We want to cover the layout from the top (z-direction) with a smooth surface to resemble a landscape in which elevations correspond to prominent entities. We therefore first generate a point set in the xy-plane, triangulate it, and finally compute z-coordinates for all points using this triangulation and the prominence of nodes.

The set of points defining the shape of the landscape is generated as follows. Consider the two-dimensional straight-line drawing of G defined by (x, y), and add $\Omega(\sqrt{n})$ equidistant horizontal and vertical lines each to the drawing. The set of points P that defines the landscape consists of all nodes of G and all intersections between edges, grid lines, or edges and grid lines. Note that $|P| \in \mathcal{O}(m^2)$.

Next, a Delaunay triangulation of P is computed. The resulting triangles will be used to render the surface, but first we have to determine z-coordinates for all $p \in P$ such that the surface covers the three-dimensional graph layout like a table cloth.

Ideally, points created from nodes of the graph are placed at the same z-coordinate as the node. On the other hand, for the surface to be smooth, points that are close in the xy-plane should also be close in z-direction. Thus consider the objective function

$$\sum_{p,q \in P} \omega_{pq} \cdot \|z_p - z_q\|^2$$

where ω_{pq} is a non-negative weight measuring the influence of q on p, which will depend on the relative distance between them. We set $\omega_{pq} = 0$, if $p = q$ or p and q are not adjacent in the Delaunay triangulation. Inspired by recent work on terrain modeling [95, 96], we compute the remaining influence weights as in Sibson's interpolant [97], i.e. by temporarily removing p from the Voronoi diagram and setting ω_{pq} to the share of p's Voronoi cell that its Delaunay neighbor acquires through p's removal.

Minimization of the above objective function is straightforward. It is also the quadratic form associated with a Laplacian matrix, but this time of the graph of the Delaunay triangulation with Sibson weights. Moreover, since the surface should cover the three-dimensional shape of the network, there are natural candidates for the z-coordinates of points stemming from a node or the intersection of an edge. Since close nodes may have very different prominence scores, we fix the z-coordinate of points that are locally maximal (i.e. whose Delaunay neighbors have smaller natural candidates) and of points on the convex hull (at ground level). Subject to these constraints, the remaining coordinates are determine so as to minimize the above objective.

Since some points are already fixed, minimization amounts to placing all other points in the weighted barycenter of their neighbors. The resulting system of linear equations has a unique solution [32], which can quickly be approximated using an iterative equation solver. Let F be the edges of the Delaunay triangulation, then we simply iterate

$$z_p^{(k+1)} \leftarrow \sum_{q\,:\,\{p,q\} \in F} \frac{\omega_{pq}}{\sum_{q'\,:\,\{p,q'\} \in F} \omega_{p,q'}} z_q^{(k)}$$

for each $p \in P$ whose coordinate has not been fixed. These are once again sparse matrix computations, and since the matrix is weakly diagonally dominant, convergence is rapid.

6.4 Results and Discussion

As a proof of concept, we have implemented our approach in C++ using the Library of Efficient Data Types and Algorithms (LEDA) [44] and OpenGL. The data set used in our examples is taken from the 2001 Graph Drawing Contest [98] and consists of all papers published in proceedings of Graph Drawing Symposia 1994–2000, complete with their mutual citations. The largest connected component is formed by 249 papers and 642 citations. (Note that this data cannot form the basis for valid conclusions about the relative importance of papers in the field of graph drawing as such. It was chosen because we are most familiar with the document corpus and could therefore evaluate much better the adequateness of our visualizations relative to the data set.)

Using our reshaped landscape metaphor, the citation network suggests several hypotheses about the nature of citations in the area of graph drawing that are readily confirmed by inspection of the underlying data. Peaks indeed indicate authoritative papers, and villages correspond to popular themes in graph drawing.

Improved graphical design (e.g., richer glyphs), more sophisticated rendering (e.g., increased realism), and powerful means of user interaction (e.g., levels of detail) would certainly be worthwhile considering for an actual system, but are beyond the scope of our work. The landscape visualization might further be extended by introducing topical area boundaries (based on the implicit surface techniques of [99]) or citation tracks (based on main path analysis [100]).

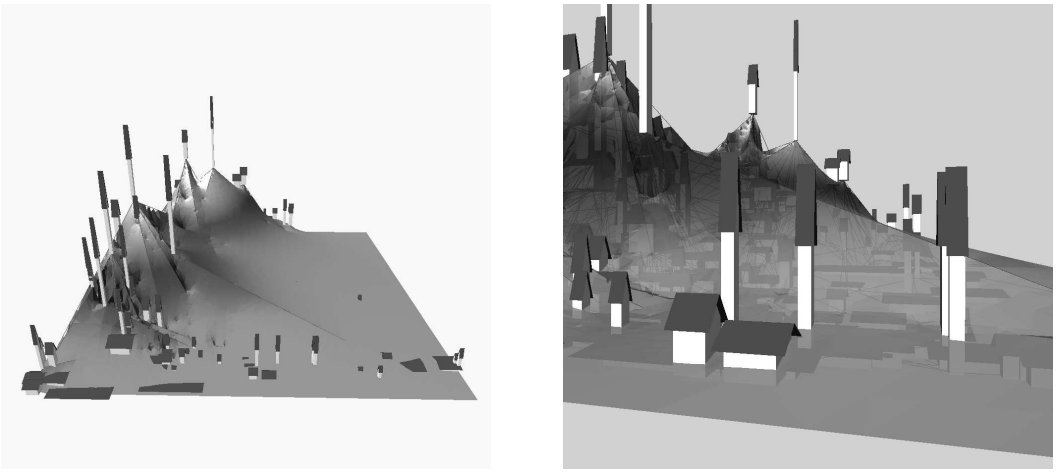


Figure 6.3: Simultaneous visualization of prominence (authority) and clustering (co-citation similarity) – overview and zoomed in to the "village" of the graph drawing contest (with partially transparent table cloth)

Chapter 7

Generating Layouts for Travel Planning Systems

In travel-planning systems for public transport [6, 5, 4], shortest-path computations form a basis for answering connection queries. They often make use of geometric speed-up techniques for shortest-path computations as presented in Chapters 2 and 3. Finding the best itinerary using schedules vehicles can thereby be modeled as a shortest-path query in the graph where the nodes are departure and arrival events and an edge represents a train connection or a waiting time at a station.

The problem we consider in this chapter has been posed by an industrial partner[1] who is a leading provider of travel-planning services for public transportation. They are faced with the fact that quite often, much of the underlying geography, i.e. the location of nodes in a network, is unknown, since not all transport authorities provide this information to travel service providers or competitors. This is particularly true if the provider of the travel information system differs from the transportation company, e.g. for foreign trains or local bus companies. The situation will become even more serious in future with the liberalization of the European railway network and the integration of local buses in door-to-door queries. Furthermore, producers of a time-table information system need to demonstrate their travel-planning software to new customers who usually do not have coordinates in a ready-to-use state. Since the reduction in query response time is important, other ways to make the successful geometric speed-up heuristics applicable are sought.

The existing, yet unknown, underlying geography is reflected in part by travel times, which in turn are given in the form of timetables. Therefore, we can construct a simple undirected weighted graph in the following way. Each station represents a node, and two nodes are adjacent if there is a non-stop connection between the two corresponding stations. Edge weights are determined from travel times, thus representing our distance estimates. Reasonable (relative) location estimates are then obtained by embedding this graph in the plane such that edge lengths are approximately preserved. Parts of these results have been published previously in [101]. This problem is closely related to drawing

[1] HaCon Ingenieurgesellschaft mbh, Hannover.

Internet latency maps and positioning algorithms for wireless ad-hoc networks. However, it is *not* our goal to reconstruct the original coordinates, but to produce coordinates with which the speed-up techniques perform well.

In the next section, we describe precisely how we model the graph from a set of timetables that we use to generate the geographic information. In Section 7.2, we tailor a specific force-directed layout algorithm for this graph and application. This algorithm is experimentally evaluated on timetables from the German public train network using a snapshot of half a million connection queries. Section 7.3 describes the experiments that we performed and Section 7.4 their results.

7.1 Estimating Distances from Travel Times

If the actual geographic locations of stations in a travel network are not provided, the only related information available from the timetables are travel times. We use them to estimate distances between stations that have a non-stop connection, which in turn are used to generate locations suitable for the geometric speed-up techniques, though in general far from being geographically accurate.

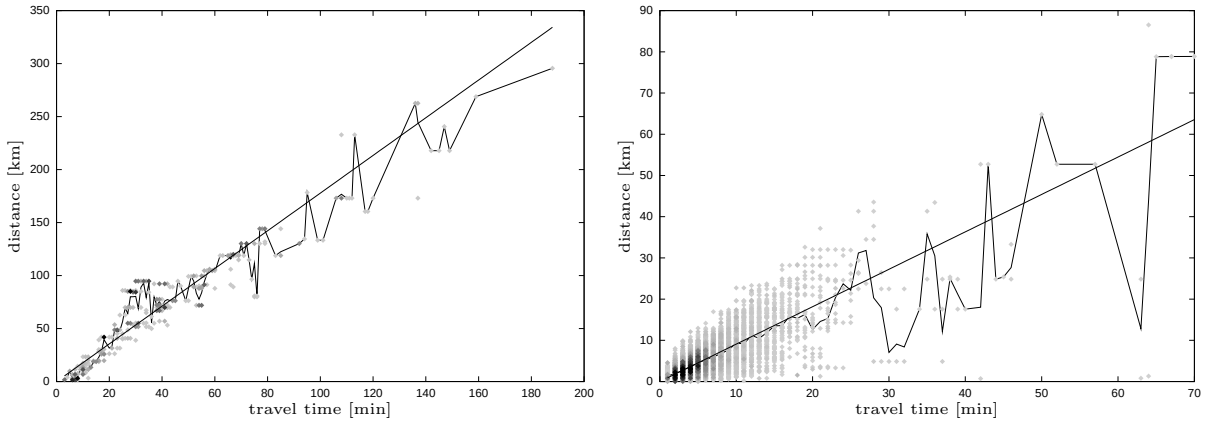
The (undirected, simple) *connection graph* of a set of timetables contains a node for each station listed, and an edge between every pair of stations connected by a train not stopping in between. The *length* $l(e)$ of an edge e in the station graph will represent our estimate of the distance between its endpoints.

The distance between two stations can be expected to be roughly linear in the travel time. However, for different classes of trains the constant involved will be different, and closely related to the mean velocity of trains in this class. We therefore estimate the length of an edge e in the station graph, i.e. the distance between two stations as follows: Consider all non-stop connections that induce this edge. We use as estimated distance the mean value of their travel time times the average velocity of the vehicle serving the connection.

Mean velocities have been extracted from the data set described in Sect. 7.3, for which station coordinates are known. For two train categories, the data are depicted in Fig. 7.1, indicating that linear approximations yield a fairly good estimation.

Note that all travel times are integers, since they are computed from arrival and departure times. As a consequence, slow trains are often estimated to have unrealistically high maximum velocities, thus affecting the modified edge lengths in the goal-directed search heuristic.

Apart from the estimation of geographic distances, a second set of edge lengths has been tested: a distance that is proportional to the average travel time of all vehicles including this edge. (It is the same as assuming that all trains have the same average velocity.) This approach can be justified by the insight that the travel-planning system minimizes only the travel time.



(a) high-speed trains

(b) local trains

Figure 7.1: Euclidean distance vs. travel time for non-stop connections. For both service categories, all data points are shown along with the average distance per travel time and a linear interpolation

7.2 A Specific Layout Model for Connection Graphs

In our particular application, it may occasionally be the case that the geographic locations of at least some of the major hubs of the network are known, or can be obtained easily. We will therefore first examine barycentric layouts (see Sect. 5.1) exploiting the fact that such hubs are typically well-distributed and thus form a scaffold for the overall network. Our approach for the more general case, described in the rest of this section, is a derivation of a multi-level force-directed layout and can therefore be viewed as an extension of a barycentric layout, too.

7.2.1 Sparsening.

We use the algorithm of Kamada and Kawai from the Sect. 5.1 as a starting point for our specific layout model. If springs are introduced between every pair of nodes, a single iteration takes time quadratic in the number of nodes. Since at least a linear number of iterations is needed, this is clearly not feasible. Since, moreover, we are not interested in a readable layout of the graph, but in supporting the geometric speed-up heuristics for shortest-path computations, there is no need to introduce springs between all pairs of nodes.

We cannot omit springs corresponding to edges, but in connection graphs, the number of edges is of the order of the number of nodes, so most of the pairs in (5.2) are connected by a shortest path with at least two edges. If a train runs along a path of k edges, we call this path a *k-connection*. To model the plausible assumption that, locally, trains run fairly straight, we include only terms corresponding to edges (or 1-connections) and to 2- and

3-connections into the potential. Whenever there are two or more springs for a single pair of nodes, they are replaced by a single spring of the average length. For realistic data, the total number of springs thus introduced is linear in the number of nodes.

7.2.2 Long-Range Dependencies.

Since we omit most of the long-range dependencies (i.e. springs connecting distant pairs of nodes), an iterative method starting from a random layout is almost surely trapped in a poor local minimum.

We therefore determine an initial layout by computing a local minimum of the potential on an even sparser graph that includes only the long-range dependencies relevant for our approach. In other words, we apply a (fairly special) multi-level variant of a force-directed layout. More precisely, we consider the subgraph consisting of all stations that have a fixed position or are a terminal station of some train, and introduce springs only between the two terminal stations of each train, and between pairs of the selected nodes that are consecutive on the path of any train. We refer to these additional pairs as *long-range connections*. In case the resulting graph has more components than the connection graph, we heuristically add some stations touched by trains inducing different components and the respective springs. After running our layout algorithm on this graph (initialized with a barycentric layout), the initial position for all other nodes is determined from a barycentric layout in which those positions that have already been computed are fixed.

To measure the impact of this modification, we computed 100 layouts with and 100 without it and compared the average of the final potential values. It turns out that the result of the algorithm using this initial layout has a 9% smaller value of the objective function.

7.2.3 Nodes of High Degree.

The method to find an initial layout of the sparser graph in the previous section can be seen as a two-level approach of the embedding algorithm: The first level is the full graph, the second level is the graph of long-range connections, which is embedded first. Although the second level already improves the final potential function, in case no station at all has a fixed position, the iterative method is still likely to be trapped in a local minimum. The introduction of a third, even smaller graph therefore leads to another improvement of the potential function of 21%. (In case of 22 given coordinates for major hubs, the improvement is only 0.1%, however.)

To determine the third level, we use again the structure of the graph: We iteratively replace all nodes of degree three or less in the second level by edges between their neighbors. More precisely, nodes

- of degree 0 or 1 are removed
- of degree 2 are replaced by an edge with length equal to the sum of the lengths of incident edges

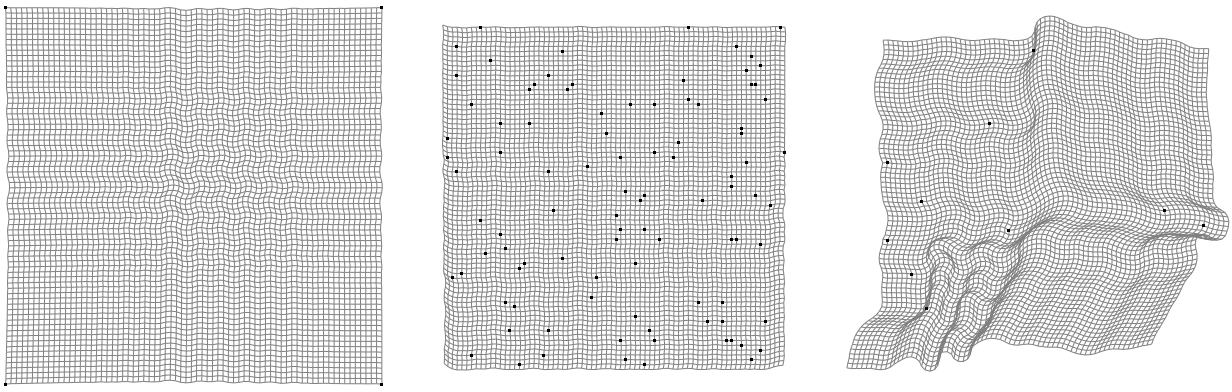


Figure 7.2: Layouts of the graph of Fig. 5.1, where fictitious trains run along grid lines, using the tailored layout for connection-graphs.

- of degree 3 are replaced by a triangle of edges, where the edge lengths are computed as if the angles between the incident edges of the node were 60°.

7.2.4 Another Dimension.

Generally speaking, a set of desired edge lengths can be realized more accurately when the number of dimensions of the Euclidean space is increased.

Several models make use of this observation by temporarily allowing additional coordinates and then penalizing their deviation from zero [102] or projecting down [103].

We use a third coordinate during all phases of the layout algorithm, but ignore it in the final layout. Since projections do not preserve the edge lengths,[2] we use a penalty function $\sum_{v \in V} c_t \cdot z_v^2$, where c_t is the penalty weight at the tth iteration, to gradually reduce the value of the z-coordinate towards the end of the layout computation.

The penalty has to be chosen large enough such that the graph is finally pressed into the plane and small enough that the result differs from the projection.[3] Let P_0 denote the potential at the beginning of the last phase. Experiments showed that a multiple of the unmodified potential that increases like the fourth power of the time gives good results: $c_t = C \cdot P_0 \cdot t^4$.

The average final potential of 100 runs of the algorithm is reduced by 16% with respect to an exclusively two-dimensional approach.

7.2.5 Summary

In summary, our layout algorithm consists of the following six steps:

[2] In our experiments with 100 layouts, the objective function triples if the graph is simply projected into the plane.

[3] In our experiments with 100 layouts, the objective function of a projection into the plane and a subsequent optimization in the plane is 7% worse.

1. barycentric layout of graph of long-range connections for nodes of high degree
2. iterative improvement
3. barycentric layout of graph of long-range connections
4. iterative improvement
5. barycentric layout of entire graph including 2-, 3-, and long-range connections
6. iterative improvement with increasing z-coordinate penalties

In each of these steps, the iteration is stopped when none of the stations was moved by more than a fixed distance. Figure 7.2 shows the results of this approach when applied to the graph of Fig. 5.1.

When it comes to ship and flight schedules it is not possible anymore to ignore the fact that the earth is not flat. Fortunately it is easy to modify the algorithm in such a way that it works on a sphere. It is sufficient to (a) use a metric that reflects the slope of the earth and (b) modify the penalty function of the projection. Whereas the latter can be done in a canonical way, the new distance should avoid expensive trigonometric calculations. It turned out that a fifth order approximation of the arcus sine multiplied by the Euclidean distance works well in practice.

7.3 Experimental Setup

For the experimental evaluation of our generated layouts, we use the (simplified) scenario of a travel planning system for public railroad transport presented in [7, 9]. It is based solely on *timetables*; for each train there is one table, which contains the departure and arrival times of that train at each of its halts. In particular, we assume that every train operates daily.

The system evaluates *connection queries* of the following kind: Given a departure station A, a destination station B, and an earliest departure time, find a connection from A to B with the minimum travel time (i.e., the difference between the arrival time at B and the departure time at A).

To this end, a (directed) *time-expanded graph* of the network is constructed from timetables in a preprocessing step. For each departure and arrival of a train there is one node in the graph. Therefore, each node is naturally associated with a station, and with a time label (the time the departure or arrival of the train takes place). There are two different kinds of edges in the graph:

Stay edges: The nodes associated with the same station are ordered according to their time label. Then, there is a directed edge from every node to its successor (for the last node there is an edge to the first node which introduces cycles in the graph). Each of these edges represents a stay at the station, and the edge length is defined by the duration of that stay.

Travel edges: For every departure of a train there is a directed edge to the very next arrival of that train. Here, the edge length is defined to be the time difference between arrival and departure. (Travel edges introduce more complex cycles.)

Answering a connection query now amounts to finding a shortest path from a source to one out of several target nodes: The source node is the first node at the start station representing a departure that takes place not earlier than the earliest departure time, and each node at the destination station is a feasible target node. In [7], a variation of Dijkstra's algorithm is used for these shortest-path computations that takes advantage of the special structure of the graph and uses several speed-up techniques. For the evaluation of the generated layouts, we focus on the purely geometric speed-up techniques of this system: *goal-directed search* (see Chapter 3) and *shortest-path containers with angular sectors* (see Chapter 2).

Our computational experiments are based on the timetables of the Deutsche Bahn AG, Germany's national train and railroad company, for the winter period 1996/97. It contains a total of 933,280 arrivals and departures on 6,884 stations, for which we have complete coordinate information (`de-org` in Fig. 7.3). To assess the quality of coordinates generated by the layout algorithms described in Sect. 7.2, we used a snapshot of queries against the central travel information server of Deutsche Bahn AG. This data consists of 544,181 queries collected over several hours of a regular working day. These benchmark data are unique in the sense that it is the only real network for which we have both coordinates and query data.

In the experiments, shortest paths are computed for the above queries using our own implementation of Dijkstra's algorithm and the angle-restriction and goal-directed search heuristics. All implementations are in C or C++, compiled with `GCC 2.95.2`.

7.4 Computational Results

From the timetables we generated the following instances:

- `de-org` (coordinates known for all stations)
- `de-22-important` (coordinates known for the 22 most important[4] stations)
- `de-22-random` (coordinates known for 22 randomly selected stations)
- `de` (no coordinates given)

For these instances, we generated layouts using the barycentric model of Sect. 5.1, the tailored model of Sect. 7.2 with estimated distances, and the tailored model with average travel time as "distance," and measured the average core CPU time spent on answering the queries, as well as the number of edges touched by the modified versions of Dijkstra's

[4]Together with the coordinate information, there is a value associated with each station that indicates its importance as a hub. The 22 selected stations have the highest attained value.

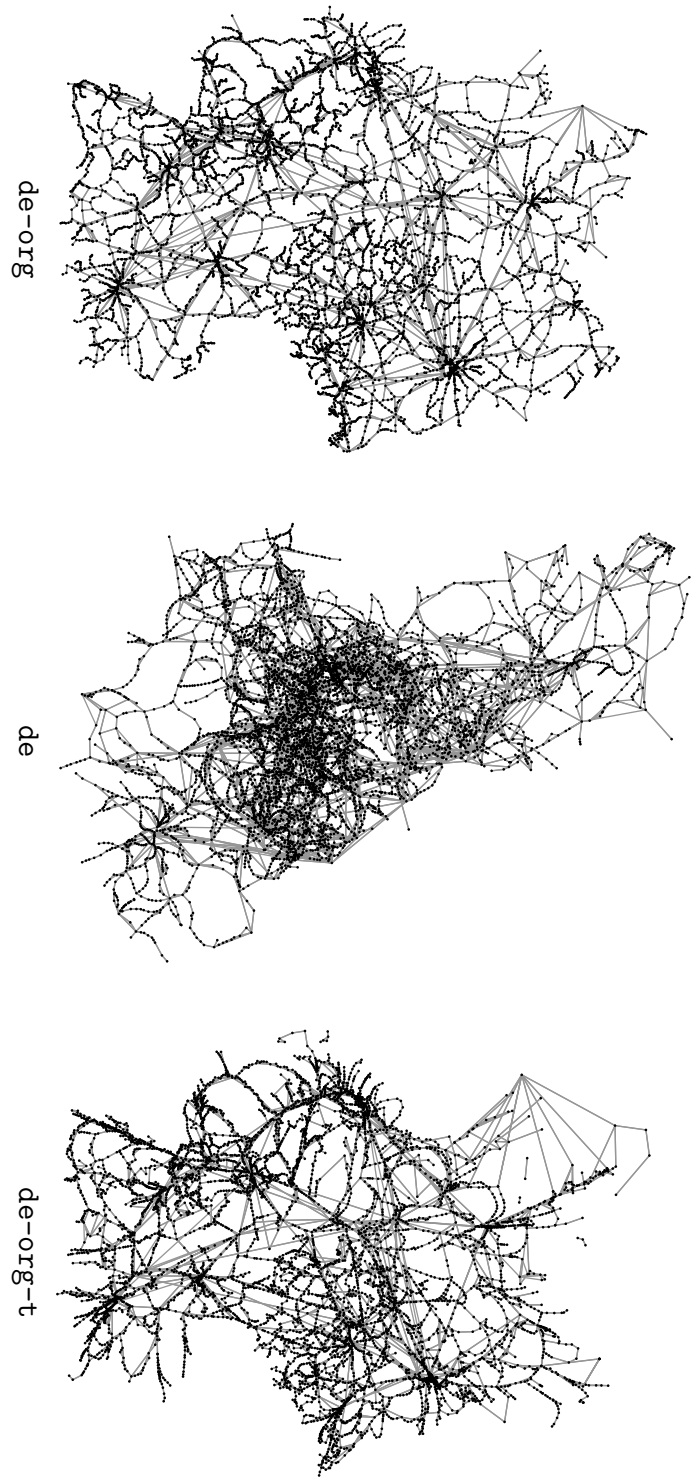


Figure 7.3: Original and generated layouts

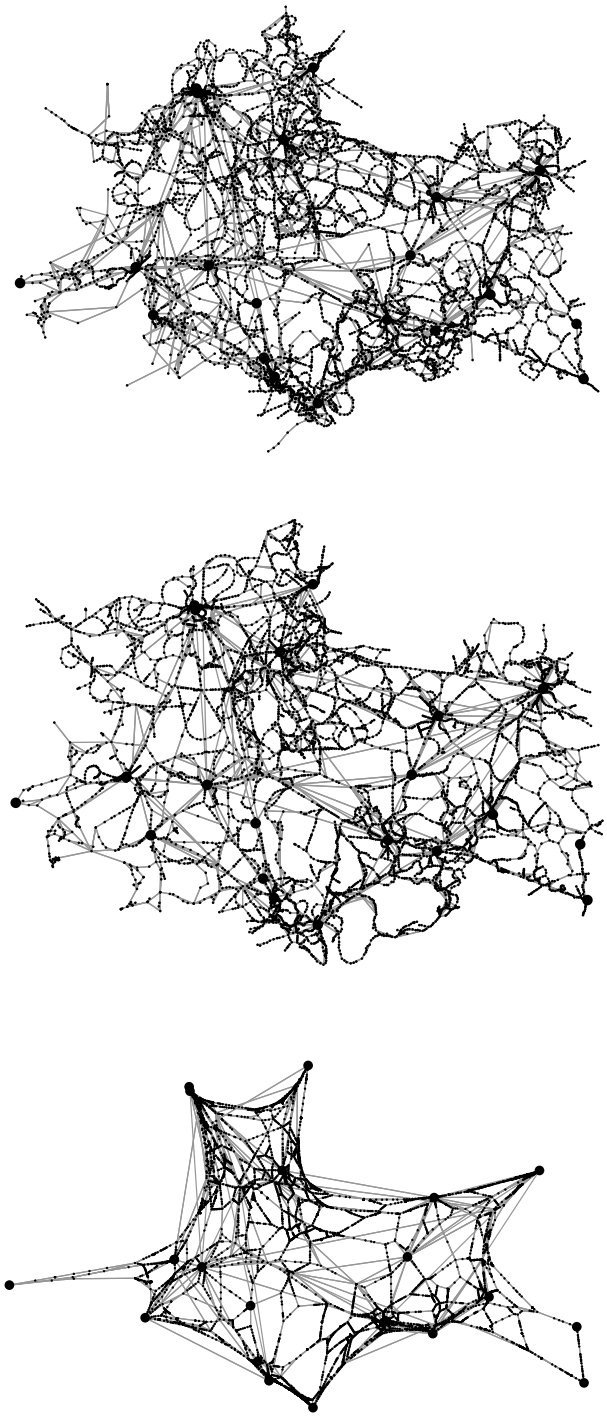
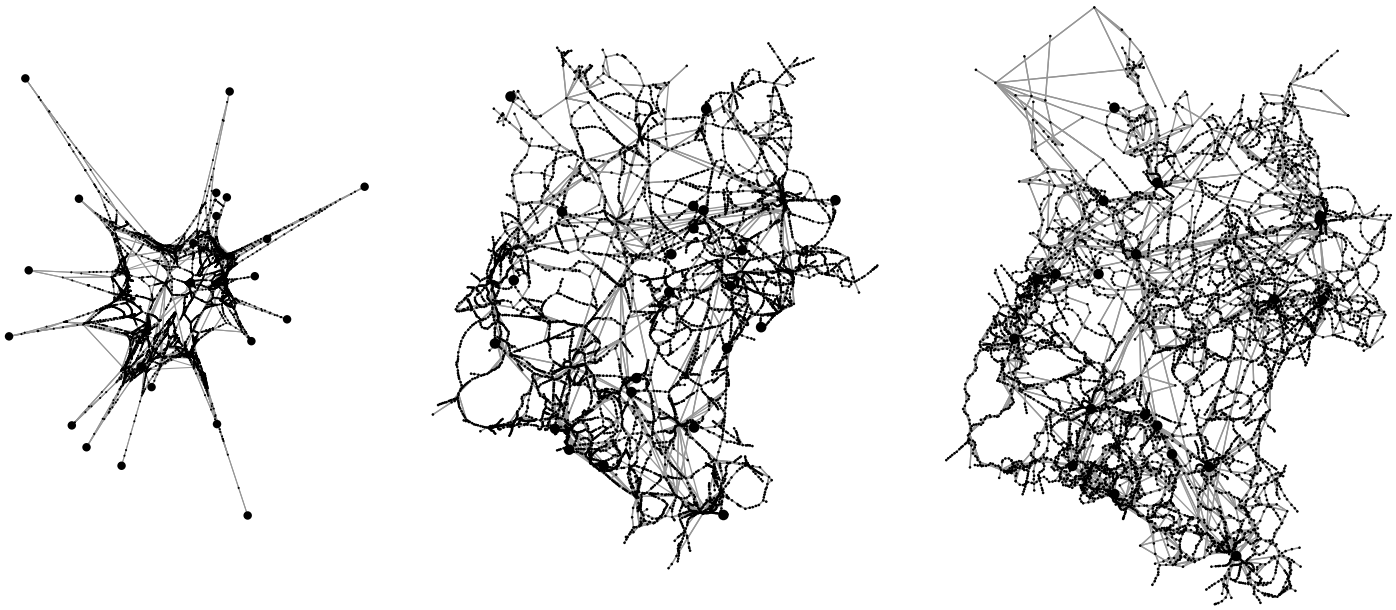


Figure 7.4: Generated layouts with provided coordinates for 22 important stations (`de-22-important`)



barycentric tailored (dist. est.) tailored (time avg.)

Figure 7.5: Generated layouts with provided coordinates for 22 random stations (`de-22-random`)

		speed-up technique					
		angles		goal		both	
instance	layout model	ms	edges	ms	edges	ms	edges
`de-org`		**17**	9177	**79**	20995	**14**	6496
`de` (Fig. 7.3)	tailored (dist. est.)	**40**	17553	**107**	28591	**43**	15167
	tailored (time avg.)	**43**	18228	**92**	24634	**38**	13888
`de-22-`	barycenteric	**20**	10464	**100**	26669	**19**	8722
`important`	tailored (dist. est.)	**19**	9844	**93**	24334	**19**	7763
(Fig. 7.4)	tailored (time avg.)	**22**	10994	**75**	20052	**17**	7412
`de-22-`	barycentric	**27**	13415	**124**	32628	**31**	13066
`random`	tailored (dist. est.)	**21**	10597	**87**	23033	**18**	7973
(Fig. 7.5)	tailored (time avg.)	**27**	11671	**77**	20853	**22**	8055

Table 7.1: Average query response times and number of nodes touched by Dijkstra's algorithm. Without coordinates, the average response time is **105** ms (33704 edges).

algorithm. Each experiment was performed on a single 336 MHz UltraSparc-II processor of a Sun Enterprise 4000/5000 workstation with 1024 MB of main memory. The results are given in Tab. 7.1, and the layouts are shown in Figs. 7.3–7.5.

The results show that the barycentric model seems to pair very well with the angle-restriction heuristic when important stations are fixed. The somewhat surprising usefulness of this simple model even for the randomly selected stations seems to be due to the fact that our sample spreads out quite well.

Another interesting observation is that the layouts according to average travel times are better for goal-directed search than the layouts that use estimated distances. This can be explained by the fact that goal-directed search uses the highest speed to calculate the lower bound to the destination, which is tighter in this case.

The tailored layout model appears to work well in all cases. The results and pictures suggest that our layout algorithm produces a reasonably good reconstruction of the traffic network with respect to the travel-planning system. Note that the average response time for connection queries compared to the average response time without coordinates is reduced by 60%, even without any knowledge of the underlying geography. With the fairly realistic assumption that the location of a limited number of important stations is known, the speed-up obtained with the actual coordinates is almost matched.

To evaluate whether the tailored model achieves the objective of preserving given edge lengths, we generated additional instances from `de-org` by dropping a fixed percentage ranging from 0–100% of station coordinates, while setting $l(e)$ to its true value. As can be seen in Fig. 7.6, these distances are reconstructed quite well.

As an attempt to avoid the small but existing shortcomings of the layout algorithms as much as possible, we generated a further set of coordinates `de-org-t` that uses the original coordinates as initialization. The layout was then modified locally to fit the travel times

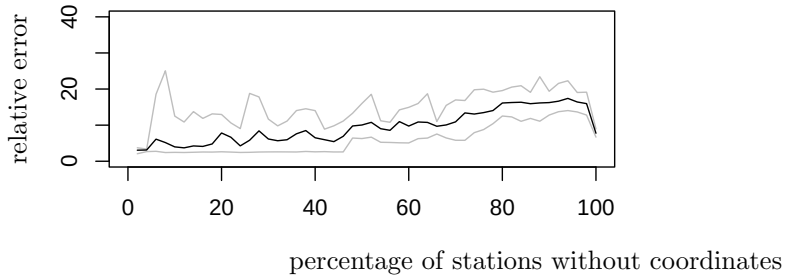


Figure 7.6: Evaluation whether the tailored layout model preserves edge lengths. Minimum, mean, and maximum relative error in edge lengths, averaged over ten instances each

		speed-up technique					
		angles		goal		both	
instance	layout model	ms	edges	ms	edges	ms	edges
`de-org-t`	modified org	**18**	9384	**71**	18782	**14**	6086

Table 7.2: Average query response times and number of nodes touched by Dijkstra's algorithm for the layout based on original coordinates.

as much as possible.

The result is depicted as `de-org-t` in Fig. 7.3 and the resulting query times are shown in Tab. 7.2. With `de-org-t`, the results are slightly better than the results with any other generated coordinates. This suggests that there is still some space for improvement of the layout algorithm, but probably not much. The main result of this last experiment, however, is the observation that the original coordinates can be improved with respect to the goal-directed search.

Chapter 8

Generating Layouts for Geometric Speed-Up Techniques

The previous investigations in Chapters 2 and 3 have shown the practical usefulness of geometric speed-up techniques that guarantee the correctness of the result for shortest path computations. However, such speed-up techniques utilize a layout of the graph which typically comes from geographic information. This chapter examines the question how geometric speed-up techniques can be used in case there is no layout given. We present an extensive computational study analyzing the usefulness of methods from graph drawing as foundation for such techniques. It turns out that using appropriate layout algorithms, significant a speed-up can be achieved.

Our baseline algorithm without any preprocessing is the bidirectional variant of Dijkstra's algorithm with binary heaps. For sparse graphs, the asymptotic running time for Dijkstra's algorithm with binary heaps is $\mathcal{O}(n \log n)$. Moreover, for our application, binary heaps have been reported to be efficient in practice [44].

In Chapter 7, a related question has been studied for the special case of a timetable information system. A scenario is considered where the geographic information typically contained in timetable data is only incomplete. The results in this chapter are more general with respect to the graphs considered, as well as the layout algorithms explored. We experiment with real world graphs from different areas and with various generated graphs. In particular, in contrast to Chapter 7, no additional information is used that might support the layout algorithms, like movement of trains or coordinates of selected stations.

The main contribution of this chapter consists in a computational study demonstrating that artificially produced layouts can indeed be used as basis for geometric speed-up techniques for shortest paths computations. For several of the graphs explored, significant speed-ups are achieved with appropriately generated layouts. Moreover surprisingly, for a number of the tested instances where layouts based on geographic information were available, the generated layouts resulted in an even better speed-up.

	Street		Rail		AS		Planar		Small W.		Preferential	
1	1429	3034	409	1215	14492	29948	2000	10000	2000	11974	2000	12000
2	2948	7128	698	1669	14689	30336	4000	20000	4000	23984	4000	24000
3	15868	33380	1650	4311	14899	30712	5999	30000	6000	35960	6000	36000
4	20036	41476	2239	5948	15078	31347	7999	40000	8000	47974	8000	48000
5	24106	53826	2348	7856	15268	33743	9998	50000	10000	59978	10000	60000
6	35802	78646	4553	14829	15415	34716	11999	60000	12000	71974	12000	72000
7	38823	79988	6848	18542	15578	35041	13998	70000	14000	83976	14000	84000
8	44439	96994	10795	29328	15763	33000	15999	80000	16000	95974	16000	96000
9	44878	90930	12070	33728	15899	33585	17998	90000	18000	107984	18000	108000
10	78947	171410	14335	39887	16037	34283	19999	100000	20000	119970	20000	120000

Table 8.1: Number of nodes and edges for all test graphs.

8.1 Experimental Setup

For our experiments, we used graphs of six different types and 10 graphs of every type. Three of the types are real data that stem from an application (Streets, Railway, and AS) while the three other types are randomly generated graphs. Furthermore, three of the types provide already a layout (Streets, Railway, and Planar), which we can use as a baseline. In detail, the six types of graphs are:

Street Networks. The street networks in our test data are derived from street maps of US cities and their surroundings. As bends are realized with piecewise straight lines, these graphs are very sparse and fairly large. Unfortunately, our data does not contain information about the street classification, so all edges are weighted by an approximation of their Euclidean length.

Railway Graphs. A node in a railway network is a station or stop. There exists an edge between two nodes, if there is a non-stop connection serving the respective stations. The edges are weighted according to their average travel time. A layout of this graph is provided by the geographic coordinates of the stations (although the edge weights are not derived from this layout).

Autonomous Systems. These graphs represent the autonomous systems topology of the Internet. An Autonomous System (AS) is a collection of routers which are under one administrative domain (e.g. UUNET, AT&T, or DFN). They present a unified face to the rest of the world in terms of accessibility and routing policies. Every node in an AS-graph represents one Autonomous System, and two nodes are connected if there is at least one physical link between the two corresponding Autonomous Systems.

The AS-graphs are generated regularly by the *Oregon Route Views Project* using traces. If $t(e)$ denotes the number of traces that crossed an edge e, we regard $\frac{1}{t(e)}$ as the edge length. Therefore, good connections that are passed by a lot of traces are regarded as short edges.

Random Planar Graphs We generated a family of random planar graphs. For given n and m, we first select n points uniformly at random in the unit square. Then, we determine a Delaunay triangulation of these points and delete edges randomly until

there are only m edges left.[1] The edge weights are the Euclidean distances in the layout that is provided by the construction.

Small Worlds. In [104], a graph model has been introduced to simulate self-organizing networks that show a small average path length (like, e.g., networks of acquaintances, the power network in the Western US, or the collaboration graph of movie actors). Starting from a regular ring lattice with n nodes and d edges per node, an edge is "rewired" with a given probability p. For our test sets we used $d = 3$ and $p = 0.1$. Edge weights are uniformly distributed over $[0, 1]$.

Graphs with Preference. A model for random graphs that produces a power law distribution of the node degrees has been presented in [105]. Nodes are added consecutively to the graph, while an incident edge of a new node is added with a certain probability. The main idea is that a new edge connects more likely to a node with a high degree. For our test sets, we generated graphs with an expected degree of 3. Edges weights are uniformly distributed over $[0, 1]$.

From all graphs, we used only the maximal connected component for our calculations to assert connectedness. (In all cases, the maximum connected component contained almost all if not all nodes.) The number of nodes and edges are listed in Table 8.1.

The graphs were drawn with the three methods described in Chapter 5:

Force-Directed Layout. For our implementation, we combined some of the methods in Section 5.1 for a multi-level realization of a force-directed layout. To create the next coarser graph in the multi-level hierarchy, edges of a maximal matching are contracted [36]. This works very well except for graphs that contain many star-like structures. (Then, only few edges are removed in the next coarser graph.) For such graphs, we therefore use an inclusion-maximal independent set as in [35].

Instead of forces according to Fruchterman and Reingold [33], we followed the approach of Kamada and Kawai [34], because these forces are better suited to represent distances in a graph. To avoid a computation and storage of all-pairs shortest paths, we restricted the forces to the nearest 30 nodes. Due to the multi-level embedding, distances are also preserved in a larger range.

Spectral Layout. We incorporated the edge weights in this approach by replacing the adjacency matrix A by a weighted adjacency matrix $W = (w_{ij})_{ij}$. Since an edge weight $l(v_i, v_j)$ represents the length of an edge, we set $w_{ij} = \frac{1}{l(v_i,v_j))}$, if $\{v_i, v_j\} \in E$, and $w_{ij} = 0$ otherwise. The diagonal matrix D uses the weighted degree in this case. (Using the weighted Laplacian substantially improved our results.)

High-Dimensional Embedding. Again, we incorporated the edge lengths, which was fairly obvious for this graph drawing technique. The high-dimensional embedding,

[1] This is very close to the generator of planar graphs in LEDA except that we use a Delaunay triangulation. We think that these graphs correspond more to our intuition of a random planar graph.

which is later projected to 2D, is generated with distances in the graph. We replaced the unweighted graph by the weighted graph—or algorithmically, the breadth-first search by Dijkstra's algorithm.

For each graph and layout, we processed shortest-path queries using bidirectional search and all eight combinations of (1) goal-directed search, (2) geometric shortest-path containers, and (3) reach-based routing (Sect. 1.2). We used exact values for the reach and not the bound provided by [49], since we admit an all-pairs shortest-path computation for shortest-path containers, too. We selected bounding boxes as object type for geometric shortest-path containers, because they are fast and simple.

We determined the average number of nodes that were inserted in the priority queue as well as the average CPU time used per query. Observe that in contrast to the second number, the first number does not depend on the type of the priority queue, implementation, compiler, system, or processor. The actual *speed-up* for a speed-up technique, graph and layout is then defined as the ratio of the respective values without speed-up technique and with speed-up technique.

The algorithms have been implemented in C++ using LEDA 4.5 (see [44]). In particular, we used the graph and binary heap data structures, vector and matrix classes and the algorithms for computing maximum matchings and Delaunay triangulations from LEDA. The programs were compiled with GCC 3.3 and the experiments were performed on a single AMD Opteron with 2.2 GHz running Linux 2.4. For a unified processing, all graphs have been converted to GraphML [55].

8.2 Computational Results

The results are depicted in Fig. 8.1–8.7. Each figure summarizes the result for a combination of speed-up techniques that is added to bidirectional search. In each figure, the left diagram illustrates the average speed-up in terms of the number of visited nodes and the right diagram illustrates the average speed-up in terms running time. Since there is no layout given for some graphs (AS-graphs, small world graphs, and graphs with preferential attachment), the speed-up for "given layout" for these graphs is missing in all figures.

The high-dimensional layout is obviously well suited for goal-directed search (Fig. 8.1). It is the best layout for small-world and street graphs and also very good for random planar graphs, although the given layout is unbeatable in this case. Note, that for street graphs the speed-up for the high-dimensional layout is higher than for the given layout. However, the overhead for goal-directed search is so large that in many cases no speed-up in terms of CPU time is achieved.

The situation is completely different for shortest path containers (Fig. 8.2). Except for AS-graphs, the speed-up in terms of CPU time is close to that for the number of visited nodes and achieves values up to 20–40. Furthermore, it is interesting to note that for shortest-path containers the generated layouts are as good as the given layouts (with the high-dimensional layout for AS-graphs being the only exception).

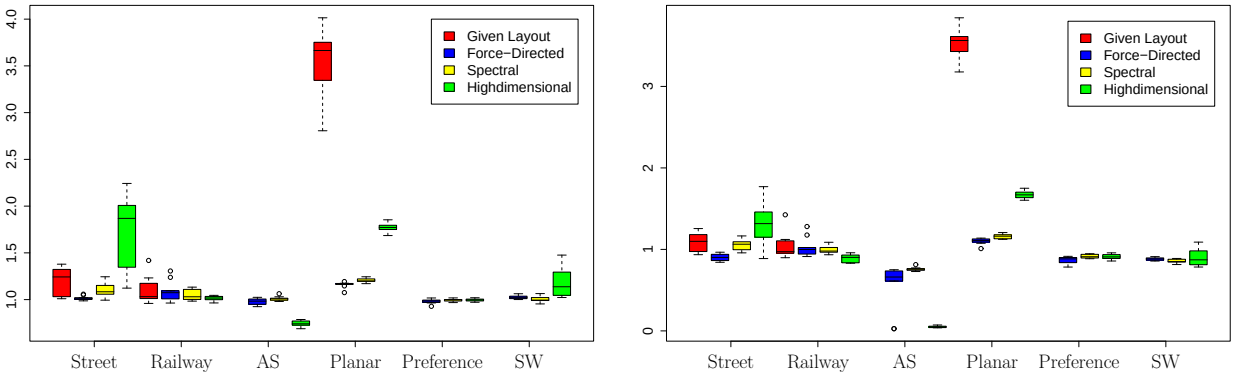
The speed-up for adding reach-based routing (Fig. 8.2) is smaller, but it is nice to see that this speed-up technique is well-suited for street graphs as originally intended [49] (but works very well for small word graphs, too). Again, all generated layouts provide a fairly good performance compared to the provided layouts.

If you combine goal-directed search with shortest-path containers (Fig. 8.4), the speed-up is dominated by the latter speed-up technique. Therefore, the results are very similar to the results in Fig. 8.2. The same is true for the combination of shortest-path containers with reach (Fig. 8.6) and the combination of all three speed-up techniques (Fig. 8.7), which reveal the same characteristics. More interesting is Fig. 8.5, which shows the combination of goal-directed search and reach. The speed-up of the two techniques add up for most graphs.

For AS-graphs, the speed-up concerning CPU time is generally not as good as for the number of visited nodes. This can be explained by the special structure of these graphs, which consists of a highly connected core to which a lot of path-like graphs are attached. The ratio of excluded and visited nodes is very high in this case, which increases the average running time per visited node.

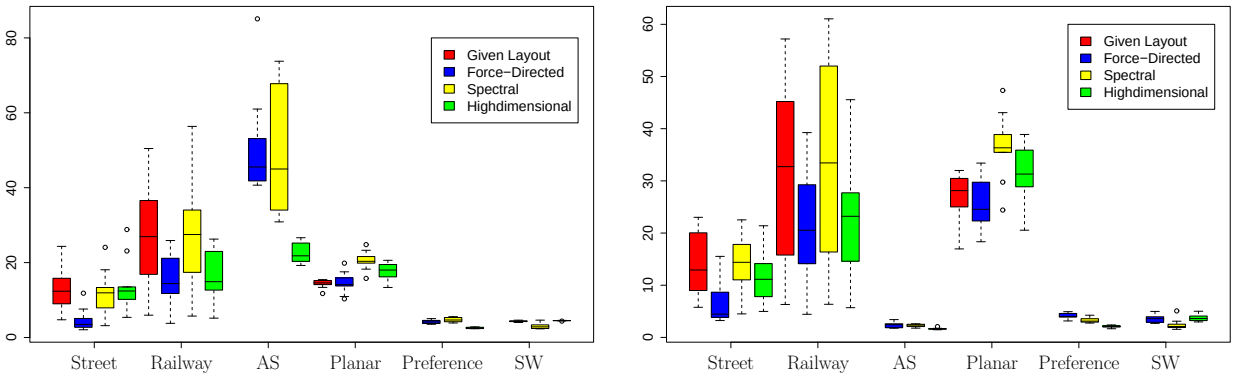
We have seen, that sometimes a fairly good speed-up of the query time is possible by first generating a layout and then applying geometric speed-up techniques. Apart from few exceptions, all three graph-drawing methods produce equally good layouts concerning geometric speed-up techniques. Since a high-dimensional layout is best suited for goal-directed search and generally faster to produce, we recommend this type of layout. It is also notable, that if a layout is already given for a graph, it is sometimes possible to generate a layout that results in a better speed-up.

Motivated by these results, it would be interesting to develop a specialized "graph drawing" method that optimizes the layout for geometric speed-up techniques. Such a layout algorithm would not necessarily produce nice drawings, but generate even better layouts to speed up shortest-path computations.



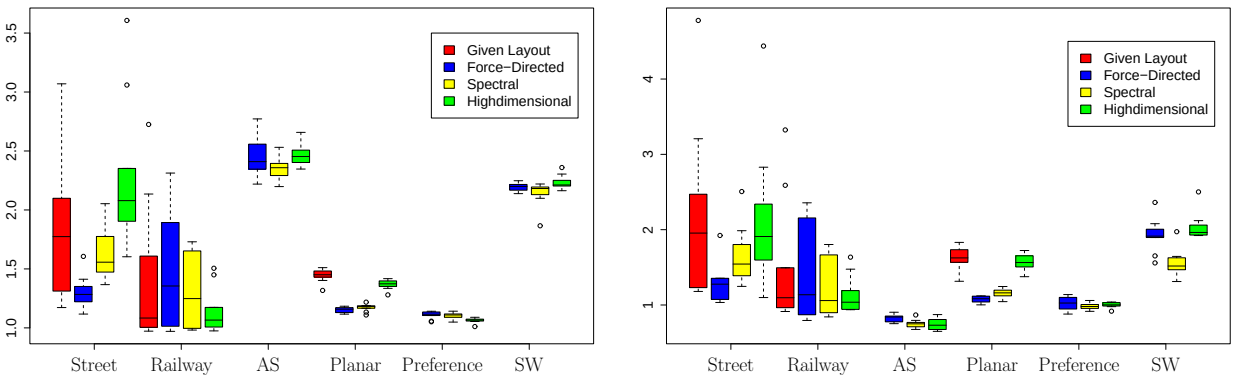
(a) # visited nodes (b) CPU time

Figure 8.1: Average speed-up for all graph classes and layouts using *goal-directed search*



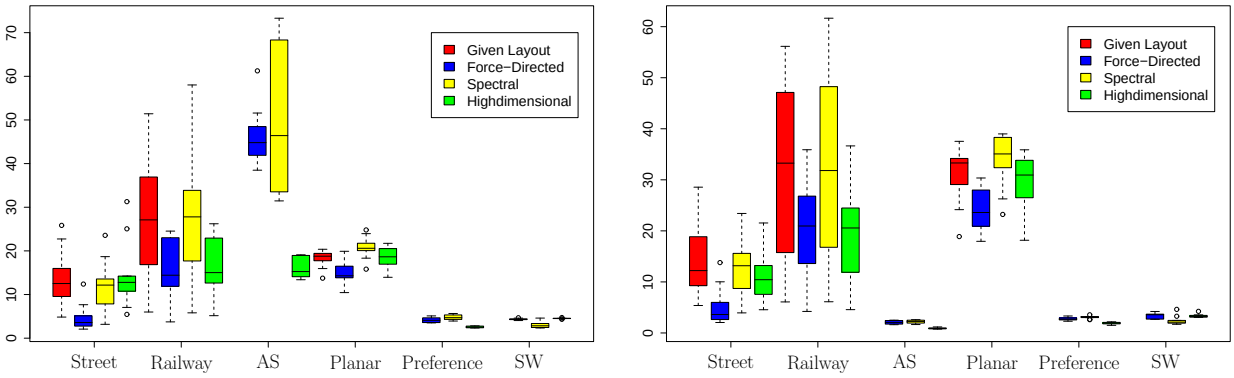
(a) # visited nodes (b) CPU time

Figure 8.2: Average speed-up for all graph classes and layouts using *shortest-path containers*



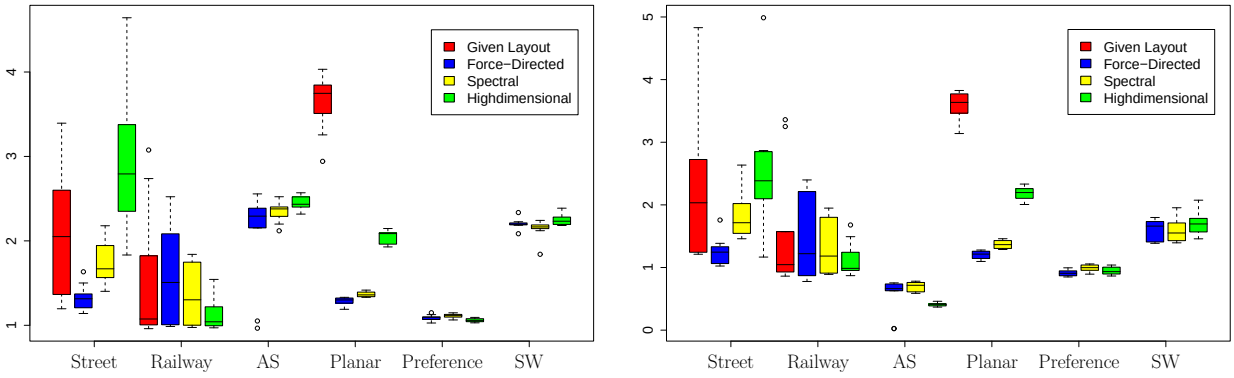
(a) # visited nodes (b) CPU time

Figure 8.3: Average speed-up for all graph classes and layouts using *reach*



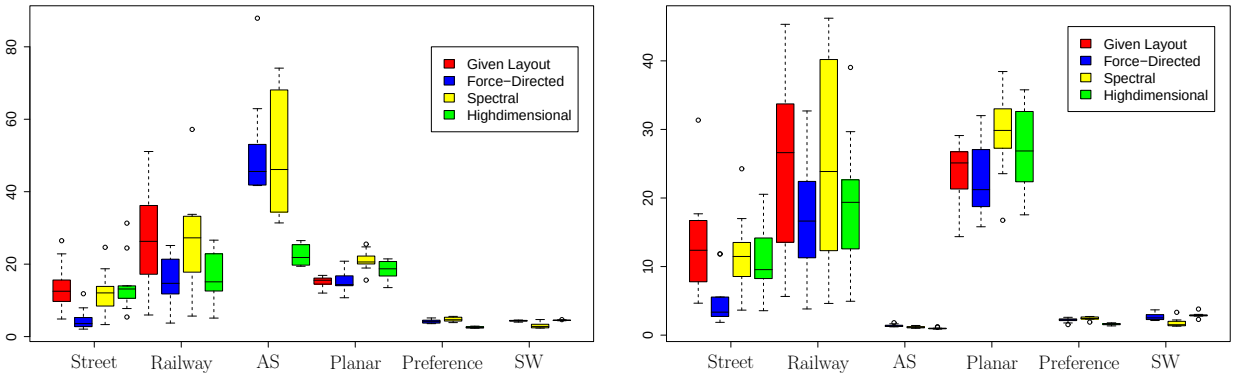
(a) # visited nodes (b) CPU time

Figure 8.4: Average speed-up for all graph classes and layouts using *goal-directed search* and *shortest-path containers*



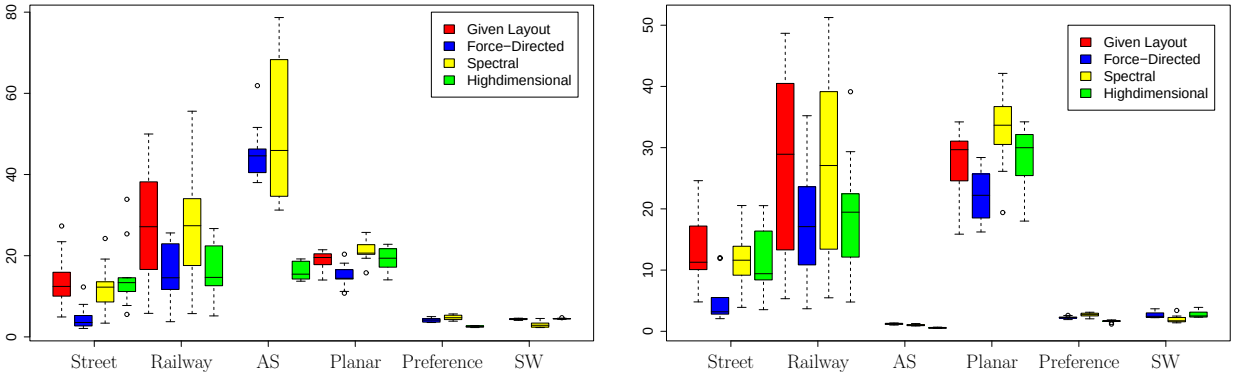
(a) # visited nodes (b) CPU time

Figure 8.5: Average speed-up for all graph classes and layouts using *goal-directed search* and *reach*



(a) # visited nodes (b) CPU time

Figure 8.6: Average speed-up for all graph classes and layouts using *shortest-path containers* and *reach*



(a) # visited nodes (b) CPU time

Figure 8.7: Average speed-up for all graph classes and layouts using *goal-directed search*, *shortest-path containers* and *reach*

Chapter 9

Conclusion

We have seen that using a layout may lead to a considerable speed-up in Dijkstra's algorithm, if one allows a suitable preprocessing. Actually, we are able to reduce the search space with a linear amount of space to only $5 - 10\%$, while even "bad" containers result in a reduction to less than 50%. The somewhat surprising result is that the simple bounding box outperforms other geometric objects in terms of CPU cycles. For bit-vectors as shortest-path containers, very good partitions can be generated with standard techniques like quadtrees or kd-trees. Furthermore, the memory consumption can be reduced efficiently for an unidirectional search with two-level bit-vectors. Since the geometric pruning is independent of the priority queue. Algorithms using a special priority queue such as [106, 107] can easily be combined with it. The decrease of the search space is in fact the same (but the actual running time would be different).

In the dynamic case, we have seen that it is possible to speed up the maintenance of geometric containers by a factor of about 2-3 while preserving optimality in almost all cases. Enlarging containers to infinity leads to a cascading effect that destroys the benefit of geometric containers. If containers are only enlarged, the presented pruning of Dijkstra's algorithm does not justify the loss of quality. It would be interesting to find other simplifications that guarantee consistent containers, but realize a good compromise between optimality and running time. Furthermore, our results suggest that it should be possible to get a speed-up factor of about 2 with an (provable) optimal update strategy. Finally, it might be possible to combine edge weight increases and edge weight decreases in a single algorithm.

For bit-vectors as shortest-path containers, the partitioning algorithm plays a crucial rule for the speed-up. Using the standard data structure of a kd-tree results in an efficient algorithm. If an unidirectional search is performed, a higher speed-up is possible with the same amount of space using two-level bit-vectors.

It is easy and efficient to combine shortest-path containers with other speed-up techniques. Of particular interest is the bidirectional search where the speed-up factors almost scale. Another hot candidate for a combination is the goal-directed search. However, the higher the number of speed-up techniques in a combination, the lesser their impact on the overall speed-up.

If a layout of the graph is not provided, we have seen that it makes perfect sense to draw the graph in order to use geometric speed-up techniques afterwards. The speed-up factors for generated layouts may be even on par with layouts that are provided by the application. In case a graph has a specific structure (like a timetable graph), it is possible to engineer a specific graph-drawing algorithm that takes advantage of this structure. In our concrete application of a timetable graph, the train lines were of particular use to generate multiple levels of the graph. Furthermore, it can be assumed in practice that parts of the layout are known in this case.

For general graphs, major graph-drawing techniques can be used to generate a layout. Apart from some exceptions, their results are suited for our application even though their original purpose is fairly different. Although force-directed and spectral layouts are also appropriate, we recommend a layout by a high-dimensional layout, because it is easy to implement and acceptably fast in practice.

Bibliography

[1] F. B. Zhan and C. E. Noon, "A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths," *Journal of Geographic Information and Decision Analysis*, vol. 4, no. 2, 2000.

[2] C. Barrett, K. Bisset, R. Jacob, G. Konjevod, and M. Marathe, "Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router," in *Proc. 10th European Symposium on Algorithms (ESA 2002)* (R. Möhring and R. Raman, eds.), vol. 2461 of *LNCS*, pp. 126–138, Springer, 2002.

[3] S. Jabbar, S. Edelkamp, and T. Willhalm, "Geometric travel planning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 6, pp. 5–16, March 2005.

[4] L. Siklóssy and E. Tulp, "TRAINS, an active time-table searcher," in *Proc. 8th European Conf. Artificial Intelligence*, pp. 170–175, 1988.

[5] T. Preuss and J.-H. Syrbe, "An integrated traffic information system," in *Proc. 6th Int. Conf. Appl. Computer Networking in Architecture, Construction, Design, Civil Eng., and Urban Planning (europIA '97)*, 1997.

[6] K. Nachtigall, "Time depending shortest-path problems with applications to railway networks," *European Journal of Operational Research*, vol. 83, no. 1, pp. 154–166, 1995.

[7] F. Schulz, D. Wagner, and K. Weihe, "Dijkstra's algorithm on-line: An empirical case study from public railroad transport," *ACM Journal of Experimental Algorithmics*, vol. 5, no. 12, 2000.

[8] M. Müller-Hannemann and K. Weihe, "Pareto shortest paths is often feasible in practice," in *Proc. 5th Workshop on Algorithm Engineering (WAE'01)* (G. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, eds.), vol. 2141 of *LNCS*, pp. 185–197, Springer, 2001.

[9] F. Schulz, D. Wagner, and C. Zaroliagis, "Using multi-level graphs for timetable information," in *Proc. Algorithm Engineering and Experiments (ALENEX'02)*, vol. 2409 of *LNCS*, pp. 43–59, Springer, 2002.

[10] G. Ramalingam and T. W. Reps, "An incremental algorithm for a generalization of the shortest-path problem," *Journal of Algorithms*, vol. 21, no. 2, pp. 267–305, 1996.

[11] G. Ramalingam and T. W. Reps, "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, pp. 233–277, 1996.

[12] D. Frigioni, "Semidynamic algorithms for maintaining single-source shortest path trees," *Algorithmica*, vol. 22, no. 3, pp. 250–274, 1998.

[13] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic output bounded single source shortest path problem," in *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, pp. 212–221, 1996.

[14] S. Even and H. Gazit, "Updating distances in dynamic graphs," *Methods of Operations Research*, vol. 49, pp. 371–387, 1985.

[15] H. Rohnert, "A dynamization of the all pairs least cost path problem," in *Proc. Symp. Theoretical Aspects of Computer Science (STACS'85)*, vol. 182 of *LNCS*, pp. 279–286, Springer, 1985.

[16] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," in *Proc. 35th ACM Symposium on Theory of Computing (STOC 2003)*, pp. 159 – 166, ACM Press, 2003.

[17] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Incremental algorithms for minimal length paths," *Journal of Algorithms*, vol. 12, pp. 615–638, 1991.

[18] V. King, "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs," in *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pp. 81–91, 1999.

[19] C. Zaroliagis, "Implementations and experimental studies of dynamic graph algorithms," in *Experimental Algorithmics* (R. Fleischer, B. Moret, and E. M. Schmidt, eds.), vol. 2547 of *LNCS*, pp. 229–278, Springer, 2002.

[20] U. 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* (M. Raubal, A. Sliwinski, and W. Kuhn, eds.), vol. 22 of *IfGI prints*, pp. 219–230, Institut für Geoinformatik, Münster, 2004.

[21] E. Köhler, R. H. Möhring, and H. Schilling, "Acceleration of shortest path computation," Tech. Rep. 42-2004, Institute of Mathematics, TU Berlin, 2004.

[22] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on systems science and cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[23] M. Holzer, "Hierarchical speed-up techniques for shortest-path algorithms," tech. rep., Dept. of Informatics, University of Konstanz, Germany, 2003. http://www.ub.uni-konstanz.de/kops/volltexte/2003/1038/.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, New York, NY: Addison-Wesley, 1995.

[25] G. Bracha and W. Cook, "Mixin-based inheritance," in *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proc. European Conference on Object-Oriented Programming* (N. Meyrowitz, ed.), pp. 303–311, ACM Press, 1990.

[26] F. J. Hauck, "Inheritance modeled with explicit bindings: an approach to typed inheritance," *SIGPLAN Notices*, vol. 28, no. 10, 1993.

[27] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[28] U. Eisenecker and K. Czarnecki, *Generative Programming in C++*, ch. 10, pp. 397–501. Addison-Wesley, 2000.

[29] J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[30] O. Spinczyk, A. Gal, and W. Schroder-Preikschat, "AspectC++: An aspect-oriented extension to the C++ programming language," in *Proc. 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)* (J. Noble and J. Potter, eds.), vol. 10 of *Conferences in Research and Practice in Information Technology*, (Sydney, Australia), pp. 53–60, ACS, 2002.

[31] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[32] W. T. Tutte, "How to draw a graph," *Proceedings of the London Mathematical Society, Third Series*, vol. 13, pp. 743–768, 1963.

[33] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software – Practice & Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[34] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[35] P. Gajer and S. G. Kobourov, "Grip: Graph drawing with intelligent placement," *Journal of Graph Algorithms and Applications*, vol. 6, no. 3, pp. 203–224, 2002.

[36] C. Walshaw, "A multilevel algorithm for force-directed graph-drawing," *Journal of Graph Algorithms and Applications*, vol. 7, no. 3, pp. 253–285, 2003.

[37] D. Harel and Y. Koren, "A fast multi-scale method for drawing large graphs," *Journal of graph algorithms and applications*, vol. 6, no. 3, pp. 179–202, 2002.

[38] K. Hall, "An r-dimensional quadratic placement algorithm," *Management Science*, vol. 17, pp. 219–229, 1970.

[39] Y. Koren, "On spectral graph drawing," in *Proceedings of The Ninth International Computing and Combinatorics Conference (COCOON'03)*, vol. 2697 of *LNCS*, pp. 496–508, Springer, 2003.

[40] D. Harel and Y. Koren, "Graph drawing by high-dimensional embedding," in *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*, vol. 2528 of *LNCS*, pp. 207–219, Springer, 2002.

[41] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[42] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[43] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[44] K. Mehlhorn and S. Näher, *LEDA, A platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[45] R. Dial, "Algorithm 360: Shortest path forest with topological ordering," *Communications of ACM*, vol. 12, pp. 632–633, 1969.

[46] A. V. Goldberg, "Shortest path algorithms: Engineering aspects," in *Proc. International Symposium on Algorithms and Computation (ISAAC 2001)* (P. Eades and T. Takaoka, eds.), vol. 2223 of *LNCS*, pp. 502–513, Springer, 2001.

[47] U. Meyer, "Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds," *Journal of Algorithms*, vol. 48, pp. 91–134, 2003.

[48] A. V. Goldberg and C. Harrelson, "Computing the shortest path: a^* search meets graph theory," Tech. Rep. MSR-TR-2004-24, Microsoft Research, 2003. Accepted at SODA 2005.

[49] R. Gutman, "Reach-based routing: A new approach to shortest path algortihms optimized for road networks," in *Proc. Algorithm Engineering and Experiments (ALENEX'04)* (L. Arge, G. F. Italiano, and R. Sedgewick, eds.), pp. 100–111, SIAM, 2004.

[50] D. Wagner and T. Willhalm, "Geometric speed-up techniques for finding shortest paths in large sparse graphs," in *Proc. 11th European Symposium on Algorithms (ESA 2003)* (G. D. Battista and U. Zwick, eds.), vol. 2832 of *LNCS*, pp. 776–787, Springer, 2003.

[51] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, "On the design of CGAL a computational geometry algorithms library," *Softw. – Pract. Exp.*, vol. 30, no. 11, pp. 1167–1202, 2000.

[52] E. Welzl, "Smallest enclosing disks (balls and ellipsoids)," in *New Results and New Trends in Computer Science* (H. Maurer, ed.), vol. 555 of *LNCS*, Springer, 1991.

[53] G. Toussaint, "Solving geometric problems with the rotating calipers," in *Proc. IEEE Mediteranian Electrotechnical Conference (MELECON 1983)* (E. N. Protonotarios, ed.), (NY), pp. A10.02/1–4, IEEE, 1983.

[54] C. Schwarz, J. Teich, A. Vainshtein, E. Welzl, and B. L. Evans, "Minimal enclosing parallelogram with application," in *Proc. 11th Annual Symposium on Computational Geometry*, pp. 434–435, ACM Press, 1995.

[55] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Scott, "GraphML progress report," in *Proceedings of the 9th International Symposium on Graph Drawing (GD '01)* (P. Mutzel, M. Jünger, and S. Leipert, eds.), vol. 2265 of *LNCS*, pp. 501–512, Springer, 2001.

[56] R. R. Wilcox, *Fundamentals of modern statistical methods: substantially improving power and accuracy*. Springer, 2001.

[57] D. Wagner, T. Willhalm, and C. Zaroliagis, "Dynamic shortest path containers," in *Proc. Algorithmic MeThods and Models for Optimization of RailwayS (ATMOS 2003)* (A. Marchetti-Spaccamela, ed.), vol. 92 of *Electronic Notes in Theoretical Computer Science*, pp. 65–84, 2004.

[58] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm, "Partitioning graph to speed up dijkstra's algorithm," in *Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005* (S. E. Nikoletseas, ed.), vol. 3503 of *LNCS*, pp. 189–202, Springer, 2005.

[59] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge Massachusetts: The MIT Press, 2001.

[60] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing archive*, vol. 20, pp. 359–392, August 1998.

[61] G. Karypis, "METIS: Family of multilevel partitioning algorithms." `http://www-users.cs.umn.edu/~karypis/metis/`, 1995.

[62] S. Shekhar, A. Kohli, and M. Coyle, "Path computation algorithms for advanced traveler information system (ATIS)," in *Proc. 9th IEEE Int. Conf. Data Eng.*, pp. 31–39, 1993.

[63] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows*. Prentice–Hall, 1993.

[64] I. Pohl, "Bi-directional and heuristic search in path problems," Tech. Rep. 104, Stanford Linear Accelerator Center, Stanford, California, 1969.

[65] H. Kaindl and G. Kainz, "Bidirectional heuristic search reconsidered," *Journal of Artificial Intelligence Research*, vol. 7, pp. 283–317, 1997.

[66] S. Jung and S. Pramanik, "HiTi graph model of topographical road maps in navigation systems," in *Proc. 12th IEEE Int. Conf. Data Eng.*, pp. 76–84, 1996.

[67] B. M. Waxman, "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, 1988.

[68] U. W. Eisenecker, F. Blinn, and K. Czarnecki, "A solution to the constructor-problem of mixin-based programming in C++," in *Proc. 1st Workshop on C++ Template Programming*, (Erfurt, Germany), October 10 2000.

[69] J. Siek and A. Lumsdaine, "Concept checking: Binding parametric polymorphism in C++," in *Proc. 1st Workshop on C++ Template Programming*, (Erfurt, Germany), 2000.

[70] R. L. Brooks, C. A. B. Smith, A. H. Stone, and W. T. Tutte, "The dissection of rectangles into squares," *Duke Mathematical Journal*, vol. 7, pp. 312–340, 1940.

[71] G. H. Golub and C. F. van Loan, *Matrix Computations*. Baltimore, MD: Johns Hopkins University Press, 3rd ed., 1996.

[72] P. Eades and N. C. Wormald, "Fixed edge-length graph drawing is np-hard," *Disrete Applied Mathematics*, vol. 28, pp. 111–134, 1990.

[73] A. Kumar and R. Fowler, "A spring modeling algorithm to position nodes of an undirected graph in three dimensions," tech. rep., Department of Computer Science, University of Texas - Pan American, Edinburg, 1994.

[74] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer Series in Operations Research, Springer, 1999.

[75] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, 3rd ed., 1999. See http://www.netlib.org/lapack/.

[76] R. Davidson and D. Harel, "Drawing graphs nicely using simulated annealing," *ACM Transactions on Graphics*, vol. 15, no. 4, pp. 301–331, 1996.

[77] I. F. Cruz and J. P. Twarog, "3D graph drawing with simulated annealing," in *Proceedings of the 3rd International Symposium on Graph Drawing (GD '95)*, pp. 162–165, 1995.

[78] C. Kosak, J. Marks, and S. Shieber, "Automating the layout of network diagrams with specified visual organization," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 24, no. 3, pp. 440–454, 1994.

[79] J. Branke, F. Bucher, and H. Schmeck, "A genetic algorithm for drawing undirected graphs," in *Proceedings of the 3rd Nordic Workshop on Genetic Algorithms and their Applications*, pp. 193–206, 1997.

[80] C. Godsil and G. Royle, *Algebraic Graph Theory*, vol. 207 of *Graduate Texts in Mathematics*. Springer, 2001.

[81] U. Brandes and S. Cornelsen, "Visual ranking of link structures," in *Proceedings of the 7th Workshop on Algorithms and Data Structures (WADS '01)* (F. Dehne, J.-R. Sack, and R. Tamassia, eds.), vol. 2125 of *LNCS*, pp. 222–233, Springer, 2001.

[82] H. D. White and K. W. McCain, "Bibliometrics," *Annual Review of Information Science and Technology*, vol. 24, pp. 119–186, 1989.

[83] U. Brandes and T. Willhalm, "Visualization of bibliographic networks with a reshaped landscape metaphor," in *Proc. 4th Joint Eurographics - IEEE TVCG Symp. Visualization (VisSym '02)*, (Barcelona, Spain), pp. 159–164, ACM Press, 2002.

[84] K. Börner, A. Dillon, and M. Dolinsky, "LVis – digitial library visualizer," in *Proc. International Conference on Information Visualization (IV 2000)*, pp. 77–81, IEEE Computer Society Press, 2000.

[85] C. Chen and R. J. Paul, "Visualizing a knowledge domain's intellectual structure," *IEEE Computer*, vol. 34, no. 3, pp. 65–71, 2001.

[86] G. S. Davidson, B. Hendrickson, D. K. Johnson, C. E. Meyers, and B. N. Wylie, "Knowledge mining with VxInsight: Discovery through interaction," *Journal of Intelligent Information Systems*, vol. 11, no. 3, pp. 259–285, 1998.

[87] A. Brüggemann-Klein, R. Klein, and B. Landgraf, "Bibrelex: Exploring bibliographic databases by visualization of annotated contents-based relations," *D-Lib Magazine*, vol. 5, no. 11, 1999.

[88] D. R. White, J. Buzydlowski, and X. Lin, "Co-cited author maps as interfaces to digital libraries: Designing pathfinder networks in the humanities," in *Proc. International Conference on Information Visualization (IV 2000)*, pp. 25–30, IEEE Computer Society Press, 2000.

[89] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the Association for Computing Machinery*, vol. 46, pp. 604–632, September 1999.

[90] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[91] P. Bonacich, "Factoring and weighting approaches to status scores and clique identification," *Journal of Mathematical Sociology*, vol. 2, pp. 113–120, 1972.

[92] M. M. Kessler, "Bibliographic coupling between scientific papers," *American Documentation*, vol. 14, no. 1, pp. 10–25, 1963.

[93] H. Small, "Co-citation in the scientific literature: A new measure of the relationship between two documents," *Journal of the American Society for Information Science*, vol. 24, pp. 265–269, 1973.

[94] M. Chalmers, "Using a landscape metaphor to represent a corpus of documents," in *Proc. European Conference on Spatial Information Theory*, vol. 716 of *LNCS*, pp. 377–390, 1993.

[95] M. Bertram, B. Hamann, K. Joy, S. Konkle, and H. Hagen, "Terrain modeling using voronoi hierarchies," in *Proc. NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods*, Springer, 2002.

[96] M. Bertram and H. Hagen, "Subdivision surfaces for scattered-data approximation," in *Data Visualization 2001. Proceedings of the 3rd Joint Eurographics and IEEE TCVG Symposium on Visualization (VisSym '01)* (D. Ebert, J. Favre, and R. Peikert, eds.), pp. 55–63, Springer, 2001.

[97] R. Sibson, "A brief description of natural neighbor interpolation," in *Interpreting Multivariate Data* (V. Barnett, ed.), pp. 21–36, John Wiley & Sons, 1981.

[98] "Graph drawing contest." `http://www.ads.tuwien.ac.at/gd2001/`, 2001.

[99] T. Sprenger, R. Brunella, and M. Gross, "A hierarchical visual clustering method using implicit surfaces," Tech. Rep. 341, ETH Zurich, 2000.

[100] N. P. Hummon and P. Doreian, "Connectivity in a citation network: The development of DNA theory," *Social Networks*, vol. 11, pp. 39–63, 1989.

[101] U. Brandes, F. Schulz, D. Wagner, and T. Willhalm, "Generating node coordinates for shortest-path computations in transportation networks," *ACM Journal on Experimental Algorithmics*, vol. 9, no. 1, p. R1, 2004.

[102] D. Tunkelang, "JIGGLE: Java interactive general graph layout environment," in *Proceedings of the 6th International Symposium on Graph Drawing (GD '98)* (S. Whitesides, ed.), vol. 1547 of *LNCS*, pp. 413–422, Springer, 1998.

[103] P. Gajer, M. T. Goodrich, and S. G. Kobourov, "A fast multi-dimensional algorithm for drawing large graphs," in *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000)* (J. Marks, ed.), vol. 1984 of *LNCS*, pp. 211–221, Springer, 2000.

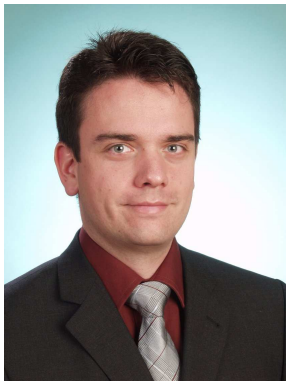
[104] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[105] A.-L. Barabàsi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 357–367, 1999.

[106] U. Meyer, "Single-source shortest-paths on arbitrary directed graphs in linear average-case time," in *Proc. Symposium on Discrete Algorithms*, pp. 797–806, 2001.

[107] A. V. Goldberg, "A simple shortest path algorithm with linear average time," in *Proc. 9th European Symposium on Algorithms (ESA 2001)* (F. Meyer auf der Heide, ed.), vol. 2161 of *LNCS*, pp. 230–241, Springer, 2001.

Index



Thomas Willhalm studied mathematics at the University of Konstanz. He spent the third year of his studies in Grenoble/France. In 1999, he graduate in mathematics with physics as a minor with an diploma thesis on using discrete mathematics to reconstruct the topology of a CAD data model.

In 2000, he started as a Ph.D. student in the Algorithms & Data Structures group in Konstanz. Part of his work was done while he was visiting the Computer Technology Institute in Patras/Greece through the Human Potential Programme AMORE of European Commision. Since 2003, he works for the Department of Computer Science at the University of Karlsruhe (TH) on engineering shortest paths and layout algorithms for large graphs.