Building Blocks for Mapping Services

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Dennis Luxen

aus Frankfurt am Main

Tag der mündlichen Prüfung: 3. Juni 2013

Erster Gutachter: Herr Prof. Dr. Peter Sanders

Zweiter Gutachter: Herr Andrew V. Goldberg, Ph.D.

Für Katrin, Charlotte und Benjamin.

Contents

Acknowledgements

First of all, I wish to thank Peter Sanders for giving me the opportunity to join his group. I am grateful for the advice, inspiring discussions, and support in the past years. Special thanks go to my office mate Dennis Schieferdecker for the great working atmosphere, and numerous discussions on algorithms in general and route planning in particular. I appreciate Dennis' eye for detail while proof reading this thesis and the lively discussions[1] in the course of the conversation. I also wish to thank Christian Vetter for proof reading and providing valuable feedback on select chapters.

I wish to thank my thank my co-authors Julian Arz, Veit Batz, Thomas Brenner, Matthias Duschl, Fletcher Foti, Robert Geisberger, Christian Jung, Daniel Karch, Tim Kieritz, Sebastian Knopp, Moritz Kobitzsch, Sabine Neubauer, Peter Sanders, Dennis Schieferdecker, Antje Schimke, Christian Vetter, Lars Volker, Paul Waddell, Renato Werneck, and Roman Zubkow. It has been fun and very instructive to collaborate on many interesting topics. Although, we did not collaborate scientifically, I appreciated the company of my colleagues Timo Bingmann, Vitaly Osipov, Christian Schulz, Jochen Speck, Nodari Sitchinava and Johannes Fischer very much. I also thank Norbert Berger and Anja Blancani for technical and administrative support. Furthermore, I thank Christian and Timo for fixing the office network when it died all of a sudden 48 hours prior to printing this thesis. The strawberry cake was much appreciated. Also, I'd like to mention my kindergarten friend Alexis Müller-Marbach who introduced me to the wonders of the world of computation when he showed me a computer game on his older brother's Commodore 64 in the early or mid 1980s. An unextinguished fire was lighted that day.

I especially wish to thank my wife Katrin and children Charlotte and Benjamin for reminding me that family, us four, is *the* most important thing, and that there is plenty of life outside of research, too. I wish to thank Deutsche Forschungsgemeinschaft (DFG) for partially supporting my resesarch with grants SA 933/1 and /2. Last but not least, I thank Andrew Goldberg for great discussions on route planning and for agreeing to review this thesis.

[1]Honorary co-authorship was offered at one point but humbly declined.

Abstract

Mapping services have become ubiquitous on the Internet in recent years. These services enjoy a considerable user base. But it is often overlooked that providing a service on a global scale with virtually millions of users has been the playground of an oligopoly – select few service providers are able to do so. Unfortunately, the literature on these solutions is more than scarce. This thesis aims to add a number of building blocks to the literature that explain how to design and implement a number of features for such services.

The core components of a mapping service are map display, route planning and further location-based services that answer sophisticated requests by using and expanding existing data structures and algorithms or by designing new ones. Using simple-minded approaches yields preliminary results fast, but hits a point of diminishing returns soon. At times it may be easy to provide these kind of services on a small scale, i.e. for small data sets and a small number of users. On the other hand, the real algorithmic challenge is scalability. Thus, it is important to provide data structures and algorithms that *scale* with two parameters. First, with the amount of data and second, with the number of users. Scalability can is dependent by the following properties of any solution:

- Queries should be as fast as possible.
- Algorithms should be able to handle imprecision in the input.
- Preprocessing should be fast with moderate space requirements to be able to reprocess on a regular basis.

Unfortunately, these requirements can be contradictory and handling only one aspect fine may not be enough to yield a feasible solution.

Geocode Matching

A mapping service is only as good as the localization mechanism it provides, even with an excellent routing component. When a user does not know the geographic coordinate of a

location, he or she is likely to give a textual descriptions. Address data formats vary wildly across the world, e.g., see [124], and may not even be distinct. In addition, the inherent fuzziness of natural language descriptions, unintended misspellings or simply omitted information are reasons why it is hard for computer systems to handle such inputs. On the other hand users expect the service to handle a certain amount of imprecision. It is this expectation that affects the user experience. In other words, if the localization feature does not provide the quality of service that is expected by the user, one may ask what good a fast routing from a to b is, if it doesn't know a or b.

Route Planning

The arguably most well-known service is route planning, i.e. point-to-point queries. Engineering of shortest path algorithms in road network has seen a large body of research in academia. Current methods are efficient and accurate.

The availability of geographic data has improved dramatically over the past decade. For example, free data sources like OpenStreetMap provide detailed digital maps for certain parts of the world and the GPS chips in omnipresent smart phones can generate an enormous stream of information that resembles the current state of the road network, e.g., traffic jams, construction sites or closed roads. To reflect this constant change of the road network, it is important to efficiently reprocess the road network (or certain parts of it) on a regular basis.

Further Services

Point-to-point queries are most important, but there is also demand for more sophisticated user requests like alternative routes, detour computation for ride sharing or more general queries for points of interests. These queries need to exploit the data structures and algorithms of the underlying routing technique to be scalable. These kind of requests have in common that they need to perform several distance computations at once. The challenge here is not to just run several distinct distance computations in parallel, but to engineer a more elegant solution that exploits for example the structure of already existing data or adds a further preprocessing step.

CHAPTER 1

Introduction

We present several building blocks to build a high-performance map-based service. The order of presentation of the chapters reflects an implementation direction to actually build an online mapping service with a number of advanced features.

1.1. Main Contributions

We describe each contribution in more detail below:

Geocode Matching. The first main contribution is an efficient server-based geocode matching engine that allows fast real-time queries. The central idea is to integrate an approximate dictionary index with the *inherent hierarchical nature of location descriptions*, e.g., of addresses. The underlying approximate dictionary index data structure allows to exploit a space-time tradeoff. In case that the running time of the query is most important, storage space can be traded. We also show that the index is robust against the length of queries. Most importantly, we do not rely on rule-based systems with fixed (locale-dependent) address formats that have to introduce rules to handle special cases, e.g. [107]. This vastly improves the maintainability of such a system. Additionally, the system is able to handle ill-formed queries, i.e. erroneous input with spelling mistakes, without a compromise in query time or solution quality. The geocode matching applies minimum weight bipartite matching, inverse document frequency and natural ranking heuristics to yield very good results in practice. The experimental results show that our approach is at least as good as existing systems, often outperforming them.

We also introduce a graph based method to disambiguate locations that captures the notion of nearness without relying on prefixed distance thresholds. We apply concepts from computational geometry to exploit the inherent hierarchical nature of the descriptions. Again, we conduct an experimental study to confirm our findings.

Distributed Preprocessing. First, we show how to implement a shared-memory preprocessing of Contraction Hierarchies that has a rather low memory foot-print in practice by applying a simple but sophisticated hashing scheme. We also show how to apply the result to build a fast and cache-efficient key-value store that can be used during preprocessing and queries of Contraction Hierarchies. The cost of memory access is not uniform across the memory hierarchy of a modern computer system. It pays off to invest CPU cycles into computation when random memory accesses (and therefore cache misses) can be saved. The experimental results confirm our findings.

Another contribution is a *distributed preprocessing* of Contraction Hierarchies. Although preprocessing of Contraction Hierarchies is reasonably fast on medium-sized networks, e.g., like the size of Western Europe, it bears considerable preprocessing effort if the road network is augmented with additional information like time-dependency, turn restrictions, or turn costs in general. This can take hours for planet-wide networks which is a problem for Internet companies that would like to provide fast turnaround times after an update to the road network. It turns out that the preprocessing of Contraction Hierarchies can be efficiently distributed onto a cluster of cheap commodity machines with good speedups using the message passing paradigm of parallelization. We are also able to significantly reduce the memory requirements for every processing node in the cluster. We also show how to efficiently perform distributed queries in such a setting with a constant number of communication rounds. An experimental study on time-dependent road networks of Germany and Western Europe confirms these results in an experimental evaluation.

We show how to apply our findings to another parallelization paradigm, namely MapReduce. Our findings show that it is technically feasible but suffers from overheads that are inherent to the setting. We develop caching mechanisms that allow us to efficiently work with large graphs while keeping only a portion of the graph in main memory.

Taking Advantage of the Contraction Hierarchies Search Data Structure. The search data structure of Contraction Hierarchies has a number of properties. It is a directed acyclic graph (DAG) and rather flat. In practice it has a height of roughly 100–150 levels. These properties can be used to either answer advanced queries or to efficiently propagate information along the entire network.

The next contribution is the efficient computation of *alternative routes* and *user equilibria* by exploiting these properties. We generate via candidate node sets for feasible alternatives that are small on average and can be applied to a legacy routing engine. This enables further features for users without exchanging the algorithmic core. Additionally, we also show how to compute scorings that are based on the topology of the network and the location of points of interests, as well as the computation of minimum detours for ride sharing.

Ride sharing is a popular way to save resources while travelling. We show how to engineer a matching algorithm that finds best matches with minimized detours. As a further improvement, we show how to perform the matching in an augmented setting where offers (drivers) and requests (riders) are time-dependent. Also, riders are allowed to make transfers between different drivers in this setting.

In real life, the duration of a route depends on many factors. One of them is the expected travel time that depends on the volume of traffic, i.e. daily rush hours. When drivers switch routes to a route with lower expected costs, the traffic patterns can change. We compute user equilibria in road networks, i.e. when unilateral changes would not help any more, by traversing the directed and acyclic search graph in a topological order to propagate information along the road network.

Reconsidering Transit Node Routing. Another contribution is the construction of a Transit Node Routing distance oracle that is solely built on top of Contraction Hierarchies. We settle the open question if such a distance oracle can be built without additional geometric information. The key idea, again, is to exploit the special properties of the Contraction Hierarchies search data structure. Our main insight is that the search spaces are compressible by using *graph Voronoi diagrams*, a method known from computational geometry. As a result, we can efficiently construct a distance oracle with single-digit microsecond query time that is proportional to the preprocessing of the road network.

1.2. Related Work

We now give an overview on the work related to this thesis. The presentation is split into parts to resemble the structure of subsequent chapters.

Approximate Text Dictionaries. The string matching problem is to find matches of an input pattern in a given text quickly. The problem has a number of applications that range from spell checking, to DNA processing, to identifying music from small excerpts among others. Algorithms can be classified into online and offline algorithms. Online algorithms are allowed to preprocess the input text, while offline algorithms are not. The general problem of approximately matching words can be further refined into two categories, namely matching elements from a set of words or matching arbitrary patterns in strings [12]. When a collection of words is given, the problem is known as *dictionary matching*.

A well-known method for accelerating exact matches in large texts is to build the suffix array [90, 132] as a full-text index data structure. It is basically a sorted array of all suffix indexes in lexicographic increasing order. Naive string sorting yields a worst-case construction time of $\mathcal{O}(n^2)$, which is impractical for large texts A well-known and much more time and space efficient algorithm is the *difference cover algorithm (DC3)* of Kärkkäinen and Sanders [110] that recursively sorts triples and runs in linear time. An altogether different principle to construct a suffix array is *induced copying*, which can also be implemented to run in linear time [150]. The induced copying based methods heavily depend on fast random access memory and are inherently sequential. Nevertheless, the currently fastest implementation *divsufsort*[1], which combines highly engineered parallel string sorting and induced copying, outperforms all competitors in practice. It is only published partially.

Approximate string matching is a variant of the problem where one searches for locations of a pattern that allows a number of errors. More precisely, it looks for the location of

[1]`http://code.google.com/p/libdivsufsort/`

approximate occurrences p' of a pattern p such that the distance between p and p' is at most k. The *edit distance or Levenshtein metric* [125] formalizes this distance into a minimal number of atomic operations that transform one string into another. The operations are additions, deletions or exchanges of a single character in a string. Computations of this metric are rather expensive as its asymptotic running time is $\mathcal{O}(n \cdot m)$ for two input strings of lengths n and m. Although approximate search in suffix array (and trees) is technically feasible, e.g. [50], the methods are rather complex and of limited practicability.

For the approximate dictionary matching problem it is natural to find an algorithm that does not compare the input to the entire dictionary but only a few entries. A so-called *filter* represents a criterion to quickly discard large portions of the search space.

The exploitation of the underlying metric space implied by the edit distance [12] is easy. The set of words is partitioned by the distance of each element to a more or less carefully chosen and perhaps random pivot element. By computing the distance to the pivot, the search space is pruned using the triangle inequality. However, this approach has limited effect, e.g. in natural language dictionaries. Distances of most dictionary elements to the pivot lie in a small range.

To cope with the limitations, different schemes were introduced from using multiple pivots to tree-like data structures. The oldest of such trees is the *BK-tree* data structure proposed by Burkhard and Keller [37] which is built recursively. A root is selected whose subtrees are identified by distance values to the root. The i-th subtree consists of elements of the dictionary at distance i to the root. The subtrees are recursively built until the number of elements in a subtree is below some threshold. Again, the triangle inequality is used to branch into or cut any subtrees. A candidate set of possible matches is built by the union of all leaves that are reached by the tree traversal. A rather weak result is that BK-trees and its refinements need $O(n^{\alpha})$, $0 < \alpha < 1$, comparisons and node traversals on average [12] for a dictionary of n entries. See Chávez et al.'s publication [44] for a survey.

More practically oriented work has focused on filtering algorithms that take linear space, but these do not have strong worst case performance guarantees. Kärkkäinen and Na [109] report on a linear space data structure that supports sub-string search, but has much larger query times compared to our result. Ukkonen [182] investigated suffix trees as a building block to solve the problem. Likewise, Cobbs [49] gives a data structure based on suffix trees with linear time preprocessing for a fixed size alphabet for searching fixed patterns. Queries to the data structure can be answered in time $O(mq + occ)$, where m is the length of the pattern, $q \leq n$ and occ is the number of occurrences.

A technique involving so called *q-grams* [181] is popular among practitioners. These q-grams are sub-words of length q and the q-gram distance (or similarity) is defined by the number of q-grams two words share. A generalization of this technique are gapped q-grams. Taking q letters from a word as before and introducing *don't care* positions define a pattern instead of a sub-word. These *don't care* positions are called *gaps* and the entire pattern with *k don't care* positions is called *k-gapped q-gram.* In [38] it is shown that 1-gapped q-grams can be extended to obey the edit distance metric. One of the major difficulties of gapped q-grams is the computation of a threshold which is the smallest number of matching q-grams between a pattern and a text. Most experimental work focuses on finding this threshold, e.g. [39, 108].

As expected for high dimensional search problems, there is a significant space-time trade-off. Cole et al. [50] give a solution for the dictionary matching problem using $\mathcal{O}(n \log^d n)$ space and answer a query in $O(m \cdot \log\log n + occ)$ for a dictionary of size n, query length m, edit distance d. Here, occ is the number of occurrences of the pattern. Mihov and Schulz [140] present a sophisticated but complicated method to solve the problem with universal Levenshtein automata. Russo et al. [166] propose a compressed index that performs well for $d = 1, 2, 3$ but needs several seconds to perform queries for larger d. The best known linear space solution needs $\mathcal{O}(m^{d-1} \log n \log\log n + occ)$ query time [42] for error $d \geq 2$. However, this solution is fairly complicated and involves large constant factors, and to our knowledge there are no implementations yet. Furthermore, any of the general-purpose approximate string matching algorithms have to be adapted to perform dictionary matching: Either the query has to be adapted to ensure that only complete words are found, or special characters have to be introduced to mark the start and end of a dictionary entry. For more information on approximate string matching we refer the interest reader to the following list of publications: [89, 109, 130, 131].

To speed up edit distance computation itself in an on-line setting, research focused on simple and practical bit-vector algorithms [192]. Words of character length n with d or fewer differences can be matched in $\mathcal{O}(nmd/w)$, where w is the word size of the machine an m the length of a query. This is done by computing the bit representation of the current state-set of the k-difference automaton. The run time is improved to $\mathcal{O}(nm/w)$ and even further refinements yield an $\mathcal{O}(dn/w)$ [146] expected-time algorithm for arbitrary large m.

Geocoding. The aim of *geocode matching* or *geocoding* is to automatically translate textual location descriptions into location codes, i.e. geographical coordinates[2]. Geocoded data used to cost several dollars per 1 000 records in the mid-eighties [118] and did not nearly provide the spatial accuracy of today's cost-free services. At that time the use of geographic information systems was limited to professionals only who were aware of the difficulties and limitations of the geocoding process [162]. Nevertheless, basic solutions to the problem have been available in geographic information systems for quite some time [51] with applications for example for route planning, validation of customer addresses [74], or surveillance and management of disease outbreaks like the yearly wave of influenza [119].

Goldberg et al. [88] give a survey on the state of the art of geocoding and identify four fundamental components in the process of geocoding: input, output, processing algorithm, and reference data set. We focus on the processing algorithm in Section 3. The input, i.e. the *query*, is the entity the user wishes to have geographically referenced. The description must contain attributes that have previously been assigned to some datum in the reference dataset that represents the geographic reference. The output usually ranges from simple geographic coordinates to two- or three-dimensional geo-spatial entities like line, polygons, poly-type, or similar.

For example, Goldberg [87] compares eight different geocoders. However, the evaluated Californian addresses are all given with high precision including city name, ZIP code, state

[2]Please note that the term *geocoding* is also used for referencing satellite images to a given cartographic projection.

and street name. User input may be erroneous, incomplete or ill-formatted and improving the quality of geocoding using approximate string matching is considered desirable in [190], but subsequently dismissed as too expensive. Nevertheless, any geocoder should return a result when it finds a perfect match and preferably return also *likely* matches along with a rating.

The literature on geocoding algorithms that match user input against a reference data set is scarce and mostly covers rule-based systems on top of existing database systems. At times it is just referred to as *the algorithm*, e.g. [88, 154]. A common approach is the method of applying perfect matching with a number of *ad-hoc rules* such as common spelling mistakes or well-known abbreviations. Karimi and Durcek [107] give a simple rule-based system that is applied on top of a commercial system. However, there is also a distinction between *deterministic* and *probabilistic* algorithms, e.g. [48, 145]. The first one returns a single an perfect match, while the latter ranks *probable results* according to an internal scoring function that is supposed to reflect the probability of partial matches, e.g. to account for spelling mistakes. The ranking and aggregation function of the United Stated Census Bureau [154] have been published more than 35 years ago. Instead of looking for perfect matches, it gives a scoring function that expresses the probability that a certain entry is a match. Potential matches that reach a certain threshold are accepted. However, a description of how to find matches without scanning the entire data set for each query is not given. Murray et al. [145] report on a hybrid system that is deterministic and combines auxiliary data sources with a simple rule-based system on top of a commercial data base. Experiments on a data set of an administrative district give successful matching rates of up to 80%. Christen et al. [48] apply *hidden markov models* (HMMs) by automatically learning the structure of Australian address data. The pre-trained HMM is able to cope with minor spelling mistakes and to expand common abbreviations, e.g. to expand *St.* into *Street.* Approximate matching is done with a q-gram based index on locality names, i.e. cities only, and the matching is done by a simple rule-based system. The experimental results show that the method achieves a successful matching rate of up to 85%.

Most geocoding services reference postal addresses only as they bear an inherent structure. As it is common in Western countries to locate and navigate by these addresses [71], it is sufficient for many applications to restrict geocoding to this simplified use-case.

Sengar et al. [172, 173] describe a system that is able to handle ill-formed queries to a certain extent by applying heuristics on top of data base SQL queries. Their system does not require any explicit country-specific rule set, but exploits the underlying geometric map data to produce a language independent representation of the data. This kind of abstraction is especially useful in areas of the world where formal address formats are non-existing, e.g., in India. However, it remains unclear if and how such systems scale to large databases at truly continental size. Likewise, Joshi et al. [102] use a statistical machine transliteration system to apply multi-lingual search to a mono-lingual system. Although the results look promising at first, but it remains unclear how they would scale on large data sets.

A problem that is related to geocoding is the *disambiguation of locations.* The system does not only have to find a location that (approximately) matches the input, but has to disambiguate between potential matches. The given name of a place may simple be not unique. Does a query for *Washington* mean Washington, D.C. or the state of Washington.

The simplest solution is to have fixed order of importance on the set of locations, i.e. larger cities have precedence over small towns. The problem with this approach is that disambiguation often has to be situation dependent, e.g. Oakland has the biggest population in the Bay Area, California, but most people associate the area with San Francisco only. Agrawal and Shanahan [6] apply machine learning techniques to build an automatic classifier from previous query logs. Earlier work, e.g. [134, 170], focused around fixed rule-based systems that decided on the assumed importance of a location.

The post-processing algorithms have moved from simple feature assignment to complex interpolation algorithms using diverse data sources. The quality of geocoding highly depends on the underlying data set, e.g. for resolving house numbers or for producing highly accurate output coordinates. Although this has been a major focus of previous literature, e.g. [88], we view this particular problem as orthogonal since once town and street have been correctly identified, we are dealing with much more local data and hence much smaller data volumes. For example, searching an odd house number not present in the database can often be done by a binary search in a sequence of the known odd house numbers followed by an interpolation, e.g. [69, 88].

Dijkstra's Algorithm and Related Speedup Techniques. Finding shortest paths in graphs has been solved by Dijkstra's algorithm [65] in the late 1950s and is taught in virtually every computer science course on data structures and algorithms. It searches radially and in iterations around the source node and computes the shortest path distance to all nodes. Note that *shortest* does not necessarily reflect the spatially shortest path, but depends on the chosen edge weights, e.g. representing travel time. It is not necessary to run the algorithm until all distances have been computed for a point-to-point query and thus it can be stopped as soon as the target node has been settled.

Dijkstra's algorithm keeps track of the *tentative distance* for each node. Initially, this distance is ∞ and the algorithm *settles* the nodes in increasing distance from the source node. The running time of the algorithm is obviously polynomial, but it does not scale well on large instances. The algorithm usually relaxes many more nodes than just the nodes on the shortest path.

A naive implementation scans the tentative distance array to find the current minimum in each iteration and runs in time $\mathcal{O}(n^2)$. It is more efficient to use a *priority queue* to keep track of the minimum, though. Much of the theoretical research focuses on priority queues to speed up Dijkstra's algorithm. For example, using binary heaps [189] improve the run time to $\mathcal{O}(n \log n)$. Fibonacci Heaps [77] improve it to time $\mathcal{O}(m \log \log n)$, while radix heaps [47] improve it further to expected time $\mathcal{O}(m + n(\log C)^{\frac{1}{3}+\varepsilon})$, where C is an upper bound on the weight of any edge. In practice, it often suffices to use basic priority queue implementations. This is especially true in the following for the *speedup techniques* to Dijkstra's algorithm that have much smaller queue sizes. A slow-down factor of only 2–3 was reported earlier by Cherkassky et al. [46] which was later also confirmed by Schultes [171]. Further results have been found for certain graph classes. Linear time algorithms are known for planar graphs [97] and undirected graphs [180]. Furthermore, linear running time with high probability is known for graphs with random edge weights that are drawn uniformly at random from the $[0, 1]$ interval [85, 139].

Point-to-point queries can be accelerated by *bidirectional search*. The search is started from source and target simultaneously until a common node has been settled in both directions. Then the shortest path distance and a corresponding path can be extracted from the data in both queues and the tentative distance array [52]. Bidirectional search accelerates the query in (nearly) planar graphs by roughly a factor of two because of the following reasoning. Consider the search space to be a circle. A single (unidirectional) circle encompassing source and target has about twice the area as two circles around source and target respectively that meet in the *middle*.

A known heuristic from the artificial intelligence community is A^* [95, 156]. Instead of searching uniformly around source (and target) the search is guided by a heuristic that tries to explore nodes only that lead *closer* to the target. It does so by computing a lower bound on the shortest path distance to the target for each node. Instead of selecting the node in each iteration with the shortest distance, it selects the node that minimizes the sum of shortest distance and the lower bound. The method does not give any worst-case guarantees for general graphs. But moderate speedups are observed in practice. There are also reports in the literature [86] that this method can lead to slow-downs as well if the heuristic does not deliver proper guidance.

Route planning in road networks has seen a lot of results from the Algorithm Engineering community in recent years. A number of techniques have been proposed that preprocess the static input graph and achieve significant speedups over Dijkstra's baseline algorithm. An early technique that provides substantial speedups is *arc flags*, also called *edge flags*; originally conceived by Lauther [122, 123]; later by Möhring et al. [141] and Köhler et al. [98, 117]. It is a *goal directed* technique. The road network is partitioned into *cells* and each edge stores flags to indicate if there exists a shortest path over it into a certain region.

The arc flags query is an extension of Dijkstra's algorithm that checks for each relaxed edge if the flag for the targets region has been set. One problem of this query is the *coning* effect. The arc flags do not provide any guidance once the search reaches the target region and the entire region is searched. Bidirectional search is able to cope with this effect when flags for the reverse graph are available. The reasoning is that the two searches are likely to meet *in between* before the coning actually happens.

The naive preprocessing algorithm is to run a one-to-all search from every node which results in an all-pairs shortest path (APSP) computation. For large graphs this is simply infeasible. A straight-forward improvement is to conduct the one-to-all searches only from cell boundary nodes, i.e. those nodes that are adjacent to nodes in other cells. While this is substantially faster, the preprocessing is still beyond any practicability for large graphs. Hilger et al. [98] provide a *Centralized Shortest Path Algorithm* that constructs a shortest path tree for all boundary nodes of a region simultaneously by storing not only a single label, but a label for each boundary node. The preprocessing is still quite pricey and takes about 17 hours and much RAM on a graph representing Western Europe with about 18 million nodes. Hilger et al. [98] report on a speedup factor of more than 2 500 over Dijkstra's algorithm for the query. Only recently, Delling et al. [58] provide an algorithm that exploits the hierarchical properties of Contraction Hierarchies (explained in Setion 2.5), that is able to compute arc flags in a matter of minutes on GPU hardware.

Searching for shortest paths in a road network has the advantage that the underlying graph has an inherent *hierarchy*, i.e. some edges are more important than others. For example, a segment on a highway carries more traffic than a dead-end street in a residential area. This fact is exploited in *hierarchical speedup techniques* to Dijkstra's algorithm to accelerate shortest path computation. *Highway Hierarchies (HH)* [168] is an exact method that captures this inherent hierarchy. It was the first technique to efficiently handle realistic road networks of continental size. The road network is said to consist of several levels. However, the importance of nodes and edges in the graph is determined automatically in a preprocessing step. This step alternates between two basic procedures. First, *edge reduction* removes so-called *non-highway edges* that would only appear on shortest paths close to source and target. Second, *node reduction* or *contraction* replaces nodes with low degree by inserting *shortcut* edges that preserve shortest path distances in the remainder of the graph. The road network shrinks geometrically during the preprocessing while it remains nearly planar and sparse. The query is essentially a bidirectional Dijkstra run with the exception that certain edges are pruned from the search when it is sufficiently far away from source and target. The reported speedups over Dijkstra's algorithm are substantial with reported average query times of around 1 millisecond.

Contraction Hierarchies (CH) by Geisberger et al. [84] is a simplification of HH that relies only on node contraction. The nodes are ordered by some measure of importance and contracted in that order. Here, *contracting* means that nodes are (temporarily) removed one-by-one from the graph and replaced by shortcut edges to preserve shortest path distances in the remainder of the graph. The resulting data structure is the union of the original edges and the created shortcuts with the property that edges only lead to more important, i.e. later contracted, nodes. Note that this graph is directed and acyclic. The query is a bidirectional variant of Dijkstra's algorithm with the only crucial difference being a modified stopping criterion. It stops relaxing edges if and only if the tentative distance of a node exceeds the upper bound that may exist for the shortest path. A detailed description of the CH preprocessing and query is given in Section 2.5. Bauer et al. [24] investigate combining goal-directed techniques and hierarchy-based method. The fastest of these methods is *CHASE* where the CH query is pruned by arc flags. CHASE queries run in the order of ten microseconds.

Route planning has been generalized to time-dependent edge-weights that model recurring events, e.g. rush-hours. The baseline algorithm is a *label-correcting* version of Dijkstra's algorithm [54], that is able to reinsert settled nodes. Delling [56] reports on a combination of shortcuts and arc flags *(SHARC)* that achieves speedups of more than 75 depending on the amount of time-dependent edge weights. Batz et al. [21] give a generalization to *time-dependent Contraction Hierarchies (TCH)* that achieve speedups over the label-setting Dijkstra variant of more than 1 200. The travel-time functions for shortcuts in TCH take up a considerable amount of space. Batz et al. [20] give a carefully designed approximation for these functions that uses an order of magnitude less space. Interestingly, shortest path queries remain exact with only a small slow-down in query time.

The reasons why speedup techniques to Dijkstra's algorithm perform so well on road networks is analyzed by Abraham et al. [2]. A graph is said to have low highway dimension if there exists a sparse set of nodes S_r for every $r > 0$ such that every shortest path longer than r includes a node from S_r. Road networks are said to have a low *highway dimension*, and a unifying preprocessing algorithm for several speedup techniques is given. We also refer the reader to a survey by Sommer [176] for an extensive report on the state of shortest path computation for certain graph classes.

Points of Interests. Searching for points of interest (POIs) is one of the major applications in *location based services (LBS)* and has been previously solved by using k-nearest neighbor point data structures in Euclidean space, e.g. by using 2D-trees [27], quad-trees [167] in main memory, or R-trees [93, 165] in external memory, among others. These systems search radially in euclidean space, which is inadequate for distances in a road network, e.g. consider a river that *naturally cuts* [60] the road network with bridges being far away. The Euclidean distance may yield a POI that is just across the river where the network distance may give entirely different locations. This is especially an issue when the edge weights in the road network represent travel times.

Searching for nearest neighbors in a road network can be solved by searching radially around a given location with a unidirectional Dijkstra, or given unit-distances with a breadth-first search. Unfortunately, there is no performance guarantee to the number of settled nodes in these searches, e.g. the entire network is searched if only $k-1$ POIs exist in total. Naive heuristics therefore prune the search when a threshold on the distance has been reached. Jensen et al. [101] report on a framework to support *dynamic* k-nearest neighbor queries of moving objects in a road network. Location information is mapped onto a graph and a search yields candidates that are subsequently verified. However, shortest path distances between the nodes of the network are precomputed and the space overhead is infeasible for large-scale instances. The problem of finding POIs can also be solved by exploiting the structural properties of a speedup technique to Dijkstra's algorithm. Geisberger [79] reports on a natural solution of storing the information of backward search spaces in buckets at nodes of the graph to allow accelerated nearest neighbor queries in a road network that obey the actual network distance without searching radially. In this setting, the number of visited nodes during a POI search does not depend on the distance to the POI but on the search space size of a node. A more formal introduction is given in Section 5.2. A similar approach is taken by Abraham et al. [1] with *HLDB* and applied to relational data base system that efficiently stores *label sets*, i.e. preprocessed search spaces. Union and intersections of search spaces are identified as the building blocks for most LBS systems available through the programming API of the data base, e.g. SQL. Queries can be evaluated in a matter of milliseconds depending on the underlying hardware of the data base system, i.e. traditional hard drives versus SSDs. However, the data base is rather static as reloading the label poses a significant overhead and requires preprocessing of the entire road network.

Rice and Tsotras [161] develop an algorithm that finds a shortest path from source to target that passes through at least one location from each of a predefined set of categories, e.g. ATMs, gas stations, post boxes, etc. Their general problem is a variant of the well-known Traveling Salesman Problem and likewise $\mathcal{NP}$-hard. Nevertheless, their algorithm has running time exponential in the number of categories, which is generally low, and is able to solve practical instances within a matter of seconds.

Alternative Routes. Computing alternative routes in road networks has been studied before for a long time. Yen [196] and Eppstein [73] investigate the k-shortest path algorithm. Unfortunately, reasonable alternatives are usually not among the first few hundred or thousand shortest paths. Also, these algorithms are not fast enough to be considered practical. Chen et al. [45] apply the *penalty method* that iteratively computes shortest paths while increasing a number of edge weights. Another natural approach is to apply multi-criteria optimization to combine two or more edge length functions [62, 94, 135]. For example, a variant of CH provides routes [80] that are optimal with respect to flexible objective functions, i.e. user preferences, but does not necessarily deliver reasonable alternatives.

Alternative paths that combine two shortest paths over a *via node* are used by *Choice Routing* [40], also called *Plateau Method.* Shortest path trees are grown from origin node s and in reverse from target node t. *Plateaus* $\langle u, \ldots, v\rangle$ running from node u to v are maximal paths that appear in both trees. They give candidates for natural alternative paths, i.e. follow the forward tree from s to u, then the plateau, and then the reverse tree from v to t. Although not entirely published, the plateau method provides alternatives of good quality in practice. Further discussion on this can be found in the work of Abraham et al. [5].

Bader et al. [11] introduce the generation of alternative graphs, which is an extension of the penalty method of Chen et al. [45]. Such graphs are a sparse subset of the road network that encode a number of feasible alternative at once.

Ride Sharing. Previous research on ride sharing focused on multiple areas. Several authors [66, 96, 151] investigated the socio-economic prerequisites of wide-spread customer adoption and overall economic potential of ride sharing. For example, Hartwig and Buchmann [96] analyze the ride sharing business case given that there exists a central service platform that can be accessed by mobile devices.

Other authors [194, 195] identified the missing spatial resolution of concurrent ride sharing services and examined a sensor network approach to metropolitan-local ride sharing offerings. Hand-held mobile devices function as nodes of the sensor network and communicate locally over short distances. Unfortunately, the work focuses on heuristic communication strategies. Likewise, no performance guarantees are possible and rides are only matched heuristically. Matching is done greedily and the first ride to go geometrically closer to the destination is taken. Note that geometric routing might lead to arbitrarily bad routes on a road network in the worst case.

Xing et al. [195] give an approach to ad-hoc ride sharing in a metropolitan area that is based on a multi-agent model and show the validity of their approach by simulation on a rather small metropolitan network. But in its current form the concept does not scale. As the authors point out it is only usable by a few hundred participants and not by several thousands or more participants that a real world ride sharing service would have.

Recently, Abraham et al. [1] provide an implementation of the single-hop ride sharing algorithm of Section 5.3.1 based on hub labeling using a technique called *double-hub indexing.* The query time does not depend on the number of offers, but on the square size of the label set of source and target, which can be asymptotically less work.

The Traffic Assignment Problem. The traffic assignment problem (TAP) has been studied for several decades. It studies the selection of routes between sources and targets and is a step in the traditional travel forecasting model. It is therefore an important tool on policy making and transportation planning. The first mathematical formulation of the TAP is generally attributed to Beckman et al. [26]. It was first given in 1956 and models the TAP as an optimization problem that seeks to find an *equilibrium*. A general behavioral assumption in the field of transportation science is that each traveller or vehicle in a road network takes a path that has least cost (or is at least perceived as such). It is further assumed that travel time is the most significant utility for route choice. We recognize the over-simplification of this model, but direct the reader to the literature on empirical research of route choice, i.e. [160].

Travellers on a road network are said to be non-cooperating and seek to minimize travel time (or any other metric) and can be understood to act as selfish agents. They switch to better routes if they become aware of it. A state where no driver can find a better route by a unilateral decision is called an equilibrium. The state of this equilibrium is the aggregate result of individual decisions and therefore also goes by the name of *user equilibrium*. It is generally assumed that under equilibrium conditions all used routes for the same origin-destination pair have the same costs, e.g. equal travel time. Also, unused routes between any origin-destination pair have higher costs than used ones.

The method of choice to solve this problem is the Frank-Wolfe-Algorithm [133], which is also known as the *convex combinations algorithm*. It was originally invented to solve quadratic programming problems. Over the years it has been applied to the traffic assignment problem, mainly because of its rather simple structure. Occurrences in the literature go back to the late 1960s [34, 144]. The major advantage of the Frank-Wolfe-Algorithm (besides its simplicity) is its low memory consumption. For example, it does not save any information on computed routes. It only counts the volume of traffic on each individual street segment. This was considered a major advantage in the early days of computation because of limited memory capabilities. The algorithm alternates between an assignment phase of the traffic demand and a minimization step to numerically approximate edge flows.

The textbook of Sheffi [174] gives an overview of the first three decades of research between 1950 and 1980. Most of the solution techniques described are still in use by practitioners today. Usually they are applied to road networks of small and medium size up to several hundred or a few thousand edges and often only on sparse subsets of highway networks which are much smaller than the full road network.

Previous work focusses on speeding up convergence of solving the traffic assignment problem by modifying the way traffic flow is distributed during the computation. For example, Bar-Gera [14] presents an algorithm to compute the equilibrium by paired alternative segments. If flow between two nodes splits into separate sub-paths then flow is shifted proportionally.

A completely different model to solve the traffic assignment problem is to apply game theory. Rosenthal [163] was the first to consider the problem by a game theoretic approach. A so-called congestion game is defined by a set of players that compete for one or more shared resources. It is said to be symmetric if all players chose among the same set of strategies. Fabrikant et al. [75] show that any symmetric congestion game can be solved in polynomial time. Relating to our case the players are travellers that compete for roads and seek to minimize travel expenses. Edge latencies, i.e. the time it takes to traverse a road segment, are defined to be non-negative, continuous and non-decreasing functions of the amount of travellers on that edge. Consider building up a given traffic flow on a road from zero flow, one infinitesimal flow path at a time. The potential function is obtained by integrating the latency experienced by each infinitesimal flow path, using edge latencies that were in effect at the moment it was routed. This potential can be easily optimized to a (local) minimum by allowing players to switch their strategy, which is a shortest route in this case. These switches are called selfish steps. Consider an improving move of one of the players to a better route. It is easy to see that the potential is lowered and that it can be brought to a local minimum by subsequent switches until no switch to an improved route for any player is possible.

Kirschner et al. [113] apply book keeping heuristics to avoid many path computations and subsequently speed up the rate of convergence on networks with less than a few thousand nodes and edges. For an excellent survey over the literature for congestion games and algorithmic game theory in general see the textbook of Nisan et al. [149] and especially Roughgarden's seminal work on the Price of Anarchy in routing games [164]. Note that the game theoretic approach prohibits any precomputation based on the metric of the edge weights, because edge weights change with every move. A single selfish step might change the edge weights enough to invalidate the preprocessed data structures and preprocessing the network for a single query is out of the question. The application of the Frank-Wolfe-Algorithm as well as the game theoretic solution need a method of path finding. Plain solutions spend virtually all of the computational effort in path finding.

To the best of our knowledge there is no publication that reports on directly exploiting inherent properties of a speedup technique to accelerate traffic assignment computations.

Transit Node Routing. Transit Node Routing (TNR) by Bast et al. [18] is one of the fastest speedup techniques for shortest path distance queries in road networks. By applying an additional, second step of preprocessing, TNR yields almost constant-time queries, in the sense that nearly all queries can be answered by a small number of table lookups. It follows an intuition: Long distance connections almost always enter an arterial network connecting a set of important nodes the *transit nodes*. Once these nodes are identified, a mapping from each node to its access nodes and pairwise distances between all transit nodes are stored.

Many TNR variants have a common drawback that preprocessing time for TNR is significantly longer than for the underlying speedup technique. Another weakness is that the locality filter requires geometric information on the position of the nodes [16, 17, 84]. The presence of a geometric component in an otherwise purely graph theoretical method is regarded as awkward. There are several examples of geometric ingredients in routing techniques being superseded by more elegant and effective graph theoretical ones, e-g- see [171], with the locality filter of TNR being the only *survivor* that is still competitive. Recently, Abraham et al. [2] give further analysis of TNR in the context of Highway Dimension that yields a natural locality filter. Further technical details on TNR are given in Section 6.

Geisberger et al. [84] uses CH to define transit node sets and for local searches, but still uses a geometric locality filter and relies on Highway Hierarchies [168] for preprocessing. In lecture slides [15], Bast describes a simple variant of CH-based preprocessing which explores a larger search space than ours and which also computes a super-set of the access nodes only because it omits post-search-stalling. No experiments are reported. The geometric locality filter is not touched. In Section 6.2 we remove all these qualifications and present a simple fully CH-based variant of TNR which yields surprisingly good preprocessing times and allows for a very effective fully graph-theoretical locality filter.

A related technique to TNR is *Hub Labeling (HL)* by Abraham et al. [3] which stores sorted CH search spaces, intersecting them to obtain the distance. Using sophisticated tuning measures this can be made significantly faster than TNR since it incurs less cache faults. However, HL need much more space than TNR then. There exist a number of variants of HL [3, 4, 61] that each introduce a specific trade-off between preprocessing time, space and query efficiency. A hierarchical variant [4] introduces a trade-off between query time and preprocessing efficiency as well as overall better space consumption. Recently, a *compressed* variant has been published by Delling et al. [61] that further reduces the memory overhead while having reasonably fast preprocessing times.

CHAPTER 2

Preliminaries

This Chapter introduces the basic data structures and algorithms alongside the notation used throughout this thesis.

2.1. Graph Theory

A graph $G = (V, E)$ models relationships between entities. It consists of a finite set of nodes V and also of a finite set of edges E. An edge $e = (u, v) \in E$ is a pair of nodes. If this pair is ordered, then the graph is directed. Otherwise it is undirected. The number of nodes is denoted by $n := |V|$, while the number of edges is denoted by $m := |E|$. As degree $deg(v)$ of a node v we denote the undirected number of edges incident to node v. $deg_{in}(v)$ ($deg_{out}(v)$) gives the number of directed edges leaving at (ending in) v. A graph is said to be *sparse* if $m = \mathcal{O}(n)$. A reverse graph $\bar{G}$ is obtained by replacing every (directed) edge $(u, v) \in E$ with (v, u).

Edge Weights and Functions. Each edge $e \in E$ is associated with a *length*, or more precisely with a non-negative cost $l(e)$: $E \to \mathbb{R}_0$ that indicates how expensive it is to traverse that edge, e.g., the travel-time necessary to traverse the edge. In static graphs the edge weight is constant. Note that in undirected graphs the edge weights are symmetric, i.e. $l(u, v) = l(v, u)$, whereas in directed graphs we can have $l(u, v) \neq l(v, u)$.

Time-dependent network instances do not have static edge weights, but rather a travel time function $f_e(\cdot) : E \times \mathbb{R} \to \mathbb{R}$. The weight function $f_e(\tau)$ specifies the travel time at the endpoint of an edge when the edge is entered at time τ. For our purposes, the edge weight functions are given as piece-wise linear functions that support three operations:

- Evaluation: Given a function f and a departure time τ, compute $f(\tau)$. A query on a naive implementation with binary search takes time $\mathcal{O}(\log n)$, whereas a query on an implementation with a bucket data structure takes expected time $\mathcal{O}(1)$.

- Linking: Given two adjacent edges $e_1 = (u, v)$ and $e_2 = (v, w)$ with weight functions f and g compute the *linked* edge function $g \circ f$, i.e. g *after* f, of the path $\langle u, v, w \rangle$. The resulting function gives $g(f(\tau) + \tau), \forall \tau$. This can be computed in $\mathcal{O}(|f| + |g|)$.
- Minimum: Given two parallel edges e_1 and e_2 with weight functions f and g compute the minimum travel time function $f' := \min\{f(\tau), g(\tau)\} : \forall \tau$ that *merges* these edges while preserving the shortest paths. This can be computed in $\mathcal{O}(|f| + |g|)$.

Thus, time-dependent edge functions are treated as abstract data types. It is further assumed that each edge obeys the FIFO property:

$$(\forall e \in E)(\forall \tau < \tau') : \tau + l(\tau, e) \leq \tau' + l(\tau', e) \ .$$

In other words, one can not arrive earlier by starting the journey later. For an in-depth overview on the modelling of time-dependent networks see the seminal article of Orda and Rom [153] on the subject.

Paths. A path $P = \langle s, v_1, v_2, \ldots, t \rangle$ in G is a sequence of nodes such that there exists an edge between each node and the next one in P. Let $p := \langle v_1, \ldots, v_k \rangle$ (or $p := v_1$–v_k) be a connected path between nodes $v_1, v_k \in V$ with $(v_i, v_{i+1}) \in E, (\forall i : 1 \leq i < k)$.

The length of a path $l(P)$ is the sum its edge weights. A path with minimum cost between $s, t \in V$ is called a *shortest* path and denoted by P_{st} with cost $\mu(s, t)$. Note that a shortest path need not be unique. A path $P = \langle v_1, v_2, \ldots, v_k \rangle$ is called *covered* by a node $v \in V$ if and only if $v \in P$. A node set $V' \subseteq V$ covers a set of paths $\mathcal{P}$ if and only if $P \cap V' \neq \emptyset : \forall P \in \mathcal{P}$. A graph is said to be strongly-connected if for every pair of nodes $s, t \in V$ there exists a path between s and t in G and a directed graph without any cycles is called a DAG. The *hop distance* between node $u, v \in V$ is the minimum number of edges of a path between u and v. If such a path does not exist, the distance is ∞.

Edge-Expansion. An *edge-expanded* graph is a directed graph that explicitly models all possible turns (u, v, w). The input graph to an edge-expanded graph is called node-based graph. A node-based graph resembles roads between junctions, while the edge-expanded graph resembles turns between roads. Thus, each directed edge of the node-based graph becomes an edge in the edge-expanded one. Likewise, the edges between edge-expanded nodes are turns. The construction of an edge-expanded graph is straight-forward. Turn-costs and turn restrictions can be modelled into the graph by adding them to the edge that models the turn or by omitting the edge entirely. As as rule of thumb, we note that edge expansion increases the graph size by about a factor of 3–4. See [185] for a description of the construction and an experimental results on memory requirements and query efficiency.

Partitions. A family of blocks, or cells, $\mathcal{C} = \{C_1, \ldots, C_k\}$ is a partition of V if every node $v \in V$ belongs to exactly one element of $\mathcal{C}$. A partition of the graphs node set V is a family of subsets. The *cut* of a partition is the number of edges which connect distinct cells, i.e. edges (u, v) for which $u \in C_i, v \in C_j, i \neq j$. A partition of a subset of V is call sub-partition.

A second-level partition is a family of cells $\mathcal{C} = \{C^0, \ldots, C^l\}$ that exists for each $1 \leq i \leq k$ such that $C_i = \bigcup_{0 \leq j \leq l} C^j$. If defined recursively over several layers, it is called a multi-level partition. A graph G is said to be bipartite if its node set $V := V_1 \cup V_2$ admits a partition such that every edge $e \in E$ connects the two different cells V_1 and V_2.

2.2. Further Conventions.

We model a road network as a directed graph $G = (V, E)$. V is a set of nodes, e.g., junctions in the road network. Edges $(v, w) \in E$ connect all pairs of nodes v, w for which there is a direct connection in the road network. Note that nodes need not necessarily be junctions, but can also be used to model the geometry of the road.

When a logarithm is given without base we assume that it is the logarithm to base 2, i.e. $\log n \equiv \log_2 n$.

2.3. Data Structures and Algorithms

We now present basic facts about data structures and algorithms that are used throughout the course of this thesis.

2.3.1. Priority Queues

A *priority queue (PQ)* is an abstract data type that maintains a set of elements E and each element $e \in E$ is associated with a key that resembles the elements importance. The data structure supports a number of basic operations:

- `insert`(e, k) inserts element e with priority key k into the PQ.
- `min`$()$ yields the element with minimum priority.
- `delete_min`$()$ removes the element with minimum priority from the PQ.
- `size`$()$ returns the current number of elements.

Addressable PQs allow an important additional operation:

- `decrease_key(k,v)` decreases the priority of key k to v.

Note that the `min` operations can also be implemented as `max` operations by inverting the sorting order. And depending on the implementation, there may be further functionality like bulk insertions, and merging of existing PQs. The implementation of a PQ determines the asymptotic running time of each of the operations. Naive implementations store the data in a simple array. If the array is sorted, then `delete_min` and `min` operations can be done in $\mathcal{O}(1)$, but insertion requires a linear scan through the array, requiring time $\mathcal{O}(n)$. Likewise, storing the data unsorted allows constant time inserts, but searching the minimum element requires a linear scan.

More sophisticated implementations keep the data in a *partially sorted* order. For example, *binary heaps* keep the data sorted in heap order [189], i.e. when n elements are stored in an array $h[1 \dots k]$:

$$h\lfloor i/2 \rfloor \leq h[i], 1 < i \leq n \ .$$

The `min` operation runs in time $\mathcal{O}(1)$, while `delete_min`, `decrease_key` and `insert` run in time $\mathcal{O}(\log n)$ in the worst-case. While there are a number of PQ data structures that have better asymptotic run times, e.g., like Fibonacci Heaps [77], the simple structure and low overhead are reasons why binary heaps are used often in practice.

2.3.2. Dijktra's Algorithm

Dijkstra's seminal algorithm [65] computes the shortest paths from a given source node s to all nodes in a directed graph with non-negative edge weights. The algorithm keeps track of distances in an array, where it stores the *label*, i.e. the distance from the source, of each node. It searches *radially* around the source and stores the current search horizon in a priority queue. In each iteration, it *settles* the node u with minimum distance from the source, i.e. removes the minimum from the queue. The edges $(u, v) \in E$ are *relaxed* and if $\mu(s, t) + l(u, v) <$ distance of v the entry in the distance array is updated. If v was entered into the priority queue previously, its key is decreased, otherwise it is inserted into the queue. See Listing 2.1 for pseudo-code representing the algorithm, given a global array `distances` where all entries are set to ∞.

If only the shortest path (distance) from a source node s to a specific target t is sought, the algorithm can be aborted as soon as t is settled. All distances updated after settling

Listing 2.1: Dijkstra's Algorithm

```
function dijkstra(s) do
  distances[s] = 0
  q.insert(s,0)
  while(!q.empty()) do
    (u,d) = q.delete_min()
    for all edges $(u,v) \in E$ do
      new_distance = d+$mu(u,v)$
      if(new_distance < distances[v]) do
        bool was_inserted = (distances[v] != $\infty$)
        distances[v] = new_distance
        if(was_inserted) do
          q.decrease_key(v,new_distance)
        else
          q.insert(v, new_distance)
        end
      end
    end
  end
end
```

t will be higher than `distances[t]` and therefore not change the result. This is called a *point-to-point* (or *one-to-one*) query. The computation can be accelerated by *bidirectional search* [52]. The search is started simultaneously from source and target, where the *backward* search runs on the reverse graph. Once a common node is settled in both search directions, no further edges are relaxed any more and the shortest path can be computed by the already gained information (inside the queues). A *search tree* is the tree that is gained by noting the tree of parent pointers when running a unidirectional search from a node s.

As argued in Section 1.2 most of the theoretical work focused on the implementation of the priority queue. But road networks are sparse, i.e. $m = \Theta(n)$, while the number of queue operations Dijkstra's algorithm is obviously bounded above by $\mathcal{O}(n+m)$. Thus, even binary heaps yield a total asymptotic run time of $\mathcal{O}(n \cdot \log n)$ for these instances. *Speedup techniques* to Dijkstra's algorithm as introduced in Section 1.2 aim to minimize the number of queue operations which make the actual implementation of the PQ data structure even less important.

A^* [95] is a heuristic improvement to Dijkstra's algorithm for point-to-point queries. The search is guided towards the goal by following the path of lowest expected total cost. Each node v is associated with two cost functions:

- $g(v)$ is the (tentative) distance from the source.
- $h(v)$ is a lower bound of the distance to the target, also called *potential*.

Function $h(v)$ must be an *admissible* heuristic. A node's potential must not overestimate the distance to the target, i.e. $\mu(s, v) + h(v) \leq \mu(s, t), \forall v \in V$. The order of the priority queue is then given by the sum $f(v) = g(v) + h(v)$. An example for such an heuristic is the euclidean distance which never overestimates the shortest path distance in a road network (with distance metric).

A^* finds the optimal path if it exists, but it does not have strong worst-case guarantees. Its performance depends on the accuracy of the heuristic. Goldberg and Harrelson [86] report on a number of variants of a sophisticated potential function, based on landmarks and the triangle inequality. The observed performance is roughly an order of magnitude higher than Dijkstra's algorithm depending on the test instance.

2.3.3. Associative Arrays and Hashing

An associative array [137] stores a set of elements. Each element e is associated with a key $k \in K$ from the set of possible keys. The associative array A supports the following basic functionality:

- `A.insert`(k, v) *adds* the value v identified by its key k into A.
- `A.find`(k) *searches* if A holds an entry identified by k and returns it.
- `A.delete`(k) *removes* the key-value pair identified by k from A.

A way to implement associative arrays is to use a *hash function* $h : K \to S$ that maps the set of keys K to a set of *hash values* S with $|S| \leq |K|$ and $S := [0, \ldots, m-1]$. This range is sometimes abbreviated with $[m]$. A *collision* occurs when distinct keys get mapped to the same hash value. The collision has to be resolved. A number of approaches have been developed to resolve collisions. *Hashing with chaining* stores data in an array A_c of linked lists. The above operations can be implemented easily. Insertions of elements e are done by adding e to the list of $A_c[h(e)]$. This can be done in constant time. Deletions are done by scanning $A_c[h(e)]$ until the element is found and then removed from the linked list. The find operation also scans $A_c[h(e)]$. Scanning takes time linear in the size of the linked list, which is linear in the total number of elements of elements in the worst case. While one could expect a much better average case in practice this is not better than storing linear lists of elements in the first place.

Hashing with (Linear) probing stores data directly in an array A_p of elements (not lists) and tries to resolve collisions by finding free spots in A_P instead of managing lists. A simple variant is the following that we describe for a basic overview of the technique. An element e is stored at $A_p[h(e)]$ or at a subsequent position. The implementation of `insert()` works similar to `find()`. Array A_p is scanned linearly beginning at $A_p[h(e)]$ until the first free spot is found. The search can be aborted as soon as a free entry is found. Implementing the `remove()` operation is trickier. We describe a rather simple solution to give a sense for the problem. An entry cannot be simply deleted, because depending on load of the table and insertion order it may break searching for elements. A simple fix is to mark elements as previously used, but deleted. Periodic reorganizations help to manage the amount of free/deleted/used space.

2.4. String Processing

A string $s := s_0, \ldots, s_k$, $s_i \in \Sigma$ of length k is a sequence of characters over an *alphabet* Σ that is the set of all possible characters. The length of a string s is the number of it characters and we denote the i-th character of string s by s_i. An *empty string* is denoted by ε. It is seen as an abstract data type that allows certain operations to get information on a string or to transform it.

We define a number of basic operations on strings: Two strings s and t are said to be *equal* if $|s| = k = |t|$ and $s_i = t_i, (\forall i \leq k)$. *Character replacement* transforms s into s' and exchanges the i-th character $s_i \in \Sigma$ of string s by $s'_i \in \Sigma$ where $s_i \neq s'_i$ such that $s' := s_0, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, s_k$. *Character deletion* transforms s into s' and removes the i-th character from s such that $s' := s_0, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k$. *Character addition* transforms s into s' and adds character $a \in \Sigma$ at position i such that $s' := s_0, \ldots, s_i, a, s_{i+1}, \ldots, s_k$.

2.4.1. Tokenizing and Inverted Indices

A text is an ordered set of strings where the strings are delimited by white space or delimiter characters. *Tokenizing* a text or input stream means that it is broken up into words or symbols, e.g., the text of a novel is tokenized by noting all its words. An *inverted index* is a simple index data structure that is useful when tokens appear several times in the text, e.g., in natural language texts. It is a mapping of tokens to the texts in which they appear. Consider the following example:

Example 1 (Inverted Index.). Two texts are given*:*

T[0]	=	*to be or not to be*
T[1]	=	*be there or be square*

The corresponding inverted index is then given as*:*

to	:	{0}
be	:	{0,1}
or	:	{0,1}
not	:	{0}
there	:	{1}
square	:	{1}

A *full* inverted index also stores the occurrences of the token in the text as a pair {textid, position}. The most common query function of such an index is `locate(x)` that returns all texts where token x occurs. The construction principle of an inverted index is straightforward. First, a forward index is built that stores lists of words, i.e. tokens, of the texts. In the next step, this forward index is *inverted* which lists the texts of a token, i.e. in which texts a token occurs. A `locate` query on the forward index would require scanning the texts sequentially, whereas a query on the inverted index can be done in constant time by resolving the ID of a token through hashing.

2.4.2. Levenshtein Distance

The *minimum edit* or *Levenshtein distance* [125] $ed(s,t)$ is a metric that measures the similarity of two strings s and t. It is the minimum number of character insertions, deletions or replacements necessary to transform string s into t. Consider $m := |s|$ and $n := |t|$ and the base cases $D_{0,0} = 0$, $D_{i,0} = i, 1 \leq i \leq m$, and $D_{0,j} = j, 1 \leq j \leq n$. The edit distance can be computed by filling a matrix D of $m \times n$ entries.

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & +0 \text{ if } s_i = t_j \\ D_{i-1,j-1} & +1 \text{ (characterreplacement)} \\ D_{i,j-1} & +1 \text{ (characterinsertion)} \\ D_{i-1,j} & +1 \text{ (characterdeletion)} \end{cases}$$

The minimum number of edit operations can then be found at entry $D_{m,n}$. The asymptotic run time to compute the edit distance is $\mathcal{O}(m \cdot n)$ and the space required is $\mathcal{O}(\min(m,n))$ since only access to the previous row is necessary and the entry of the previous column is stored in the current row. The edit distance of two strings is zero if and only if the two strings are identical. And it is at least the length difference and at most the length of the longer of the two strings.

2.5. Contraction Hierarchies

This section explains Contraction Hierarchies (CH) [84], a very successful speedup technique to Dijkstra's algorithm. CH heuristically order the nodes by some priority function and *contract* them in this order. Here, contracting means that a node is removed from the graph and as few edges as possible are inserted to preserve shortest path distances. The original edges are augmented by the shortcut edges. A query, which is essentially a bidirectional variant of Dijkstra's algorithm, only needs to follow edges that lead to more important nodes. Hence, the data necessary to answer a query forms a directed acyclic graph (DAG).

When a node is contracted, it is (temporarily) removed from the graph and shortcut edges are inserted to preserve shortest path distances between the remainder of the graph. The node that is selected next for removal is selected by an online heuristic since it is too expensive to precompute all priorities before actually contracting any node. In particular, Bauer et al. [22] show that it is actually $\mathcal{NP}$-hard to compute a hierarchy, i.e. a node ordering, that minimizes the size of the respective CH data structure.

When a node v is contracted, a local search is conducted to verify if a shortcut for path $\langle u, v, w\rangle$ may be omitted. This heuristic is called *witness search* and essentially runs a shortest path computation between the pairwise combinations of nodes u and w. Shortcut edges are omitted if this returns that there is exists a path $\langle u, \ldots, w\rangle$ that is shorter than the (potential) shortcut, i.e. $\mu(\langle u, \ldots, w\rangle) \leq (l(u, v) + l(v, w))$. Note that superfluous shortcuts do not influence the correctness of the constructed data structure, but unnecessarily inflate the search space of a query. Hence, the witness search is allowed to have a one-sided error, i.e. false negatives. For example, the witness search can be pruned by an upper bound on the number of settled nodes, or on a hop distance from the source. We refer the interested reader for a sound analysis of correctness to [84].

The priority function is a linear combination of a number of terms. While one can think of all sorts of complicated ingredients to that combination, recent implementations, e.g., [128, 129], use rather simple heuristics with great success. Three easy terms have been identified. First, the so called *edge quotient* is considered, i.e. the number of edges that are deleted and added by contracting a certain node. Second, the (related) *quotient of original edges* is considered, i.e. how many original edges are represented by a shortcut. And last but not least, the so-called *depth* of a node is considered, which represents the maximum hop distance from a given node to a previously contracted node using only already inserted shortcuts always leading to earlier contracted nodes. Then, the actual linear combination is as easy to compute as

$$\alpha \cdot \text{edgeQuotient} + \beta \cdot \text{originalEdgeQuotient} + \gamma \cdot \text{nodeDepth}, \text{ with } \alpha, \beta, \gamma \in \mathbb{N}.$$

Node priorities are recomputed for all neighbours of contracted nodes right after their (temporary) removal from the graph. The preprocessing is parallelized by identifying independent sets of nodes that can be contracted without interfering with each other. While the remaining graph (containing the nodes not yet contracted) is non-empty, an independent set I of nodes is identified to be contracted next. Note that any independent node set could be used in principle, but it is reasonable to approximate the ordering a sequential algorithm would use, i.e. contract the nodes in roughly the same order of importance. An important ingredient to the performance of sequential contraction is to select nodes *uniformly*, e.g. by growing (graph) Voronoi regions [84]. Therefore, those nodes that are locally minimal with respect to a heuristic importance function within their local 2-hop neighbourhood are selected for contraction. Obviously, this hop distance is sufficient to ensure that independent nodes can be contracted in parallel without interfering with each other. Vetter [184] reports that this turns out to be a valuable compromise between a low number of iterations and good approximation of the behaviour of sequential contraction. A distributed variant of the parallelization approach will be examined in Section 4.

The resulting data structure of the Contraction Hierarchies preprocessing is the union of the original graph and the set of shortcut edges. A shortest path computation on this data structure is essentially a bidirectional version of Dijkstra's algorithm that considers only edges to more important nodes, i.e. so-called upward edges. Note that this graph forms a directed acyclic graph (DAG) where edges only lead to more important nodes, i.e. later contracted nodes. The set of *forward* edges in this DAG is denoted by $G^{\uparrow}$ and the set of *reverse*, i.e. *backward*, edges by $G^{\downarrow}$. The length of a shortest path from node u to v in the forward (backward) search space is denoted by $d^{\uparrow}(u, v)$ ($d^{\downarrow}(u, v)$). The only crucial difference is the stopping criterion of the bidirectional search that continues adding nodes into the priority queues when a common node is found in both search spaces, but stop when tentative distances of added nodes exceed the lower bound that may exist for a shortest path. A shortest path then goes over a *middle node* that is settled in both (half-)searches and for which CH guarantee correct labelling in both search directions. See Figure 2.1 for a visualization of the CH data structure and its query.

Although the search spaces explored in CH queries are rather small in practice, there is a simple technique called *stall-on-demand* [171] that further prunes the search spaces. We use a simplified version of that technique, which leads to queries as fast as those reported in [84] but is much simpler and does not need additional data structures. For every node v that is the end point of a relaxed edge (u, v) it is checked if there exists a reverse edge (w, v), where the tentative distance of w plus the edge weight of (w, v) is less than the tentative distance of u plus edge (u, v). This is done by simply scanning the edges incident to node v. If such a node w exists, edge (u, v) can't be part of a shortest path and thus v is not added into the queue. See Figure 2.2 for a visualization of stalling in an exemplary forward search.

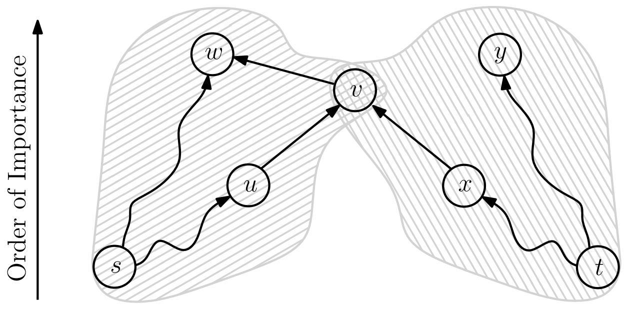


Figure 2.1.: CH Query for Nodes $s, t \in V$. Forward and Backward Search Spaces Meet at Node v.

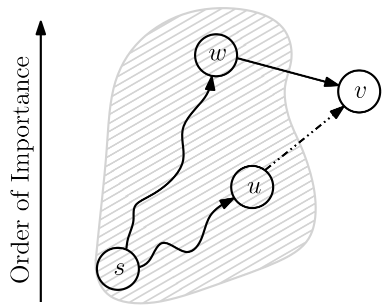


Figure 2.2.: Node v is not Inserted into the Priority Queue When the Length of Path $\langle s, \ldots, w, v\rangle$ is at Most the Length of the Newly Discovered Path $\langle s, \ldots, u, v\rangle$.

While the resulting distance of a CH query is the same as the distance returned from a plain unidirectional Dijkstra run on the uncontracted graph, the returned path may consist of shortcut edges. Thus, these shortcuts need to be *unpacked* to retrieve the *full path*. This is done by a recursive procedure that unpacks a shortcut by splitting it into the two edges from which the shortcut was built. Note that it is sufficient to store the *middle node*, i.e. the contracted node from the creation of each shortcut edge, to unpack every path and that this information may be omitted in case one is interested in distances only. It is easy to see that the middle node of a shortcut is less important, i.e. has smaller priority, than any of its endpoints. See Figure 2.3 for a visualization.

Time-Dependent Preprocessing. We use the same approach as Batz et al. [21] for time-dependent preprocessing and consider only upper and lower bounds during the witness searches. We need to create a shortcut if we only find a witness that has a higher maximum travel time than the lower bound of our potential shortcut. If we find a witness with minimum travel time below the lower bound of our potential shortcut, we do not have to insert a shortcut. The travel time functions of the two input edges are linked to generated the travel time function of the shortcut. Otherwise, we conduct a *profile search* on a corridor of visited nodes during the witness search. We only omit a shortcut if we find a profile that is a lower bound for our potential shortcut. For further explanation of the inner workings of the edge weight data type and the witness search, we refer the interested reader to the

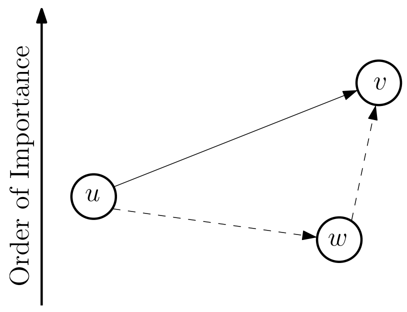


Figure 2.3.: Edge (u, v) is Unpacked into Edges (u, w) and (w, v).

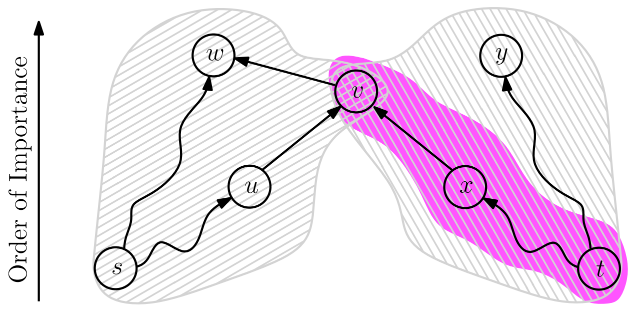


Figure 2.4.: Nodes of the Backward Search are Marked and Subsequently Explored by the (Resumed) Forward Search.

publication of Batz et al. [21] and to the brief recapitulation of Section 2.1.

Time-Dependent Query. A bidirectional and time-dependent query has to overcome the issue that the arrival time is not known. The forward search is easy to adapt as the departure time is known, though. Each relaxed edge is evaluated for exactly one point in time and the resulting search is very similar to the time-**in**dependent case. We have to solve the problem of how to conduct a bidirectional search, when the arrival time is unknown. This is solved by splitting the search from the target into two search phases. First, the *backward search* explores the reachable search space by a BFS-like search that marks a *corridor* of all potentially reachable nodes. This exploration can be done by an (adapted) static Dijkstra query and by noting upper and lower bounds for the edge cost functions, we can prune the search at some nodes, e.g., arriving at a node before actually departing is impossible. Again, the search space is small on average and the rationale is that only a modest number of nodes will be marked. Conceptionally, the forward search is resumed in a *downward search* on these marked nodes and eventually finds a path. See Figure 2.4 for a visualization of this query process.

2.5.1. Computing Distance Tables

Computing a table of all pairwise shortest path distance between a set of nodes can be trivially done by running a quadratic number of queries. While this is already significantly faster with Contraction Hierarchies than with a naive implementation of Dijkstra's algorithm, this table can be computed much more efficiently with the algorithm of Knopp et al. [114] as they report construction times of a few seconds only even for large tables. The algorithm was originally conceived for a different speedup technique to Dijkstra's algorithm, but its underlying principles are more general and can also be applied to Contraction Hierarchies [84]. The main observation is that when computing a quadratic number of pairwise distances a recurring set of important nodes gets settled in nearly all searches.

The algorithm basically consists of two phases of *half searches*. Consider a set of sources S and a set of targets T. Also, consider an initialized, but empty distance table D, i.e. all entries set to ∞. In the first phase, a simple CH backward search is conducted for each node $t \in T$. During each of these *half searches*, the pair $\big[t, d^{\downarrow}(v,t)\big]$ is noted for each settled node $v \in G^{\downarrow}$. Thus, the algorithm notes the distances to the target nodes at each and every potential middle node that is settled during the backward search phase. The information that is stored at each node v is called a *bucket* b_v, which in a technical sense is a simple unsorted, but dynamic array.

In the second phase of the distance table computation, the forward searches are conducted for all nodes $t \in S$. Whenever a node v is settled at distance $d^{\uparrow}(s,v)$, its bucket b_v is scanned. Given a bucket element $\big[t, d^{\downarrow}(t,v)\big]$, the corresponding entry in the distance table $D[s,t]$ is updated if the following condition holds:

$$D[s,t] > \underbrace{d^{\uparrow}(s,v)}_{\text{forw. search}} + \underbrace{d^{\downarrow}(v,t)}_{\text{bucket entry}} \ .$$

Note that this condition is tested for each bucket and every entry encountered during the second phase of the algorithm. The search spaces are small. Hence, the size of the buckets will be small, as will be the number of nodes that carry a non-empty bucket at all.

For a proof of correctness consider the node pair $(s,t) \in V$, with $s \in S$ and $t \in T$. We show that the meeting node v of a simple CH query for (s,t) is found with correct forward and backward labels inside the bucket structure. Assume that the $|T|$ backward searches are finished. Since the backward search for distance tables is the same as for a CH query, it is easy to see that there exists a bucket entry $\big[t, d^{\downarrow}(v,t)\big]$ with the correct backward distance for middle node v. The forward search settles the middle node as well by the same argument. Hence, the forward distance $d^{\uparrow}(s,v)$ is also correct. By correctness of the CH query we know that $\mu(s,t) := d^{\uparrow}(s,v) + d^{\downarrow}(v,t)$ for the correct middle node v. We refer the interested reader to a much more detailed discussion of the correctness in [114].

Computing such a distance table with the above algorithm is a matter of mere seconds since only $O(|S| + |T|)$ half searches have to be conducted. The quadratic overhead to initialize and update the distance table entries is close to none in practice. It is easy to see, that computing a one-to-many (or many-to-one) query is a special case of the general algorithm to compute distance tables.

While the algorithm has been available to the Algorithm Engineering community for quite some time, the fast generation of distance tables has only been recently when it was picked up by the economic research community [70] to show that travel times and clustering of businesses correlate on the large scale.

2.6. Test Instances

Table 2.1 lists the test data instances that are used in the experimental evaluations along with their size, date of origin, as well as a remark. The instances are grouped into time-dependent and static graphs and sorted within their group by origin and node size.

The test instances come from a number of sources. Dictionary *dict-moby* is the text of Melville's classic novel *Moby Dick* available from Project Gutenberg [157], *dict-town* is a list of town names extracted from OpenStreetMap in February 2009, *dict-eng* is a list of

Dictionary	Entries	avg. length	Rf.	[MB]	Remark
dict-moby	37 924	9	[157]	0.31	Melville's classic *Moby Dick*
dict-town	47 339	10	[152]	0.49	German town names
dict-eng	213 557	10	[138]	2.20	Webster's Dictionary
dict-wiki	1 812 365	9	[187]	17.06	Wikipedia article titles

Adress Data	City Tokens	Street Tokens	Rf.	Year	Remark
ptv-addr	≈ 76 000	269 000	[158]	2009	Germany

Graph	n	m	Rf.	Year	Remark
ptv-europe-td	≈ 18.0M	≈ 42.6M	[158]	2006	6% time-dependency
ptv-ger-mw-td	≈ 4.7M	≈ 10.8M	[158]	2006	8% time-dependency, midweek
ptv-ger-sun-td	≈ 4.7M	≈ 10.8M	[158]	2006	3% time-dependency, sunday
ptv-europe	18 029 721	44 826 256	[64]	2006	Western Europe
ptv-euro-dist	18 029 721	44 826 256	[64]	2006	West. Eur., distance metric
ptv-germany	4 378 446	9 574 514	[64]	2006	Germany
ptv-belgium	463 514	1 093 544	[64]	2006	Belgium
osm-planet	758 206 383	1 842 527 702	[152]	2012	Entire World
osm-germany-4	35 024 256	43 790 686	[152]	2013	Germany
osm-germany-2	33 927 089	86 477 642	[152]	2012	Germany
osm-berlin	288 755	844 550	[152]	2012	Berlin, Germany
osm-germany-3	15 139 753	29 492 546	[152]	2011	Germany
osm-germany-1	6 344 491	13 513 085	[152]	2010	Germany
osm-bay-area	≈ 355.000	≈ 466.000	[152]	2010	Bay Area, California, USA

Table 2.1.: Basic Properties of the Test Instances Used in Benchmarks.

entries from Webster's Dictionary [138], while *dict-wiki* is a list of Wikipedia [187] article from February 2009.

Graphs prefixed with *ptv-* have been made available by PTV AG [158] for 9th DIMACS challenge on shortest paths [64]. The length and the respective type out of 13 road categories are available for each edge segment. The edge cost metric resembles expected travel time which is derived from the length of an edge and the road category.

Several graph instances are time-dependent. The resolution of the time functions is 15 minutes. Instance *ptv-europe-td* has about 6% time-dependent edges with an average number of more than 26 supporting points each. Instance *ptv-ger-mw-td* reflects average midweek (Tuesday through Thursday) traffic for Germany while *ptv-ger-sun-td* reflects the more quiet Sunday traffic patterns. Both data sets were collected from historical data and enriched with data from traffic simulations. The German Midweek scenario has about 8% time-dependent edge weights while the Sunday scenario consists of about 3% time-dependent edges. These instances are the de-facto standard benchmark instances for a number of time-dependent shortest path algorithms, e.g., [20, 56, 19, 83].

Graphs prefixed with *osm-* are extracted from OpenStreetMap. The data of Open-

StreetMap is freely available for download. While the data set is changing constantly, it is possible to download the so-call *full history dump* and extract the database from a given point in time in the past. A basic tutorial how to extract graphs is covered in Appendix E.

The instances *osm-planet*, *osm-germany-4*, *osm-germany-2*, *osm-baden*, and *osm-berlin* are edge-expanded, i.e. each possible turn is explicitly modelled and U-turns are forbidden, e.g., [185]. Moreover, existing turn restrictions present in the input data are preserved.

The data set *ptv-addr* is commercial data from 2009 comprising all German street and town names provided by PTV AG. It contains about 12 000 cities, 108 000 towns, 80 000 town names, 76 000 town name tokens, 1 350 000 streets. About 560 000 of the street names contain the token "`Strasse`"[1]. In total, we see about 444 000 distinct street names comprised by approximately 269 000 street name tokens. A street name consists of 2.5 tokens on average, while town names consist of 1.1 tokens on average. The raw data set occupies about 30 MiB of space.

2.7. Machines

The following machines are used in the subsequent chapters to conduct experiments:

Machine A is an Intel Core i7-920 Quad-Core processor running at 2.667 GHz with 12 GB of main memory and 8 MB of L3 cache. The machine is used in the experiments of Sections 3.2, 3.3, 5.5, and 6.

Machine B is a cluster of 200 compute nodes. Each node has two Intel Xeon X5355 Quad-Core processor running at 2.667 GHz with 16 GB of main memory and 2x4 MB of L3 cache. The nodes are interconnected by an InfiniBand 4XDDR network that has a latency of less than 2 microseconds and a peak point-to-point bandwidth of more than 1300 MB/s. This machine is used in the experiments of Section 4.3.

Machine C is an AMD Opteron 6212 processor running at 2.60 GHz with 2×4 full and 2×4 CMT cores, and with 128 GB of main memory as well 8×16 KB L1 cache, 4×2 MB L2 cache and 16 MB of L3 cache. This machine is used in the experiments of Section 4.2.

Machine D is an Intel Core i7-860 Quad-Core processor running at 2.80 GHz with 16 GB of main memory and 8 MB of L3 cache. The machine is used in the experiments of Section 3.4.1.

Machine E is an Intel Xeon X5690 with six cores running at 3.47 GHz, and with 256 GB of main memory as well as 12 MB of L3 cache. The machine is used in the experiments of Section 5.2.

Machine F is an AMD Opteron 270 running at 2.0 GHz with 8 GB of main memory and 2×1 MB of L2 cache. The machine is used in the experiments of Section 5.3.

[1]The German word for *street*.

Machine G is an AMD Opteron 6168 processor with 4×12 cores running at 1.9 GHz, and with 128 GB of main memory as well as 6 MB of L3 cache. The machine is used in the experiments of Sections 5.3 and 5.5.

Machine H is an AMD Opteron 8350 processor with 4×4 cores running at 2.0 GHz with 64 GB of main memory and 2 MB of L2 cache. The machine is used in the experiments of Section 5.6.

Machine I is an Intel Xeon X5550 CPU with 8 cores running at 2.667 GHz, and with 48 GB of main memory as well as 8 MB of L3 cache. The machine is used in the experiments of Section 3.2.

Note that Machine B is a cluster of 200 compute nodes. We denote a core as a processing element (PE) throughout this thesis. Appendix D gives a tabulated summary of the machines used.

Further Reading. We refer the interested reader to the textbook of Mehlhorn and Sanders [137] for further information on basic algorithms and data structures.

CHAPTER 3

Geocoding of Locations

3.1. Central Ideas

The process of geocoding is to automatically transform textual location descriptions into geographical coordinates. This process has been available in geographic information systems for quite some time [51] with applications in route planning, validating customer addresses [74], or surveillance and management of disease outbreaks like the yearly wave of influenza [119], among others.

However, with its ubiquitous use in modern web services, e.g., [29, 91], requirements have become more severe: Since most of these services are free, geocoding servers must handle huge streams of queries at very low cost.

The problem has many applications. For example, Google's 'Did you mean' feature catches typos in search queries. But in some settings, the uncertainty is higher and therefore one is not interested in the best match, but also in other matches which are similar to the query. In a geocoding setting one wishes to also map misspelled location descriptions to coordinates.

The underlying problem of searching approximate matches in a dictionary arises in many fields. Most common is the search for the so called best match. Each word in the dictionary is represented by a string of characters over a finite alphabet Σ. The Levenshtein distance [125] $ed(a, b)$ defines a metric between two words $a, b \in \Sigma^*$ and is used in this work to compute the distance between two words.

The most trivial algorithm to solve the problem is scanning sequentially through the input list and noting the best match(es) at each entry. The running time is obvious and consists of a linear number of distance computations and searching an entire directory on a standard desktop computer takes only a few seconds even for dictionaries up to a few hundred thousand or even a million entries. But in many settings this is too much, because queries arrive in a high frequency. For example, a web search engine only has a few milliseconds to process a single request and does not have the time to do exhaustive searching in a large dictionary. At the same time, users expect instantaneous answers. Finally, inputs are frequently fragmentary, contain misspelled names or specify combinations of town and street that are inconsistent with the database. A service is likely to be more popular and useful if it tolerates such imprecisions.

We focus on the algorithmic aspects of the problem to map information about town and street to a database entry for the intended street. We assume the input or *reference data* to be structured in the following way:

- It consists of strings from a natural language.
- The data has a *spatial overlap*.

By spatial overlap we mean that elements of our data set correspond to geometric entities that are either close to each other or have a non-empty intersection. The latter point can be expressed in layman's terms that a building (coordinate) is at a street (polyline) in a city (polygon) belonging to some administrative area (polygon, too), etc. The geometric entities in this example are either close to each other (building, street) or overlap (street, city).

We aim to exploit the two properties from above to collect a few candidates for further inspection. While companies offering such services have naturally worked on this problem intensively, we are not aware of academic work that offers the required combination of low latency, high throughput, and error-correction for large address data sets. Our original aim was to make reasonable methods available to a cooperating company and to the academic community. It turns out that our approach achieves better solution quality than the market leaders at low costs so that it might also help the industry to improve their services.

The remainder of this chapter is structured as follows. First, we give a description of our index data structure that efficiently supports approximate queries in a dictionary. An experimental evaluation shows the performance of our approach and the accompanying implementation. Second, we apply this index as a basic building block to solve the problem of geocoding and show the efficiency of the resulting data structure in an extensive experimental evaluation on a real-world data set.

References

The contents of the following chapter are based on the following publications: Section 3.2 is based on joint work with Daniel Karch and Peter Sanders [105, 106]. Sections 3.3 and are based on joint work with Christian Jung, Daniel Karch [104], Sebastian Knopp and Peter Sanders [103]. Wordings of these publications are used in this thesis.

3.2. Fast Similarity Search

Deletion Neighborhood. We present a filtering technique for the dictionary matching problem called *Fast Similarity Search (FastSS)* [179], which in turn is a generalization of the single error method proposed by Mor and Fraenkel [142]. For a given integer d and a word $w \in \Sigma^*$ the *d-(deletion-)neighborhood* $\mathcal{N}_d(w)$ is defined as the set of all sub-words of w with

Listing 3.1: Computing the d-deletion neighborhood of a string s.

```
function d_neighborhood(const d, const s)
  $\mathcal{N}'_d(s) := \emptyset$
  if $0 == d$ then
    return $\mathcal{N}'_d(s)$
  end
  for i=0 to $|s|$ do
    $\mathcal{N}'_d(s) \cup$ d_neighborhood($d-1$, del($s, i$))
  end
  return $\mathcal{N}'_d(s)$
end
```

exactly d deleted positions. Each element of $\mathcal{N}_d(w)$ is called a *residual string*. Furthermore, a string w is called *originating string* for residual r if and only if $r \in \mathcal{N}_d(w)$. We obtain an exact filter for a set of words $\mathcal{S}$ by pre-computing the d-neighborhoods of strings in $\mathcal{S}$. As a filtering function, we obtain $F(q) = \{s \in \mathcal{S} : \mathcal{N}_d(s) \cap \mathcal{N}_d(q) \neq \emptyset\}$. The correctness of this definition follows from the following Lemma:

Lemma 1. *If two words $u, v \in \Sigma^*$ are within a distance d from each other, then there exists a word w which has length at least $|u| - d$ and consists of letters from u and v in their original order. Assume that u is at least as long as v.*

We use the concept of *Ordered Edit Sequences* [131] to show the claim. Our proof is simpler and more intuitive than the proof from [179].

Proof. Recall that the edit distance is said to be the minimal number of edit operations to transform one word $u \in \Sigma^*$ into another $v \in \Sigma^*$. The set of operations available for any single transformation are $op = \{\mathsf{ins}, \mathsf{del}, \mathsf{chg}\} : \Sigma \cup \{\epsilon\} \to \Sigma \cup \{\epsilon\}$ with $v = op_d(op_{d-1}(\ldots(op_1(u))\ldots))$. The sequence $\rho(u, v) = (op_1, op_2, \ldots, op_d)$ is called edit sequence and we call it ordered if the operations are applied from left to right. We define $pos(\cdot)$ to give the position of an operation within the edit sequence. In other words $\forall i : (pos(op_i) \leq pos(op_{i+1}))$.

By definition of the edit distance metric there exists an edit sequence of minimal length. Now, we can show Lemma 1. Since $ed(u, v) \leq d$ it follows that the length of a minimal ordered edit sequence is at most d, which means $|\rho_{min}(u, v)| \leq d$ is the length of a minimal edit sequence. Wlog, we consider string v as the case for u is symmetric. This implies that v is changed at no more than d positions. By replacing these at most d positions of v with delete operations, we get a string w, which has length at least $|u| - d$ and preserves the letter ordering from u and v. $\square$

The actual computation of a d-deletion neighborhood can be specified as a recursive procedure as shown in pseudo code in Listing 3.1. It uses a given *atomic* sub-routine `del`(s, i) that removes the i-th character from a string s. The asymptotic runtime of this procedure is $\mathcal{O}(|s|^d)$, which is constant in practice for small values of d and $|s|$).

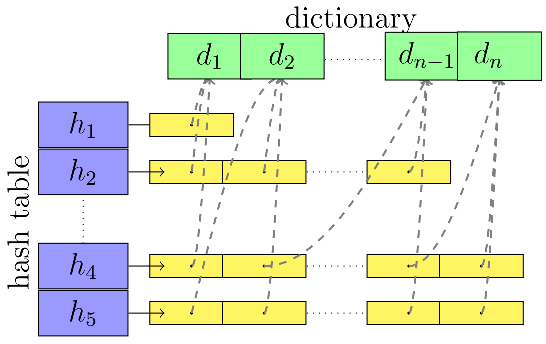


Figure 3.1.: Approximate string matching data structure.

Basic Data Structure. A static index data structure is generated in a precomputation phase that can be queried during an on-line phase. We insert a number of values into a hash table that is part of our data structure. The structure utilizes store pointers to originating strings as the values of a hash table entry. If any hash value has more than one originating dictionary entry then the corresponding pointers are stored in a list. Figure 3.1 sketches the internal structure of the index.

Query. For an input query q and maximum distance d, the d-neighborhood and its hash values are computed. If any element of the query's residuals is also an element of the data structure then the pointers to the originating dictionary entries give a set of candidates. Each of those might be an approximate match. Once the candidate set is completely built, it is searched exhaustively by computing the edit distance of each candidate to the query. By removing all elements from the candidate set whose distance is larger than the threshold d we get the set of all dictionary members that are at most a distance d away from query q. Perhaps there exists an additional order on the candidates stemming from the application. The algorithm can be adapted to not only return the best match, but also a list of those candidates that are sufficiently close.

Assume that a dictionary has been precomputed by storing the deletion neighborhood of every dictionary entry in a key-value storage and that the neighborhood of a key accessible by `fetch`$(\cdot)$. The top-k query algorithm is given in Listing 3.2, given a subroutine $\texttt{ed}(s_1, s_2)$ that computes the Levenshtein distance of two strings s_1 and s_2. Note that this algorithm can be more efficient in practice for small values of k than a simple quick-select. The container `top_list` that holds a tentative result and is implemented as a non-addressable max-priority queue. The resulting asymptotic run time is

$$\mathcal{O}\left(\overbrace{|q|^d}^{\text{del. nbh.}} + \overbrace{\max(|\mathcal{N}_d(q)|, |\mathcal{C}|)}^{cand.fetch} + \overbrace{|\mathcal{C}| \cdot \log k \cdot \sum_{0 \le i \le |\mathcal{C}|} T_{\texttt{ed}(q,c_i)}}^{\text{verification of candidates}}\right)$$

$$= \mathcal{O}\left(|q|^d + |\mathcal{C}| \cdot \log k \cdot \sum_{0 \le i \le |\mathcal{C}|} T_{\texttt{ed}(q,c_i)}\right) \quad .$$

This asymptotic running time is depends on the number of candidates the `fetch`$(\cdot)$ operation yields as well as the time $T_{\texttt{ed}(a,b)}$ it takes to compute the Levenshtein distance for a given pair of elements. If we assume the strings to be of limited size we can see this cost as constant, although the asymptotic run time of the Levenshtein distance computation is $\mathcal{O}(n \cdot m)$. The expected number of candidates is analyzed in a later section.

Preprocessing. We compute the d-neighborhood of each element of the input dictionary and insert the resulting information into our index data structure. Doing this preprocessing naively and storing all residual strings in a tree-like search data structure takes up an enormous amount of space while the number of residual strings grows exponentially with the distance parameter d. Instead, we use hash function $h : s \to \mathbb{N}$ and reduce each element of the residual neighborhood into an integer number to save space. We insert pointers to the originating dictionary entries into the hash table at the respective hash values of all residual strings. Therefore, only constant space is needed per residual string regardless of the length of that string. Listing 3.3 details the preprocessing of an entire dictionary, given a key-value data structure D that maps strings of characters to an array of integers. We now present a generalization of the algorithm that significantly reduces the space overhead by introducing an tuning parameter.

Algorithmic Generalization. We can further limit the number of elements that are inserted into the index while still staying exact. To do so, we split long originating strings in half, compute their residual strings with half the number of errors, and adapt the query algorithm, which is explained in the remainder of this Section. Instead of generating $\binom{|s|}{d}$ hash values

Listing 3.2: Running a top-k query q against the Dictionary D with distance at most d.

```
function match(D, q, d, k)
  $\mathcal{N}_d(q) := \texttt{d\_neighborhood}(d, 0, q)$
  $\mathcal{C} := \emptyset$
  for $i$ in $\mathcal{N}_d(q)$ do
    { fetch candidates from key-value storage }
    $\mathcal{C} := \mathcal{C} \cup \texttt{fetch(i)}$
  end
  { initialize empty max-pq }
  $\texttt{top\_list} := \{\}$
  for each $c$ in $\mathcal{C}$ do
    if $\texttt{ed}(c,q) \leq \texttt{top\_list}.max()$ then
      if $\texttt{top\_list}.size() \geq k$ then
        $\texttt{top\_list}.pop()$
      end
      $\texttt{top\_list}.insert(c, \texttt{ed}(c,q))$
    end
  end
  return result
end
```

we insert only

$$\binom{|s|}{\lfloor \frac{d}{2} \rfloor} + \binom{|s|}{\lceil \frac{d}{2} \rceil}$$

values for a split dictionary entry s. The *generalized d-neighborhood* of $w' \in \Sigma^*$ is the set of residuals that is found by computing all combinations of $\lceil \frac{d}{2} \rceil$ deleted characters for the first and second halves of w'.

The generation of the index works similar to the way the generalization works. But note that we have to pay some extra care at query time, because insertions and deletions that transform words can take place at arbitrary positions. As a consequence, we can not rely on the length of a query q to decide whether it has been split or not. Instead of splitting a query q of length l at a fixed position, it is split several times in half at positions in the interval of $\lceil \frac{l}{2} \rceil \pm \lceil \frac{d}{2} \rceil$. Also, the allowed error is halved to compensate for shorter strings. We split any dictionary token if it has length greater or equal than a threshold m. If the length of a query word is within $m \pm d$ then the index is also searched for the non-split string. See Section 3.2.1 for an experimental analysis of the threshold value m, which indicates whether or not to split a word.

Our method is still correct, i.e. does not lose exactness, since we can show the existence of at least one common residual string for either the prefix or the suffix of a split query word. Consider the following definitions. Let $w \in \Sigma^*$ be an entry of dictionary D and let d be the maximum allowed error. Furthermore, let $u = p(w)$ and $v = s(w)$ denote the first and second halves of the split word w. Prefixes u and suffix v are indexed as explained above, while q is the query. Any query q is split at several positions as explained above and we define $\mathcal{P}(w)$ to be the set of first and $\mathcal{S}(w)$ to be the set of second word halves. The following Lemma formalizes correctness:

Lemma 2. *Let $q \in \Sigma^*, w = uv$ with $ed(w, q) \leq d$. Consider $\mathcal{P}(q)$ ($\mathcal{S}(q)$) to be the set of $\lceil \frac{d}{2} \rceil$ many prefixes (suffixes) of q that are generated for each query to the index by the generalized neighborhood. Then there exists at least one pair (p', s') with $p' \in \mathcal{P}(q)$, $s' \in \mathcal{S}(q)$ and $p' \circ s' = q$ of prefix-suffix-elements for which either $ed(u, p') \leq \lceil d/2 \rceil$ or $ed(v, s') \leq \lceil d/2 \rceil$. It suffices to test the split positions from the interval $\lceil \frac{|q|}{2} \rceil \pm \lceil \frac{|d|}{2} \rceil$ to find that pair.*

In other words, we can be always sure that a common residual between any query q and all index entries $e \in D$ with maximum error d is found for which $ed(q, e) \leq d$ holds. The Lemma follows directly from the pigeon hole principle, but we give a full formal proof here.

Listing 3.3: Precomputation of a dictionary D.

```
for $s$ in $S$ do
  $\mathcal{N}_d(s)$:= compute-neighborhood(d, s);
  for $i$ in $\mathcal{N}_d(s)$ do
    D[s].append(i)
  end
end
```

Proof. Wlog, consider the set of prefixes of q. If $ed(w,q)=d$, then there is an element $p \in \mathcal{P}(q)$ for which $ed(u,p)=j \leq \lfloor d/2 \rfloor$. Then there also exists a sequence of operations $op_1,\ldots,op_d \in \{\mathsf{ins},\mathsf{del},\mathsf{chg}\}$ such that $(op_d \circ \cdots \circ op_1)(w)=q$. Wlog, we can assume that $op_1,\ldots,op_j$ introduce the j errors in $u=p(w)$. Hence, the suffix $v=s(w)$ remains untouched after the first j edit operations. Since there remain only $d-j \leq \lfloor d/2 \rfloor$ edit operations, it follows that $ed\left(v, s_{|v|}(q)\right) \leq \lfloor d/2 \rfloor$. The case for $ed\left(v,s(q)\right) > \lfloor d/2 \rfloor$ follows from symmetry. $\square$

Consider the edit sequence S that transforms w into q and that has length at most d, s.t. $ed(w,q) \leq d$. String w is split at position $\lceil \frac{|w|}{2} \rceil$ into $w = p \circ s$. Note that the lengths of p and s differ by at most 1. Sequence S is applied to $w = p \circ s$ and yields $q = p' \circ s'$. Hence, either $ed(p,p') \leq \lceil \frac{d}{2} \rceil$ or $ed(s,s') \leq \lceil \frac{d}{2} \rceil$ or both. The algorithm has to split query q exactly into p' and s' to guarantee that a match is found. Assume that it doesn't suffice to test the interval $\lceil \frac{|q|}{2} \rceil \pm \lceil \frac{|d|}{2} \rceil$ to find the correct splitting position. Then p' is either shorter than $\lceil \frac{m}{2} \rceil - \lceil \frac{d}{2} \rceil$ or longer than $\lceil \frac{m}{2} \rceil + \lceil \frac{d}{2} \rceil$. Assume $|p'| < \lceil \frac{m}{2} \rceil - \lceil \frac{d}{2} \rceil$. Then we deduce the following:

$$\begin{aligned}
&\Rightarrow |s'| > \lceil \frac{m}{2} \rceil + \lceil \frac{d}{2} \rceil \\
&\Rightarrow |p'| + \lceil \frac{d}{2} \rceil < \frac{m}{2} < |s'| - \lceil \frac{d}{2} \rceil \\
&\Leftrightarrow |p'| + \lceil \frac{d}{2} \rceil < |s'| - \frac{d}{2} \\
&\Leftrightarrow |p'| < |s'| - 2 \cdot \lceil \frac{d}{2} \rceil \\
&\Leftrightarrow |p'| - |s'| < -2 \cdot \lceil \frac{d}{2} \rceil \\
&\Leftrightarrow |s'| - |p'| > 2 \cdot \lceil \frac{d}{2} \rceil \ .
\end{aligned}$$

This implies that the lengths of s' and p' differ by more than $2 \cdot \lceil \frac{d}{2} \rceil$. But then edit sequence S has to be longer than $2 \cdot \lceil \frac{d}{2} \rceil$ operations, because length difference is a lower bound for edit distance. The other case for $|p'| > \lceil \frac{m}{2} \rceil + \lceil \frac{d}{2} \rceil$ follows by the same line of argumentation.

It is obvious that at least one common residual string is generated for strings $w, w' \in \Sigma^*$ when split like described above. For a given constant d the number of entries in the hash table is linear in the length of each inserted word. Note that in general, for higher (query) distances than the value used in precomputation, the index is not exact any more.

See the latter part of Section 3.2.1 for an experimental analysis of the generalization that shows it uses half the space than our implementation of the original algorithm and maintains stable query performance. Wu and Manber [193] use partitioning into $d+1$ pieces to match one of the pieces with no error, while Navarro and Baeza-Yates [148] gave a recursive partitioning scheme for fast on-line approximate string matching.

Analysis. Our variant of the approximate index makes heavy use of hashing as we argued before. We analyze the expected performance penalty of our approach coming from hash collisions to estimate the penalty that is inherent to this approach. First, we consider the case that we do not split the input string. Note that this resembles the approach taken by Bocek et al..

For each dictionary entry of length ℓ, we insert at most $\binom{\ell}{d}$ constant size entries into the hash table. The hash table needs $O(1)$ space per element since the bit size of each entry is of constant size in practice as $\log(i), i \in \mathbb{N}$ grows slowly. We obtain overall linear space consumption for $d = 1$, because $O(\ell)$ constant size hash table entries are stored for a dictionary entry of size ℓ.

The query time obviously depends on the number of identified candidates. We now analyze an upper bound to the number of dictionary entries that we have to consider for a given query q of length ℓ. Our model is the following: The dictionary has n entries drawn uniformly at random from Σ^ℓ, with $|\Sigma| = \sigma$. The probability that the i-th residual of q equals the j-th residual of a given word w of the dictionary is given by

$$Pr[X] := \frac{1}{\sigma}^{\ell-d} = \sigma^{d-\ell}.$$

Note that there exist $\binom{l}{l-d}$ distinct originating strings that share such a residual. We apply the linearity of expectation to get an expected number of

$$\mathbb{E}[X] = n \cdot \binom{\ell}{\ell - d} \cdot \binom{\ell}{d} \cdot \sigma^{\ell-d} = n \cdot \binom{\ell}{d}^2 \cdot \sigma^{\ell-d} \tag{3.1}$$

residuals to consider. As a consequence of Equation 3.1, we can expect a speedup over the naive linear scanning algorithm that is proportional to the size of the dictionary by applying the Markov bound. More precisely, we get an upper bound of the probability that the expected number of residuals to consider is not a fraction of n. Let $c > 0$ be a constant.

$$P\left[X \geq \frac{n}{c}\right] \leq c^{-1} \binom{\ell}{d}^2 \sigma^{d-\ell} . \tag{3.2}$$

The splitting optimization can be analyzed in the same way, which gives us an expected value of

$$\mathbb{E}[X_m] = n \binom{\lceil \ell/2 \rceil}{\lceil d/2 \rceil}^2 \sigma^{\lceil d/2 \rceil - \lceil \ell/2 \rceil}$$

for a dictionary in which every word gets split, as well as

$$P\left[X \geq \frac{n}{c}\right] \leq c^{-1} \binom{\lceil \ell/2 \rceil}{\lceil d/2 \rceil}^2 \sigma^{\lceil d/2 \rceil - \lceil \ell/2 \rceil}.$$

We conclude that the original variant does not suffer from hash collisions, but that we have to chose the threshold m carefully. If m is too small, we artificially increase the number of collisions. See Section 3.2.1, where we experimentally analyze the behavior of the algorithm for varying splitting parameters.

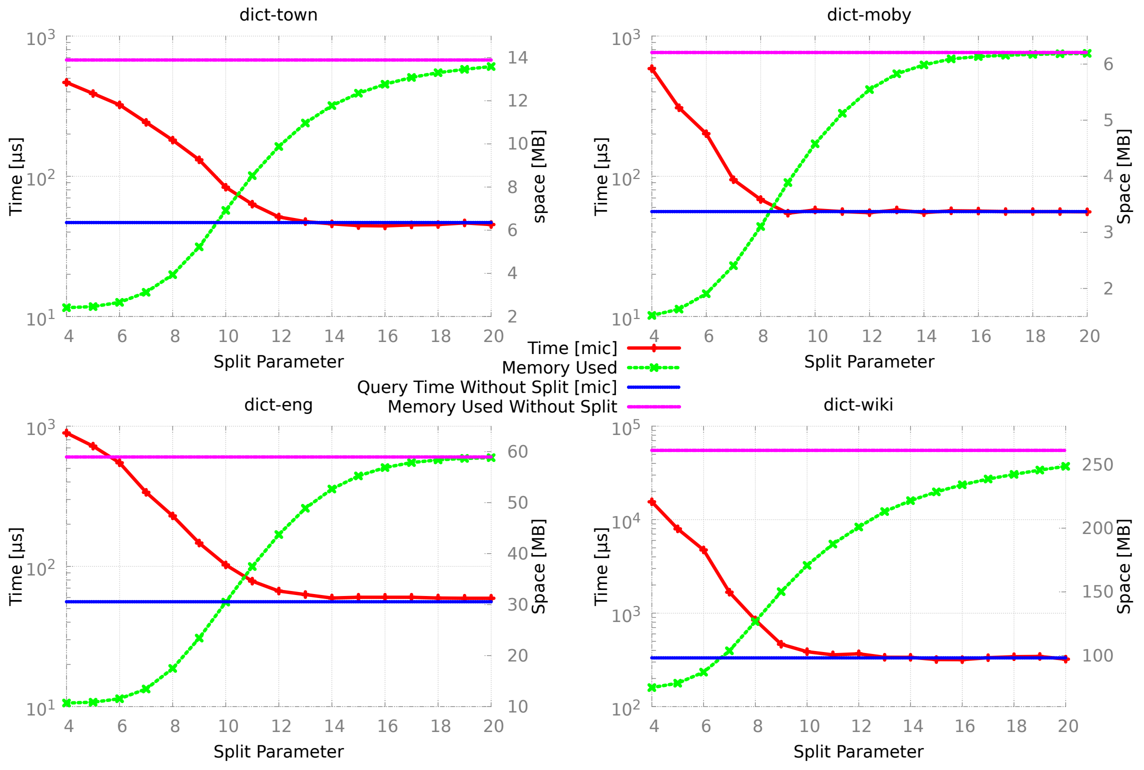


Figure 3.2.: Analysis of the Splitting Parameter for $d = 2$

3.2.1. Experimental Evaluation: Improved Fast Similarity Search

Implementation Details and Methodology. We implement the data structure, the construction and query algorithms in C++ using GCC Compiler version 4.3.2. We hashed all residual strings with the built-in hash function of the Boost library v1.36 to a 32-Bit Integer and chained with a simple linear congruence function to get values from a smaller interval.

All of our tests were conducted on a single core of Machine I, running a version 2.6.27 Linux kernel. We compare the performance of our optimizations against our own implementation of the basic variant for reasons of fairness as it is significantly faster than the numbers from literature. The dictionaries used in the experiments are *moby dick*, *town*, *english*, and *wikipedia*. All results were averaged over a number of queries of perturbed dictionary entries.

The exhaustive search, i.e. the verification, of a candidate set is done by a simple implementation of the Levenshtein distance. It computes a band of width $2d + 1$ only. This way we compute the distance exactly only if it is smaller than d and return otherwise as soon as we get a certificate that the distance is larger than d. Since we need $O(1)$ to fill a cell in the distance table, we can verify a candidate in $O(d \cdot l)$, where l is the length of the shorter word. In the experiments it took less than a microsecond to verify any single candidate.

Preprocessing Space. We analyze the amount of distinct residuals that are generated for each value of $m \in 1, \ldots, 30$ and the average duration of a single query against this index. To

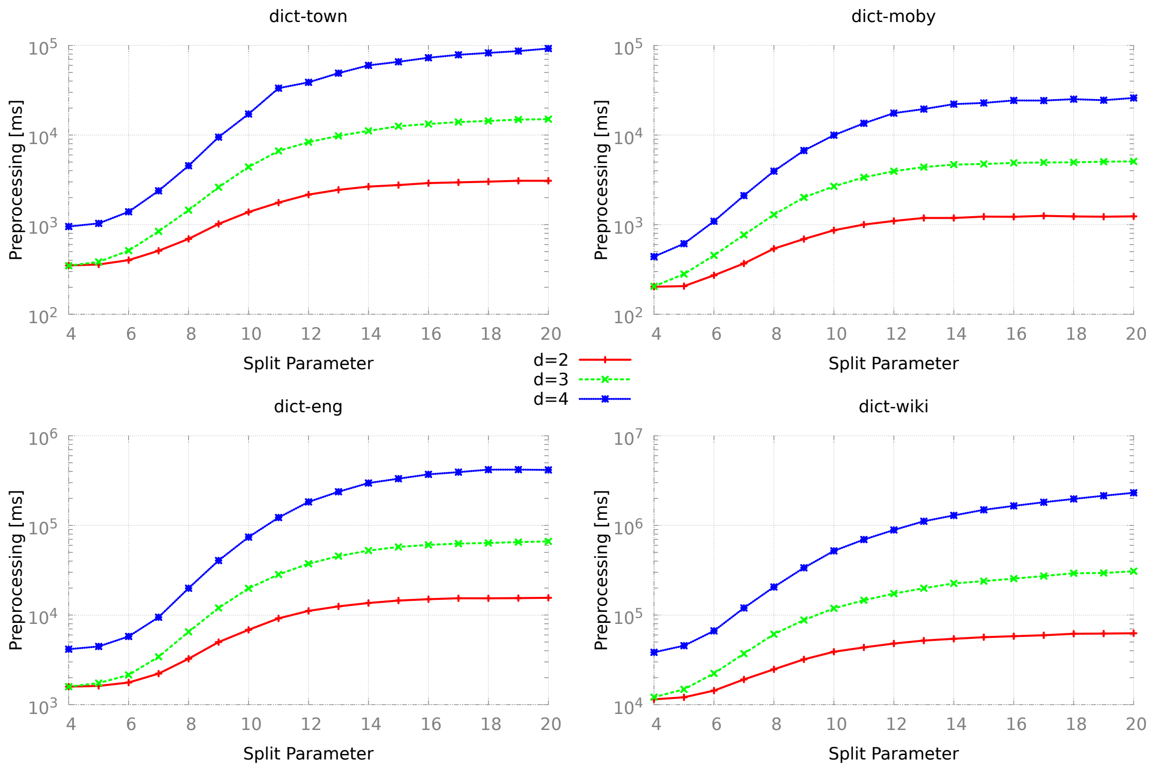


Figure 3.3.: Analysis of the Preprocessing Depending on the Splitting Parameter m.

do so, we averaged over 1 000 randomized queries. Both value $m = 1$ and $m = 30$ resemble worst cases. We present the results in the plots of Figure 3.2 for edit distance $d = 2$. Other distances show similar behavior. Note that we omitted the lower and upper values of m for clearer arrangement, because for the values $1, \ldots, 5$ $(20, \ldots, 30)$ nearly all (none) strings get split. We present selected plots that show the experiments. Note the logarithmic scales for query times. In all the experiments we see that there is a trade-off between the memory consumption and average query time. The split parameter functions as an adjusting value to choose between size of the index and query performance. Our analysis shows that the index size can be halved by degrading the speed of an average query within acceptable limits only. Especially, when splitting is restricted to those dictionary entries whose length is larger than the average, we can halve the memory consumption of the index. The query performance is virtually unaffected.

Preprocessing Time. We investigated preprocessing times with and without splitting parameter set. The preprocessing was run for values $d = 2, 3, 4$ on all of our data sets. Figure 3.3 reports on the numbers. The preprocessing is roughly ten times faster for reasonable values of the splitting parameter than without any splitting. Mainly this is because we do not store any additional information besides pointers to dictionary entries.

	dict-moby		dict-town		dict-eng		dict-wiki	
d	mem	proc	mem	proc	mem	proc	mem	proc
0	0.25	0.061	0.46	0.156	2.36	0.886	14.41	7.131
1	1.33	0.320	1.79	0.576	8.55	3.450	55.84	32.287
2	4.57	1.272	6.91	2.483	30.49	12.596	170.79	107.289
3	9.78	4.044	15.18	7.458	61.37	36.309	342.18	270.506
4	16.09	14.647	27.20	28.144	105.75	117.970	603.35	922.521

Table 3.1.: Preprocessing Duration and Index Size Depending on the Maximum Allowed Edit Distance d.

Query Performance. We conduct experiments on each list for maximum distances of $d = \{0, \dots, 4\}$ to test the query performance for varying number of allowed errors. A distance of $d = 3$ is already large and even larger distances deliver matches that already look arbitrary to the human eye for natural languages. During each query we generate the candidate set, verified each member of the set and reported a best match found. Each test run picks 1 000 elements from the dictionary and introduces up to d errors at random. First x is drawn uniformly at random from $\{0 : d\}$, and then x uniformly chosen operations of insertions, deletions or replacements are performed. The splitting parameter is set to $m = 10$. The query times and search space sizes are averaged. Table 3.1 and Table 3.2 report on these experiments. Column *Mem* is the size of the index in [MiB], *proc* the duration of index creation in seconds, while *query* is the average duration of a single query in microseconds, and *cand set* gives the average size of the candidate set.

The *query* column shows the time for the actual query in microseconds and *cand set* is the number of elements in the candidate set on average. We see the expected rise in the number of candidates that have to be verified by the algorithm. We briefly compared the observed number of collisions against the expected number from our analysis in Section 3.2. The observed number was always lower as the expected one since our analysis is an overestimate of the actual collision rate. In some cases we observed the order of a magnitude less collisions

	dict-moby		dict-town		dict-eng		dict-wiki	
	query	cand	query	cand	query	cand	query	cand
d	[μs]	set	[μs]	set	[μs]	set	[μs]	set
0	2	1	0	2	0	1	1	1
1	5	5	8	9	8	6	34	25
2	84	61	99	99	122	46	502	702
3	553	606	644	613	644	502	7 019	9 900
4	2 974	3 376	7 250	3 720	7 250	4 520	$55{\cdot}10^3$	$65{\cdot}10^3$

Table 3.2.: Query Efficiency and Candidate Set Size Depending on Maximum Allowed Edit Distance d.

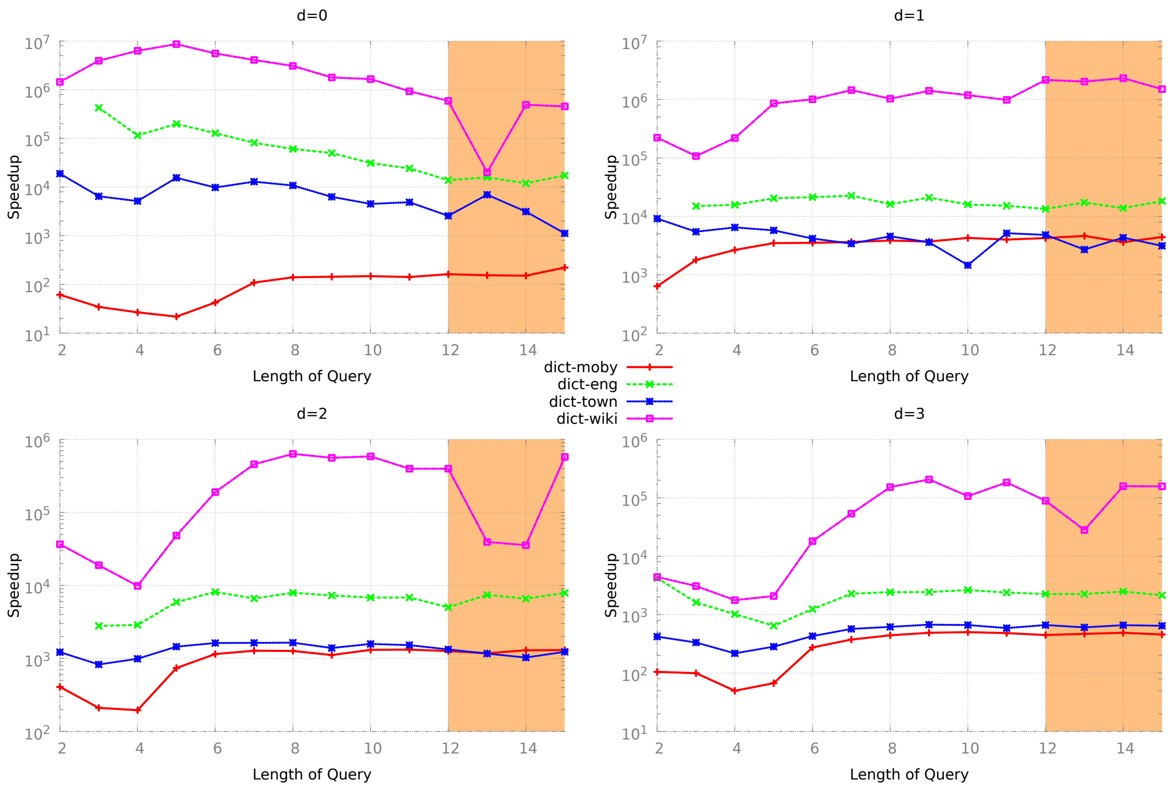


Figure 3.4.: Speedup over The Naive Algorithm Depending on Query Length and Maximum Allowed Edit Distance d.

than expected.

Next, we look at the speedup that we get compared to the naive algorithm of scanning the entire dictionary. We run queries with varying length against the dictionaries of our test instances for a edit distances $d \in \{0, \ldots, 3\}$. Each query is a randomly selected from the dictionary and changed with d random edit operations. The query length are from the range of $[2 : 15]$. Figure 3.4 gives the plotted results. Here, we define the speedup to be the number of candidates as a fraction of the dictionary size. We observe a bit variance of the speedups, but note that we see the better speedups the bigger the input dictionary is. This is expected behavior from the analysis of Equation (3.1). We see rather good speedups for queries of length up to 12. Generally speaking, the less error the dictionary has to cope with, the better the speedups. And we see best numbers for searching the Wikipedia dictionary with no error as expected. For queries with length greater than 12 we observe more variance for the large Wikipedia dictionary. This area is marked in the plot by the background color. The fluctuation of the speedup has several reasons though. First, queries get rather long, we expect a number of hash collisions that increase the candidate size. Second, since we are dealing with a natural language, we cannot expect the data to be as (randomly) benevolent as in the aforementioned analysis. However, we conclude that we see substantial speedups in practice over the naive algorithm of linear scanning.

		$m = \infty$	$m = 10$	Bocek et al.	BK-tree
preprocessing	[ms]	2 649	**349**	5–7.5·10^3	**183**
avg. query	[μs]	114	**18**	100–200·10^3	935
total size	[MiB]	9.8	**1.5**	20	**0.25**

Table 3.3.: Reproducing a Previous Results from Literature on Randomly Generated Instance of 10 000 Words. Best Results Printed in Bold.

When comparing our results with previous experiments of Bocek et al. [179] in Table 3.3, we see that our implementation performs better by about an order of magnitude in all important areas. Although we know that our numbers are measured on different hardware, they give an impression on the performance. For example, we repeat the experiment on a random dictionary of 10 000 words. Note that the case of $m = \infty$ corresponds to Bocek et al.'s algorithm. They also propose several improvements that *either* perform fast or have low space consumption, but not both at the same time. Since the results of the experiments are only available as plots we have to estimate the values. We do so in a benevolent way and compare the best of their values in each category against our implementation with and without splitting of tokens.

Comparison against BK-Trees. A previous experimental evaluation of the classical BK-tree data structure and several variants [143] reports on the size of the search space that is visited depending on the error distance. Those experiments were conducted on a set of 100 000 English words. We see a nearly linear growth of the visited search space for BK-trees going up from about 5% for edit distance 0 to slightly more than 40% for a distance of 4. We are able to confirm the high number of candidates reported previously with our own BK-tree implementation on our smallest and also on our largest test instance. The size of the visited search space in our experiments is always less than 1% and much less than the search space size for the best BK-tree variant.

Table 3.4 reports on the results of this experiment in more detail. BK-trees have the fastest preprocessing and a low memory footprint. On the other hand, our preprocessing is less than a factor of two slower for a reasonable value of m. The query is more than 50

	dict-moby		dict-wiki	
d	query [μs]	cand set	query [μs]	cand set
1	198	197	1 258	1 184
2	3 586	4 127	94·10^3	116·10^3
3	8 722	10·10^3	374·10^3	486·10^3
4	13 083	15·10^3	862·10^3	802·10^3

Table 3.4.: Evaluation of the query performance of BK-trees.

times faster as the number of candidates in BK-trees is quite high even for small allowed error distances. Thus, the filtering effect of exploiting the metric space is quite low. Also, we conclude that simple BK-trees are not suitable for a setting where queries arrive at a high rate and need to be answered as fast as possible.

Sounding a Note of Caution. We see a potential source of performance problems with our experiments as we tested on dictionaries of words from a natural language. The elements in the dictionary are rather short words that also have similar sizes. The higher the allowed error distance d is, the shorter residual strings get for these rather short words. This leads to longer indices lists in the hash table, as it is more likely that two distinct words have common residual strings. This also explains the generally larger number of candidates for higher values of d in our experiments.

3.3. Efficient Error-Correcting Geocoding

We assume that the reference data for geocoding contains an inherent structure, e.g., streets belong to towns and districts. Furthermore, we assume that a user is more likely to omit information then to over-specify a query.

Inverted Indices and Town Lists

We view place and street names as (very short) documents containing a sequence of *tokens* separated by white space, commas or hyphens. Thus, we can use methods known from full text search to support fast geocoding. In particular, we build two *inverted indices*, i.e. the town index maps tokens appearing in town names to the towns using that token in their name and the street index maps tokens appearing in street names to all streets containing this token.

In addition to the above inverted index, we precompute the set towns(s) of town IDs containing a street with name s and also a set towns(t) of town IDs with name t. This translation to town IDs enables us to quickly determine which combinations of town and street name correspond to actual addresses.

Ignoring Light Tokens. While indexing by token makes the index more convenient to use, it introduces a serious problem. A street query of the form "`New Hollywood Street`" returns *every* street that matches *any* of the tokens "`New`", "`Hollywood`", or "`Street`". In fact, about 25% of all street names in out data set match against the token "`Street`"[1], therefore we would get a really big candidate set. A simple and naive solution to the problem is the use of a list of *stop words* [**?**], also called *negative dictionary*, which filters tokens that would are deemed to be of poor quality. For example, words like `and` or `to` are such words for any text in English literature. They just occur too often to carry meaningful information with a reasonable likelihood. As such, stop words do not get indexed. The manual generation

[1]It is actually the German word *Straße*, which means street in English.

of stop words is seen as error-prone as it reflects the creators subjective views. Therefore, the automatic generation of stop word lists has been the focus of research, e.g. [188]. This approach works with reasonable success, but has limited applicability in our case.

As mentioned above, we view place and street names as very short documents and the real issue with stop words in our setting is that they are *global* and apply to the entire index. But tokens that may be irrelevant for one name may be relevant for another. Thus, the importance of a token depends on the setting. Consider the following example. Suppose that our reference data contains two streets named “`New Rhododendron Alley`” and “`Alley Street`”. Then “`Alley`” is arguably *relatively unimportant* compared to “`Rhododendron`”, but it is more important when compared to “`Street`”, because it occurs less often. To avoid this problem, we turn to a concept that does not use a global list stop words, and which is often used in information retrieval and text mining, e.g. [13, 43, 191]:

Definition 1 (Inverse Document Frequency (IDF)). *The* inverse document frequency *of a token c with respect to a set M of strings (town or street names in our case) is defined as*

$$\mathrm{IDF}(c) := \log \frac{\sum_{x \in M} |\,\mathrm{tokens}(x)|}{|\,\{x \in M : c \in \mathrm{tokens}(x)\}\,|}$$

where $\mathrm{tokens}(x)$ *is defined as the set of tokens making up string x.*

Tokens that occur very often in the document (such as “`Street`”) receive a lower IDF weight than those that appear only infrequently. Tokens that receive a high weight are more helpful in identifying the correct string, because they match fewer strings in the index.

We use these observations to our advantage by making further assumptions on user behavior: When a user enters an address that they want to have geographically referenced, they may leave out parts of the address that they deem irrelevant, but they will probably enter those parts of the query that non-ambiguously defines what they are looking for. In our example, the user may leave out either “`New`” or “`Street`”, but they most definitely won’t leave out the token “`Hollywood`”, which is also the token with the highest IDF weight among those three. If we expect the user to enter the most important part of an address, it is not necessary to have said address be referenced also by the remaining, unimportant tokens. I.e. we don’t want to find the street “`New Hollywood Street`” by the token “`Street`”, because we expect that the more descriptive token “`Hollywood`” is entered anyway. Let

$$w_t(s) := \frac{\mathrm{IDF}(t)}{\sum_{t' \in \mathrm{tokens}(s)} \mathrm{IDF}(t')}$$

be the *relative weight* of the token t in the string s. If we decide that the query must contain tokens that make up a fraction μ of the total weight, then we can ignore the lightest k tokens, if the sum of their weights is not greater than some tuning parameter μ. Example 2 illustrates the intuition behind the application of IDF.

Example 2 (Application of IDF). *Assume the following relative weights for the street names “*`New Hollywood Street`*” and “*`Alley Street`*”:*

- w_t(“`New`”) $= 0.26$*,* w_t(“`Rhododendron`”) $= 0.6$*, and* w_t(“`Alley`”) $= 0.14$*, and*

- w_t("`Alley`") $= 0.72$, w_t("`Street`") $= 0.28$.

For a threshold of $\mu > 0.4$ the tokens "`New`*" and "*`Alley`*" is ignored for the first street name, while "*`Street`*" is ignored for the second name only.*

Hence, a user searching for "`Alley`" is presented with "`Alley Street`" in this simplified example. Note that this *local* definition of importance is very different from the stop words. Instead of ignoring some tokens entirely, we do so only very selectively.

Token-Based Approximate Indices. We build indices supporting approximate search on the lists of *tokens* appearing in town names and street names respectively. Note that we do not build an index for every separate town city or region. This design decision has two crucial advantages. First, the dictionaries are much smaller than the full data base as the data exhibits some redundancy. In particular, our input data set of Germany contains 1.35 million streets but only roughly 219 000 distinct tokens for street names. Thus, we can afford super-linear space to some extent. Second, token based indices can easily handle queries that drop part of the town or street name. For example, most users just type "`Frankfurt`" when they are looking for "`Frankfurt am Main`". In other words, the generally used name may not be the official/administrative one. The approximate index of Section 3.2 for a token set M with maximum error d_i can be queried with a string q and returns a set $M_q \subseteq M$ of tokens that have edit (Levenshtein) distance at most d_i.

Multi-Field Search

We first concentrate on the case where a query consists of two strings typed into separate fields for town name and street name. Note that in this case it is easy and largely orthogonal to allow additional fields for house number or ZIP code. After a normalization step, we tokenize the query and construct the deletion neighborhood as explained above. We try three increasingly sophisticated ways to obtain candidate sets $C_\mathcal{T}$ and $C_\mathcal{S}$ of towns and streets that *could* match against the query. Each of these steps allows an increasing number of errors. After each of these attempts, we combine the intermediate candidate sets into *compatible* candidate addresses from $C_\mathcal{T} \times C_\mathcal{S}$. We stop as soon as we have found a satisfying solution. The following paragraphs explain each of these phases. We detail the individual phases and give a technical description afterwards in Listing A.1.

Initialization Phase. The input strings are first scanned and transformed into a set Q of tokens and normalized to lower-case. This is also the place where some culture-specific preprocessing can be done. In our German implementation, there is only one such speciality: The German words for "`Street`", "`Lane`", ... are sometimes used as a separate word and sometimes as a suffix and nobody really knows which version is correct in every case. Hence, compounds with these suffixes are broken into a normal form with separate tokens. This even works when the suffixes are misspelled or abbreviated.

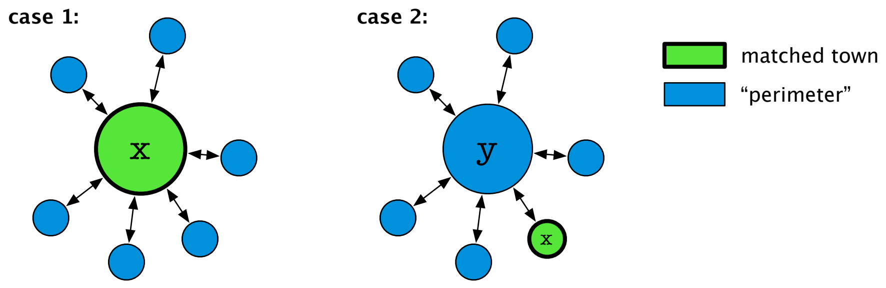


Figure 3.5.: Two Cases of Periphery Search. Matching Districts to City (left) and Matching of City to one of its Districts (right).

Partially Exact Town Match. Following the successful principle of "make the common case fast", we use a simplified special treatment for the case of a *partially exact town match* where at least one sufficiently rare token of a town candidate is exactly matched. If this already yields a plausible result, we stop. For example, in the query "`Franfrt am Main, Römerberg`", "`Frankfurt`" was misspelled. But the token "`Main`" is an exact match and therefore we consider the set of towns that contain this token somewhere in their name before we look at all possible approximate matches. If we find a street similar to "`Römerberg`" in one of the candidate cities, we return it. Note that assume that only relevant city tokens are in the index since we filter the tokens to insert by IDF.

Periphery Search. Assume that we successfully identified partially exact town matches during the first phase, but could not match a street in any of these towns with a sufficient rating. Then we extend the scope of exact search to the *periphery*: If the input specifies a city, we try all its districts, if it specifies a district, we try the city it belongs to and all its districts. We assume that each town is either a principal town, i.e. a city, or a district. If a candidate x is a city, we search for the street also in all districts of x. If a candidate x is a district of the city y, we search for the street also in y as well as in all of its districts. See Figure 3.5 for an illustration of the two cases.

If the town provided by the user is matched against a candidate x which is subsequently corrected to a town y in the periphery of x, we still calculate the rating for x, because the name of y generally does not match anything in the query string and would lead to a low rating. We give a way to automatically construct this proximity information when it is not available in the input data in Section 3.4.

Approximate Search. When there are no or no good partially exact matches or when even periphery search does not find a good candidate, additional candidates are computed using the approximate indices for towns and streets. If a town candidate found specifies a district x of a city y, we also add y to the candidates. However, we do not do a full scale approximate periphery search because this could easily yield results that are hard to understand.

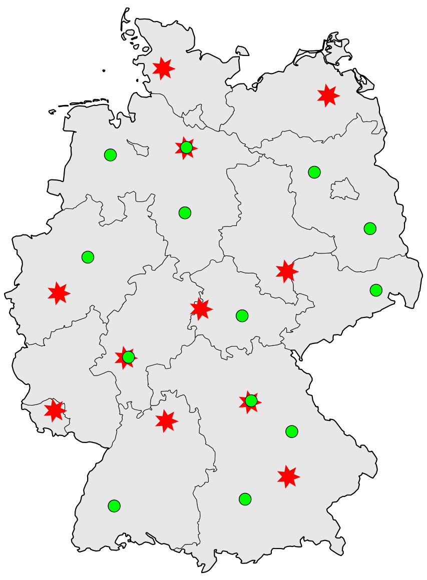


Figure 3.6.: Town candidates are indicated by stars, street candidates by circles. Markers without geospatial proximity. can be dropped.

Compatible Candidates. After partially exact matching, periphery search, or approximate search, that all treat towns and streets separately, we generate address candidates where town and street are *compatible* with each other, i.e. these candidates either spatially overlap or are very close. A pair $(t, s) \in C_{\mathcal{T}} \times C_{\mathcal{S}}$ is compatible if a street with name s is present in some town with name t, i.e., we have to compute the set

$$C_{\mathcal{T}\times\mathcal{S}} := \{(t, s) \in C_{\mathcal{T}} \times C_{\mathcal{S}} : \text{towns}(t) \cap \text{towns}(s) \neq \emptyset\} \quad .$$

There are various ways to do this more efficiently than the naive way of computing $|C_{\mathcal{T}}| \times |C_{\mathcal{S}}|$ set intersections. As a first step, we can completely drop a town candidate $t \in C_{\mathcal{T}}$ if no town with this name contains any street with a name in $C_{\mathcal{S}}$. Vice versa, a street candidate $s \in C_{\mathcal{S}}$ can be dropped if there is no town with name $t \in C_{\mathcal{T}}$ that has a street with name s. Figure 3.6 visualizes this process.

While these tests may sound complicated on the first glance, involving a lot of string comparisons, we can reduce it to operations on precomputed sets of town IDs as follows: Let $\text{towns}(C) := \cup_{c \in C} \text{towns}(c)$ where $C = C_{\mathcal{T}}$ or $C = C_{\mathcal{S}}$. We can drop a town candidate t if $\text{towns}(t) \cap \text{towns}(C_{\mathcal{S}}) = \emptyset$. Vice versa, we can drop a street candidate s if $\text{towns}(s) \cap \text{towns}(C_{\mathcal{T}}) = \emptyset$. The experiments of Section 5.5.3 show that dropping incompatible candidates is very effective and reduces the overall runtime by a factor of 3. In a second step, we generate the set of compatible candidate pairs

$$C_{\mathcal{T}\times\mathcal{S}} := \{(t, s) \in C_{\mathcal{T}} \times C_{\mathcal{S}} : \text{towns}(t) \cap \text{towns}(s) \neq \emptyset\} \quad .$$

Consider dictionaries for the city and street tokens (`street_index` and `city_index`) that have been preprocessed for a maximum distance d. Also, consider functions `tokenize`()

that returns the tokens of a string, `normalize`() that returns a normalized string, and `get_cities`() that returns the (indices of) cities where a street name occurs while `periphery`() returns the (indices of) cities or rather districts in the vicinity of another city. Assume that a `rating`() function evaluates the quality of a potential result. We explain the details of our rating in the following. Listing A.1 in Appendix A gives the pseudo code of the query.

Single-Field Search

In the previous sections we focused on separate fields for town and street because multi-field search is easier to program, and one should expect that it reduces errors. From the users points of view, however, it is more convenient to enter a query into a single text field, with street and town in arbitrary order, possibly delimited by white-space or a comma.

However, in order to compare multi-field search and single-field search and also in order to compare our approach with existing Internet services, we have also implemented a simple version of single-field search with an emphasis on quality. Our solution is based on the plausible hypothesis that the token sequence resulting from a single-field query resembles the following two regular expressions:

$$\mathsf{streetToken}^{*}\mathsf{townToken}^{+} \text{ or } \mathsf{townToken}^{+}\mathsf{streetToken}^{*},$$

i.e. strings of street and town tokens are contiguous and there is at least one token designating a town. We exhaustively try all $2m - 1$ possible ways to split a token sequence of length m and call a multi-field search for each of them. Consider the following example to illustrate the splitting process.

Example 3 (Splitting of "`Oxford Street London`").

Town	*Street*
`Oxford`	`Street London`
`Oxford Street`	`London`
`Oxford Street London`	
`Street London`	`Oxford`
London	**Oxford Street**

Only the last line would return a perfect rating as we might have hoped. This query is a bit lucky though since there is no "`London Street`" in "`Oxford`" which, if it existed, would also receive a perfect rating.

Rating Candidates

After we have dismissed most of the search space, we are left with a hopefully small set of compatible address candidates $(t, s) \in \mathcal{T} \times \mathcal{S}$. These are then *rated*. The result is interpreted using two threshold values $\underline{\rho}$ and $\overline{\rho}$. Ratings below $\underline{\rho}$ are considered unsatisfactory. If all results are unsatisfactory, more extensive search is done (after partially exact matching or periphery search) or, when everything failed, an empty result is returned. In contrast, if a candidate with rating $\geq \overline{\rho}$ is found, the search returns successfully without further attempts at refined searching. Depending on the application we can then return the top ranked candidate or a list of good candidates. To develop a rating heuristic, let us recall the different kinds of errors that we want to compensate for:

1. Typing errors

2. Missing or redundant tokens

3. Inconsistent pairing of a street and a town.

Since periphery search and candidate filtering have already dealt with inconsistent candidates, we are left with the first two issues. The first step on the way to a robust rating heuristic is to align the query to a candidate, i.e. find a good mapping from the tokens in the query to the tokens in the candidate (see Section 3.3). Based on this mapping, we then compute the actual rating.

The rating is computed separately for town and street by the same method and combined by the arithmetic mean afterwards. Hence, the following explanation details the town rating only. There is one small asymmetry however that we call *filter by edit distance*: Since there are usually less candidate towns than candidate streets, we first filter out candidates that are already unsatisfactory because they do not sufficiently well fit the town description in the query.

Matching the Query to a Candidate. To match the town tokens $q \in Q$ given by the users to the tokens of a candidate town name $c \in C$, we solve a *minimum weight perfect matching problem* on a bipartite graph. If $|Q| \leq |C|$ we add $|C| - |Q|$ *dummy* nodes to Q and obtain the matching graph $G = (Q \cup C, Q \times C)$ where the weight of edge (q, c) is the edit distance between q and c if c is not a dummy node and 0 if c is a dummy node. Edit distances take misspellings into account and dummy query nodes go a long way to model missing tokens in the query. Similarly, but perhaps less importantly, if $|Q| > |C|$ we add $|Q| - |C|$ *dummy* nodes to C. This matching problem can be solved in time $\mathcal{O}(n^3)$ in the number of tokens, e.g. [7, 120]. Moreover, the considered graphs are very small so that solutions can be computed quickly in practice. See Figure 3.7 for an illustration.

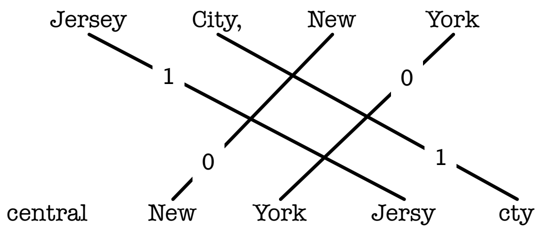


Figure 3.7.: Candidate (top) is Matched Against Query (bottom). Edge Labels Symbolize the (Pairwise) Edit Distances Between Tokens.

The Rating Heuristic. Based on the matching, we calculate a rating for each candidate. The ranking shall define an order on the set of candidates. We assume that the rating can be expressed by a numeric value. It should take into account the following considerations:

- Each token that matches with at most d errors should be awarded some points. It makes sense to choose d larger than the error bound d_i for the approximate index since space or index access time is no issue for the pairwise distance computations used for the rating function.
- Tokens that could not be matched with at most d errors should not be awarded points and may even be punished.
- The user is more likely to omit information (either because they forget it or because they deem it unnecessary) than to over-specify the query. Therefore, tokens in the query that don't match anything in the candidate should be punished higher than candidate tokens that don't match anything in the query.
- The rating should be a real number in the interval $[0, 1]$, with one denoting a perfect match.
- The heuristic should be able to distinguish between tokens that are *important* and tokens that do not provide much information.

Rather than directly using the edit distance, we also want to take into account the lengths of compared words, since the rate of error that can be introduced into a word with a constant number of changes depends on its length. Consider the following:

Definition 2 (Token Similarity). *Given two tokens q and c. Then their similarity is given as*

$$\mathrm{sim}(q,c) := \begin{cases} 1 - \frac{\mathrm{editDistance}(q,c)}{|c|}, & \textit{if } \mathrm{editDistance}(q,c) \leq d \\ 0, & \textit{else} \end{cases} .$$

We normalize the error rate by the length of c since candidates are entries that are actually present in our database. In order to take the *importance* of a candidate token into account, we use its inverse document frequency.

Let M denote the set of edges (d, c) between query tokens and candidate tokens that were matched with edit distance $\leq d$. Let U denote the set of unmatched query tokens, i.e., those tokens that could not be matched to any candidate token with at most d errors. We can now define our rating function

$$\begin{aligned}
\operatorname{rating}(Q,C) &:= \gamma \operatorname{rating}^Q(Q,C) + (1-\gamma) \operatorname{rating}^C(Q,C) \\
&\text{where} \\
\operatorname{rating}^Q(Q,C) &:= \frac{\sum_{(q,c)\in M} (\operatorname{sim}(q,c))^\alpha \operatorname{IDF}(c)}{\sum_{(q,c)\in M} \operatorname{IDF}(c) + |U| \operatorname{IDF}_{avg}}, \text{ and} \\
\operatorname{rating}^C(Q,C) &:= \sum_{(q,c)\in M} \operatorname{IDF}(c) / \sum_{c\in C} \operatorname{IDF}(c)
\end{aligned}$$

where IDF_{avg} is the average over the IDF-values of all town tokens. The term $|U| \,\mathrm{IDF}_{avg}$ expresses that the unmatched queries should have matched somewhere but we have no idea where – so we use an average IDF-value. The parameter α is used to adjust how important it is to have similar matches. Notice that rating^Q is not influenced by the number of unmatched *candidate* tokens. This is why we compute a convex combination of rating^Q with rating^C which penalizes unmatched candidate tokens. The parameter $\gamma \in [0,1]$ specifies the relative weight. Usually we want to give more weight to the matched parts of a query, therefore we choose $\gamma > 1/2$.

3.3.1. Experimental Results: Error-Correcting Geocoding

Methodology and Implementation Details. We implement the algorithms and data structures described above in C++, using GCC's C++ compiler version 4.3.2 with full optimization. The experiments were performed using a single CPU core on Machine A, running Linux kernel version 2.6.27. Experiments are done on test instance *ptv-addr*, a commercial data set from 2009 by PTV AG for this evaluation. The indexed dictionary occupies about 327 MiB of main memory, which is roughly an order of magnitude more than the input data.

The following tuning parameters are chosen: We ignore light tokens comprising up to 40% of the cumulative IDF of a name. The correction limit of the approximate dictionaries and pairwise edit distance computations is limited to $d = d_i = 2$ in order to keep space consumption low. Candidate matching uses the Hungarian method [7] using the implementation by [120]. The rating function takes similarities between matched words to the power $\alpha := 2$. For the convex combination

$$\operatorname{rating}^Q(Q,C) = \gamma \cdot \operatorname{rating}^Q(Q,C) + (1-\gamma) \cdot \operatorname{rating}^C(Q,C)$$

we choose $\gamma := 3/4$. The threshold for a satisfactory rating is $\underline{\rho} := 1/2$ and a *good* rating starts at $\overline{\rho} = 4/5$.

We use a set of existing, *relevant* addresses R, and a set of non-existing, *irrelevant* addresses I. A relevant address is sampled by first choosing a random street name s and then picking a random town from $\operatorname{towns}(s)$. An irrelevant address is composed of randomly chosen town and street names such that combination which accidentally occur in the database are rejected. Ideally, we would like to return correct results for relevant address queries and no result for irrelevant address queries. To generate a simple random query, it would be easiest to just insert, delete or substitute random characters in an existing address. The errors that are introduced this way, however, are unlikely to resemble the errors that a human would make while entering a query through a keyboard. We identify several sources of errors to generate input sets with more realistic errors.

Typing errors are very common and we distinguish between:

- swapped characters

  Example: "`Frankfurt`" → "`Frankfrut`"

- missing characters

  Example: "`Frankfurt`" → "`Franfurt`"

- superfluous or wrong characters, mostly closely located to the correct character on the keyboard (here in terms of the German QWERTZ-layout).

  Example: "`Frankfurt`" → "`Frankdfurt`" or "`Frankdurt`"

Depending on the respective language there are several phonetic error sources:

- doubled characters where there should be a single character

  Example: "`Dublin`" → "`Dubblin`"

- single character where there should be two of the same

  Example: "`Cardiff`" → "`Cardif`"

- The SOUNDEX algorithm, e.g [116] pp. 391–92, identifies classes of characters such that different characters from the same class differ only slightly in their pronunciation.

  Example: `z` ≡ `s`, "`Zaragoza`" → "`Saragosa`"

- Two consecutive vowels that occur in the same syllable are called a *diphthong*. In German, for example, several different diphthongs sound the same or similar:
  - `ei` ≡ `ey` ≡ `ay` ≡ `ai`
  - `eu` ≡ `äu` ≡ `oy` ≡ `oi`
  - ...

  Example: `Hoyerswerda` → `Heuerswerda`

Preliminary Experiments. We use several techniques, as described in Section 3.3, to make sure that the number of candidates stays small and that we don't have to perform too many edit distance computations. To see if these techniques are necessary and how each of them affects the query time, we have performed a number of experiments. The techniques are:

- Filter Incompatible Candidates (FIC): We keep only those town and street candidates that are geographically compatible.

ILT	FIC	FED	[ms]
×	×	×	570.00
×	×	✓	566.00
×	✓	×	199.00
×	✓	✓	126.00
✓	×	×	10.68
✓	×	✓	10.58
✓	✓	×	3.45
✓	✓	✓	2.09

Table 3.5.: The Effect of The Features ILT, FIC and FED on Query Efficiency.

- Filter by Edit Distance (FED): We have an additional filtering stage that drops some candidates before doing a full rating evaluation. A candidate can eliminated when the town names are an unsatisfactory match.
- Ignore Light Tokens (ILT): We can ignore some tokens during the construction of the index due to their weight in comparison to the other tokens. E.g. the candidate "`New Hollywood Street`" is represented only by "`Hollywood`", because the other two tokens occur so frequently in the dictionary that they would not be of much help to distinguish this candidate from others.

As we can see in Table 3.5, disabling ILT absolutely destroys the performance. To see why this is so, consider the most frequent token in the street dictionary, "`street`". If we randomly choose any street, the probability that it contains the token "`street`" is about 1/3. Without ILT, *any* query that contains the token "`street`" returns *all* candidates that contain this token. Hence, if we randomly choose a candidate and query the index with this candidate, we expect a candidate set that contains more than 1/9-th of all streets on average. This amounts to almost 50 000 candidates for our data. In our experiments, the actual number of candidates was even bigger, as we observe 67 000 candidates on average. The query time drops by a factor of almost 100 when we enable ILT. FIC gives us another boost of factor 3 and FED makes a difference only when used in conjunction with FIC. None of these features has a noteworthy effect on the memory requirements of the index, therefore all of them are enabled by default.

Query Results. In order to introduce k errors into an address, we introduce $\lceil k/2 \rceil$ errors into the street string and $\lfloor k/2 \rfloor$ errors into the town string. To introduce an error, we first pick a random error class, then a random token, and then a random position. All distributions are uniform. Here we report experiments on 1 000 relevant and 100 irrelevant addresses. We classify the results returned by the index as follows:

- *True Positive (TP):* A relevant address that is correctly identified.
- *True Negative (TN):* An irrelevant, i.e. inexisting, address that is either not matched at all, or a correct partial result , e.g., the correct town is matched.

error	relevant			irrelevant		Time
no.	TP	FN	II	TN	FP	[ms]
0	1 000	0	0	93	7	3.02
1	989	10	1	95	5	2.75
2	988	11	1	94	6	2.44
3	928	66	6	94	6	2.40
4	854	140	6	99	1	1.79
5	557	431	12	97	3	1.59

Table 3.6.: Multi-Field Search Matching Rates and Query Efficiency for Random Addresses – 1 000 Relevant and 100 Irrelevant Ones.

- *False Positive (FP):* An irrelevant, i.e. inexisting, address where the index does return a result.
- *False Negative (FN):* A relevant address where the index does return a result.
- *Incorrectly Identified (II):* A relevant address that returns an incorrect result, i.e. another relevant address.

The matching rates, along with the query times, are shown in Tables 3.6 and 3.7. Multi-field search works extremely well, both with respect to result quality and query time. Only at five errors, when three errors are introduced into the street name, we see a sharp increase of false negative results. At this point, the approximate street index often fails to find the right result because its error limit is set to $d_i = 2$. Interestingly, query times *decrease* with the number of errors. The reason is that we have to check a smaller number of candidates both in the approximate index and when rating candidates.

Single-field search for relevant addresses works almost as well as multi-field search. The only noticable difference is that a small fraction of the false negative results mutates into incorrectly identified results. For irrelevant addresses, our current implementation seems to be too aggressive though because it returns a significant number of false positives. On the first glance it looks paradoxical that we get a larger number of output errors when there

error	relevant			irrelevant		Time
no.	TP	FN	II	TN	FP	[ms]
0	1 000	0	0	52	48	26.07
1	989	10	1	63	37	23.33
2	986	13	1	74	26	19.72
3	927	66	7	75	25	18.44
4	856	125	19	80	20	16.69
5	560	414	26	86	14	14.31

Table 3.7.: Single-Field Search Matching Rates and Query Efficiency for Random Addresses – 1 000 Relevant and 100 Irrelevant Ones.

d	Bing	Google	Ours
0	87	97	100
1	78	54	98
2	71	3	98
3	59	3	94
4	55	2	86

Table 3.8.: Rate of Resolved Queries for Several Geocoder APIs.

are no spelling errors in the input. But the reason is simple: without spelling errors we obtain higher ratings for the generated candidates and thus it becomes more likely that the result is accepted. This indicates that the result quality could be improved by increasing the threshold for accepting a result. Single-field query times are an order of magnitude larger than for multi-field search. This is not surprising, since our current implementation naively factors a single-field search into several multi-field searches.

Figure 3.8 shows the distribution of query times for the same set of 5 × 1 100 queries. 90% of all multi-field queries finish in less than 5 ms. The maximum query time observed is 106 ms. Note that for the server scenario we are considering, we want very low *average* query time to achieve high throughput and low cost. Occasional slower queries are no problem as long as they do not lead to noticeable delays for the user. Also, a duration of 100 ms is well below the delays users are accustomed to experience due to network latencies anyway. Although single-field search is an order of magnitude slower on the average, this slow-down does not translate into a proportional increase of the slowest query times – we still remain below 406 ms. This indicates that the query times of the generated multi-field sub-queries are not strongly correlated.

We also compared against existing geocoding services[2]. To do so, we took the first 100 relevant queries from the above query set and ran them against the publicly available APIs of Bing and Google Maps. Table 3.8 reports on the number of true positive results. Google

[2]Experiments run on March 29th, 2013.

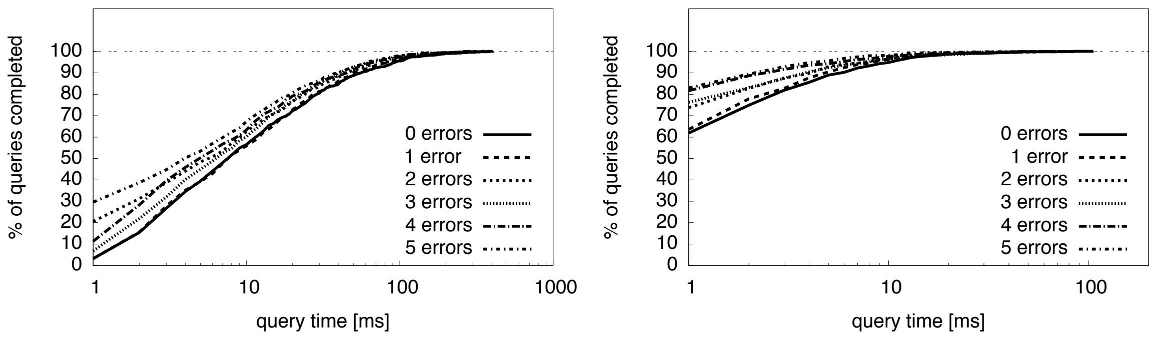


Figure 3.8.: Query Times of Single-Field (left) and Multi-Field (right) Search Depending on the Numbers of Distortions.

works very well for undistorted inputs – the three remaining errors could perhaps be attributed to differences in the input data. But already for a single error, the recognition rate drops to 54 % and completely collapses for $d \geq 2$. Bing already has significant deficits at $d = 2$ but fails more gracefully for distorted inputs. Still, the number of failed answers is an order of magnitude larger than for our system. It should be noted that the subjective performance of Google with interactive use on Google-Maps is much better than over the API. In particular, the auto-completion mechanism works very well but is obviously not applicable to the API.

Real-world Queries. To get an impression of the quality of the results, we also did experiments with real-world input, i.e. actual queries that have been provided by users of the existing geocoder of PTV AG. The test data consists of 1 383 multi-field queries that were logged from the users of PTV AG. The results were pre-classified by the company into five categories. We briefly describe the categories.

- Exact: Queries where each token can be matched without errors to a token in the result. The differences allowed between query and result are those that are handled by a normalization phase.

  Example: "`london tally road`" → "`London, Tally Road`"

- Partially Exact: Queries where each token occurs in the result, but not each token of the result string occurs in the query.

  Example: "`london tally`" → "`London, Tally Road`"

- High/Medium/Low: Queries that contain errors. The labels High, Medium, and Low were assigned depending on the *confidence* of the existing system to have a correct interpretation of the input.

  Example: "`Lodon; Tall Rd`" → "`London, Tally Road`" is Medium.

We tested our address search with these queries checking correctness of our result manually. The results are classified into three categories:

- Strong Match (s): A result that is unquestionably correct. In many cases the query was non-ambiguous and easy to verify.
- Weak Match (w): A result that is not correct, but has successfully identified parts of the query, e.g., the town.
- No Match (n): Either a result that is definitely incorrect, or no result at all.

In total, the test data consists of 1 383 queries, classified into 844 Exact, 357 Partially Exact, 125 High, 41 Medium, and 16 Low queries. Since they had to be verified by hand, we picked random samples of at most 100 queries per class. To make the comparison fair, we removed a number of queries that were not resolvable, because our index does not contain points of interest, e.g., shopping malls, water parks, etc. There remained 100 exact and 99 partially

		Bing			Google			Ours		
category	#	s	w	n	s	w	n	s	w	n
Exact	100	81	5	24	100	0	0	100	0	0
Partial	99	77	20	2	96	2	1	99	0	0
High	81	60	16	5	65	10	6	77	2	2
Medium	34	7	1	26	16	11	7	27	6	1
Low	15	1	1	12	9	4	2	13	1	1

Table 3.9.: The Match Rates of Several Geocoding APIs on Real-World Queries.

exact queries, as well as 81, 34 and 15 queries classified as high, medium and low. The results are shown in Table 3.9.

We run the queries against the public APIs of Google and Bing. [3] Our system outperforms the Google and Bing API in all categories. As with the real world inputs, Google is similarly good for (partially) exact queries but looses ground for erroneous inputs. For example, for category *High* our system returns correct results for all but 4 of the inputs whereas Google fails for 16 inputs – a factor four in the failure rate. An interesting difference to the random inputs is that now Google consistently outperforms Bing – also for the inputs with errors.

3.4. Neighborhood Graph

In most countries, city names are not unique. For example, Wikipedia knows 41 Springfields, 5 of them in Wisconsin. We present an efficient data structure here that can be used to find plausible nearby cities for disambiguation. Applications could either use the data structure to propose disambiguating places or they could match them to inputs of the user such as "`Springfield near Kansas City`".

What makes a certain city a plausible match? It should be large, and it should be close to the city it is supposed to disambiguate. Hence, we are facing a bi-criteria optimization problem. A save way to handle such a situation is to consider all cities that are not dominated by any other city.

Definition 3 (City Dominance). *City x dominates y with respect to city t if it is both closer to t than y and larger than y.*

This problem is known under several names: Finding *Pareto optima*, *vector maxima* [121], or a *skyline* [31]. If the cities are sorted by size, it is easy to solve the problem with a full scan of all cities – outputting a city if it is closer than all previously inspected ones. However, looking at all cities may still be too slow on large data sets for a practical application.

We encode the required information into a graph. Assume the cities are numbered 1 to n by decreasing size. There is no need to store all towns, only those cities big enough to serve as a reference place. A convenient way to define the size of a city here is to just count the

[3] Originally, this was done in June 2011 and we reran them prior to finishing the thesis. We did not observe significantly different results but noticed that the API of Bing has become sensitive to semicolons in the input.

Listing 3.4: Finding Pareto optimal cities in the neighborhood graph D^*.

```
function pareto_optima($q$)
do
  output $u$
  for each neighbor $v$ of $u$ in $D^*$, with $u < v$ in increasing order do
    if $||q - \mathsf{position}(v)||_2 < ||q - \mathsf{position}(u)||_2$ then
      $u := v$
      break
    end
  end
while no closer node found
end
```

number of streets. Let $D_i := (\{1, \ldots, i\}, E_i)$ denote the Delaunay triangulation [28] of the i largest cities[4].

Then we consider the *neighborhood graph* $D^* := (\{1, \ldots, i\}, \cup_{i=1}^{n} E_i)$. Depending on the application, we may interpret D^* as a directed acyclic graph where all edges go from smaller to larger indices (downward) or vice versa (upward). The intuition here is that the Delaunay triangulation encodes a natural concept of proximity. Directing the edges *upward*, i.e., towards larger cities allows us to find such cities. Obviously, it is not enough to just consider the Delaunay triangulation D_n of all n cities since we would usually end up in a dead end of a medium sized cities whose neighbors are all smaller. The following theorem states that the union D^* of n Delaunay triangulation solves this problem very effectively:

Theorem 1. *Consider any map position (x, y). The Pareto optima with respect to city size and closeness to position (x, y) forms a downward path from node 1 to the city closest to (x, y). This path can be found with the simple greedy algorithm depicted in Listing 3.4.*

Proof. By induction from 1 to n, analogous to the work of Birn et al. [30]. □

Of course it is important how long it takes to construct D^* and how much space it takes. If the size of a city is assumed to be independent from its position, the problem is the same as in *randomized incremental construction* of a Delaunay triangulation [92] – we get a linear number of edges in time $\mathrm{O}(n \log n)$.

In case, the *importance* of places is unknown, there exists a simple generalization of the above method. We construct a Voronoi diagram of the entire set of places and run an adapted BFS query on the corresponding graph. All query candidates are inserted into the queue with hop distance 0 and then a BFS is run. It prunes the search when a predetermined threshold on the hop distance has been reached. For each settled node that corresponds to an *acceptable* match, we compute a ranking that can be based on hop distance and probably a number of other features and return a ranked list of the top k matches.

[4] Note that for convenience we work with Euclidean geometry here.

Description	Count
city	87
town	2 404
village	37 850
hamlet	34 850
isolated dwelling	927
farm	580
suburb	8 424
neighborhood	1
Total	85220

Table 3.10.: Extracted Place Features. Hierarchically Sorted from Top to Bottom.

3.4.1. Experimental Evaluation: Neighborhood Graph

We implement the hierarchical neighborhood graph in C++ using GCC's C++ compiler version 4.7.2. The implementation uses the incremental Voronoi diagram construction algorithm of the CGAL[5] library version 4.02. The (informal) OpenStreetMap mapping rules define a hierarchy of *places*[6] that resembles the size of the place. Experiments were run on Machine D, running Linux kernel version 3.5.0. Table 3.10 gives an overview of the number of extracted features and in the order of the hierarchy of importance.

We build a neighborhood graph D^* for all 85 220 *places* in our input. The construction takes 2.58 seconds, including initial sorting. The resulting graph consists of 255 635 directed edges, i.e. about 3 directed edges per node which is similar to what we would expect for random city sizes. About 25 nodes are reachable from a city on average, saving a factor 3400 compared to a full scan of the city table, even if we do not fully use Theorem 1.

[5]`http://www.cgal.org` – accessed February, 25th 2013
[6]`http://wiki.openstreetmap.org/wiki/Key:place` – accessed February, 25th 2013

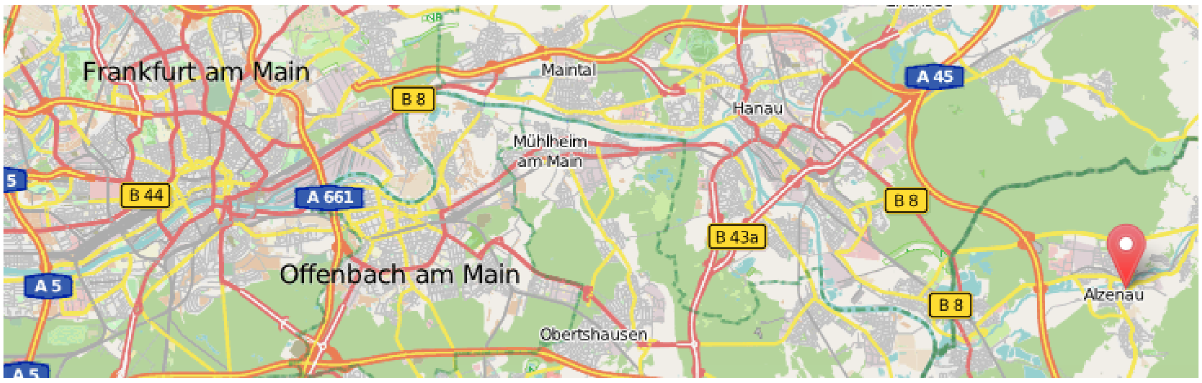


Figure 3.9.: Example From the Experiments: Town of *Alzenau* (marker) is Close to the City of *Frankfurt am Main*. (Image © OpenStreetMap Contributors, CC-BY-SA).

Viability of Computational Results. We also performed a small user study to test how reasonable the results of our modelling are. We randomly picked 48 towns from a map of Germany and asked a non-expert in the field of geography or computer science to attribute a larger reference town to the one we picked. In 46 of these cases, a pareto-optimal reference town was chosen. Therefore, we conclude that the results are reasonable. Figure 3.9 shows an example from the experiments.

3.5. Concluding Remarks

Improved Fast Similarity Search. We improved a method for approximate string matching in a dictionary. We developed algorithmic optimizations that provide a tuning parameter to choose between space consumption and running time while having overall lower preprocessing duration. Additionally, the performance has been validated experimentally by comparison against BK-trees and the baseline version of FastSS.

We see possibilities to speed up the verification of the candidate set using a more sophisticated implementation with bit-parallelism [99] and SIMD instructions. This technique has been successfully used by [78]. However, only about half of the time of the algorithm is actually spent in the verification phase with the computation of the edit distance. Likewise there might be opportunities to speed up the precomputation, in particular, using fast, incremental computations of hash functions and using parallelization. Also, it might be interesting to use data compression techniques to further reduce the storage requirements.

Error-Correcting Geocoding. We presented algorithms and data structures for error correcting geocoding that yield instantaneous answers at costs negligible compared to the overheads for displaying maps answers, etc. over the Internet. Perhaps most surprising is that we still have a high match rate with as much as four errors and this is much better than current web geocoders. Another pleasant surprise was that our combination of powerful data structures and general rating functions yields a considerably simpler solution than several rule-based industrial solutions we have heard about.

Although our experiments have so far focused on commercial German road data, we believe that they are easy to adapt to other Western industrial countries. In particular, these countries have similar address systems and language conventions. Points of interest (POI) like gas stations or restaurants can be incorporated by treating them like a street, town or house number depending on the context. Finding street intersections is relatively easy once we have geocoded the two intersecting streets.

The basic ingredients – fast approximate dictionary search, token matching and scoring functions might also help in other settings like in countries with more complicated addresses or less structured reference data. However in that case we should expect more errors, longer query times, and the need for further heuristics. In the future, work should focus on speeding up the query speed of single-field search, too, as it is the most obvious use case in practice. From an implementation point of view, we see further possibilities in applying highly tuned sub algorithms for matching, edit distance, and integer set intersection. An interesting challenge would be to have dynamic data structures that handle data sets which evolve over time. For example, it is a higher incentive for the individual to contribute crowd-sourced data sets, like OpenStreetMap, when the times are low that it takes a change to appear in a service.

Neighborhood Graph. Our concept of neighborhood graph is a promising approach to disambiguate queries involving frequent town names. Proximity is not determined by a predefined threshold, but by the input data itself. It captures the fact that certain areas a more densely populated than others.

The experimental evaluation as well as the user study show its potential. Scoring functions could even improve the quality of matches. In particular, we may want to restrict the set of reported cities to those in the same "region" (e.g., county, state, or nation) as the query town. Since Delaunay triangulations encode neighbors *in all directions*, we suspect that with appropriately defined "locally well behaved region shapes" the desired output still consists of nodes reachable in D^*. Another interesting aspect is that the set of nodes reachable through the neighborhood graph D^* may contain further "interesting" nodes.

CHAPTER 4

Distributed Preprocessing of Road Networks with Contraction Hierarchies

4.1. Central Ideas

Dijkstra's seminal algorithm solves the shortest path problem in polynomial time, but it does not scale well in practice to large instances. For example, a query for a shortest path on a continental-sized network can take several seconds to complete even on the fastest currently available hardware. Contraction Hierarchies is a speedup technique to Dijkstra's algorithm. In general, server based route planning in road networks is powerful enough to find shortest paths in a matter of milliseconds, even if detailed information on time-dependent travel times is taken into account, e.g., [21, 56, 84, 147]. However, this requires considerable amounts of memory on each query server and hours of preprocessing for large-scale road networks. This setting is problematic since Internet services with global coverage would like to work with transcontinental networks, detailed models of intersections, and regular re-preprocessing that takes the current traffic situation into account.

Preprocessing algorithms for speedup techniques to Dijkstra's shortest path algorithm were shown to be fast enough to be used in practice in a server-based scenario [129]. Ever bigger networks are available with crowd-source data sets like OpenStreetMap. Services may want to offer a seamless coverage of transcontinental networks such as EurAsiAfrica or the Americas. Second, one would like to move to more and more detailed network models with an ever increasing fraction of time-dependent edge weights and perhaps even multi-node models for every road intersection that allow to model turn-penalties, traffic-light delays, and even more. First studies indicate that detailed turn modelling increases memory requirements by a factor of 3–4 [185]. On top of this, one would also like to recompute the preprocessed information frequently in order to take information on current traffic (e.g. traffic jams or temporary road closure) into account. Preprocessing large networks such as the road network of entire world takes a considerable amount of time.

The preprocessing of Contraction Hierarchies is very *local.* Decisions whether to select a node for contraction and whether to insert a particular shortcut are made by inspecting a tiny portion of the network only. For example, it suffices to look at only 2 000 nodes in a network of millions of nodes and edges to decide if a shortcut is necessary as well as to inspect a 2-hop neighborhood around a node to decide if it should be selected for contraction. We use this locality to our advantage to devise data structures used during preprocessing that require space that is sub-linear in the number of nodes of the input graph without virtually any impact on the quality. We achieve much better cache locality that uses the fact that memory access is not uniform across the memory hierarchy of caches and main memory. Distributing the work load to a cluster of machines enables the preprocessing of large networks with a economically justifiable budget since the cost of purchase of several smaller computers is less than the cost of a single machine with equivalent performance. Also, we can distribute the processing of queries onto a cluster of machines. This lowers the computational load on computers effectively, and therefore allows one to use cheap commodity hardware for computation intensive tasks.

This Chapter is structured as follows. Section 4.2 exploits the locality of the witness search during Contraction Hierarchies preprocessing. We apply build an time and space efficient key-value storage for the priority queue that is able to achieve a space consumption that is sub-linear in the number of compute processes. We apply this technique to construct an efficient tie-breaking mechanism with performance guarantees. Section 4.3 shows how to conduct CH preprocessing on a cluster of cheap machines using the message passing paradigm of parallelization.

References

The contents of the sections in this chapter are based on the following publications: Section 4.2 is based on joint work with Dennis Schieferdecker [128]. Section 4.3 is based on joint work with Tim Kieritz [111], Peter Sanders and Christian Vetter [112]. Wordings of these publications are used in this thesis. The author especially expresses his appreciation of the work of Veit Batz [21] who provided much of the infrastructure code used in Section 4.3, but humbly declined co-authorship of the subsequent publication [112].

4.2. Preprocessing With Constant Space per Core

The contribution of this section is twofold. First, it describes a shared-memory parallel implementation of the Contraction Hierarchies preprocessing that uses constant space per witness search and per core in practice. Second, we apply the result to tie-breaking to construct a fast and space efficient tie-breaking oracle for independent set selection. An experimental analysis of each approach shows how well-engineered data structures and algorithms are able to deliver better performance by consequently exploiting caching effects while executing a larger number of CPU operations.

4.2.1. Engineering of the Witness Search Heuristic

We build on the introduction to Contraction Hierarchies preprocessing as given in Section 2.5. Recall that the subgraphs that are explored during witness searches are rather small compared to the entire graph instance. For example, the implementation at hand prunes witness searches at 1 000 nodes for simulations and at 2 000 nodes for actual contractions. The parallel preprocessing obviously needs a priority queue per thread which (internally) stores its content in some table. The easiest implementation is an array of size linear in the number of nodes of the graph. But this array has to be instantiated for every preprocessing thread $t \in T$ and thus the overall space requirement would be $\mathcal{O}(n \cdot T)$ which becomes easily infeasible on current multi-core architectures. Likewise, any solution to save memory must show *good* scalability, i.e. any solution with constant factor slowdown essentially wastes (a portion) of the available processing power.

Tabulation hashing is a simple hashing scheme that dates back to as early as the late 1960s when first published by Zobrist [197] and the late 1970s when rediscovered by Carter and Wegmann [41]. It uses simple table lookups and *exclusive or (*`XOR`*)* operations. Later Patrascu and Thorup [155] gave a theoretical analysis of the scheme. Tabulation hashing interprets input keys as a string of c characters $x_1, \ldots, x_k$. For each of the possible character positions a random table $T_i, 1 \leq i \leq k$ of size $2^{c/k}$ is initialized and the following hash function is used:

$$h(x) = T_1[x_1] \oplus \ldots \oplus T_c[x_c].$$

The probability of a hash collision is small, e.g. Carter and Wegmann [41] show that tabulation hashing is 3-independent which is a strong property given the simplicity of the approach.

Implementation Details. Our variant of tabulation hashing splits inputs of size 32 bits into two words of size 16 bit, i.e. the most and least significant halves. Thus, two lookup tables with 2^{16} entries have to be filled with pairwise distinct random numbers. This is done by initializing the tables consecutively with numbers $0, \ldots, 2^{16} - 1$ and then applying a random shuffle. Hashing any input is done as follows. Input x is assumed to be 32 bits wide. x is split into the most and least significant halves, a lookup is performed for each half and then combined by a `XOR` operation. The work necessary to perform a query is constant. The overhead of initializing these arrays can be neglected, since this has to be done only once and the associated work is linear in the size (and number) of the lookup tables. See Figure 4.1 for an illustration of the tabulation hashing scheme. Note that we assume input words that are 32 bits wide. However, an extension to 64 or even 128 bits is easy to implement.

The range of the hash function is 2^{16} which is obviously of much larger cardinality than the set of at most 2 000 explored nodes. The number of expected collisions is tiny as shown in the analysis. The observed rate of collisions in practice is less than 10^{-5}. Nevertheless, it is necessary to use a collision resolution strategy, since a hash function points only to a records location and not to the record itself. It seems obvious to use linear probing [115] as resolution strategy for the following two reasons. First, the expected number of collisions is small and so is the expected number of cells in the hash tables that have a non-vacant neighbour. Second, the next cells are very likely to lie in the same cache line as the original cell and therefore linear probing is virtually cost-free.

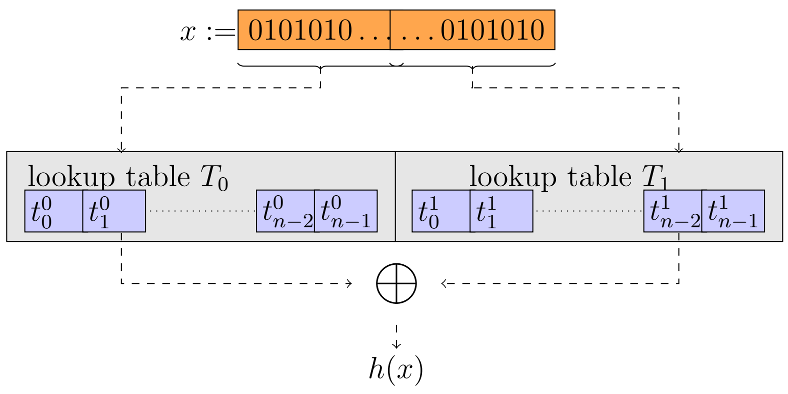


Figure 4.1.: Schematic Visualization of the Tabulation Hashing Scheme.

The implementation is straight-forward from the description. The queue implementation at hand is a binary heap that maintains a partially sorted heap of the keys along-side an index into a key-value store, where the actual elements reside. In the simplest case this key-value store is a simple array that has the size of the maximum number of distinct elements that could be inserted. We replace this simple storage array with a smaller sized array that has the range of the hash function. A hash value is generated for each input key and its value is stored at the corresponding index of the storage array. Collisions, i.e. when the cell is not empty, are resolved by linear probing. After each witness search the storage table of the priority queue is reinitialized. While this seems non-obvious at first sight, one has to pay special attention for the reinitialization of the storage array. Resetting an array to initial values is expensive as it either involves a reallocation or a sweep over the memory or even both. Therefore each cell has a local time stamp that indicates the time when it was written last. Initially, the global time stamp is zero and incremented each time the storage table is cleared. This way, it is not necessary to actually zero out any memory every time, and it suffices to do a simple comparison during collision resolution. The zeroing step has to be done only in the case that the timestamp overflows which happens every four billion times when using a 32 bit integer. The amortized overhead is negligible. The implementation uses 4 bytes each for key and value as well as for the time stamp which yields cell sizes of 12 bytes and therefore an overall memory consumption of 384 kilobytes per queue for the storage table. We even halved the size of the table to store only $32\,768 = 2^{15}$ entries, which is half the range of the hash function. Preliminary experiments show that the collision rate was virtually unaffected by decrease while the memory consumption halved.

It is not necessary to implement an explicit deletion operation for our use-case. Elements that have been inserted into the queue simply do not get deleted from the queues underlying key-value store until the entire queue is flushed. Therefore, it has not been implemented.

We briefly experimented with a *replacement strategy* to do collision resolution, too. The data element that we keep in the storage array is essentially the (tentative) weight of a node. Consider an insertion of an element e and that $h(e) = h(f)$ for some already inserted element f. Instead of searching for a new position to insert an element e, we replace the entry at $h(e)$ with the maximum of the tentative distances of e and f. This is feasible in our case

	duration [s]			
	ptv-germany	ptv-europe	osm-germany-2	osm-planet
boost	397.00	1384.42	1876.40	47 978.80
google	175.97	643.02	1215.67	22 278.10
plain	**68.10**	**278.93**	**822.21**	21 873.58
xor-witness	116.65	363.86	827.48	**16 030.90**

Table 4.1.: Comparison Against Several Hash Table Implementations.

since the purpose of the witness search is not to find actual paths but to indicate if there exists some path between nodes u and w that is shorter than the one over the middle node v. The witness searches are allowed to have a one-sided error, i.e. we are allowed to have false negatives: If it indicates that it didn't find a shorter path than the one over the middle node, then this must not be true. We just add too many shortcuts in this case, but do not break the correctness of the CH search graph. We observe essentially the same experimental results for this variant within the margin of error.

Experimental Evaluation of the Constant-Sized Heap Storage. We implement the algorithm in C++ using the GCC compiler version 4.6.1 with full optimizations. Experiments are done on 12 cores of Machine C, running Linux kernel version 3.0.0. The graph instances used are *osm-planet*, *osm-germany-2*, *ptv-europe*, and *ptv-germany*. Recall that the *osm*-graphs are edge-expanded. The implementation of CH already includes the tabulation hash based tie breaker from above. Preprocessing was run for the same instances as before and compared against two *standard* hash table implementations. The first one is an implementation from the Boost[1] C++ library version 1.4.6, namely `boost::unordered_map`. This hash table is said to be close to the implementation of GCC C++11 standard hash table implementation. The second implementation is Google's sparsehash[2] library version 1.10, namely `google::dense_hash_map`[3]. This hash table has the reputation of being among the fastest hash table implementations. Note, we also use the tabulation hashed tie-breaker for the *google* and *boost* variants. The presented numbers are averaged over three runs.

Table 4.1 gives the results from the experiments on a number of input graphs where *xor-witness* denotes the implementation of our approach. The reference values of the plain implementation are given in line *plain*. Best values are printed in **bold font**. First, we see that the performance of the (general purpose) boost hash library is not competitive. It is the slowest among the implementations and has a slowdown of about 3–5 compared to the fastest solution. Next, we observe that the Google hash library is indeed fast as its slowdown becomes less the larger the graph instance are. Compared to *xor-witness* we see a slowdown of about 30% in instance *osm-planet*- Compared to the plain it has a slowdown of about 2% on that instance.

[1]`http://boost.org` – accessed April, 1st 2013
[2]`http://code.google.com/p/sparsehash` – accessed April, 1st 2013
[3]The library naming reflects implementation details, but we note the irony.

Next, we see that our approach works the better the larger the test instance is. We observe that the results of the plain implementation and *xor-witness* are virtually identical on instance *osm-germany-2* and on *osm-planet* our solution is faster by 25–30%. This is a strength of our approach as it appears to scale better with ever-growing data sets like OpenStreetMap. Google's hash table implementation fare quite all-right compared to the plain implementation as it is slightly slower on instance *osm-planet* by about 2%. Again, the boost solution is not competitive any more.

4.2.2. Tie-Breaking in Constant Time and Space

The application of tabulation hashing is not limited to be applied to the witness searches of preprocessing. As mentioned briefly before, the role of tie-breaking is to facilitate the decision which node to contract only when neighbouring nodes have equal priorities during the selection of the independent set. A tie-breaking mechanism cannot be an arbitrary decision process but has to fulfil certain properties as the following paragraph shows.

Definition 4 (Node Ordering and Tie-Breaking). *Consider two nodes $u \neq v \in V$. A node u is smaller than node v from the k-neighbourhood if $p(u) < p(v)$ or if $u \prec v$ in case $p(u) = p(v)$, where $\prec$ defines an order on the nodes. The order (or tie-breaker) is called consistent if and only if $u \prec v = \neg(v \prec u)$.*

One can show that the property of consistency is an essential property of any correct Contraction Hierarchies implementation. Consider the contraction of a single node to be a basic operation during the preprocessing.

Lemma 3. *Contraction Hierarchies preprocessing with an inconsistent tie-breaker does not terminate for all inputs.*

It suffices to show that there exists an input graph and an inconsistent tie-breaker for which no node is selected during an iteration.

Proof. Consider a triangle of three nodes a, b, c each of degree two with equal priority. Further assume that the tie-breaker is inconsistent with $x \prec y = 0$, $\forall x, y \in \{a, b, c\}$. No node will be selected to be an element of the independent set that is to be contracted. Thus, the contraction does not terminate. □

The easiest implementation of a consistent tie-breaker is a *random permutation* of the node IDs and a subsequent renumbering of the graph. A random shuffle of node IDs implies linear work in the number of nodes and edges. We call this tie-breaker a *bias array* From a theoretical point of view, one could argue that this is as good as it gets since the work is constant per decision. On the other hand, the constants associated with such a scheme may render it impractical. Doing a random shuffle and a subsequent renumbering on the nodes of a large graph, e.g., for the entire planet, can be prohibitively expensive in practice. Preliminary experiments preceding show that this can take as long as contracting the first 20–25% of the nodes. Running a parallel shuffling algorithm may help, but a general a disadvantage of random shuffling is that it breaks any inherent cache-efficiency that the data has. Also, we note that such an implementation still has $\mathcal{O}(n)$ space requirements. In real-world data sets node IDs are given in the order in which they are created, i.e. consecutive numbering of the nodes of an entire street when it is added into the data set. There are conflicting interests for the numbering of the nodes. On one hand, the strength of CH is that its data structure is quite different from the intuition of a hierarchy of road types. And thus, one would want a preprocessing that is independent from any existing ordering or presentation of the input data. On the other hand, the preprocessing mostly consists of small graph searches and one would like the data to display a certain amount of locality, i.e. close-by nodes have close-by IDs, to leverage caching effects.

The bias array can be found in the implementations of [112, 184]. Array A is populated with numbers $0, \ldots, n-1$ and randomly shuffled at the beginning of the preprocessing. This yields a precomputed pairwise distinct random number for each node v in the graph. When a tie-break is necessary for nodes i and j then the values of $A[i]$ and $A[j]$ are compared. We look at the situation that comparison of any two elements is based on the input ordering and define the following:

Definition 5 (Self-Reliance of Tie-Breaking). *A tie-breaking ordering is called self-reliant, if its outcome is irrespective of any input numbering. An ordering is said to be ε-self-reliant from the node ordering, if the comparison of any two elements is independent from the input ordering with high probability, i.e. the probability is at most $1 - \epsilon$.*

The above straight-forward implementation of tie-breaking has one major disadvantage in practice that is a direct result of its simplicity. While one expects this tie-breaker to be fast, the number of cache misses is large. The bias array is much larger than any cache size even for medium-sized graphs and one must expect an expensive cache miss for each call to the tie-breaking rule, even if the data exhibits some locality preserving node numbering. A preliminary experiment with a memory debugging tool revealed that most of the accesses to the bias array were actually cache faults, and the number of clock cycles wasted in a single cache miss easily amount to a few hundred [67].

4.2.3. A Fast Self-Reliant Tie-Breaker With High Probability

The following tabulation hashing-based scheme gives the basis of a tie-breaking mechanism that takes constant time to evaluate and uses constant space only. It is not only independent with high probability, but surprisingly fast in practice and even faster than the above simple schemes. We build a tie-breaker for two nodes $a, b \in V$ in the following way. The hash values $h(a), h(b)$ of a and b are compared and in the (unlikely) event of a hash collision, a and b are compared directly. More formalized we have:

Definition 6 (Tie-Breaking by Tabulation Hashing). *Given a (tabulation) hash function $h : U \to [m]$ and two elements $a, b \in U$, then the boolean expression*

$$a \prec b := [h(a) < h(b)] \vee [(p(a) \equiv p(b)) \wedge (a < b)]$$

obviously defines an order on the elements of U.

We want the probability of hash collision less or equal than ε to achieve best results. For fixed keys $x_1 \ldots, x_k \in U$ and a randomly drawn hash function $h \in H$, the hash values $h(x_1), \ldots, h(x_k)$ are independent random numbers. Recall that tabulation hashing is 3-independent. Thus, the probability of a hash collision is less than 2^{-k} and thus the order it defines is random with high probability. Only in the rare case, of a collision the ordering is derived from the IDs of the nodes.

Implementation Details. A query to the tie-breaking oracle is straight-forward to implement by reusing the implementation of the witness search heuristic of Section 4.2.1. It is expanded into the following tie-breaking algorithm by implementing Definition 6. Consider the code fragment of Listing 4.1. The entire tie-breaking mechanism, including hashing, uses as few as 22 assembly instructions on an X86 CPU in practice, when letting the compiler optimize the code. See Appendix B for the actual assembly listing. Most interestingly, it is possible to evaluate the `if`-statement without any branching by using conditionally set flags in the register[4]. We note that the optimization is compiler dependent and that a conditional jump would pose no serious performance penalty as one would expect it to be almost always predicted correctly by the CPU. On Machine C (AMD Opteron 6212, Bulldozer v1 architecture) the code of Listing 4.1 takes approximately 350 cycles to be evaluated. The space requirement for this tie-breaking mechanism is 256 KB of RAM, which fits into the L2 cache of any recent X86 processor, and L2 cache latency is approximately ten cycles. Table 4.2 gives the results of experiments of running times of CH preprocessing either with bias-array based tie-breaking or tabulation hash-based tie-breaking.

4.2.4. Experimental Evaluation

The experiments on the performance of the tabulation hash based tie-breaker have been evaluated on Machine C using the same methodology as the previous experiments.

Table 4.2 reports on the impact of the hashing scheme on the duration of the preprocessing. Line *plain* gives the result for the bias-array based and *xor-witness* denotes the preprocessing of the previous section. Best values are given in **bold font**, again.

The performance of the *xor-break* is the fastest in all experiments. The experiments show that a tie-breaking mechanism based on tabulation hashing not only reduces the memory

[4]X86 assembly instruction `setg`

Listing 4.1: Tabulation-based Tie-Breaker

```
function bias(const NodeID a, const NodeID b)
    unsigned short hasha = h(a);
    unsigned short hashb = h(b);
    if(hasha != hashb)
        return hasha < hashb;
    return a < b;
end
```

	duration [s]			
	ptv-germany	ptv-europe	osm-germany-2	osm-planet
plain	68.10	278.93	822.21	21 873.58
xor-witness	116.65	363.86	827.48	16 030.90
xor-break	**65.86**	**259.85**	**628.96**	**15 815.10**

Table 4.2.: Results for the Tabulation Hash-Based Tie-Breaking Scheme.

requirements, but also that it pays off to trade some processing cycles for much better cache efficiency. An extended profile run was conducted on the smaller edge-expanded instance *osm-berlin*. This was done using the `cachegrind` plug-in of Valgrind[5], a tool for (memory usage) debugging and profiling. Examining larger instances is impractical since the tool entirely simulates the cache hierarchy of a modern processor, which takes orders of magnitude longer than running on real hardware. However, the experiment revealed that while the overall instruction count increased by less than 1%, the number of (simulated L1 and LL) cache misses dropped significantly by more than 20%.

The benefits of tabulation hash based tie-breaking are twofold. The speedup is only minor for the node-based graphs, but grows with the size of the graph and its complexity, i.e., for the edge-expanded graphs. Generally, we see that the larger graph, the better the efficiency of the tabulation hash based methods. Variant *xor-break* is the fastest in all experiments. This is in contrast to the experiments of the previous section where we observed best results of tabulation hash based methods only for the largest of the instances. The larger the graph the more cache faults occur for the *plain* variant because the storage table of the priority queue grows linear in the number of nodes. And here, the gap between *plain* and *xor-break* becomes larger the larger the graph gets. Contrary, the gap between *xor-witness* and *xor-break* decreases as the road networks grow in size. We see that the tabulation hashing approach is especially effective to limit the space requirement of the witness search.

4.3. Distributed Preprocessing by Message-Passing

We lift existing bottlenecks of CH preprocessing by giving a distributed memory parallelization of (time-dependent) Contraction Hierarchies (TCH). Storage requirements are critical because TCHs introduce many additional edges which represent long paths with complex travel-time function. Furthermore, TCH queries have small search spaces mostly concentrated around source and target node and hence exhibit a degree of locality. It is this degree of locality that makes it attractive for a distributed implementation.

In this chapter, these issues are addressed by describing an approach that distributes both preprocessing and query computation to a cluster of inexpensive machines, each equipped with limited main memory. The available large cumulative memory allows large networks while the cumulative processing power allows fast preprocessing and accelerated query throughput. For example, on a medium sized network, 64 processes accelerate TCH preprocessing

[5]`http://www.valgrind.org` – accessed April, 6th 2013

by a factor of 28 to mere 160 seconds, reduce per process memory consumption by a factor of 10.5 and increase query throughput by a factor of 25. The distributed implementation of time-dependent Contraction Hierarchies (DTCH) of this Chapter is based on Batz' *sequential* time-dependent Contraction Hierarchies (TCH) [21].

4.3.1. Distributed Node Ordering and Contraction

Consider the shared-memory parallel Contraction Hierarchies preprocessing that identifies independent sets as described in Section 2.5. We generalize this approach to a distributed memory parallel preprocessing that we describe in the following. We use the following notation. We compute a partition of the input graph nodes where each cell $\mathcal{C}_i$ is associated with a process $i \in \{1, \dots, p\}$ with the aim to optimize for cells of roughly equal size as well as minimizing the number of cut edges. Process k is said to be *authoritative* for all nodes in cell $\mathcal{C}_k$. The authoritative process for node v is denoted by $A^{(v)}$.

During the preprocessing, the following invariant is maintained: Every process stores its local partition cell $\mathcal{C}_i$ plus the nodes within a certain *neighbourhood*, s.t. every witness path of hop length h is contained within the cell and its neighborhood. This neighbourhood is also called *halo* $\mathcal{H}_i$ *of cell* i.

Lemma 4 (Halo Size of Cell i). *A halo which contains every node with hop distance less of equal than $\ell := \lfloor h/2 \rfloor + 1$ suffices in an undirected graph such that all witness paths of hop length h can be found within $\mathcal{C}_i \cup \mathcal{H}_i$.*

To show the claim, we look at the maximum hop distance of a node k on a witness path. Furthermore, we show that all witness paths explored during the contraction of node v with at most h hops do not leave the halo.

Proof. Consider node v that is next to be contracted and that witness searches are conducted for each path $p := \langle u, v, w \rangle$ in the (remaining) graph. Now, consider a path p where $v \in \mathcal{C}_i$ and $u, v \in \mathcal{H}_i$ as well as a witness path p' to p that does not include any nodes from $\mathcal{C}_i$.

Now, there exists a node k on path p' that has maximum hop distance to v. When limiting all witness paths p' to a hop distance of at most h, we see that node k can be at most $\lfloor h/2 \rfloor + 1$ hops away (from v by either going through u or w). Hence, all witness paths of length at most h are covered by a halo of ℓ hops. □

Figure 4.2 gives a visualization of the argument. Lemma 4 can be extended to directed graphs by considering edges independent of their direction. In other words, node v is in the ℓ-neighbourhood of some node set $\mathcal{C}_i$ if there is either a (directed) path with at most ℓ edges from v to a node in $\mathcal{C}_i$ or a directed path with at most ℓ edges from a node in $\mathcal{C}_i$ to v.

While the remaining graph (containing the nodes not yet contracted) is non-empty, an independent set I of nodes is identified to be contracted next. It suffices to inspect the nodes in a 2-hop neighborhood to check for independence and every cluster node obviously stores enough halo information to check the nodes under its authority. The nodes within the independent set I can be contracted in any order without affecting the final outcome. Therefore, the contraction of the nodes in an independent set can be obviously conducted

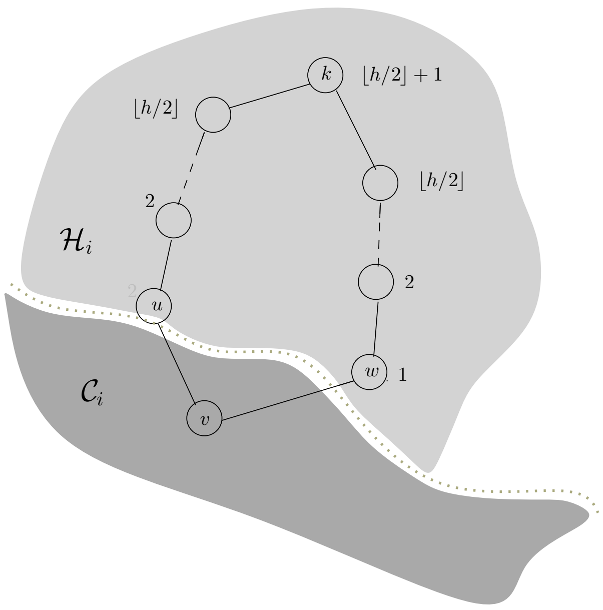


Figure 4.2.: Node k is at most ℓ hops away from node v.

in a distributed system. While any independent set could be used in principle, nodes should be contracted in increasing order of importance, as argued in Section 2.5.

If the halo information is available for all nodes of an independent set, node contraction can be performed completely independently on each process, possibly using shared-memory parallelism. When this is done, all newly generated shortcuts (u, w) are communicated to the authoritative processes of u and w which in turn forward them to processes with a copy of u or w in a halo. Note that this implies bookkeeping and that messages between the same pair of processes can be joined into a single message in an actual implementation. This way, two global communication phases suffice to exchange information on new shortcuts. Then,

Listing 4.2: Distributed Procedure halo_repair

```
function halo_repair(i, max_dist)
  do
    $U := \emptyset$;
    MPI_Recv( new edges $E_{\mathcal{C}_i}$ incident to $\mathcal{C}_i$);
    for $(u,v)$ in $E_{\mathcal{C}_i}$ do
      if ($hop(v) <$ max_dist) and ($hop(v) < h[v]$) then $h[v] = hop(v)$; $U = U \cup v$; end
      if ($hop(u) <$ max_dist) and ($hop(u) < h[u]$) then $h[u] = hop(u)$; $U = U \cup u$; end
    end
    for $u$ in $U$ do
      MPI_Send(request all edges incident to $u \in U$ from process $A(v)$);
    end
  while($U \neq \emptyset$);
end
```

Listing 4.3: Distributed Procedure contract(v)

```
function contract(i)
  $U :=$ all bordering nodes of $v$;
  $n := |\mathcal{C}_p|$
  MPI_Send(request all edges incident to $u \in U$ from process $i$);
  for j in 0 .. n do
    update_priority(j);
  end
  while ($n > 0$) do
    halo_repair();
    $V_p :=$ identify_independent_set();
    for $v$ in $V_p$ do
      contract(v);
    end
    $n := n - |V_p|$
    MPI_Send(edges $(u,w)$ to $A(u)$ and $A(w)$);
    MPI_Send(remove edges $(u,v)$ and $(v,w)$)
  end
end
```

the halo-invariant has to be repaired since new shortcuts may result in new nodes reachable within the hop limit. This is done in an approach with two phases. First, a local search from the border nodes in $\mathcal{C}_i$ establishes the current distances of nodes in the halo. Here, bordering nodes of cell $\mathcal{C}_i$ are those nodes that have a neighbour outside $\mathcal{C}_i$. Then, nodes in the halo with a new hop distance $< \ell$ request information on their neighbours using a single global message exchange. This process is iterated until the full ℓ-hop halo information is available again, i.e. there is no node any more whose hop distance got decreased in the previous step. Obviously, this repair operation takes at most ℓ global communication phases.

Consider a function `hop(·)` that computes the hop distance of a given node from the border of a cell and $h[v]$ gives the currently stored hop distance for a given node v. If the distance has not been computed before or the node is not part of the halo then we set $hop(v) := \infty = h[v]$. Listing 4.2 gives pseudo-code for the halo repair algorithm, where $E_{\mathcal{C}_i}$ is initialized with the border nodes of cell $\mathcal{C}_i$ in the first iteration and `max_dist` is an upper bound to the hop distance.

Next, consider the function `contract(·)` that contracts a single node as well as utility functions `update_priority(·)` and `identify_independent_set()`. The first function updates the priority of a node by evaluating its (local) ℓ-hop neighbourhood, while the latter one identifies a contractable independent node set in $\mathcal{C}_i$ by a number of local searches. Listing 4.3 gives the pseudo code for the distributed contraction step.

At the time of completion of the contraction phase, each compute process not only stores the nodes in $\mathcal{C}_i$, but also the nodes reachable from a node within region $\mathcal{C}_i$ using upward edges only, i.e. the forward and backward search spaces. Note that these upward edges may be in both directions, forward as well as backward. Forward or backward search for a node in $\mathcal{C}_i$ can be done locally using this data structure as is explained in the next section.

4.3.2. Distributed Query

The distributed a query reuses the same partition of the nodes of the graph that was also used previously during the preprocessing. In addition to the nodes of cell $\mathcal{C}_i$, each process also keeps track of forward and backward search spaces of its nodes in the CH data structure.

A time-independent query for this distributed system is easy to conceive. A request enters the system at possibly any process and is sent to the authoritative process for s. If that process is the authority to node t as well, the entire query can be answered locally by a single process. If not forward and backward search spaces are computed on the respective authoritative compute nodes and a search space intersection is performed after aggregating the information. A node with minimum sum of forward and backward distance in the intersection is the middle node of a shortest path between source and target.

A distributed time-dependent earliest arrival query works in a similar way. Otherwise a time-dependent forward search is conducted starting at node s in $G^{\uparrow}(s)$ on process $A(s)$. The arrival time at all reached nodes by this search is sent to process $A^{(t)}$ which has all the information needed to complete the query. As in the sequential time-dependent query algorithm, $A^{(t)}$ marks a corridor of edges in the backward search space in $G^{\downarrow}(t)$. Finally, the forward search is resumed at the nodes in the corridor reached by the forward search. It needs only to explore marked edges of the corridor. Note that the searches can eliminate some nodes that cannot be on a shortest path using the search space pruning technique of stall-on-demand [84].

The backward search can be made goal directed, too. When the BFS-like backward exploration, as explained in Section 2.5, marks reachable nodes it stores with them upper and lower bounds of the *arrival time* at these nodes. The lower bounds can be recycled to give an admissible heuristic for A* search, since it never overestimates the arrival time.

4.3.3. Experimental Evaluation

Implementation Details and Methodology. We implement the aforementioned algorithms and data structures in C++ using Intel's C++ compiler version 10.1 with full optimization, and OpenMPI 1.3.3 as implementation of the message passing paradigm. The experiments are performed on a varying number of PEs on Machine B. Note that PEs are on separate compute nodes on purpose to minimize internal communication. Experiments are done on test instances *ptv-europe-td*, *ptv-ger-mw-td*, and *ptv-ger-sun-td*. The partitioning of the input graphs is done by the same partitioner as SHARC [23], which is kindly provided by Daniel Delling. The running time for graph partitioning is not included but negligible in our setting.

Distributed Contraction

Figure 4.3 shows the numbers of speedups and execution times obtained in the experimental evaluation. Please note the logarithmic y_1 scales of the plots. The speedup of the distributed implementation is compared to the sequential variant. Interestingly, for the case of the German road network, the sequential code from [21] yields very similar values on Machine B. Since the European road network cannot be contracted on less than 4 nodes, because of

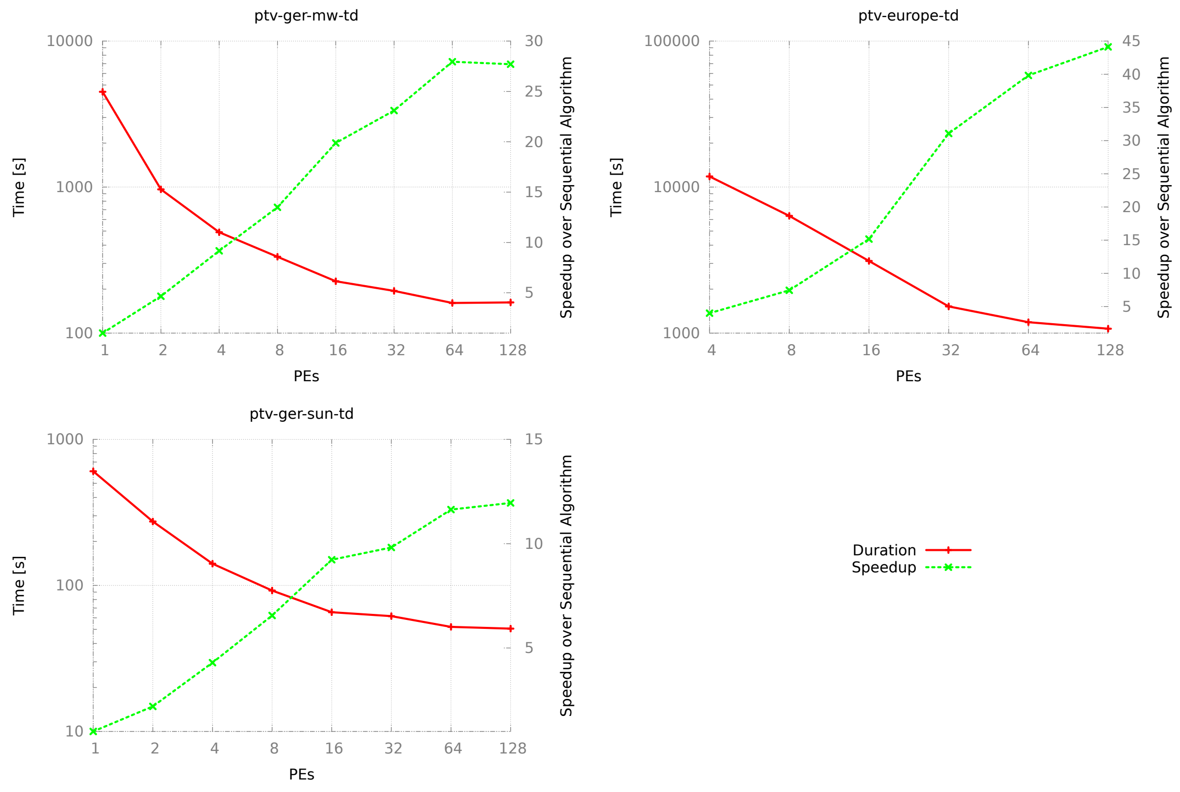


Figure 4.3.: Running Time and Speedup of Distributed Contraction Depending on the Number of PEs.

memory requirements, this execution time is used as a baseline.

For the German midweek road network the implementation scales well up to 16 processes. Indeed, a slight super-linear speedup can be observed which in turn can be attributed to a larger overall capacity of cache memory. The speedup achieved depending on the number of processes rapidly declines for more than 32 processes. We observe this point of diminishing returns and observe as well that running the distributed preprocessing on more than 64 processes does not make sense any more. The experimental results for the German Sunday network show a similar behaviour although both overall execution times and speedups are smaller. This is expected behaviour since the limited time-dependence on just 3% of the edges incurs less overall work during the contraction. It is also expected behaviour that the larger the networks become, the better observed scalability is. Hence, for the European network scalability is better than for the German network. We see a good to reasonable speedup for up to 32 processes that still shows some improvement when increasing the number of processes from 64 to 128.

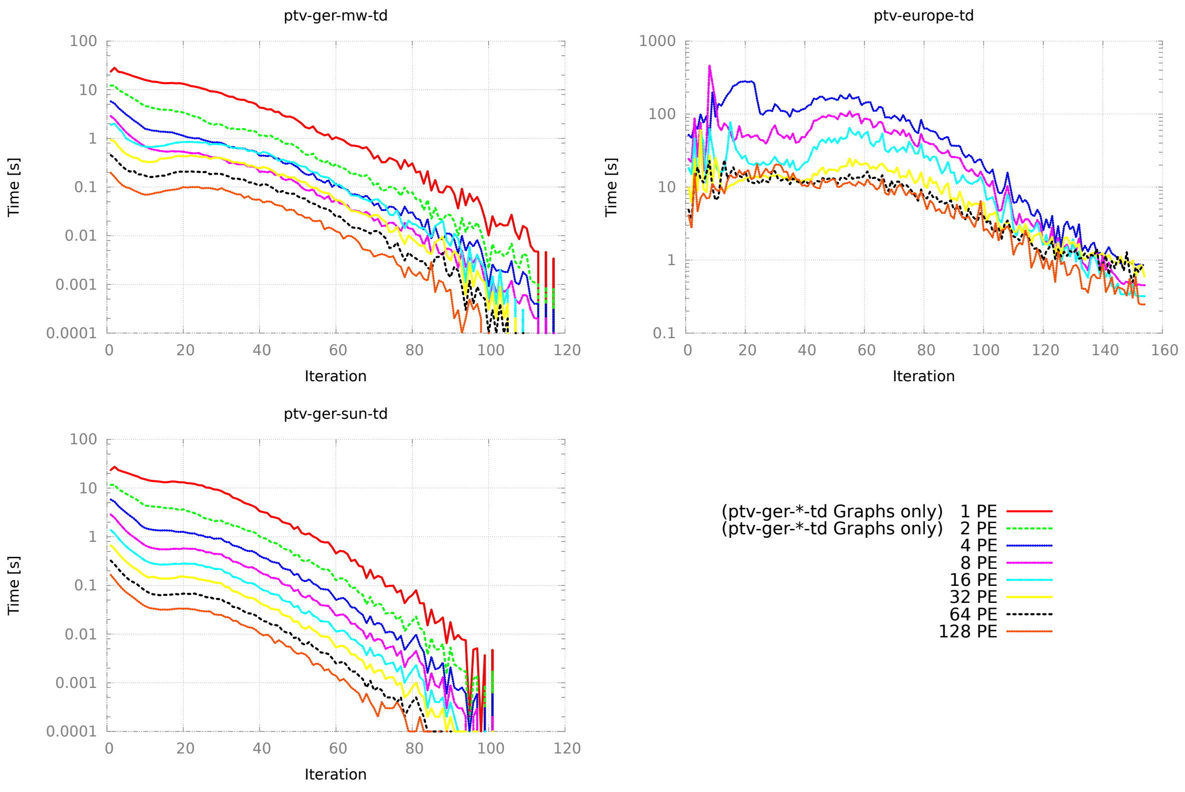


Figure 4.4.: Duration of the Contraction Step Depending on the Number of PEs.

Time Spent in each Iteration of the Contraction

Figure 4.4 gives the time spent to conduct witness searches and to actually contract nodes. The plot indicates that the execution times per iteration exhibits a certain level of variance for the European network. Closer inspection indicates that this is due to partitions that are (at least in some iterations) much more difficult to contract than others. Figure 4.5 then shows the overhead it takes to find the independent node set in each iteration depending on the number of PEs used. Note the logarithmic scale on the y-axis and that we only give plots for two of three instances because the third one shows a very similar behavior. We observe that the running time approaches a minimum which is reached at about 32 processes for the German instances and at about 64 processes for the European test instance. The time needed to process the European graph instance is about an order of magnitude larger as the instance has a bigger graph. We also observe that the time spent depends on the number of PEs and approaches a minimum, which is quite stable for the latter third of the iterations. One would expect the time to keep decreasing more significantly as the graphs only get smaller, but we see the following reason for these rather *stable* timings when looking at the raw numbers. Most of the nodes are contracted in the first dozen iterations. The graph that has to be inspected does not shrink significantly for later iterations from one to the next. Thus, the impact on the time needed to find an independent set is less significant for later iterations.

Next, we analyze the memory requirements for each process that is necessary to distribute queries. Note that memory requirements during contraction time are much lower in the implementation since edges incident to contracted nodes are written out to disk. As explained above, there is an overhead mainly because every process holds the complete search space for each node under its authority. In Figure 4.6 the memory consumption is visualized in two ways. First, the maximum memory required for any single process is analysed. Second, it is analysed how this maximum m compares to the sequential memory requirements s by looking at the ratio of $p \cdot m/s$ which quantifies the blow-up of the overall memory requirements. The first observation is that although the maximum m decreases, the memory blowup only remains in an acceptable range for around 16–32 processes. It shows that a blow-up factor of around 2 is common. Although both implementations have a relatively small set of important nodes that show up in many queries, the main difference is that the weight functions of edges incident to this small set of nodes bears highly complex travel-time functions. This is somewhat expected behaviour since shortcuts created at the end of the contraction resemble a large number of original edges.

Batz et al. [20] have solved this problem by an approximation of the piece-wise linear functions based on the algorithm of Imai and Iri [100]. The edge weight functions of shortcuts are replaced with less complex upper and lower bounds, while edge weight functions that resemble an original edge at one point in time keep their exact time-dependency. The query computes a corridor of candidates that is guaranteed to contain a path from source to target with earliest arrival time. The packed corridor is traversed by a BFS and unpacked. Thus the corridor has correct time-dependent edge weights. Subsequently a time-dependent Dijkstra finds the actual shortest path in the corridor. The speed of a query is only moderately slower by a factor of 1.2–8.2 on average but the space consumption is decreased by a factor of 2.3–8.4 depending on the input instance.

We[6] note that the query can be partially simplified and accelerated, because the forward

[6]The author thanks Veit Batz for the fruitful discussion on the topic.

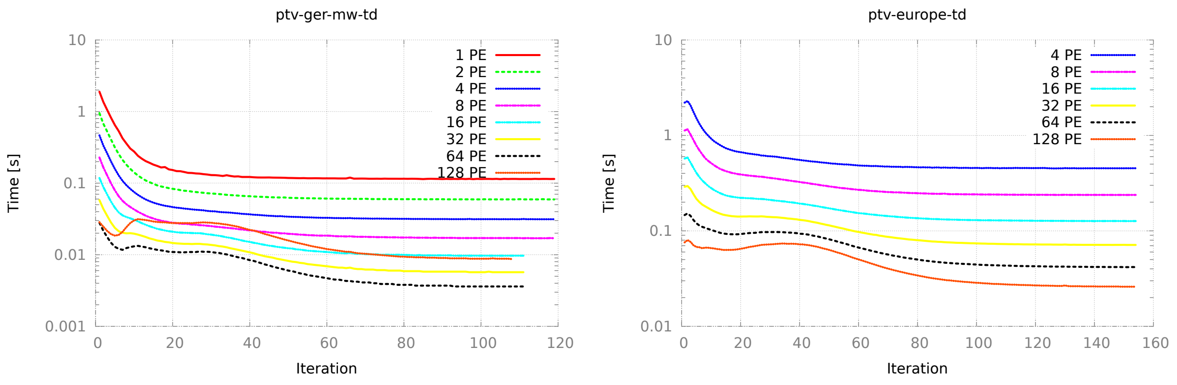


Figure 4.5.: Duration of Finding an Independent Set in Each Iteration Depending on the Number of PEs.

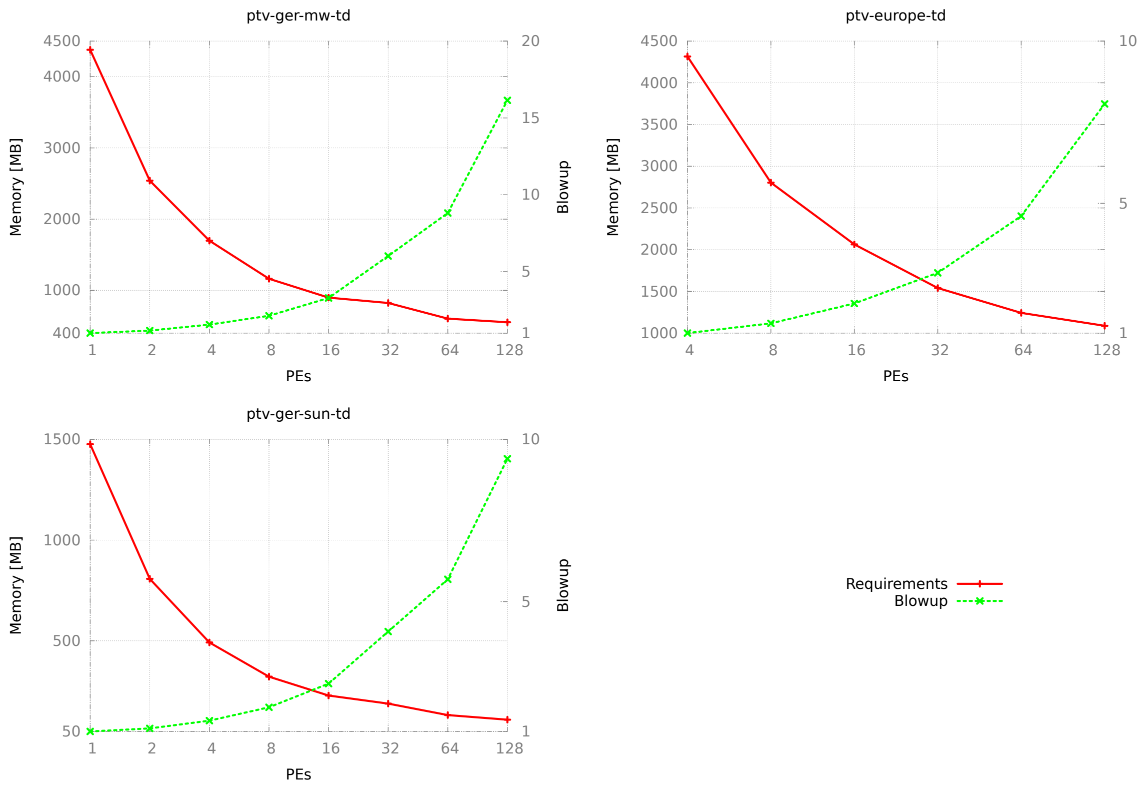


Figure 4.6.: Memory Requirements on each Compute Node Depending on the Number of PEs.

and backward portions of the corridor are DAGs. Recall that computing shortest paths in a DAG can be done by traversing the DAG in a topological sorting order. We propose to traverse the corridor with a DFS (instead of the BFS) that settles a node once all it predecessors are settled and unpack shortcuts in this order. Note that this is similar to the *unpack on demand* improvement of Batz et al.. When the target node of an unpacked edge is settled, we can *propagate* the minimum of all predecessor travel time label of the sources to the target node.

Distributed Query

The running times for earliest arrival queries are averaged over a test set of 100 000 queries that were precomputed using a plain label-correcting time-dependent Dijkstra. For each randomly selected start and destination node, a random departure time was selected. The length of a resulting shortest path was saved for verification purposes. The query and its running times can be evaluated in two distinct settings. The first setting is batched, also called *parallel setting*, where the distributed system is loaded with all queries and the time is measured until all queries have been answered. Here, queries have to wait in a queue when a processor is busy. The second setting is single query performance, also called *serial setting*.

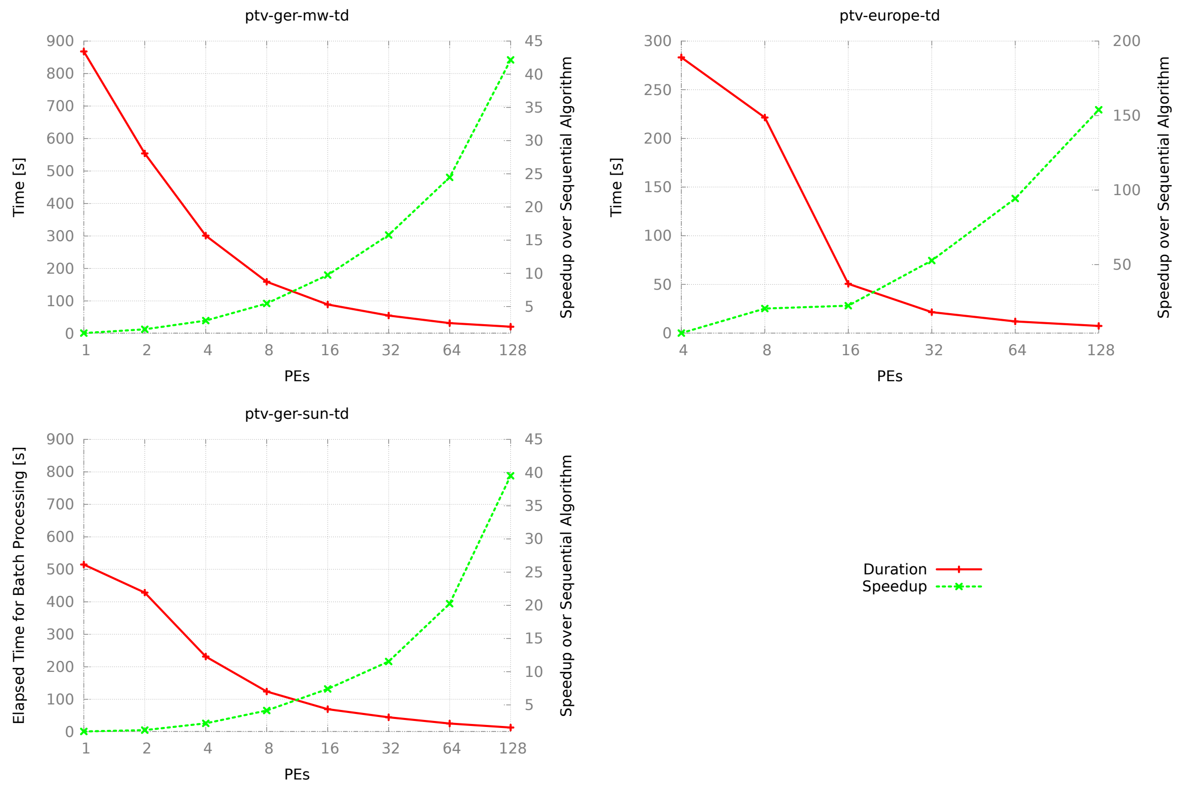


Figure 4.7.: Average Run Time [ms] of a Single Query in a Batch Run of 100 000 Queries Depending on the Number of PEs on the German road networks

The distributed system processes queries on after another.

Figure 4.7 shows the speedup obtained for performing all 100 000 queries in the batched setting. This figure is relevant for measuring system throughput and, thus, the cost of the servers. Scalability for the German networks is even better than for the contraction phase with efficiency near 50 % for up to 32 processors.

When looking at raw time that is spent in search space exploration we observe a super-linear speedup in all experiments. Notably, only the experiments on the European network even show super-linear speedup in the plot of Figure 4.7. We attribute this behavior to two reasons. First, there are cache effects combined with complex travel time functions. The European graph is large and the resulting travel time functions on higher levels of the hierarchy are rather complex. The more processes are used the higher the cache locality of the search spaces that are stored on each process. The reason for *not* experiencing super-linear speedup for batched queries on the German networks might is that the super-linear speedups on the smaller data set is dominated by communication and idle overhead on the German instances. Still, the amount of super-linear speedup remains astonishing and shows that a small portion of the overall data is used in many queries. Second, the distribution of the randomized work load is unbalanced for the experiments on the European graph, i.e. some processors finish sooner than others. This effect becomes less significant, the more

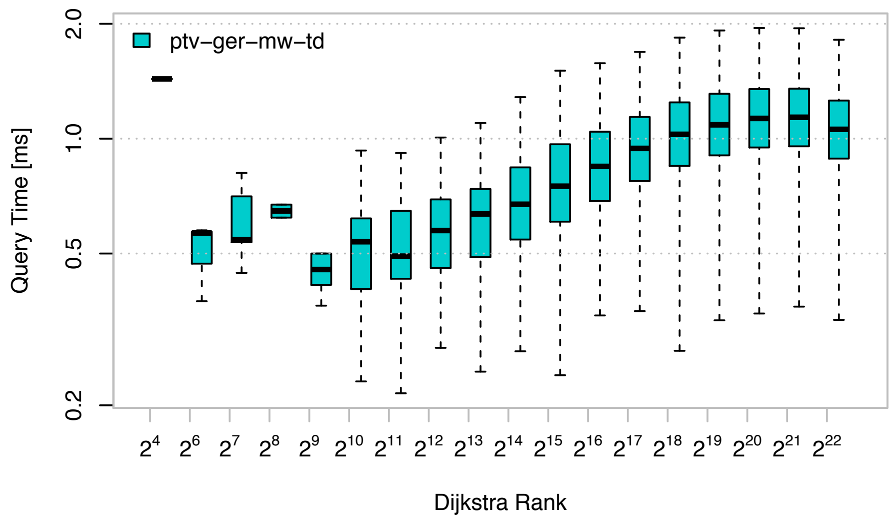


Figure 4.8.: Rank Plot for Queries on Instance *ptv-ger-mw-td* with 16 PEs

processors work on the batched queries, which may be due to the quality of the partition and also due to network latency anomalies that appear from time to time with the MPI implementation.

Next, the single query behaviour is analysed. It is measures how much time each individual distributed shortest path query takes. To do so, a detailed look into the query time distribution of Figure 4.8, 4.9 and 4.10 is conducted using the well-established methodology of Sanders and Schultes [168]. It shows a plot of the individual query times of 100 000

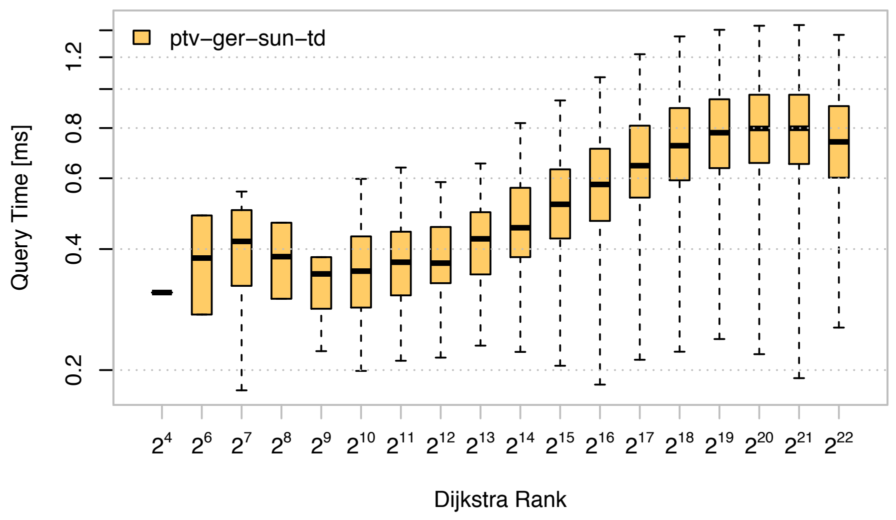


Figure 4.9.: Rank plot for Queries on Instance *ptv-ger-sun-td* with 16 PEs

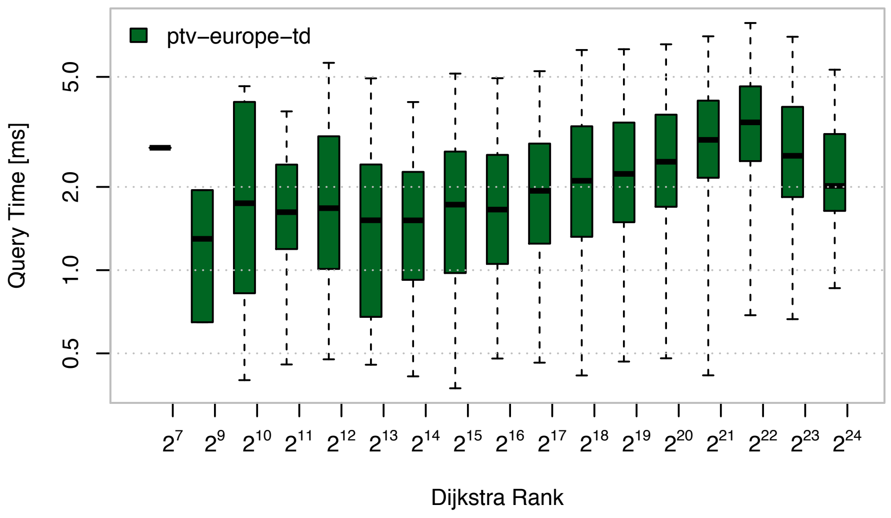


Figure 4.10.: Rank plot for Queries on Instance *ptv-europe-td* with 16 PEs.

randomly chosen distributed queries on each of the road networks with the property that a plain label-correcting time-dependent Dijkstra settles 2^i nodes, rounded down to the next smallest power of two. We report on an exemplary experiment using 16 processors only. Note that the results for other numbers of processors are similar.

The graph depicting the German Midweek Scenario shows a typical distribution of query times as shown in Figure 4.8. Most of the queries are answered in much less than two milliseconds on average except for a few outliers. The result of the experiments on the German Sunday network are given in Figure 4.9. The limited time-dependency of this particular instance is reflected in the much faster query times. The queries are answered well below a five milliseconds excluding a few outliers. Again, the European network shows a slightly different behaviour. Results are shown in Figure 4.10. The overall query latencies of roughly 2 milliseconds are good. See Appendix C for a combined rank plot. We do no see any drastic outlier in any of the experiments, which we observe as good maximum query time behavior.

The average message length is also measured over all queries and amounts to roughly 3–4 kilobytes depending on the test instance. This is the amount of data needed to communicate the result of the forward search phase. It is a somewhat expected size since the search spaces

instance	query [ms]	message [byte]
ptv-ger-sun-td	0.79	2796
ptv-ger-mw-td	1.13	3732
ptv-europe-td	3.21	4312

Table 4.3.: Average Query Performance and Message Overhead on 16 PEs.

sizes are rather small and consist of a few hundred nodes. Batz et al. [21] give a number of 520 settled nodes in total for an earliest-arrival query on average for the German test instance. We note that the experiments are done on a cluster with a fast InfiniBand interconnection network. But we expect a similar performance on a cluster with slower, but cheaper network hardware. For example, Gigabit Ethernet is a commercial off-the-shelf product standard and has a latency of about 100–150 microseconds and a bandwidth of 80–90 MB per second, in practice. Transferring 4 KB of data should therefore take only about 150-200 microseconds. When we compare that to the average query times on 16 PEs of our instances given in Table 4.3, we see that the expected slow-down is bearable.

A parallel system exhibits somewhat larger query times because of inevitable latencies. But these latencies are still negligible compared to the latencies common to the Internet which are at least an order of magnitude larger and often amount to dozens of milliseconds. But local queries may constitute a significant part of the queries seen in practice. Therefore, we conclude that our average, as well as maximum query times are more than suitable for an online service.

4.4. Concluding Remarks

Preprocessing With Constant Space per Core. We presented an algorithmic tuning parameter between preprocessing efficiency and space requirements. Carefully chosen and engineered data structures and associated algorithms allow for greater flexibility during the preprocessing of large real-world road network instances. The high performance of the tabulation hashing applications is attributed to much better cache locality. This has been leveraged during data structure design and implementation. Applying tabulation to the witness search effectively decreases the memory requirements per core with performance improvements for very large graphs. Applying it to tie-breaking gives a reasonable speedup in preprocessing efficiency. If memory is of essence and preprocessing time is less important, then only the tabulation hash based storage for the witness search may be applied. Or if time is limited even both may be applied.

In the future, we would like to look into further space-time trade-offs that can be used by the heuristics that are used in the Contraction Hierarchies preprocessing phase. For example, we see opportunities to move the witness search into a *witness oracle* that does not actually compute shortest paths for a given metric, but rather exploits the (metric-independent) topology of the underlying graph of the road network.

Distributed Preprocessing by Message Passing. We gave a description of a distributed-memory parallel CH variant and successfully demonstrated how to distribute time-dependent Contraction Hierarchies to a cluster of machines with medium sized main memory. The description as well as the experimental evaluation include the necessary algorithms and data structures for preprocessing and query. For large networks we approach two orders of magnitude in reduction of preprocessing time and at least a considerable constant factor in required local memory size. With respect to the targeted applications, we are certain that reduced turnaround times for including up-to-date traffic information are realistic.

For future work, we believe that there is considerable room for further improvement: Reducing per cluster node memory requirements and more adaptive node contraction that is able to interrupt PEs that take much longer in an iteration than others. We would also look into the impact of improved graph partitions with a better size of the cut, e.g. as reported by [59, 169]. For example, partitioning a graph into a high number cells that can be preprocessed independently may yield an even better scalability of the preprocessing. Extending our approach to further parallelization paradigms like Map-Reduce seems to be a challenging field of work, too.

CHAPTER 5

Taking Advantage of the Hierarchy

5.1. Central Ideas

Today's requirements for routing services, be it in-car or as a web-service, ask for more than just computing the shortest or quickest paths. These additions are more complex queries on the preprocessed network that can answer requests for alternative routes, evaluations of a location based on the network topology or compute matchings for a ride sharing application.

The search spaces of the CH search graph are small on average and can be efficiently preprocessed by exploring the search graph. Preprocessed search spaces can be easily tested for non-empty intersection and corresponding paths can be evaluated. These are the basic building blocks in this chapter for the aforementioned complex queries on the road network that previously either took too long or were infeasible.

The most common queries are point-to-point queries already explained in Section 2.5, many-to-one queries and many-to-many queries that can be answered efficiently with Contraction Hierarchies by precomputing search spaces. As we see later, one-to-many queries are a special case of many-to-many queries. This is applied in Section 5.2, where a search for points of interest and subsequently a *walkability index* is developed. Precomputed search spaces are also used in Section 5.3 that reports on a ride sharing application.

We present a fast algorithm with preprocessing in Section 5.5 for computing multiple good alternative routes in road networks. Our approach is based on single via node routing on top of Contraction Hierarchies and achieves superior quality and efficiency compared to previous methods. The algorithm has negligible memory overhead.

Sweeping the CH search data structure is presented in Section 5.6. We allocate information on a vast set of paths in the road network without explicitly touching every single road segment during path assignment. The information is propagated only once through the entire network, independent of the number of paths.

References

The contents of this chapter are based on the following publications: Section 5.2 is based on joint work with Fletcher Foti and Paul Waddell [76]. The single hop ride sharing approach of Section 5.3 is based on joint work with Robert Geisberger, Peter Sanders, Sabine Neubauer and Lars Volker. The work started as a student research project [81] and was later reimplemented by Geisberger [82] to run additional experiments. The multi-hop approach is based on joint work with Florian Drews [?]. The experimental evaluation was conducted as part of a bachelor thesis [68]. Section 5.5 is based on joint work with Dennis Schieferdecker [127] and Section 5.6 is based on joint work with Peter Sanders [126]. The description of POI index generation are given for sake of completeness, as well as single-hop ride sharing of Section 5.3 which was presented in [79], too. Wordings of the above publications are used in this thesis.

5.2. Points of Interest and Walkability

Assume that points of interests (POIs) are located at the nodes of a graph. For sake of simplicity we describe the case of a single category of POIs only, but the description is easily transferable to multiple categories. Querying the k-nearest (or all) POIs for a given location can be done naively by running an unidirectional variant of Dijkstra's algorithm on the graph. While this is technically feasible, this method does not scale well in practice, because most explored nodes are not POIs and especially in rural areas, the searches may have to cover rather large areas. Instead, we enrich the search graph data with preprocessed search spaces for the nodes that correspond to POIs and show how to conduct an efficient query for the nearest k POIs using CH.

We now describe the natural method of (conceptually) storing information from backward search spaces in buckets at the nodes of the graph. To index the POIs, the backward CH search space $G^{\downarrow}_{p_i}$ of each POI p_i node is explored. Each node holds a *bucket* that keeps track of the nearest POIs from these searches. More precisely for each POI p_i we do the following: The backward search space is explored. For each settled node v the entry $[p_i, d^{\downarrow}(p_i, v)]$ is inserted into the respective bucket if its backward distance belongs to the smallest k entries. Note that k entries suffice.

To search for the set of the k nearest POIs for a given node s, the forward search space $G^{\uparrow}_s$ is explored. Consider the settled nodes v' of this search with their forward distances $d^{\uparrow}(s, v')$. The search keeps track of the k *nearest* entries by scanning the bucket of v' and updating its list of candidates C such that it contains up to k entries with smallest total distance.

$$C := \min_k \underbrace{d^{\uparrow}(s, v')}_{\text{forw. search}} + \underbrace{d^{\downarrow}(v', p_i)}_{\text{bucket entry}} \ .$$

The search can be pruned as soon as the furthest POI in the candidate set is closer to the source node than the node that would be fetched next from the priority queue of the search. It is easy to see, that this method can be adapted to store multiple categories of POIs by introducing further buckets at the nodes. The correctness of the search follows by the same argument as given in Section 2.5.1 for the many-to-many query algorithm.

Recently, *walkability scores* have become popular among estate agents to capture the ability to conduct daily routine duties by walking. The most prominent example is *Walkscore* by FrontSeat Software[1] that expresses a location's walkability score by a number between 0 and 100. The walk score methodology has been published before [175] and it is mainly a weighted average of about two dozen POI queries for roughly 10 distinct POI categories. For example, it considers the distances to the first grocery store, the first 10 restaurants, 2 coffee shops among others. Contrary to the work of Rice and Tsotras [161] one is not looking for the distance of a path that covers all nearest POIs, but rather looks at the shortest path distance from a location to *all* the points of interest.

Experimental Evaluation. We implement the above algorithm in C++ using GCC's C++ compiler with full optimizations. The experiments are done on Machine E and the test instance is *osm-bay-area*. The road network has been extracted from OpenStreetMap data as have been the POIs. The ratio of POIs to network nodes is roughly 1 to 10. POIs that were not connected to the road network are mapped to their nearest node.

First, we compute the CH search graph and then process the hierarchy to hold additional POI information as explained above for each of the categories and for the upper bounds on the number of nearest POIs as defined from the queries. The query algorithm has been adapted to query for all categories simultaneously since distinct queries would essentially repeatedly explore the same search space. Since the data structure is read-only, we distribute the queries onto all of our cores and expect nearly linear speedup. Computing the score for all nodes in the network can be done in 1.2 seconds on 12 cores and an individual query aggregates 23 variables each in roughly $50\mu s$, which also includes the actual computation of the score.

5.3. Ride Sharing

The full results on the results of single-hop ride sharing in the following section, which was originally done in cooperation, are already presented by Geisberger [79]. The presentation at hand is a recapitulation that focuses on this author's contribution while highlighting the overall results.

5.3.1. Single-Hop Ride Sharing

Today's ride sharing services still mimic a better billboard that list offers and allow to search for the origin and destination city, sometimes enriched with radial search. Finding connections is possible between a small set of designated places only. We overcome this limitation by implementing a dynamic many-to-many route planning algorithm that builds on Contraction Hierarchies and is able to handle a massive amount of distance computations. Finding a connection between big cities is quite easy as these places can be easily looked up from a small list of designated cities. But when you want to go from a small town to another small town you may run into problems that the places are not available.

[1] `http://www.walkscore.com` – accessed July 21st, 2012

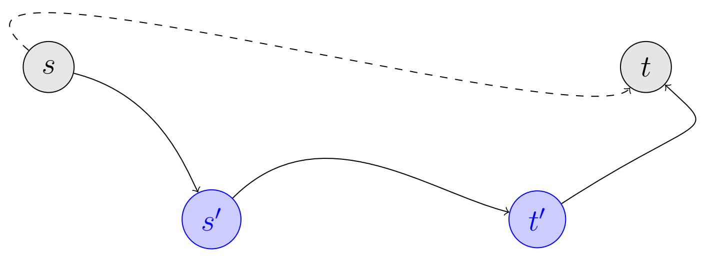


Figure 5.1.: Request (s', t') and matching offer (s, t) by allowing a detour.

We solve this interesting problem by presenting a fast algorithm that computes the offers with the smallest detours with respect to a request. Our experiments show that the problem is efficiently solvable in times suitable for a web service implementation. For realistic database sizes we achieve real-time lookup durations, which on average is less than a microsecond per single distance. We are able to further reduce running times by a factor of three under certain behavioral assumptions that allow us to prune the search space.

For many services an offer only fits a request if and only if origin and destination locations and the possibly fixed route of driver and rider are identical. We call such a situation a *perfect fit*. Some services offer an additional radial search around origin and destination and fixed way-points along the route. Usually, only the driver is able to fix the route in advance. The existence of these additions shows the demand for better route matching techniques that allow a small detour and intermediate stops. We call that kind of matching a *reasonable fit*. However, previous approaches used only compute the perfect fits.

We present an algorithmic solution to the situation at hand that leads to better results independent of the user's level of cooperation or available database systems. For that, we lift the restriction of a limited set of origin and destination points. Unfortunately, the probability of perfect fits is close to zero in this setting. But since we want to compute reasonable fits, our approach considers intermediate stops where driver and passenger might meet and depart later on. More precisely, we adjust the drivers route to pick up the passenger by allowing an acceptable detour.

Definition 7. *We say that an offer $o = (s, t)$ and a request $g = (s', t')$ form a* reasonable fit *if there exists a path $P = \langle s, \dots, s', \dots, t', \dots, t\rangle$ in G with $l(P) \leq \mu(s, t) + \varepsilon \cdot \mu(s', t')$, with $\varepsilon > 0$.*

Figure 5.1 visualizes the described situation. The solid lines symbolize the distances that are driven, while the dashed one stands for the shortest path of the driver that is actually not driven at all in a matched ride.

If we model a rider's detour having the same cost as a driver's detour, then the situation is symmetrical. The ε in Definition 7 depicts the maximal detour that is reasonable. Applying the ϵ to the rider's path gives the driver an incentive to pick up the rider. We look at a simple pricing scheme from algorithmic game theory. The so-called *fair sharing rule* [149] states that players who share a ride split costs evenly for the duration of the ride. Additionally, we say that drivers get compensated for their detours directly by riders using the savings from the shared ride. Implicitly, we give the driver an incentive to actually pick the passengers up at their start s' and to drop them off at their destination t'. Formally, we have that a match is economically worthwhile if and only if there exists an ε for which

$$\mu(s,s') + \mu(s',t') + \mu(s',t') + \mu(t',t) - \mu(s,t) \leq \varepsilon \cdot \mu(s',t') \ . \tag{5.1}$$

It is easy to see that any reasonable passenger will not pay more for the drivers detour than the gain for the shared ride, which is at most $\frac{1}{2} \cdot \mu(s',t')$. Otherwise, it would be cheaper for the passenger not to join the ride at all.

5.4. Algorithmic Details

For a dataset of k offers $o_i = (s_i, t_i)$, i=1..k, and a single request $g = (s', t')$, we need to compute the $2k + 1$ shortest path distances $\mu(s', t')$, $\mu(s_i, s')$ and $\mu(t', t_i)$. The detour for offer o_i is then $\mu(s_i, s') + \mu(s', t') + \mu(t', t_i) - \mu(s_i, t_i)$.

We adapt the algorithm of Knopp et al. [114] for distance table computation and solve our problem by computing for each s_i the forward search space $G^{\uparrow}(s_i)$ in advance and store it. More precisely, we do not store each $G^{\uparrow}(s_i)$ separately, but we store *forward buckets*

$$B^{\uparrow}(u) := \{(i, d^{\uparrow}) | (u, d^{\uparrow}) \in G^{\uparrow}(s_i)\} \tag{5.2}$$

with each potential meeting node u. To compute all $\mu(s_i, s')$ for the request, we compute $G^{\downarrow}(s')$, then scan the bucket of each node in $G^{\downarrow}(s')$ and compute all $\mu(s_i, s')$ simultaneously. We have an array of tentative distances for each $\mu(s_i, s')$. Initially, the distances are ∞, and we decrease them while scanning the buckets. The decrease happens along the lines of the decrease of a key during the run of Dijkstra's algorithm:

$$\mu(s_i, s') = \min_{u \in V} = \{d^{\uparrow} + d^{\downarrow} | (i, d^{\uparrow}) \in B^{\uparrow}(u), (u, d^{\downarrow}) \in G^{\downarrow}(s')\} \tag{5.3}$$

Symmetrically, we compute *backward buckets* $B^{\downarrow}(u) := \{(i, d^{\downarrow}) | (u, d^{\downarrow}) \in G^{\downarrow}(s_i)\}$ to accelerate the computation of all $\mu(t', t_i)$. Computing distances is very space- and cache-efficient, because it stores plain distances and scans consecutive pieces of memory. The single distance $\mu(s', t')$ is computed separately.

Adding and Removing Offers. To add or remove an offer $o = (s, t)$, we only need to update the forward and backward buckets. To add the offer, we first compute $G^{\uparrow}(s)$ and $G^{\downarrow}(t)$. We then add these entries to their corresponding forward/backward buckets. Note that this is very similar to the preprocessing of distance tables or POI indices in Section 5.2.

Entries of a bucket are stored unordered and to remove the offer, we need to remove its entries from the forward/backward buckets. This makes adding an offer very fast, but removing it requires scanning the buckets. Scanning all buckets for a request is prohibitively slow as there are too many entries. Instead, it is faster to compute $G^{\uparrow}(s)$ and $G^{\downarrow}(t)$ again to obtain the set of meeting nodes whose buckets contain an entry about this offer. We then just need to scan those buckets and remove the entries.

Assume that ε is given, then we exploit the fact that we need to obtain $G^{\uparrow}(s')$ and $G^{\downarrow}(t')$ for the computation of $\mu(s',t')$. For $(u,d^{\uparrow})$ in $G^{\uparrow}(s')$ holds $d^{\uparrow} \geq \mu(s',u)$ and $(u,d^{\downarrow})$ in $G^{\downarrow}(t')$ holds $d^{\downarrow} \geq \mu(u,t')$. We compute the distance $\mu(s',t')$ first, and temporarily keep $G^{\uparrow}(s')$ and $G^{\downarrow}(t')$ that we obtained during this search.

Experimental Evaluation. We implement the above data structures and algorithms in C++ using the GCC Compiler 4.3.2 with full optimizations. The experiments are done on Machine F, running Linux kernel 2.6.27. The test graph instance is *osm-germany-1*.

To test our algorithm, we obtained a dataset of real-world ride sharing offers from Germany available on the web. We match the data against a list of cities, islands, airports and the like, and ended up with about 450 unique places. We test the data and checked that the lengths of the journeys are exponentially distributed. This validates assumptions from the field of transportation science. We assume that requests would follow the same distribution and chose our offers from that dataset as well.

To extend the data set to our approach of arbitrary origin and destination locations, we *perturbe* node locations of the data set. For each source node s we unpack the forward search space $G^{\uparrow}(s)$ up to a distance of 3 000 seconds of travel time. From that unpacked search space we randomly select a new starting point. Likewise we unpack the backward search space of each destination node up to the distance and pick a new destination node. Distorting the location in this way preserves the distance distribution of the original data.

The results of the experiments are given in Table 5.1. Column *type* specifies the input data type as described above, *#offers* gives the number of offers that are used in the experiment, while *bucket size* gives the space overhead in addition to the underlying CH. Columns *add offer* and *rem. offer* give the average duration how long it takes to either add or remove an offer. The columns under the heading *match request* show how long it takes to match a request against the data set for varying lengths of detours. Note that the detour is given relatively with respect to the length of the shortest path of the request.

The size of the occupied space scales linearly with the number of offers in the data set. This is expected behavior since the number of buckets touched is independent of the number

	# offers	bucket size	add offer	rem. offer	match request [ms] $\varepsilon =$							
type		[MiB]	[ms]	[ms]	0.0	0.1	0.2	0.3	0.4	0.5	1	∞
pert.	10^4	28	0.24	0.29	0.9	1.1	1.3	1.5	1.6	1.8	2.7	4.1
pert.	10^5	279	0.24	0.30	4.4	6.1	8.1	10.2	12.1	14.0	25.1	43.4
unpert.	10^4	32	0.26	0.32	1.1	1.3	1.6	1.7	1.9	2.1	2.8	4.3
unpert.	10^5	318	0.27	6.26	5.6	7.9	10.4	12.4	14.5	16.1	26.3	44.6
random	10^4	31	0.25	0.30	1.1	1.3	1.5	1.7	1.9	2.1	3.5	4.3
random	10^5	306	0.26	0.32	6.0	7.8	10.1	12.6	15.4	18.5	34.9	45.1

Table 5.1.: Performance of our Algorithm for Depending on Types of Offers, Requests, Number of Offers as well as Varying values for ε.

#	max. detour $\varepsilon =$					
offers	0.0	0.1	0.2	0.3	0.4	0.5
10^3	0.00	0.70	0.90	0.98	0.99	1.0
10^4	0.00	0.94	0.99	0.99	1.0	1.0
10^5	0.00	0.99	1.0	1.0	1.0	1.0

Table 5.2.: Matching Rates Depending on Allowed Detour and Data Set Size.

of offers. It depends only on the underlying speedup technique. We see that adding and removing offers does not depend on the number of offers in the data set, which is also expected behavior since the algorithms do not depend on the size of a bucket. The duration of matchings grows with the allowed detour. For realistic amounts of offers, this time grows to as much as 45 milliseconds which is acceptable for an online service where one would like answers that are perceived as instant. Additionally, we observe the robustness of our method because all three types of data behave similarly in the experiments.

Next, we have a look at the matching rates that our algorithm achieves. Table 5.2 gives the results of an experiment, where we note the fraction of requests for which we found a match by varying detour and data base size. Note that $\varepsilon = 0$ resembles the case of city to city routing only. We see that the allowed detour is a tuning parameter to increase matching rates when the number of offers is small. A moderate detour of 20% allows to match about nine out of ten requests for 10^3 offers. For a larger number of offers, the detour can be even smaller and still nearly all requests find a match. Users will measure the quality of a match by the detour. In other words, the smaller the detour for the driver the better the match and the identified tuning parameter allows to increase this quality indicator with a growing number of offers in the system.

5.4.1. Multi-Hop Ride Sharing

We like to give users of a ride sharing system even greater flexibility. Instead of allowing the riders to join one and only one driver, we allow them to transfer between drivers, i.e. ride with one driver for some time and then transfer to another one. The algorithm of Section 5.3.1 can be extended to handle more than one hop by checking more distances, but this leads to a combinatorial explosion for all the pairwise distances that must be checked. But multiple hops may lead to connections that would have been impossible otherwise.

Modelling Multiple Hops. As a first step to allow more realistic transfers, offers are now associated with a departure time and implicitly with an arrival time, too. To reduce complexity, we define a sufficiently high number of *stations* in the road network where riders switch the car, i.e. perform the *hop*. Therefore, *multi-hop offers* as well as *multi-hop requests* are represented by triples $\langle s, t, \tau \rangle$, where s is a start station, t a target station, and τ the time of departure. The earliest arrival time at t is given by $\tau' := \tau + \mu(s, t)$. Note that travel times on the road network are time-*in*dependent, whereas ride sharing on top of this network is modelled with time-dependency.

For now, we say that a driver has only one empty seat to share but that a rider can make several transfers. We note that a rider and driver may have to wait for some time to actually join a ride. We denote the waiting time at station s_i for a given match m by $\omega_m(s_i)$ and the duration of path $P := \langle s_0, s_1, \dots, s_t\rangle$ by

$$d(m) = d(P) := \sum_{i=0}^{t} \left(\omega_m(s_i) + \mu(s_i, s_{i+1})\right) \ .$$

The raw travel time (without any waiting) $\mu(s_1, s_t) := \sum_i \mu(s_i, s_{i+1})$ is the sum of the individual travel times. As we are dealing with a scenario that accounts for departure and arrival times, we have to account for the fact that waiting periods must not be infinite. The rider should not have to wait too long for pick up. And the driver is usually already taking a detour to pick up the rider as argued previously. Thus, waiting times should be limited by a factor $\delta \geq 0$. It models the maximum percentage of time relative to the shortest path distance offer (request) that rider (or driver) are willing to wait in order to share the ride. We define reasonable delays for rider and driver:

Definition 8 (Reasonable Delay). *A match m is said to have* reasonable driver's delay *if the waiting does not exceed a relative threshold, i.e.*

$$\delta_d \geq d(P) - \frac{d(P) - \mu(s,t)}{\mu(s,t)} = \frac{d(P)}{\mu(s,t)} - 1. \tag{5.4}$$

Let $r = (s', t', \tau, \delta_r)$ be a request for a set of offers O and $P := \langle s_1, \dots, s_n\rangle$ a path where the potential (sub-)rides $m_i = (s_i, s_{i+1}, \tau_i)$, fit together such that $\tau_i + d(g_i) \leq \tau_{i+1}$ is called a fit. *Likewise, a ride m is said to have* reasonable rider's delay δ_r *if the waiting does not exceed a relative threshold, i.e.*

$$\delta_r \geq \frac{d(P) - \mu(s_1, s_t)}{\mu(s_1, s_t)} = \frac{d(P)}{\mu(s_1, s_t)} - 1. \tag{5.5}$$

It is called a reasonable fit *if the driver and rider have reasonable delays and if the drivers detour is reasonable, too, as modelled in Equation 5.1 of the previous section.*

In other words, a fit is a sequence of rides, that allows the rider to travel from the origin to the destination via (potentially) multiple hops. A fit that minimizes the rider's delay δ_r is called *best fit.* Figure 5.2 gives a visualization. The rider waits 5 minutes at s_1 and reaches the destination with a relative delay of $\delta_r = 1/3$.

Slotted Time-Expanded Graph. Our approach to ride-sharing and to finding a best fit has striking similarities to the problem of finding fastest train connections in a railway/timetable network. The drivers provide time-dependent connections between station through their offers, but note that the underlying road network is assumed to have static travel times.

We adapt the common technique of (simplified) time-expanded graphs [159] (TEGs). Such a graph models the timetable information into its topology. There exists a node for every

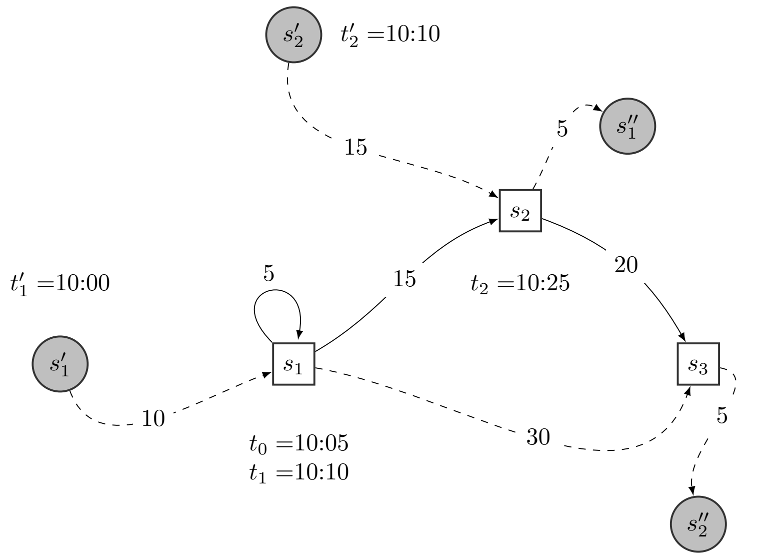


Figure 5.2.: Example match. Offers $o_1 = (s'_1, t'_1, \tau'_1)$ and $o_2 = (s'_2, t'_2, \tau'_2)$ for request $r = (s_1, s_3, \tau_0)$ yield a 2-hop ride $g_1 = (s_1, t_1, \tau_1)$, $g_2 = (s_2, s_3, \tau_2)$ given by path $P = \langle s_1, s_2, s_3 \rangle$.

event in the timetable, i.e., one node for every *departure* or *arrival* of a train. For every connection, there exists a *train edge* which connects a *departure node* of station S_1 with an *arrival node* of station S_2 at time τ. Each edge is associated with a weight that is the travel time. Note that stations $S =: \{s_1, \ldots, s_k\}$ are represented by sets of nodes and not by single nodes and the set of nodes is sorted according to the time of the event they represent. So-called *transfer edges* connect the time-ordered nodes of a station by edges to model waiting within the station. Our idea is to build a TEG that resembles a super set to all *potentially* reasonable fits which are induced by the set of offers. By potentially we mean that we insert an edge if there *could* be some later request for which it might be reasonable. We will give a more detailed explanation in the following sections. A best multi-hop fit can then be computed by an earliest-arrival query in the resulting graph. A TEG is a directed and acyclic graph (DAG) since edges move forward in time. On one hand, it suffices to run a graph traversal like BFS on the graph to discover shortest paths. On the other hand, we prune the search space by a goal-directed graph search as we show in the following.

Travel itineraries are generally error-prone. Unexpected events like traffic conditions, accidents, severe weather, etc. make it hard to predict travel times with certainty. Shared rides are especially frail to delays as they are mostly organized between private partners that are all but bound to a service level agreement. Thus, one may not assume exact travel times for our ride sharing approach. For that reason, we adapt the TEG concept to what we call a *Slotted Time-Expanded Graph* (STEG). Our idea is to introduce time-slots which chop continuous time into discrete and equal-sized ranges. We postpone departure and arrival events until the end of any time-slot. As a consequence, we are inserting waiting periods at the stations. This waiting period is a backup to compensate for unexpected delays while still being able to make a transfer. We formalize this description by the following definition:

Definition 9 (Slotted Time-Expanded Graph (STEG)). *A STEG is a directed acyclic graph $G = (V, E)$ which maintains a time range of $t_r = s_l \cdot s_c$, where $s_l \in \mathbb{N}_+$ is the length of a time-slot and $s_c \in \mathbb{N}_+$ is the total number of slots. A node $u := (w, i) \in V$ resembles station w at time-slot i. Directed edges $e := (u, v, o) \in E$, where $u := (a, i)$ and $v := (b, j)$, resemble (potential) rides with offer o going from station a to b with respect to the corresponding time-slots, i.e. $i = \lfloor \tau_a / s_l \rfloor$ and $j = \lceil \tau_b / s_l \rceil$ for departure time τ_a and arrival time τ_b. Transfer edges model waiting at the station such that a time-slot node of a station is connected to the next slot in time.*

Adding an edge into a STEG implies that *some* waiting time may be added to an offer and we notice that rides that may have been reasonable without waiting may be rendered unreasonable, while the driven detour may still be perfectly fine. On one hand, this approach delays traveling by design, but on the other hand, it adds *some* reliability to making transfers and thus reflects an arguably more realistic scenario.

Adding and Removing Offers. Initially, we model an empty system with some set of stations $\mathcal{S}$ that will be formalized below. The STEG holds a node for each time slot per station and all transfer edges are present. When an offer is added to the system, we insert a number of edges into the STEG. An edge $e = (u, v, o)$ has to be added if the driver of offer o *could* take some rider from u to v with reasonable detour and delay. While this could result in $|\mathcal{S}| - 1$ outgoing edges for each offer's departure node, we add only those edges that represent potentially reasonable routes. More precisely, we omit those edges that violate the driver's delay and detour constraints.

Therefore, we introduce (hopefully smaller) sets $\mathcal{S}_1$ and $\mathcal{S}_2$ which resemble candidate sets. The sets exploit the triangle inequality and are defined as follows:

Definition 10 (Reasonable Station Candidates). *Consider an offer $o = (s, t, \tau, \delta, \varepsilon)$ that specifies source, target, departure time, as well as upper bounds for delay and detour. Consider the set of all reasonable rides $G := \{(s', t', \tau') | s', t' \in \mathcal{S}\}$ for offer o. A super set of the source stations from which reasonable routes depart for an offer o is given by*

$$\mathcal{S}_1 := \{s' \in \mathcal{S} \mid \mu(s, s') + \mu(s', t) \leq (1 + \delta) \cdot \mu(s, t)\}\,.$$

A super set of the target stations at which reasonable routes arrive is given by

$$\mathcal{S}_2 := \{t' \in \mathcal{S} \mid \mu(s, t') + \mu(t', t) \leq (1 + \delta) \cdot \mu(s, t)\}\,.$$

The definitions for both sets are very similar and it is easy to see that this superset covers all reasonable fits. The needed distances can be computed easily by two one-to-many queries on the underlying road network. If there exists a distance table for all pairwise distances between the stations then a one-to-many query boils down to a column (line) scan. Note that not all combinations in $\mathcal{S}_1 \times \mathcal{S}_2$ are feasible for every offer. Therefore, we have to verify each candidate before actually adding the edge. The complete algorithm is given as pseudo-code in Listing 5.1.

Listing 5.1: Adding a Multi-Hop Offer

```
function add_offer($o := (s, t, \tau, \delta, \varepsilon)$) do
  $\mathcal{S}_1, \mathcal{S}_2 = \{\}$
  foreach $s', t' \in \mathcal{S}$ do
    if $\mu(s, s') + \mu(s', t) \leq (1+\delta) \cdot \mu(s, t)$ then
      $\mathcal{S}_1 := \mathcal{S}_1 \cup s'$
    end
    if $\mu(s, t') + \mu(t', t) \leq (1+\delta) \cdot \mu(s, t)$ then
      $\mathcal{S}_2 := \mathcal{S}_2 \cup s'$
    end
  end

  { insert an edge for each potentially reasonable ride }
  foreach Station $s' \in \mathcal{S}_1$ do
    foreach Station $t' \in \mathcal{S}_2$ do
      $g := (s', t', \tau)$
      $\tau' := \tau + \mu(s', t')$
      $P := \langle s, s', t', t \rangle$
      if is_reasonable($g, P, \delta, \varepsilon$) then
        $u := (s', \lfloor \tau / s_l \rfloor)$
        $v := (t', \lceil \tau' / s_l \rceil)$
        insert_edge($u, v, o$)
      end
    end
  end
end
```

Removing offers is necessary when a ride has been matched or in case a driver wishes to retract the offer. Scanning for edges to delete is cost intensive as it has to scan the outgoing edge lists of $|\mathcal{S}|$ stations. Instead we do a *lazy delete*, which means that we mark the offer itself as deleted and simply ignore its edges. The deletion of edges is done during queries where edges get scanned anyway and, thus, the cost is amortized over later operations. We note that mutual exclusion is necessary during deletion in a multi-threaded system.

Matching. We compute a *best fit* for a given request by running an earliest arrival query using Dijkstra's algorithm on the STEG as briefly mentioned before. Note that we can still use the label setting variant of Dijkstra's algorithm. Consider that distances on the underlying road network are available upon request. The query can be pruned at all nodes that violate delay constraints. First, for every edge that is relaxed during the search, we get a (tentative) value for the target node of the edge. We do not insert nodes into the priority queue that violate any threshold, called pruning rule 1. But we can also apply a second, more stricter pruning by using the underlying road network. For each settled node v, we compute the shortest path distance $\mu(v, t')$ in the road network graph. These distances can be looked up in constant time in a table of all pairwise distances between stations. We assume that we could leave v right away by a direct connection to the target. Obviously, this straight connection is a lower bound to the actual distance, i.e. an optimistic estimate. If this *lower* *bound path* violates our constraints, we do not relax any edges of v, called pruning rule 2.

It is easy to see that both pruning approaches only discard non-optimal nodes. Interestingly, the second approach is an admissible heuristic for goal-directed search with A^* as it never over-estimates the costs. We use this finding in our implementation and its experimental evaluation in the following section.

Experimental Evaluation. We implement the above data structures and algorithms in C++ using the GCC Compiler version 4.3.2 with full optimizations. The experiments are done on a single core of Machine G. The test graph instance is *osm-germany-3*. We preprocess the road network using the shared-memory parallel Contraction Hierarchies processing of Vetter [184]. A binary heap is used as the priority queue data structure. In addition we compute a distance table for all pairwise distances between the stations used for pruning. This table is computed with the algorithm of Knopp et al. [114] as described in Section 2.5.1.

In absence of realistic test data, we define rider and driver to split costs evenly and to have equal preferences for detour and delay, i.e. $\delta := \delta_r = \delta_d$. We generate 10 000 stations for our data set. The locations of stations are selected uniformly at random with the assumption that node density strongly correlates with population density. Source and target locations for our trips are then drawn uniformly at random from the stations. According to a case study conducted by the Federal Ministry of Transport, Building and Urban Development of Germany [35] the vast majority of trips take place during the day between 6 am and 8 pm. We pick the departure time of any trip to be from this interval to simulate a single day. The length of a time slot in the STEG is 30 minutes. This gives an expected waiting time of 15 minutes at the station which we see as reasonable. Since the starting time of all slots are synchronous, it suffices to represent edge weights by a multiple of the slot length.

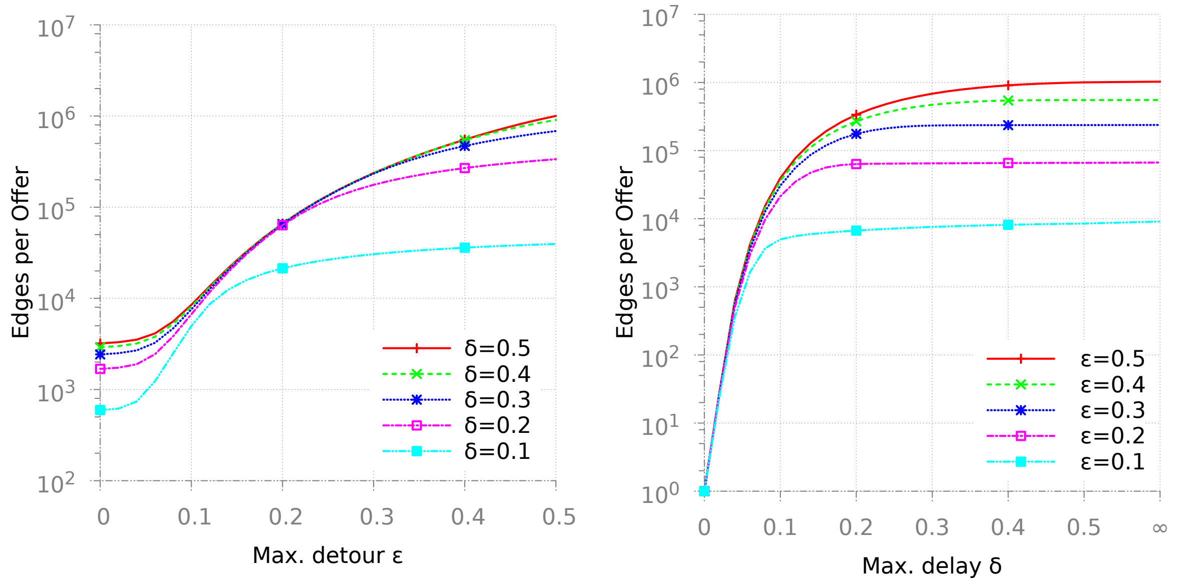


Figure 5.3.: Number of Edges per Offer Depending on Tuning Parameters

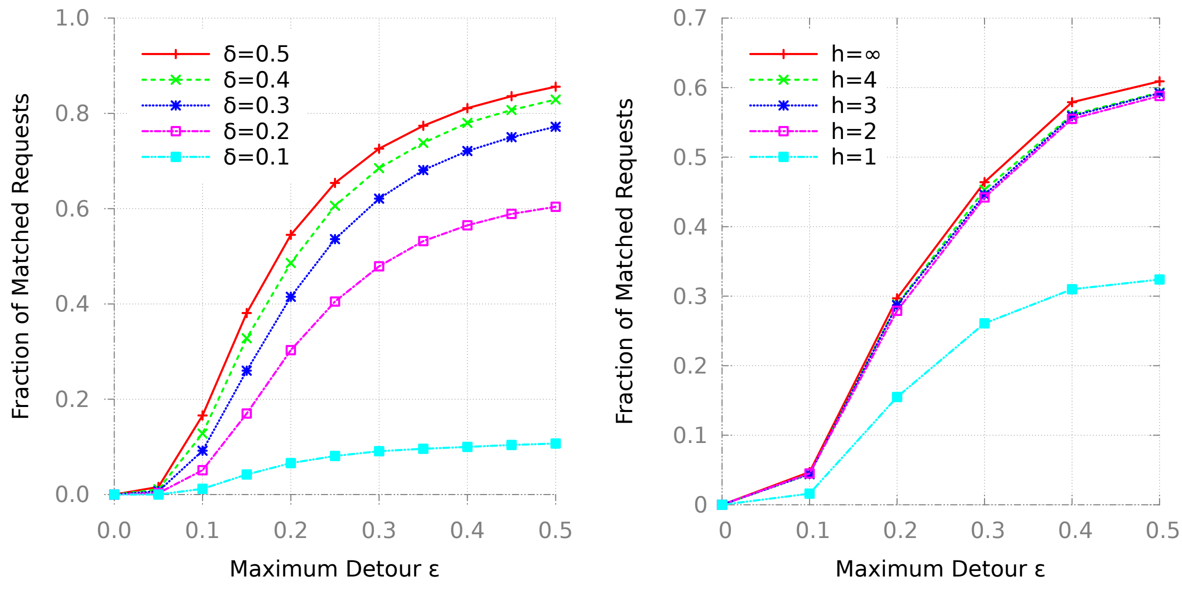
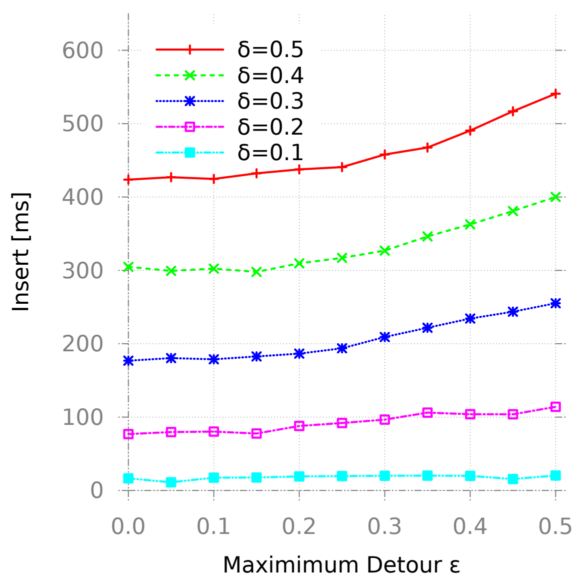


Figure 5.4.: Matching Rates Depending on Tuning Parameters (left) and on Number of Hops for a Fixed Delay of $\delta = 0.2$ (right).

We present three experimental results. First, we evaluate the number of edges inserted into the STEG depending on reasonable values for the delay factor δ and for the detour factor ε. We add 1 000 random offers and average over the number of inserted edges. Figure 5.3 gives a plot of the results. Note the logarithmic scale of the y-axis. We observe on the left hand side of Figure 5.3 that varying ε has a significant effect on the number of edges even when the allowed delay is high. On the right hand side, we observe that there is a point where further increasing the delay δ hardly increases the number of inserted edges. This appears to be true for all curves, i.e., all plotted maximum detours. As a rule of thumb we can say that it happens when maximum detour and delay thresholds are equal ($\delta \approx \varepsilon$).

Second, we experiment on the matching rate against a given set of requests. We insert 10 000 random offers into the STEG. We vary the thresholds for allowed detour and delay, while we look into the impact of a threshold on the number of hops as well. Again, we assume that driver and rider have a common interest and share the same delay and detour factors. Figure 5.4 reports on these experiments. We observe that the matching rate depends on the maximum allowed delay as a larger threshold makes less attractive rides reasonable. The higher the threshold, the more edges are inserted into the STEG and finding a suitable path in a dense graph is more likely than in a sparse one. As expected, we see the best matching rates at $\delta = 0.5$. We see a steep increase in the fraction of matched rides when increasing from $\delta = 0.1$ to 0.2 and after that increases are not as significant.

On the right hand side of Figure 5.4 we fixed the delay to the reasonable value $\delta = 0.2$ and look at the matching rate depending on the number of allowed hops. Results for other values of δ are similar. We manage a hop counter that is increased only when sharing a ride. It is not increased for transfer edges. (Nodes are not inserted into the priority queue if their



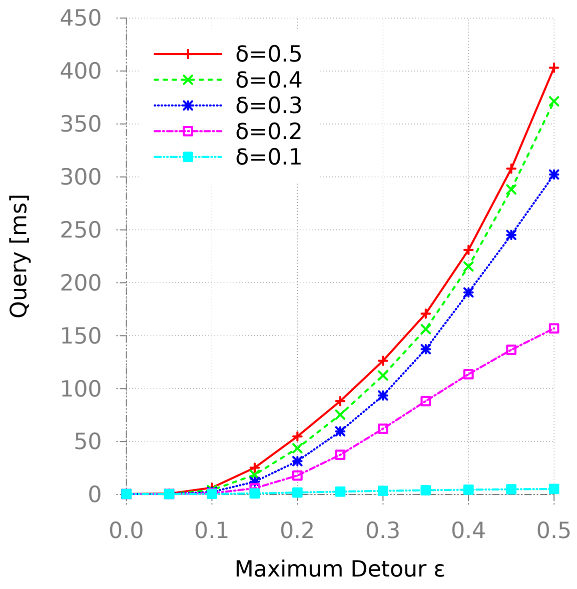


Figure 5.5.: Time to Compute a Match (left) and Time to Add an Offer (right) Depending on Tuning Parameters.

hop count exceeds $h \in \{1, 2, 3, 4, \infty\}$ in the respective experiment.) We see a significant increase in the matching rate when going from one hop to two hops, but we do not see any significant changes when further increasing the hop count. This means that searching for shared rides with three or more hops does not give significantly better results on average. We are surprised by the negligible improvements of more hops.

Third, we evaluate the performance of offer insertion as well as matching of queries. Removing an offer takes constant time as it just sets a flag in the list of offers and removes its edges in subsequent query operations, when they are encountered. We insert 10 000 randomly generated offers into an empty STEG and run the same amount of random queries against it without removing matched rides. Note that the space of the STEG has been preallocated to avoid costly resizes of the underlying basic data structures. The expected number of inserted edges per inserted offer is known from the previous experiment and we reserve 25% more space to be on the safe side. We implement Dijkstra's algorithm for comparison. Our implementation prunes the search such that it does not insert any nodes into the queue that violate the delay constraints by pruning rule 1. Our implementation of A^* on the other hand uses a distance table of all stations to look up lower bounds for goal direction and also prunes the search when it reaches a node that can be proven as sub-optimal by pruning rule 2.

Figure 5.5 reports on these experiments, where the query is run with A^*. On the left hand side of the figure, we observe that there is a significant difference between the results for $\delta = 0.1$ and $\delta = 0.2$ with respect to query time. The query time seems to be rather stable for lowest delay threshold independent of the detour, while the query time rises significantly for larger threshold values of detour and delay. But this is expected behavior as the search space of the shortest path query is small for $\delta = 0.1$. On the other hand, it is significantly

		Dijkstra			A^*			
	Edges	#sett.	#scan.	Query	#sett.	#scan.	Query	
$\delta = \varepsilon$	$[10^9]$	$[10^3]$	$[10^6]$	[s]	$[10^3]$	$[10^6]$	[s]	speedup
0.1	0.1	17	9	0.14	0.035	0.05	0.001	140
0.2	1.2	24	98	3.54	0.121	0.82	0.047	75
0.3	3.8	23	285	13.50	0.112	2.02	0.173	78
0.4	7.9	22	562	31.32	0.089	3.39	0.332	94
0.5	13.6	22	947	58.55	0.085	5.31	0.564	103

Table 5.3.: Query Performance of Our Algorithm.

larger for higher values of δ and we also see an amplification effect: The number of inserted edges is higher and the search is pruned at later distances. One effect adds to the other. On the right hand side of Figure 5.5, we see the time necessary to add an offer on average. We observe that the times are mostly independent of the maximum detour ε but that it depends on the value of δ. This is expected behavior the number of edges per offer depends on ε and most of the work that is done during insertion operations is to add edges into the graph.

Table 5.3 reports on the experiment that compares path searches with Dijkstra's algorithm versus our A^* variant with pruning by lower bounds on the distance. As we preprocess the distance table beforehand, getting a lower bound for a distance to a target requires just one memory access and is virtually for free during the search process. We observe speedups over Dijkstra's algorithm of 75–140. This is about an order of magnitude more than what we would expect from previous work, e.g., [25, 55] report factors of 7.0–8.5, when evaluating the performance of *uniALT* on a long distance timetable network. The speedups vary, because the graph remains relatively sparse at first, i.e. $\delta = \varepsilon = 0.1$. Increasing to $\delta = \varepsilon = 0.2$ infers more than order of magnitude more of scanned edges for our A^* variant. This is due to the amplification effect explained before. When further increasing the threshold value, the graph becomes even more dense and as a result the goal direction of our algorithm becomes more efficient. The query times of our search are at most just above half a second in all experiments, which can be arguably seen as fast enough for an online service in practice. This is especially true for tighter constraints. We conclude that the selection of parameters is a tuning parameter that gives a trade-off between query time, size of the data structure and achievable matching rates.

Practical Considerations. Note that we expect users of such a system to attribute themselves to a number of nearby stations within their reach. Generally, fixed source and target stations are replaced by small sets of sources and targets that are close to the drivers real location(s). These sets are generated by one-to-many queries, e.g., by the one from Section 5.2 or more realistically by a multi-modal, multi-criteria one-to-many search, e.g., by adapting the algorithm of Delling et al. [57]. When generating sets $\mathcal{S}_1$ and $\mathcal{S}_2$ we adapt the pruning to include feasible stations and in the subsequent pairwise verification, we adapt the check if a pair is reasonable. For a rider, we adapt our path search by starting simultaneously from a set of sources and stop once a reasonable target station has been settled. The search is then started from these nodes simultaneously and the stopping criterion must be adapted to stop the search once a *suitable* target station has been settled. The pruning during the path search is adapted likewise.

Time-dependency of the underlying road network can be handled as well. For example, the time-dependent variant of Contraction Hierarchies by Batz et al. [21] is able to serve (nearly) as a drop-in replacement for the underlying speedup-technique. We note that the edge-weights between stations in the STEG must be computed with the time-dependent routing, but an approximated time-dependent distance table that only stores distances for the departure times, i.e. end of time-slots, is easy to compute. This approximated and time-dependent distance table can then also serve to compute the lower bounds that are necessary for A^* pruning.

It is also possible to model more than one rider per driver. In this case, a driver has a certain capacity that is associated with an offer. Once a rider joins in, we remove this offer from the system and add two new offers with adjusted thresholds for detour and delay, and with decreased capacity. The first offer resembles the path from the drivers source to the first pickup point, while the second offer resembles the ride from the drop-off location of the rider to the drivers target location. Note that the thresholds must be set relative the already matched ride.

Minimum waiting times at a station can be modelled for the driver by choosing a later target node when inserting an edge. It is also easy to model such waiting times for the rider by adapting the search.

5.5. Candidate Sets for Alternative Routes

The following section is structured as follows. We show how to engineer previous algorithms to provide reasonable alternative paths with better efficiency. Then, we build on the results and introduce the notion of *candidate via nodes* to further speed up the computation by an order of magnitude. We show how to perform query variants and how to conduct the preprocessing efficiently. Finally, we conduct an experimental study on the performance and quality of our method.

5.5.1. The Baseline Algorithm

Abraham et al. [5] define a class of *admissible* single via node alternative paths. For a given s–t-pair and *via node* v the (via) path P_v is a concatenation of the two shortest paths s–v and v–t. The shortest path between s and t is called P_{opt} and the length of a path P_v is denoted by $l(P_v)$. A via path P_v has to be reasonable to be considered as a viable alternative and thus must obey three heuristic, but natural properties:

First, P_v has to be significantly different from P_{opt}. This states that the total length of the edges both paths share must only be a fraction of the length of the optimal path. Second, P_v has to be *T-locally optimal (T-LO)*, which means that every sufficiently short subpath P' of P_v must be a shortest path. In other words, every local decision along the alternative path must be reasonable. This is formalized by two properties. Every sufficiently short subpath

$P' \subseteq P_v$ with $l(P') \leq T$ has to be a shortest path. If P' is a subpath of P_v and P'' is obtained by removing endpoints of P' then P' must also be a shortest path if $l(P') > T \land l(P'') < T$ holds. Third, the alternative path needs to have limited stretch. A path P_v is said to have $(1+\varepsilon)$ *uniformly bound stretch* (UBS) if every subpath $P' \subseteq P_v$ has stretch of at most $(1+\varepsilon)$. As such, every alternative should only be a fraction longer than a shortest path. Given parameters $0 < \alpha < 1$, $0 \leq \gamma \leq 1$, and $\varepsilon \geq 0$ as well as the above properties, we formalize the description as:

Definition 11 (Admissible path). *A single via node alternative path P_v between s and t is an admissible alternative if it satisfies the following three conditions:*

a) $l(P_{opt} \cap P_v) \leq \gamma \cdot l(P_{opt})$ (limited sharing),

b) P_v is T-locally optimal for $T = \alpha \cdot l(P_{opt})$ (local optimality), and

c) P_v has $(1+\varepsilon)$-UBS (uniformly bounded stretch).

These measures require a quadratic number of shortest path queries to be verified, which is not feasible for a real-time setting. Thus, more practical algorithms are needed that have a more narrow focus on easy computability. There exists a quick 2-approximation *(T-test)* for T-local optimality. Given a via path P_v and a parameter T, let x be the closest node on s–v that is at least T away from both v and s. Likewise, y is the closest node on v–t that is also at least T away from s. A path P_v is said to *pass the T-test* if the portion of P_v between x and y is a shortest path.

Abraham et al. [5] give a practical solution based on a bidirectional Dijkstra (BD), called *X-BDV*, to compute single via paths that are reasonable and good alternatives. The algorithm incorporates ideas from the plateau method. An *Exploration Dijkstra* identifies potential alternative paths: A (forward) shortest path tree is grown from s, and another (backward) tree from t, until all nodes are settled that are not farther than $(1+\varepsilon) \cdot l(P_{opt})$ away from the root of their respective tree. Note that no admissible path can be any longer. Each node v that is settled in both search trees becomes a via node candidate and three measurements are computed in linear time: $l(P_v)$, the length of via path P_v, $\sigma(P_v)$, the amount of sharing of P_v with the optimal route, and $pl(P_v)$, the length of a longest plateau containing v. Note that if $pl(P_v) > T$, the T-test is always successful. These more practical measures are used to sort all candidates in non-decreasing order according to the priority function $f(P_v) = 2 \cdot l(P_v) + \sigma(P_v) - pl(P_v)$. The first path P_v is returned that is approximately admissible for which the following definition holds:

Definition 12 (Approximately Admissible). *A path P_v between s and t is approximately admissible if the following three conditions hold:*

1. $\sigma(P_v) < \gamma \cdot l(P_{opt})$ (limited sharing),

2. successful T-test for $T = \alpha \cdot l(P_v \backslash P_{opt})$ (local optimality), and

3. $l(P_v \backslash P_{opt}) < (1+\varepsilon) \cdot l(P_{opt} \backslash P_v)$ (small stretch).

Local optimality and stretch are defined with respect to the detour of the alternative. The above method yields the algorithm *X-CHV* [5] when combined with Contraction Hierarchies. The forward and backward (CH) search spaces of nodes s and t are explored. Nodes v in the forward search space are reached with a *forward distance* $l^{\uparrow}(P_{sv})$ and nodes in the backward search space with a *backward distance* $l^{\downarrow}(P_{vt})$. For nodes v that occur in both search spaces a preselection is run. Nodes are discarded, if the sum of forward and backward distance is longer than a certain fraction of the length of the shortest path: $l^{\uparrow}(P_{sv}) + l^{\downarrow}((P_{vt}) < (1+\varepsilon) \cdot l(P_{opt})$. Note that these distances are not necessarily correct but upper bounds. It is tested if the *approximated overlap* $\sigma^{apx}(P_v)$ is no longer than a certain fraction of the length of the shortest path: $\sigma^{apx}(P_v) < (1+\varepsilon) \cdot l(P_{opt})$. Additionally, the following condition concerning the stretch must hold:

$$l^{\uparrow}(P_{sv}) + l^{\downarrow}(P_{vt}) - \sigma^{apx}(P_v) < (1+\varepsilon) \cdot (l(P_{opt}) - \sigma^{apx}(P_v)).$$

Remaining candidates are ranked according to the priority function of X-BDV. The exact path $\langle s..v..t \rangle$ is computed for nodes v in that order. The first node for which the properties of Definition 12 hold is selected as via node.

The success rate of X-CHV is inferior to X-BDV since search spaces are much narrower. To cope with the smaller success rate, Abraham et al. [5] introduce a relaxed exploration phase: The exploration query is allowed to search more nodes than the plain CH query. Let $p_i(u)$ be the i-th ancestor of u in the search tree. The *x-relaxed X-CHV query* prunes an edge (u, v) if and only if v precedes all vertices $u, p_1(u), \ldots, p_x(u)$ in the order of the CH. Note, the x-relaxed variant of *X-CHV*, with $x \in \{0, 3\}$, is the baseline of our work. This section ends the recap of previous work.

5.5.2. Engineering the Baseline Algorithm

We now build on top of the baseline algorithm. Recall that it is a two step approach. A bidirectional *Exploration (CH) Dijkstra* searches for via node candidates that are subsequently tested for admissibility using a number of point-to-point shortest path queries, which we call *Target (CH) Dijkstras.* A natural approach to apply engineering is to handle the Target Dijkstras by faster methods than the Contraction Hierarchies query algorithm: We use *CHASE* that computes these queries by exploiting additional arc flags [24]. This does not apply to Exploration Dijkstras, because search spaces would be too narrow. Storing all shortcuts pre-unpacked speeds up path computation as well. Both optimization have equal impact and result in an algorithm with query times of less than half of plain X-CHV. We refer to this straight-forward engineered baseline algorithm by *X-CHASEV.*

The analyses of Abraham et al. [2] show that speedup-techniques to Dijkstra's algorithm work especially well on certain classes of graphs in which all shortest paths out of a region are *covered* by a small node set. This theoretical analysis leads to the following assumption:

Assumption 1 (limited number of alternative paths). *If the number of shortest paths between any two sufficiently far away regions of a road network is small, so is the number of* *plateaus for Choice Routing [40]. Likewise the number of admissible paths of the algorithm of Abraham et al. [5] is small and can be covered by a small number of nodes.*

Single-Level Via Node Candidates We partition the graph and apply bootstrapping to generate *via node candidate sets* for pairs of partition cells. Here, bootstrapping means that the query algorithm which is used later on to actually compute an alternative path is used during preprocessing as well.

Assume that for each pair of non-neighboring cells, we have computed a set of via node candidates. Computing an alternative path for a given s–t-query now becomes straightforward. We loop over all nodes v in the via node candidate set of the pair of cells of s and t. For each v we check whether P_v is approximately admissible using the properties of Definition 12. The first approximately admissible path found is returned as the result. If no candidate is viable or if the size of the candidate set is zero, no alternative path is returned.

In an s–t query between neighboring cells or within a single cell we perform X-CHASEV as fallback instead. The reason for this is that the number of candidates between those pairs of cells and within a single one can be numerous. It is faster to use the fallback algorithm than to check preprocessed node sets in most of these cases.

Preprocessing via node candidates starts with a partition of the underlying road network. A number of such schemes have been proposed before. We do not focus on that sub-problem but refer to [60, 169] instead. A set of via node candidates is generated greedily for each pair of cells by generating alternative paths for all pairwise combinations of their border nodes. For each cell pair, we keep a tentative via node set that keeps track of the candidates identified during preprocessing so far. When an admissible alternative can be found over a node of the tentative set, we continue to the next pair of border nodes. If on the other hand no admissible alternative is found over the nodes of the tentative set, we run X-CHASEV as bootstrapping to identify one. Whenever such a fallback run results in a new via node, it is inserted into the set of tentative via nodes. This continues until all border node combinations have been tried.

Multi-Level Via Node Candidates The above method works well when source and target node are in distinct and non-neighboring cells. Preprocessing candidate sets for source and target nodes within the same cell would inevitably lead to either rather large sets. Therefore, we propose a multi-level partition to compute via node candidates for neighboring pairs of cells as well as within a single cell. The graph is further partitioned into an order of magnitude more cells. The finer partition respects the coarser one in the sense that the nodes of a fine cell belong to one and only one of the coarse cells. We do not run full preprocessing for all pairs of fine cells. This would induce a number of additional preprocessing steps, which is quadratic in the number of cells. We run the same preprocessing algorithm as before but only on the following subset of all fine cell pairs. We run it on the pairs for which origin and destination are too close together to get a small enough via node candidate set, i.e. in the same coarse cell or in neighboring ones. Note, we preprocess each non-neighboring fine cell pair that either belongs to the same or to a pair of neighboring coarse cells. This implies only an amount of additional preprocessing work that is linear in the number of fine cells for reasonable partitions.

Consider that via node candidates have been preprocessed as outlined above. A query recurses to the multi-level partition for nodes of two neighboring coarse cells or when searching a path between nodes of the same coarse cell. When origin and destination are within the same or in neighboring fine cell, plain X-CHASEV is run as fallback. The cells of the fine partition are smaller, and origin and destination are generally close to each other, which leads to fast fallback query times.

We use a partition of 128 cells for the arc flags of X-CHASEV and explain why more cells do not deliver significantly further improved results: The number of explored nodes during a CHASE query that do not belong to the shortest path is negligible as reported by Bauer et al. [24]. Hence, we do not see any benefit of investing time into the generation of arc flags for 1 024 cells.

Further Engineering

The preprocessing is easily adaptable to shared-memory parallelism by preprocessing all cell pairs of a partition independently. This parallelization scales almost linearly with the number of processors until the memory bandwidth is reached. Most preprocessing runs verify the existence of a via node, i.e. reproduce a previously identified via node. Sampling effectively decreases the preprocessing time when the sample is of reasonable size. Preliminary experiments show that running such a sampled preprocessing on 1/16 of all of the pairs of boundary nodes for each cell pair results in only slightly inferior query performance.

Much effort during preprocessing is spent in search space exploration. The search space of each boundary node is required repeatedly. This can be hastened by about a factor of three by storing the search spaces of boundary nodes. Another tuning parameter is the order in which the nodes are stored in the tentative sets. We order by the number of how often a node occurs as a via node during preprocessing. This order is not necessarily the best of all orders. It depends on the order in which the pairs of boundary nodes are visited. Computing a best among all possible sorting orders, independent of the visiting order, is feasible and leads to slightly superior query times, but is computationally expensive. Note that selecting a via node greedily is of course faster since the first viable node is used, while selecting the via node that yields a best quality alternative is more expensive. Queries can be further accelerated by storing (forward and backward) search spaces of the via node candidate sets and also by storing the shortcuts preunpacked, as mentioned before.

5.5.3. Experimental Evaluation

We implement the above algorithms in C++ using GCC's compiler with full optimizations. The experiments are conducted on two separate machines. Queries are processed on a single core of Machine A, running Linux kernel version 2.6.34. Parallel preprocessing is done on Machine C, running Linux kernel version 2.6.38. Machine C and has roughly half the single-core performance of Machine A. Timings are done using the clock cycle counter available in virtually all 64-bit x86-based CPUs.

		performance		path quality					
		time	success	UBS[%]		sharing[%]		locality[%]	
p	algorithm	[ms]	rate[%]	avg	max	avg	max	avg	min
1	X-BDV	11 451.5	94.5	9.4	52.5	42.7	79.9	77.0	26.2
	X-CHV	1.2	75.5	9.2	48.1	44.7	80.0	74.8	26.3
	X-CHASEV	0.5	75.5	9.2	48.1	44.7	80.0	74.8	26.3
2	X-BDV	12225.8	80.6	11.5	43.0	60.0	80.0	78.6	27.0
	X-CHV	1.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0
	X-CHASEV	0.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0
3	X-BDV	13330.9	59.5	13.2	52.9	68.1	80.0	76.2	25.9
	X-CHV	2.3	14.2	10.0	33.4	65.0	79.9	84.3	30.9
	X-CHASEV	1.0	14.2	10.0	33.4	65.0	79.9	84.3	30.9

Table 5.4.: Query Performance of Algorithms for Alternatives $p = 1, 2, 3$.

Our test instance is *ptv-europe* and we partition the graph into 128 cells using the algorithm of Sanders and Schulz [169], yielding an average edge cut of 6 360 and 91.8 boundary nodes per cell. Note that their partitioner does not necessarily yield connected partitions. On average each cell is adjacent to 5.2 neighboring ones. Our finer partition into 1 024 cells has an edge cut of 25 715 with an average of 46.5 boundary nodes and 5.3 neighbors. All figures are based on 10 000 random but fixed queries, unless otherwise stated. To compare against the results of [5], we use the same quality parameter values. Minimum (detour based) local-optimality is set to $\alpha = 0.25$, maximum sharing to $\gamma = 0.8$, and maximum stretch to $\varepsilon = 0.25$. We test the performance of our algorithm in terms of both efficiency and quality according to Definition 11 in the following experiments.

Engineered Baseline Algorithm. We compare the performance of our engineered baseline algorithm, X-CHASEV, against X-BDV and X-CHV. The results of Table 5.4 report on the query performance and path quality of the engineered baseline algorithm. As described in Section 5.5.2 the engineered baseline algorithm is faster by a factor of two than the other algorithms. We reimplemented both X-BDV and X-CHV algorithms. A direct comparison

				candidate sets					
				p=1		p=2		p=3	
		time	size	empty	avg.	empty	avg.	empty	avg.
x	level	[h]	[kiB]	[%]	size	[%]	size	[%]	size
0	single	1.1	859	2.6	4.4	12.7	5.1	30.5	4.4
	multi	1.7	3 669	6.2	6.1	17.4	5.9	36.9	4.2
3	single	2.3	1 742	1.4	6.7	3.0	10.2	10.8	11.5
	multi	4.3	8 909	1.1	12.2	4.9	15.0	11.6	14.2

Table 5.5.: Performance of Unrelaxed ($x = 0$) and 3-relaxed ($x = 3$) Preprocessing.

		performance		path quality					
		time	success	UBS[%]		sharing[%]		locality[%]	
p	algorithm	[ms]	rate [%]	avg	max	avg	max	avg	min
1	X-CHASEV	0.5	75.5	9.2	48.1	44.7	80.0	74.8	26.3
	single	0.1	80.7	9.8	48.1	48.5	80.0	75.8	26.3
	multi	0.1	81.2	9.9	48.1	48.6	80.0	75.8	26.3
2	X-CHASEV	0.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0
	single	0.3	50.8	10.7	40.4	57.1	80.0	80.3	26.3
	multi	0.3	51.2	10.7	40.4	57.0	80.0	80.4	26.3
3	X-CHASEV	1.0	14.2	10.0	33.4	65.0	79.9	84.3	30.9
	single	0.4	24.8	10.7	41.0	59.9	79.9	82.5	27.9
	multi	0.4	25.0	10.7	41.0	59.8	79.9	82.6	27.9

Table 5.6.: Query Performance with Preprocessed Candidate Sets.

against the numbers of Abraham et al. [5] is unfair, since the heuristics of the underlying CH are different. X-BDV has the highest success rate and, of course, the highest query times by several orders of magnitude. This makes X-BDV unsuitable for any practical setting in which speed is a factor. The success rates of all three algorithms drop with the number of alternatives. The average path quality measures are very similar for all algorithms and identical for X-CHV and X-CHASEV by design. This is expected behavior.

Preprocessed Candidate Sets. Table 5.5 reports on the performance of the preprocessing required for the single- and multi-level algorithms. Preprocessing is run in parallel for up to three alternatives with relaxation either off ($x = 0$) or set to $x = 3$. Row *multi-level* denotes the results of adding a finer partition compared to just the single-level approach. Numbers are listed for alternative $p = 1, 2, 3$ and only pertain to candidate sets of non-neighboring,

		candidate sets		
		v.cand.	fallb.	avg.
p	algorithm	[%]	[%]	tested
1	X-CHASEV	-	-	-
	single	92.4	4.9	1.9
	multi	96.5	0.6	2.0
2	X-CHASEV	-	-	-
	single	91.6	2.6	2.8
	multi	93.8	0.3	2.9
3	X-CHASEV	-	-	-
	single	88.7	1.1	3.8
	multi	89.7	0.1	3.8

Table 5.7.: Candidate Set Quality Depending on Number of Alternatives.

	p=1			p=2			p=3		
level	time [ms]	success rate[%]	avg. tested	time [ms]	success rate[%]	avg. tested	time [ms]	success rate[%]	avg. tested
X-BDV	11 451.5	94.5	-	12 225.8	80.6	-	13 330.9	59.5	-
X-CHV	3.4	88.5	-	4.3	64.7	-	5.3	38.0	-
X-CHASEV	2.7	88.5	-	3.2	64.7	-	3.8	38.0	-
single	0.2	90.0	2.22	0.4	70.2	3.8	0.6	44.0	5.6
multi	0.1	90.0	2.3	0.3	70.4	4.0	0.5	44.2	5.8

Table 5.8.: Query Performance of Multiple Algorithms with 3-relaxation.

non-equal pairs of cells. We note that preprocessing can be done on server hardware in a few hours for all of the experiments. The relative speedup on 48 cores is only about 28 due to the memory-bandwidth bottleneck, which is about 60% of the perfect linear speedup. The space overhead is more or less negligible. Even for relaxation with $x = 3$ and multi-level partition the amount of additionally data is less than 9 MiB. Multi-level preprocessing shows a higher average number of candidates per cell pair as only pairs close to each other are processed. Fewer candidate sets remain empty using the relaxed algorithm.

X-CHASEV without candidate sets is compared to single- and multi-level candidate sets. Table 5.6 gives basic performance numbers. Algorithms with preprocessed candidate sets have query times well below 0.5 ms on average even for the third alternative, which is more than practical. We see that the multi-level optimization even improves the success rate, while the path quality remains at high level. Fallback rates to the baseline are generally low, 95% of the queries are covered by preprocessed via node candidates. We tested on omitting the fallback entirely for this setting and observe that results do not degrade noticeably. A third partition level would not give any significant further improvements to the performance of the query. Results of the 3-relaxed variant of the query are given in Table 5.8. Numbers

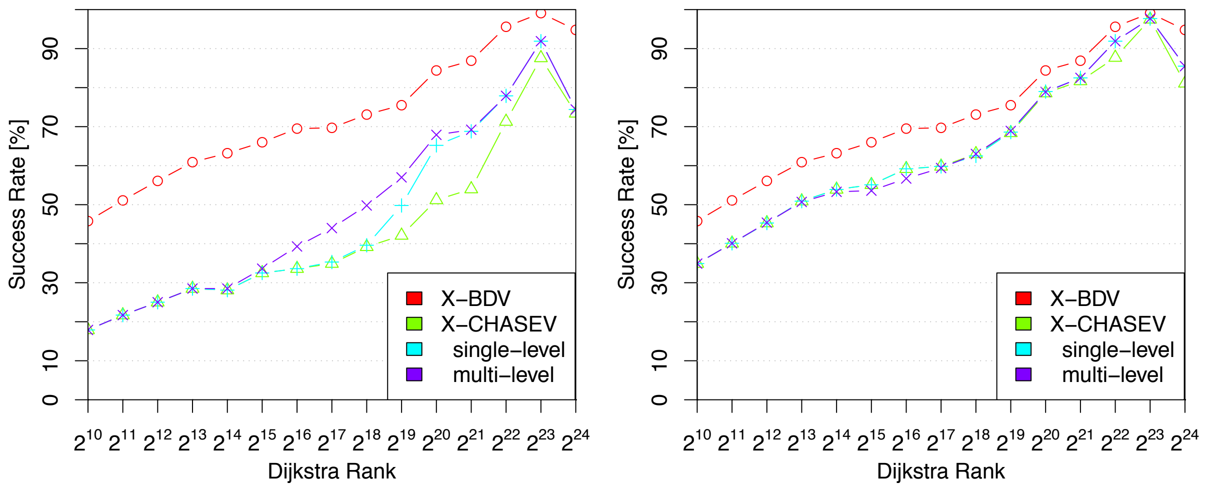


Figure 5.6.: Success Rates Depending on Dijkstra rank: Unrelaxed ($x = 0$, left) and 3-relaxed Algorithm ($x = 3$, right). Each Data Point Represents 1 000 Queries.

for X-BDV and X-CHV are shown for reference. We omit path quality since it is virtually unaffected and remains high. The success rate further improves especially for the second and third alternative. Using precomputed candidate sets is faster by an order of magnitude than X-CHASEV and naturally much faster than the original method. We identify two reasons. A) an expensive (relaxed) Exploration Dijkstra has to be done only in the rare case when a fallback is needed. B) the average number of nodes to be tested as via node candidates is small and always less than half a dozen. Our single- and multi-level approaches deliver consistently higher success rates than the (engineered) baseline with the more speedup the more relaxation is applied.

The Dijkstra rank of node v with respect to a node s is i if v is the i-th node removed from the priority queue of a unidirectional Dijkstra started at s. Figure 5.6 shows success rates with varying Dijkstra ranks to test performance for local and long range queries alike. Success rates (left) are consistently equal or better for our algorithms than for the baseline. With relaxation (right) the numbers get even closer to the rates of X-BDV. The difference is less than 10%. Success rates are compared to X-BDV as the quality “gold standard” even though its computation is prohibitively high.

5.6. Hierarchy Decomposition: Accelerated User Equilibria

The following section is structured as follows. First, we give an introduction into the basic concept of traffic equilibria. Then, we formulate an optimization problem and show how to solve that problem with the help of Contraction Hierarchies. Finally, we conduct an experimental study on the performance of our approach.

5.6.1. Traffic Equilibria

The road traffic of an entire day for a certain region can be understood as a flow with sources and sinks on the road network. Traffic has the tendency to evade regularly clogged roads and other bottlenecks, especially with modern on-board navigation devices that are able to interpret traffic information. Assuming perfect knowledge for all drivers, one might suspect traffic to shape itself in a way such that all used routes between any two points on the road network have equal latency. Although these traffic patterns rarely occur in real life, they are a handy tool to predict the general traffic situation. For small networks, these patterns can be easily computed, but road networks that model entire countries are still a hurdle because Dijkstra’s algorithm does not scale. Thus, the known techniques have only been applied to either small networks or small extracts of a much larger network. We solve this problem for country-sized road networks by combining a gradient descent method for the problem with current research on fast route planning by exploiting the special properties of CH. The computation of the gradient needs a large number of shortest paths computations on the same weighted graph, which means that the expense for preprocessing can be amortized if the number of shortest paths computations is sufficiently large. This leads to a significant overall speedup compared to running Dijkstra’s algorithm for each demand pair. Also, our study shows the robustness of CH on road networks at equilibrium state.

Traffic is often seen as a mere stream of cars. Consider the following picture: On a typical day of work, traffic flows from the suburbs into inner cities in the morning and back from them in the evening. Or on national holidays, a stream of cars and buses flows perhaps to resort towns or recreation areas close to the metropolitan areas. Naturally, some roads are more crowded than others since traffic is not equally distributed over the road network. As a matter of fact, traffic has a natural tendency to shift itself to alternatives if it is more convenient for a driver to take another route. Drivers seek to minimize travel time (or any other metric) and can be understood to act as selfish agents. They switch to better routes if they become aware of it. Assuming all drivers have full knowledge one is interested in how the traffic distributes itself over the road network. This problem is known as the traffic assignment problem and is a major application in the field of transportation planning. Visually speaking, it is the process that finds edge latencies in a road network that are the result of many individuals competing for transportation. We assume travellers to take least cost or (under some metric) shortest paths between their origins and destinations. The problem at hand has been the subject of research since the early 1950s. Wardrop's [186] first principle states the properties for a so-called *user equilibrium state*, which resembles the natural tendency of traffic to take a way of least resistance.

Definition 13 (Wardrop's User Equilibrium). *A set of flows along the edges of a road network is said to be in a user equilibrium state (UE) when the two conditions of the following definition are met.*

1. *If two or more paths between the origin s and the destination t are actually travelled, then the cost of each path between s and t actually used must be the same.*
2. *There does not exist any path between s and t that is of less cost and unused.*

Finding a traffic pattern for which the above conditions hold is called *traffic assignment problem*. Solutions to this problem have a wide range of applications, for example in transportation management or in traveller information systems. Also, the real-time computation of equilibria states can be used as traffic forecasts and for traffic steering. Basic traffic jam avoidance is a feature of nowadays navigation devices.

Unfortunately, this feature is not as well-developed in practice as it is advertised. Consider an example: A traffic jam is reported for a certain highway and drivers are advised to leave their route by switching to an alternate road nearby. When many drivers leave the highway, the road nearby is also clogged. This is not an academic example, but happens every day. Germany's biggest auto-mobile club ADAC[2] reports in a large scale study [183] that most towns close to a highway suffer from increased pass-through traffic because of jam evaders.

Unfortunately, today's jam evading features of navigation devices is limited. ADAC also reports a field study [33] that shows the inferiority of current approaches for traffic jam evasion. Not only the current traffic situation has to be considered to give better guidance around traffic jams, but also how the traffic will evolve. Routing on a road network that is at equilibrium could solve this problem as it is said to be a good estimate of routes that make not only economical sense but are also perceived as good alternatives to a clogged route.

[2]Germany's *and* Europe's largest auto-mobile club: `http://adac.de` – access March, 31st 2013

5.6.2. Problem Formulation

As usual, we model a road network as a graph $G = (V, E)$ where V is a set of nodes and $E \subseteq V \times V$ is a set of directed edges or less formally the set of street segments. Additionally, each edge carries a certain amount of traffic that we call flow. Each edge e is labelled with an edge weight that is the volume of traffic v_e on e, also called *flow*, and an edge cost function f_e. Given $|V|$ nodes in a network, let nodes $1, \ldots, p$, $p \leq |V|$ be a subset of nodes which are either origin or destination of a so-called *demand set*.

We view the nodes of the graph as the places where traffic passes by, enters or leaves the system. We define the set of demands D as a set of triples (i, j, k), where $i, j \in V$ and $k \in \mathbb{N}$. The nodes i and j indicate origin and destination nodes and k the number of units that demand to flow between these nodes. Flow on a certain road segment is said to be the ratio of the current and maximum number of vehicles on that segment at a given average speed.

An Optimization Problem. In [174] it has been shown that the traffic assignment problem can be solved as a minimization problem. The objective function of the underlying optimization problem is based on total edge flows and the resulting edge weights. Assume that their exists a *solution* traffic flow $\mathbf{x} = (x_0, \ldots, x_m)$ that indicates the flow on each edge $e \in E$ under equilibrium conditions, i.e x_e is the flow on edge e. The following mathematical program defines the optimization problem by minimizing the sum of the integrals of the edge weight function under a given flow assignment.

$$\min z(\mathbf{x}) = \sum_{e \in E} \int_0^{x_e} f_e(x_e) \ \mathrm{d}v_e \tag{5.6}$$

with the following constraints that the sum over all observed flows between any two nodes equals the total demand between those nodes. For k paths between any to nodes $, v \in V$ and q many trips between u and v

$$\begin{aligned} \sum_k v_k^{uv} &= q_r s, v_k^{uv} \geq 0 \ , \forall u, v \in V \\ \sum_e v_e &= \sum_k \sum_u \sum_v v_k^{uv} \cdot \delta_{e,k}^{uv} \ , \forall e \in E \end{aligned}$$

holds, whereby $\delta_{e,p}^{uv}$ is an indicator variable that is 1 if edge e is on a path $p = \langle u, \ldots, u \rangle$ from u to v. Note that there is no incentive to make a unilateral switch of paths under such an assignment. As Sheffi [174], pp. 60, notes, “this function does not have any intuitive (...) interpretation. It should be viewed as a strictly mathematical construct”. This minimization problem can be solved by applying the Frank-Wolfe-Algorithm [133, 174]. In each step of the algorithm the approximation of the solution is replaced by a new approximation that is obtained by gradient descent towards the optimum. We now explain this algorithm.

Initialization and Iterative Improvement. The initialization is an *all-or-nothing assignment* of the demand set where each demand is assigned to the edges of the shortest paths using free flow speed on the edges. In other words, travellers choose the routes that would be best if they were the only travellers on the road network. These free flow usages are counted and edge weights are re-evaluated with respect to the flow on the edges and these edge weights are taken as the initial solution X^0. In each iteration, a next (tentative) assignment Y^i is computed and combined with the previous solution to get a better approximation.

More formally, the n-th iteration starts with an update of edge weights by evaluating $f_e(v_e)$ for each edge. Next, an all-or-nothing assignment distributes the so-called auxiliary flow $Y^n = (y_1^n, \dots y_{|E|}^n)$ on the network. The new approximation in the $n+1$-th iteration

$$X^{n+1} = X^n + \alpha^n \cdot (Y^n - X^n)$$

is obtained by computing a scaling factor α^n that minimizes Equation (5.6). Note that computing Y^n is straight-forward one α^n has been identified as X^n is known from the previous iteration. We solve

$$\alpha^n = \min_{0 \le \alpha \le 1} \sum_{e \in E} \int_0^{v_e^n + \alpha(y_e^n - v_e^n)} f_e(v_e)\ \mathrm{d}v_e \tag{5.7}$$

at each iteration. The series of solutions $X^i, i \ge 0$, is known to converge to the solution of the traffic assignment problem. Once we reach an equilibrium state the edge's weight difference between iterations will be zero by definition. We are trying to approximate this situation by searching for a scaling factor α^n that minimizes the sum of edge weight changes between the iterations. Since we know the derivative of Equation 5.7, we can solve that step with a search strategy to find a minimum of the derivative. This is also known as *line search*.

The search for α^n is solved with a certain error threshold by the bisection method of Bolzano. It approximates the zero of a continuous function by binary search. For a given interval $[a, b]$ and $c = (b+a)/2$ we examine if our solution is either in $[a, c]$ or $[c, b]$ and descent recursively until we have reached a certain accuracy. The method is easy to implement. Again, we refer the reader to the textbook of Sheffi [174] for in-depth explanations and for the correctness of the method.

Edge Cost Function. If the travel time between any two nodes was a constant independent of the flow in between then we could solve the problem easily. It would suffice to compute the shortest path for each element of the demand set. Of course, this view neglects reality and the effect that flow, or in other words dense traffic, has to the average speed on a road segment. The denser the traffic gets the more careful drivers have to be not to cause an accident by running into a decelerating car in front. Likewise, the denser the traffic the more cars are affected by one's own driving manoeuvres [178].

To model the situation more realistically, the edge cost function has to be increasing, continuous and non-linear. Several good edge cost functions have been proposed. A simplified function is the Bureau of Public Roads [36] function (BPR). This function was derived from empiric observation and takes road length, speed limit and capacity as parameters. Although it is easy to compute, its curves are not asymptotic to any maximum capacity value, which is in stark contrast to reality. To overcome this shortage Davidson [53] proposed a family of functions that is based on queuing theory. It is defined as

$$t_e = t_e^0 \cdot \left[1 + J \cdot \frac{x_e}{c_e - x_e} \right]$$

where t_e^0 denotes the travel time at zero usage, c_e the capacity of the street segment and x_e the current usage. J is a tuning parameter to control the shape of the curve. Other classes of road functions have been proposed, too, e.g. the class of *conical volume-delay functions* [177]. For an earlier survey on edge cost functions see [32]. But on the other hand the Davidson function models basic relationships between usage and resulting travel times and it is easy to compute. From a computational point of view, any of these edge cost functions work similarly.

Convergence Criterion. Convergence can be based on a number of criteria. Clearly, one would like to stop once the edge weights do not change any more between iterations. The easiest choice is to stop after a fixed number of iterations, but this entirely neglects solution quality. A natural choice would be to use the change of the objective function as convergence test. This might be misleading, since the lengths of individual paths might differ significantly while the sum of the lengths is relatively stable. Therefore, the stopping criterion is based on how much the path length for each demand differs between two iterations.

$$\mathcal{C} := \max_{d \in D} \left| \frac{\mu^n(d) - \mu^{n-1}(d)}{\mu^{n-1}(d)} \right| \tag{5.8}$$

where D is the set of demands and $\mu^n(d)$ is the length (cost) of the path in iteration n for demand d. This stopping criterion indicates the quality of the approximation of the equilibrium much better from a behavioral point of view than a simple sum of all edge weights. Furthermore it ensures that the computation is only stopped once the weights of all edges have settled down. We will look at the impact of choosing the maximum over the average in the experimental evaluation of Section 5.6.4.

5.6.3. Integration into Contraction Hierarchies

To solve the optimization function with the Frank-Wolfe-Algorithm, as explained in Section 5.6.2, it needs a number of shortest path computations on the same weighted graph to assign traffic flow to edges. A naive implementation of the optimization algorithm is technically easy and straight-forward with any algorithm that computes shortest paths, i.e. Dijkstra's algorithm. A more efficient approach is explained below.

Note that a shortest path computed by the bidirectional CH query consists of shortcuts. Although the length of any shortest path is optimal, it has to be unpacked to retrieve the edges of the original graph. Usually, unpacking is done by a recursive method. The edges of the packed path are pushed onto a stack and while the stack is non-empty an edge is popped. If it is a shortcut then the two edges forming that shortcut are pushed onto the stack. Otherwise the popped edge is inserted into the resulting unpacked path. The recursive unpacking runs fast in time linear in the length of the unpacked path. When compared to a plain Dijkstra's algorithm, using a CH black box with path unpacking is still several orders of magnitude faster. We can do better with the following method, which has a running time independent of the size of the demand set. The method is not recursive and relies only on the number of shortcuts in the hierarchy.

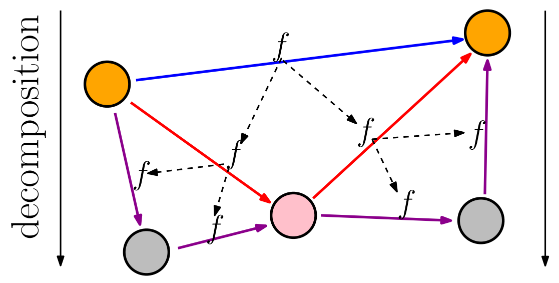


Figure 5.7.: Flow f is distributed through dashed edges to all underlying edges of a shortcut.

If paths are unpacked at each time they get computed then the heavily traversed edges would be touched many, many times to increase usage counters. We modify the CH path computation to do a *hierarchy decomposition* in which each shortcut is unpacked only once in a certain order. At first, we do not unpack the paths at all, but count the volume of flows of on edges of the packed path without unpacking any shortcuts. To do so, each original and shortcut edge is equipped with a counter to record the number of times it is part of a shortest path. This number is incremented during the path computation each time an (shortcut) edge appears in a path. The computational effort to do so is low as each path usually consists of a few shortcuts only. Since we do not need to keep track of the routes actually chosen by travellers, we just count the number of times each individual edge or shortcut is used. After all paths have been computed the hierarchy is decomposed by unpacking all shortcuts and assigning the load to the (shortcut) edges that lie underneath. See Figure 5.7 for an illustration of the process of *hierarchy decomposition*. The only prerequisite to the correctness of this approach is to decompose the shortcuts in a topological sorting order of the CH search graph: A shortcut edge $e = (u, v, w)$ is unpacked if and only if any other shortcut in which it may appear as first or second segment has been unpacked previously. We do so in the opposite order in which the shortcuts were created during the preprocessing of the hierarchy, but note that orderings that are based on DFS, like the one used in PHAST [58], could also be used to propagate the information in parallel.

Another *simple* solution would be to decompose the DAG level by level and to use a FIFO queue that stores the shortcuts to be unpacked in the current iteration. In each iteration, the queue holds the shortcuts discovered by unpacking the previous levels

It is obvious from the order of decomposition (and from the fact that the CH data structure is a directed acyclic graph) that no shortcut needs to be unpacked more than once and no edge usage is lost. The order of shortcut creation is easy to record during preprocessing and takes up a negligible amount of space only. The reverse order of insertion defines the order of the decomposition. Note that the order can also be determined by a topological search.

There exist speedup techniques to Dijkstra's algorithm that support dynamic updates. But they are no feasible options here because the number of edge weight changes is too high. Our resulting algorithm performs very well as we see in Section 5.6.4. The additional space overhead for shortcut order and edge usage counters is rather small.

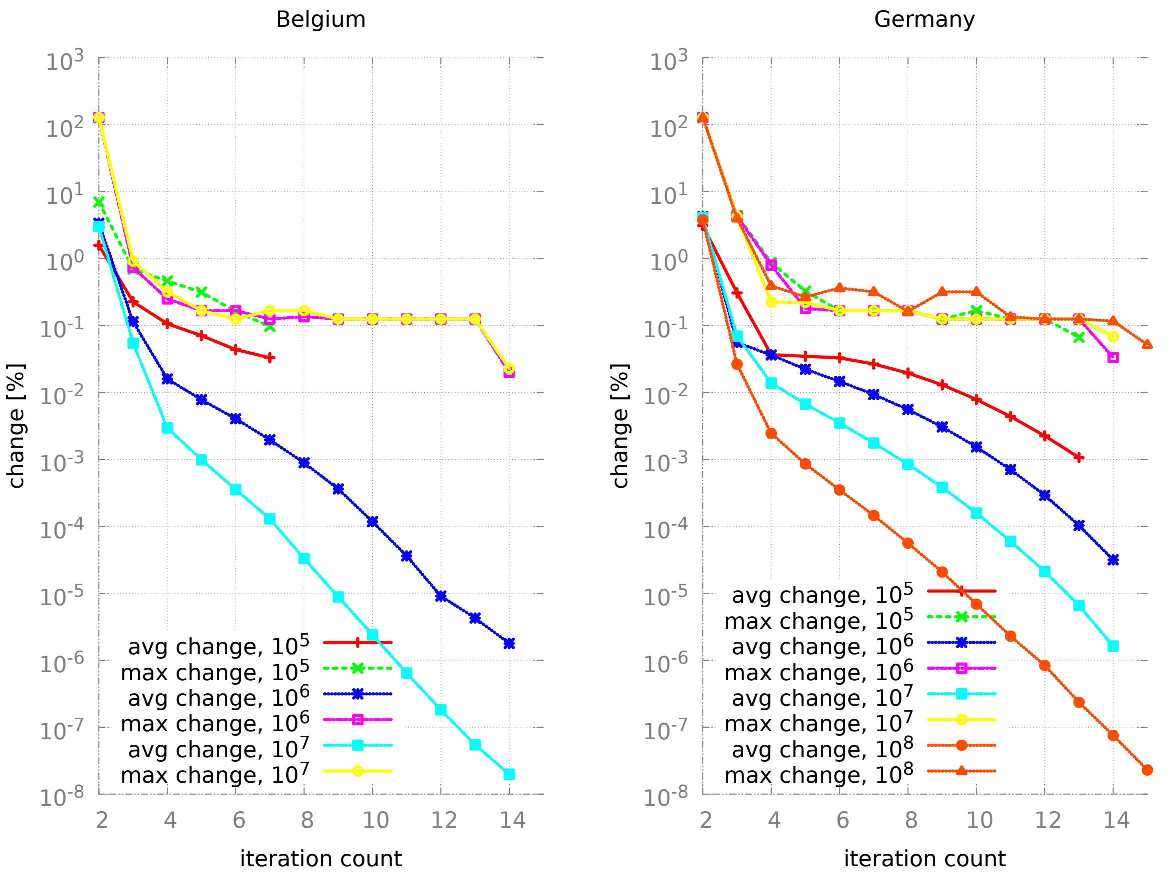


Figure 5.8.: Value of Stopping Criterion Depending on the Number of Iterations for the Belgian (left) and German (right) Road Network.

5.6.4. Experimental Evaluation

We implement our algorithm and data structures in C++ using GCC v4.3.2 compiler and full optimizations. All tests are done on a single core of Machine H, running Linux kernel version 2.6.27. The evaluation was done on test instances *ptv-germany* and *ptv-belgium*. The free flow speeds have been derived from category and length of an edge. Capacity is implied by the category of an edge, which is an oversimplification but unavoidable because of lack of realistic data sets.

We pre-generate randomized lists of origin-destination pairs, also called *the set of demands* or *demands* for short. To the best of our knowledge we are not aware of any high resolution trip generation algorithms coming from transportation science that cover entire countries. A simple trip generation model is presented that generates traffic demand realistic enough to show the validity of the technical approach. During a personal conversation with an ADAC representative we were told that the distances actually travelled are geometrically distributed with an expected distance of 40 kilometres. This number is roughly confirmed by [35] which states that only 2% of all traffic is over distances longer than 100 kilometres. We conjecture

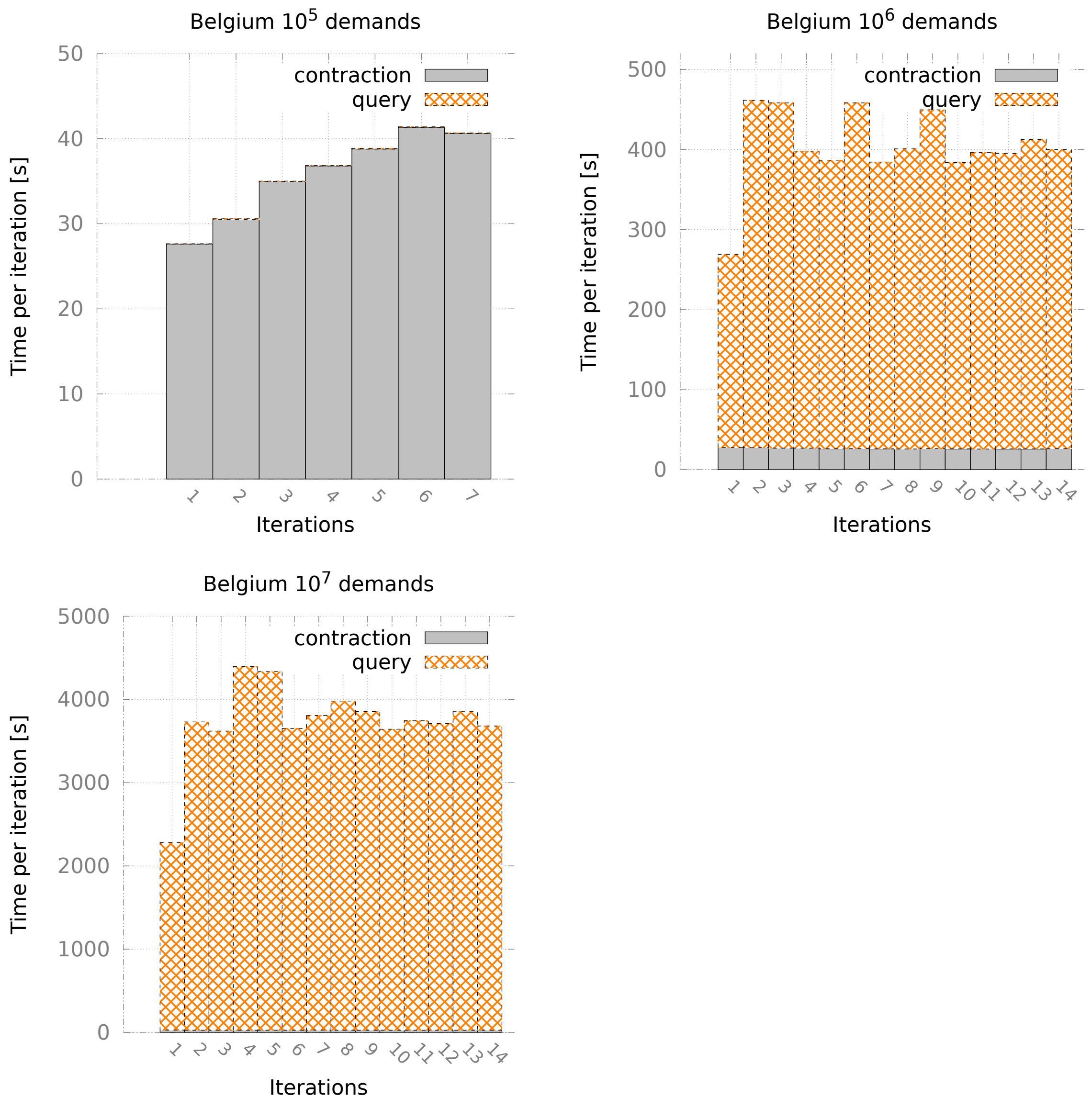


Figure 5.9.: Running times for Belgian network.

that the population density correlates strongly with the density of a road network. Therefore, we choose the starting points uniformly and at random from the set of all nodes. Since we know the travel distance distribution, we draw a geometrically distributed distance. A ball is grown around each starting node s using a unidirectional Dijkstra search. When an edge is relaxed we check the distance its end node has from the source. If the distance of the end node is equal to or more than the travel distance that was drawn before, we accept the node as the target t of s and insert the triple $(s, t, 1)$ into the demand set. Each demand is given equal weight, i.e. it resembles a single unit of flow.

Note we also implemented a simpler iterated all-or-nothing assignment. The method

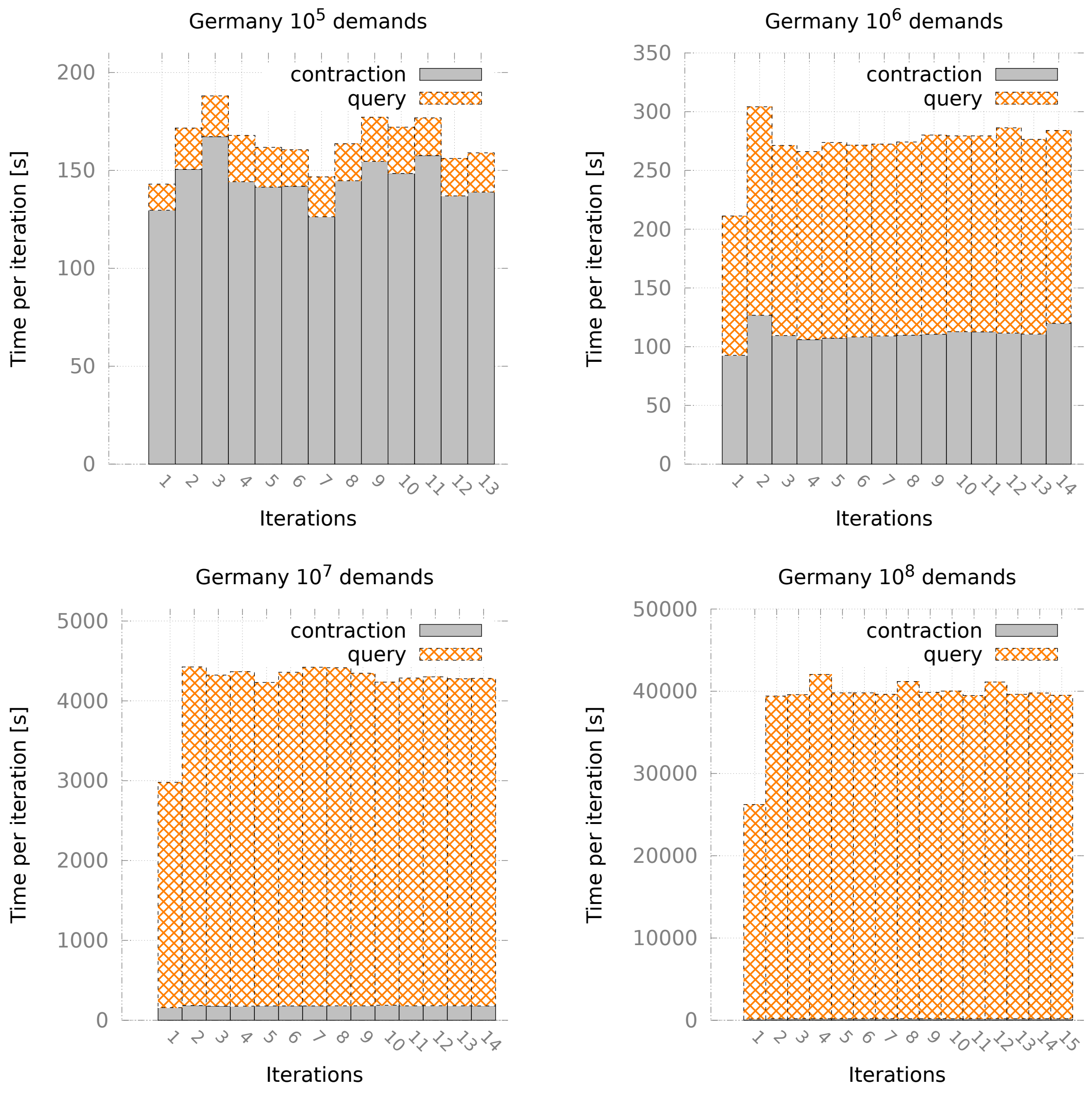


Figure 5.10.: Running times for German network.

starts with a feasible flow on the network. Then edge costs are recalculated for the flow, which is observed on each edge. The flow is reassigned to the changed network and the process is reiterated until a specified number of iterations is completed. We did not observe any convergence with this technique even for large numbers of iterations. In contrast, we observed oscillation of route choice and quickly deemed the approach infeasible. Likewise, an incremental loading where a subset of the demands is assigned proved infeasible as well. Again, convergence did not occur.

We pre-generated lists of $10^5, 10^6$ and 10^7 demands for the networks of Belgium and Germany and in addition also a list of 10^8 demands for the road network of Germany to reflect the larger size of the graph. There are reports [35] of an average of $2.8 \cdot 10^8$ trips taking place daily Germany in 2008, but not necessarily all by car. We computed the user equilibria for all demand sets on the respective graphs. The line search approximation parameter was set to 10^{-10} and the dampening factor J of the Davidson edge cost function to 0.25.

See Figure 5.8 for the results on the development of the stopping criterion depending on the number of iterations for several sizes of demand sets. We naturally determine the value of the stopping criterion in each iteration and check if it meets the threshold. Although, Equation 5.8 only calls for the computation of a maximum, we also compute the average for reference. The stopping value is quickly approached in each of the experiments. We observe that the stopping value of less than 0.001% is approached for each of the demand sets in a similar way while the average error drops significantly with larger demand set sizes. The average value of the stopping criterion approaches threshold values much more quickly than the maximum. This confirms our decision that the average is too optimistic: There are still significant changes in the volume of traffic flow for some of the edges, and the average does not report them. The number of iterations is smaller when using the average as stopping criterion, but the total number for the maximum criterion is within reasonable limits.

The traffic assignment changes the edge weights of the underlying graph. This is a direct consequence of Wardrop's User Equilibrium from Definition 13. Hence under equilibrium state all used routes for a certain origin-destination pair have equal travel times. This *flattens* the natural hierarchy of the road network that is exploited during the contraction phase. Less shortcut edges can be omitted from the search data structure, because for many shortcuts there is now a shortest path that actually lies on it. Thus, the preprocessing should take longer as it is harder to decide if a certain shortcut is needed or not. Likewise, the query times should rise. But surprisingly, this effect is not significant. We observe CH to be robust on road networks at equilibrium state.

The larger the graph and the demand set, the more evident this phenomenon gets. The larger the numbers of queries, the less important preprocessing and the road network size gets, which can be seen in Figures 5.9 and 5.10. We observe that the preprocessing dominates the query phase so strongly on the Belgian test set with 10^5 demands that it is not visible in the plot at all. Likewise, the preprocessing time for the 10^8 case on the German data set is not visible. We conclude that preprocessing times are more than bearable for sufficiently large demand sets. We also conclude that it make sense for large demand sets to switch to a different speedup-technique to Dijkstra's algorithm that may have higher preprocessing times, but better query efficiency. For example, the variant of Transit Node Routing of Section 6 may be such a candidate to speed up queries by an order of magnitude while the preprocessing takes only twice as long. The query time of Transit Node Routing should be even less affected than CH by the flattened hierarchy under equilibrium. For small demand sets, on the other hand, the speedup technique of *Customizable Route Planning* [59] may be interesting. The graphs node set is partitioned once and the edge weights can be updated in near real-time even for continental sized networks. The query is still reasonably fast. Recent improvements [63] further accelerate the time needed for customizations of road networks.

5.7. Concluding Remarks

Points of Interest and Walkability. We showed how to build a POI index based on CH and how to apply it to aggregate scorings based on the road network. The resulting algorithm is fast enough to run as a real-time online service that answers a stream of queries one by one and close to near-interactive speeds for entire metropolitan areas. In addition to that the POI index is independent of the weighting of the scoring function and thus it allows to tailor the scores to more specific user defaults.

A visual comparison by Foti et al. [76] of the distribution of scores on a plotted map of the Bay Area reveals that it is nearly identical to a similar one published on `walkscore.com`, which shows its good applicability, too. In the future we would like to explore a dynamic setting where POIs change over time, e.g. they are added or deleted from the data set or valid for a certain time only. Likewise, we would like to move away from a static scoring function but allow the user to provide a personalized scoring. This reflects the fact that certain POIs are not important for certain users and the other way round.

Ride Sharing. We developed an algorithmic solution to efficiently compute detours to match ride sharing offers and request for single hop ride sharing. This improves the matching rate for the current city-to-city scenario. In the new scenario for arbitrary starting and destination points, our algorithm is the first one feasible in practice, even for large datasets.

We introduced a modeling of multi-hop ride sharing where users travel between a number of *stations* that is related to timetable networks for public transportation. We developed data structures and algorithms to efficiently compute matches in this model that allow a rider to reach a destination via a number of transfers. We evaluate the practicability of the algorithm with respect to the influence of tuning parameters. One interesting result of our algorithm is that increasing the number of transfers does not lead to significantly superior results in practice. Our algorithms are suitable for a web service with potentially a large number of daily users. Our approaches increase the quality of matches compared to present-day services, and thus, increase user satisfaction. This could foster the habit of ride sharing in general beyond the obvious incentive of low costs for traveling.

In the future we would like to explore an augmented multi-modal scenario that does not feature a single mode of transport but multiple ones. For example, we would like to integrate not only car sharing, but car rental as well as public transportation, which is important in metropolitan areas.

Candidate Sets for Alternative Routes. We introduced via node candidate sets. We showed their compact size, their efficient precomputation on large-scale networks and report one order of magnitude faster queries. We also show that success rates are higher than for previous algorithms with negligible memory overhead. As a result of our extensive experimental evaluation, we conclude that our assumption holds that the number of admissible single via node alternative routes between regions is small. There are a number of interesting directions for future work. We would like to explore the amount of preprocessing that is necessary to match the success rates of X-BDV. Also, we would like to explore if there are tradeoffs between preprocessing or query time and success rates. We are interested to see if the via node candidate sets can be reduced without loss of quality by either applying further refinement to the existing techniques or by developing new preprocessing techniques. Further, we want to look into possibilities on how to generate the same candidate sets in reasonable time that using X-BDV during preprocessing would give.

Accelerated User Equilibria. We showed the feasibility of large scale traffic assignment on graphs that cover entire countries. We presented an application of sweeps on the CH search graph as a building block for a large scale optimization problem. Our algorithm exploits the special properties of the search data structure of CH which enables us to solve the problem with better efficiency than pure path computation with unpacking of each computed path. As an interesting side-effect, we observe that CH shows a certain robustness against the weight changes during the iterations. It appears that the hierarchical properties of the road network are not violated.

Traffic steering is an application that may use a road network at equilibrium as input. Likewise, we would like to look into a road network at equilibrium state as input to the generation of alternative routes as well as alternative route graphs. Also, we would like to move beyond a single mode of transport. Related to that is the integration of a multi-modal speedup-technique to Dijkstra's algorithm, which would enable us to incorporate the modal split of everyday traffic into one single black box of path computation.

CHAPTER 6

Transit Node Routing Based on Contraction Hierarchies

6.1. Central Ideas

Routing in road networks has applications that go beyond the computation of single shortest paths. For example, computing user equilibria on a road network requires the batched assignment of an entire set of shortest paths over a number of iterations, as briefly mentioned in Section 5.6. Depending on the number of paths to compute, it makes sense to invest more time into preprocessing the road network if subsequently the query time is lowered. We present such a technique in the following.

Transit Node Routing (TNR) captures the notion that routes will (almost) always enter the arterial network if origin and destination are sufficiently far away. The entrance into the arterial network is made through a set of important nodes – the *transit nodes*. The set of these entrances for a particular node is small on average. Once the transit nodes are identified, which will be explained below, a mapping from each node to its access nodes into the arterial network and pairwise distances between all transit nodes are stored. Preprocessing needs to compute additional information for a further building block, the so-called *locality filter*. The filter indicates whether the shortest path might not cross any transit nodes, requiring an additional local path search. By preprocessing the graph even further than other speedup techniques, it yields *almost* constant-time queries, in the sense that almost all queries can be answered exactly by a small number of table lookups. A small fraction of queries is answered by a fallback algorithm, which is also cheap on average.

References

The contents of this chapter are based on the following publications: It is the result of joint work with Julian Arz [8] and Peter Sanders [9, 10]. Wordings of these publications are used in this thesis.

6.2. Contraction Hierarchies Based Transit Node Routing

TNR in itself is not a complete algorithm, but a framework. A concrete instantiation has to find solutions to the following degrees of freedom: It has to identify a set of transit nodes. It has to find access nodes for all nodes. And is has to deal with the fact that some queries between nearby nodes cannot be answered via the transit nodes set. In the remainder of this section, we define and introduce the minimal ingredients for the generic TNR framework, conceive a concrete instantiation and then discuss an efficient implementation.

Definition 14. *Formally, the generic TNR framework consists of*

1. *A set $\mathcal{T} \subseteq V$ of transit nodes.*
2. *A distance table $D_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \to \mathbb{R}_0^+$ of shortest path distances between the transit nodes.*
3. *A forward (backward) access node mapping $A^{\uparrow} : V \to 2^{\mathcal{T}}$ ($A^{\downarrow} : V \to 2^{\mathcal{T}}$). For any shortest s–t-path P containing transit nodes, $A^{\uparrow}(s)$ $\big(A^{\downarrow}(t)\big)$ must contain the first (last) transit node on P.*
4. *A locality filter $\mathcal{L} : V \times V \to \{true, false\}$. $\mathcal{L}(s,t)$ must be true when no shortest path between s and t is covered by a transit node. False positives are allowed, i.e., $\mathcal{L}(s,t)$ may sometimes be true even when a shortest path contains a transit node.*

Note that we use a simplified version of the generic TNR framework. A more detailed description can be found in Schultes' Ph.D. dissertation [171]. We outline a generalization to multiple layers of transit nodes in Section 6.4. During preprocessing $\mathcal{T}$, $D_{\mathcal{T}}$, $A^{\uparrow}$, $A^{\downarrow}$, and some information sufficient to evaluate $\mathcal{L}$ is precomputed. An s–t-query first checks the locality filter. If $\mathcal{L}$ is true, then some fallback algorithm is used to handle the local query. Otherwise,

$$\mu(s,t) = \mu_{\mathcal{T}}(s,t) := \min_{\substack{a_s \in A^{\uparrow}(s) \\ a_t \in A^{\downarrow}(t)}} \{d_{A^{\uparrow}}(s,a_s) + D_{\mathcal{T}}(a_s,a_t) + d_{A^{\downarrow}}(a_t,t)\}. \tag{6.1}$$

Our TNR variant (CH-TNR) is based on CH and does not require any geometric information. We start by selecting a set of transit nodes. Local queries are implicitly defined and we find a locality filter to classify them. For simplicity, we assume that the graph is strongly connected. In Section 6.4 we discuss what needs to be done to handle the general case.

Selection of Transit Nodes. CH order the nodes in such a way that nodes occurring in many shortest paths are moved to the upper part of the hierarchy. Hence, CH is a natural choice to identify a small node set which covers many shortest paths in the road network. We choose a number of transit nodes $|\mathcal{T}| = k$ and select the highest k nodes from the CH data structure. This choice of $\mathcal{T}$ also allows us to exploit valuable structural properties of CHs. A distance table of pairwise distances is built on this set with a CH-based implementation of the many-to-many algorithm of Knopp et al. [114].

Finding Access Nodes. We only explain how to find forward access nodes from a node s. The computation of backward access nodes works analogously. We show that the following simple and fast procedure works: Run a forward CH query from s. Do not relax edges leaving transit nodes. When the search runs out of nodes to settle, report the settled transit nodes.

Lemma 5. *The transit nodes settled by the above procedure find a superset of the access nodes of s together with their shortest path distance from s.*

Proof. Consider a shortest s–t-path $P := \langle s, \dots, t\rangle$ that is covered by a node $u \in \mathcal{T}$. Furthermore, assume that u is the highest transit node on P. A fundamental property of CHs is that we can assume P to consist of upward edges leading up to u followed by downward edges to t. Moreover, the forward search of a CH query finds the shortest path to u otherwise the path would lead over another transit node and thus u would not be the highest. Thus, a CH query also finds a shortest path to the first transit node v on P. It remains to show that the pruned forward search of CH-TNR preprocessing does not prune the search before settling v. This is the case since pruning only happens when settling transit nodes and we have defined v to be the first transit node on P. □

The resulting superset of access nodes is then reduced using *post-search-stalling* [171]: For all nodes $t_1, t_2 \in A^{\uparrow}(v)$, if $\mu(v, t_1) + D_{\mathcal{T}}(t_1, t_2) \leq \mu(v, t_2)$, discard access node t_2.

Lemma 6. *Post-search-stalling yields a set of access nodes that is minimal in the sense that it only reports nodes that are the first transit node on some shortest path starting on s.*

Proof. Consider a transit node u that is found by our search which is not an access node for s, i.e., there is an access node v on every shortest path from s to t. By Lemma 5, our pruned search found the shortest path to v but did not relax edges out of v. Hence, the only way u can be reported is that it is reported with a distance larger than the shortest path length. Hence, u will be removed by post-search-stalling. □

6.2.1. Omninode Based Locality Filter.

Intuitively, nodes which are *close* to each other in the graph share (a subset of) access nodes. While this is often the case, it is not true in general. The problem is that access nodes are not *projectable*: If a non-transit node z is found during the search space exploration from node u, the access nodes of z do not have to be access nodes of u.

In order to make the node set projectable, we build a super set of the access nodes, so-called *omninodes* $\odot : V \to 2^{\mathcal{T}}$. This super set is the set of *first* transit nodes that are reachable from a node if all edges in the hierarchy were bidirectional. In other words, as if the forward and backward search starting both at node v would settle exactly the same set of nodes. To compute the omninodes, we explore the graph, treating it as undirected: For every node u, run an upward exploration which scans both forward and backward edges and adds settled transit nodes to $\odot(u)$. It does not even matter in which order the search space is traversed, as long as it traverses all nodes of the search space. In preliminary experiments depth-first search (DFS) runs slightly faster than a breadth-first search (BFS). Note that DFS, BFS, and a CH half-search without stalling return in the same sets of omninodes. Definition 15 defines the locality filter $\mathcal{L}_{\odot}$ by set intersection:

Definition 15. $\mathcal{L}_{\odot}(s,t) :\Longleftrightarrow \odot(s) \cap \odot(t) \neq \emptyset$

The above construction returns a projectable node set, which is easy to see: Assume a node z is in the explored space of a node u. Then the space explored if starting at z is a subset of the space of u. This implies that in that case $\odot(z)$ is a subset of $\odot(u)$.

Lemma 7. *$\mathcal{L}_{\odot}(s,t)$ fulfils Definition 14.*

Proof. First we state that for all $u \in V \backslash \mathcal{T}$, $\odot(u)$ is obviously non-empty: There is a path from u to an arbitrary transit node a and the first transit node on that path is an omninode for u. The proof of the Lemma is then done by contra position. Let, for s, $t \in V \backslash \mathcal{T}$, $\mu(s,t) \neq \mu_{\mathcal{T}}(s,t)$, thus $\mu(s,t) < \mu_{\mathcal{T}}(s,t)$. Then the meeting point of a CH-query from s to t, m, cannot be a transit node. As m is found from both s and t with a CH-search, it is also found with a full upward exploration. Hence, $\odot(m) \subset \odot(s)$ and $\odot(m) \subset \odot(t)$, therefore $\odot(m) \subset (\odot(s) \cap \odot(t))$, and in conclusion $\odot(s) \cap \odot(t) \neq \emptyset$. □

We could also generalize the computation of the omninodes by projecting transit nodes similar to the sweeping technique of PHAST [58]. The l levels of the hierarchy are swept through from level l down to level 1. For each node u in the current level, if u is a transit node, we set $\odot(u) = u$. In the other case, we set $\odot(u)$ as the union of the omninode sets of node u. We use a k-way merge for the merging step, resulting in sorted arrays of omninode sets. For real world data sets, this algorithm runs already in only a few seconds, as opposed to several minutes for the naive approach. To conclude, we have now reduced the locality filter to the test whether two sorted sets are disjoint. This test can be accomplished in linear time with a simple sweeping step: Start by comparing the first values of both arrays. If they are not equal, the lower element is compared to the next element in the other array. Proceed until either the end of one array is reached (and return *empty*) or two elements compare equal (and return *not empty*). Unfortunately, this method yields an average omninode set of more than 70 nodes. We show in the following section how to construct a locality filter that requires an order of magnitude less overhead.

6.2.2. Search Space Based Locality Filter.

Consider a shortest path query from s to t. Let $S_{\uparrow}(s)$ denote the sub-transit node search space considered by a CH query from s, i.e., those nodes v settled by the forward search from s which are not transit nodes. Analogously, let $S_{\downarrow}(t)$ denote the sub-transit-node CH search space backwards from t. If these two node sets are disjoint, all shortest up-down-paths from s to t must meet in the transit node set and hence, we can safely set $\mathcal{L}(s,t) =$ false. Conversely, if the intersection is non-empty, there might be a meeting node below the transit nodes corresponding to a shortest path not covered by a transit node. Thus, a very simple locality filter can be implemented by storing the sub-transit node search spaces which are computed for finding the access nodes anyway, and then computing $\mathcal{L}(s,t) = S_{\uparrow}(s) \cap S_{\downarrow}(t) \neq \emptyset$.

Lemma 8. *The locality filter described above fulfils Definition 14.*

Proof. We assume for $s, t \in V \backslash \mathcal{T}$, the distance $\mu(s,t) \neq \mu_{\mathcal{T}}(s,t)$ and thus $\mu(s,t) < \mu_{\mathcal{T}}(s,t)$. Then, the meeting node m of a CH-query is not a transit node, and it has to be in the forward search space for s, $\S_{\uparrow}(s)$ *and* in the backward search space for t, $S_{\downarrow}(t)$. Hence, $S_{\uparrow}(s) \cap S_{\downarrow}(t) \neq \emptyset$. □

The average size of these search spaces are much smaller than the full search spaces, e.g., 32 instead of 112 in the main test instance from Section 6.2.3. For the locality filter, only node IDs need to be stored. Compared to (uncompressed) hub labelling which has to store full search spaces and also distances to nodes, this already saves an significant amount of space.

If we are careful to number the nodes in such a way that nearby nodes usually have nearby numbers, the node numbers appearing in a search space will often come from a small range. We precompute and store these values in order to facilitate the following *interval check*: When $[\min(\bar{S}_{\uparrow}(s)), \max(S_{\uparrow}(s))] \cap [\min(S_{\downarrow}(t)), \max(S_{\downarrow}(t))] = \emptyset$, we immediately know that the search spaces are disjoint. As the sole locality filter, this range compression would yield too many false positives, but it works sufficiently often to drastically reduce the average overhead for the locality filter. In the following we discuss a much more accurate lossy compression of the search space bases locality filter that results in a remarkably efficient filter that is also easy to implement.

Graph Voronoi Label Compression. Note that the locality filter remains correct when we add nodes to the search spaces. We do this by partitioning the graph into regions and define the extended search space as the union of all regions that contain a search space node. This helps compression since we can represent a cell using a single ID, e.g., the ID of a node representing the region. This also speeds up the locality filter since instead of intersecting the search spaces explicitly, it now suffices to intersect the (hopefully smaller) sets of cell ids. Hence, we want partitions that are large enough to lead to significant compression, yet small and compact enough to keep the false positive rate small. Our solution is a purely graph theoretical adaptation of a geometric concept. Our cells are *graph Voronoi regions* of the transit nodes. Formally,

$$\mathrm{Vor}(v) := \{u \in V : \forall w \in \mathcal{T} \setminus \{v\} : \mu(u,v) \leq \mu(u,w)\}$$

for $v \in \mathcal{T}$ with ties broken arbitrarily. The intuition behind this is that a positive result of the locality filter means that the search spaces of start and destination are somewhat *close* to each other. Computing the Voronoi regions is easy, using a single Dijkstra run with multiple sources on the reversed input graph, as shown by Mehlhorn [136]. We call this filter the *graph Voronoi filter*.

6.2.3. Experimental Evaluation

We implement our algorithms and data structures in C++ using GCC's compiler version 4.6.1 with full optimizations. The tests are done on Machine A, running Linux kernel version 2.6.34 The test instance is *ptv-germany*.

Our CH variant implements the shared-memory parallel preprocessing algorithm of Vetter [184] with a hop limit of 5 or 1 000 settled nodes for witness searches and 7 hops or 2 000 settled nodes during the actual contraction of nodes. The priority function is

$$2 * \text{edgeQuotient} + 4 * \text{originalEdgeQuotient} + \text{nodeDepth}.$$

Results for further instances can be found in Section 6.2.6. The following experiments are conducted with a transit node set of size 10 000, if not mentioned otherwise, because key results from previous work were based on the same number of transit nodes, e.g. [17].

The following design choices are used in our experiments. Forward and backward search spaces are merged into one set for the locality filter. Forward and backward access node sets are also merged into one set. Note that these two sets are distinct in our implementation.

As the ID of a node does not contain any particular information, node IDs can be changed to gain algorithmic advantages. This *renumbering* is done by applying a bijective permutation on the IDs, in order to ensure that each ID stays unique. We alter the labels of the nodes in V so that $\mathcal{T} = \{0, \ldots, k-1\}$. By proceeding this way, we can easily determine if a node v is a transit node or not (during further preprocessing and during the query): $v \in \mathcal{T}$ if and only if $v < k$.

Node Renumbering. We examine renumbering strategies separately for the transit nodes set $\mathcal{T}$ and for the remaining part of the CH search graph $V \backslash \mathcal{T}$. We follow two aims for renumbering here. One is to make table lookups faster for non-local queries, while the other aim is to make local queries as fast as possible. Therefore, we treat both parts of our data structure with different strategies. If the access node IDs of a node are from a more compact interval, the cache efficiency of the table lookups is increased. Consider a number of access nodes $|A^{\uparrow}(s)| = a$ and $|A^{\uparrow}(t)| = b$ for source and target nodes respectively. The obvious worst case is $a \cdot b$ cache misses, while the best case is $min(a, b)$ misses only. This happens when all $max(a, b)$ entries are in one cache line.

The lower, non-transit node portion of the CH search graph is renumbered. It is used for local queries only and thus it has no effect on non-local queries. It also does not influence the preprocessing in our experiments and we attribute that to the fact that search spaces are similar when nodes are close to each other and the fact that input ordering already exhibits a *good* locality. Table 6.1 gives results on different renumbering strategies that we

		Query	
Preprocessing Strategy	Duration [s]	Local Search [μs]	Total [μs]
(greedy) DFS Increasing	16.9	27.4	1.38
(greedy) DFS Decreasing	16.9	32.2	1.41
Input Level Ordering	8.9	38.4	1.45

Table 6.1.: Different Renumbering Strategies for the Remainder of the Graph $V \setminus \mathcal{T}$. The Transit Node Set $\mathcal{T}$ is in Input Order.

	Preprocessing				Query		
	Exploration		Voronoi	Total	LS	LS	Total
Hops	[s]	$\lvert S\rvert$	$\lvert S\rvert$	[byte / node]	[%]	[μs]	[μs]
0	301	93.0	29.3	296	2.36	30.4	2.15
1	**149**	31.8	8.0	211	0.58	27.4	1.38
2	446	28.1	6.3	204	0.41	26.0	1.35
3	3237	**27.9**	**6.1**	**203**	**0.40**	**25.7**	**1.32**

Table 6.2.: Preprocessing Effiency Depending on Different Hop Depths for Stall-on-Demand.

detail in the following. The *(greedy) DFS* orderings renumber the graph according to a (modified) depth-first graph traversal (DFS), while the input level ordering preserves the partial ordering of the levels as described above for the transit node set. For every node an upward DFS is conducted that relaxes edges in the CH search graph leading to more important nodes. More specifically nodes in $V \setminus \mathcal{T}$, which are not yet renumbered, are explored. The actual renumbering happens during the backtracking step, i.e. we renumber a node if and only if all of its successors are already renumbered. The actual IDs can be assigned in increasing $(k, \dots, n)$ or decreasing order $(n, \dots, k)$. Column *Duration* gives the duration of the renumbering while *Local Search* gives the average running time of a local search. Column *Total* gives the average running time over all TNR queries. We see from the results that the DFS strategy with increasing IDs show best efficiency of local queries.

During the search for the access nodes, stalling of nodes is used to decrease the search space sizes. We tested different variants, varying in the number of hops the stalling does look ahead to find a witness for a wrong distance. A higher number of hops on the one hand increases preprocessing time, but on the other hand decreases the search spaces, speeding up the locality filter construction. Table 6.2 shows that while an increase of the hop depths from 1 to 2 manages to decrease space overhead and query times, a further increase from 2 to 3 is inadvisable: Preprocessing is about 43 minutes longer, and yields only have better results. Local searches take less time with higher hop depths. This is an interesting observation because the stalling during preprocessing should not affect local searches. An explanation is that the omitted local searches (due to a more exact locality filter) have a higher distance.

$\mathcal{T}$ is renumbered with the so-called *input-level strategy*, while $V \setminus \mathcal{T}$ is ordered by the (greedy) DFS Increasing strategy. The interval check accelerates the average running time of the locality filter. Prior to running the sweeping step, we check in constant time if the two intervals overlap or not with the interval check.

Scalability. We test the scalability of parallel preprocessing for a varying number of cores in Table 6.3. The raw results of parallelizable parts (preprocessing, distance table generation and exploration) show a quite high variance of about 10%. Hence, we measured the preprocessing five times and averaged over all runs. The values reported in column *Total* are the sum of the respective averages. Column *Cores* gives the numbers of cores used. Columns *CH*, *Dist. Table*, *Exploration* measure time, speedup and efficiency of the respective parts. The bottom line reports on four CPUs with activated hyper-threading (HT).

	CH			Distance Table			Exploration			Total		
Cores	[s]	Spdp	Eff.	[s]	Spdp	Eff.	[s]	Spdp	Eff.	[s]	Spdp.	Eff.
1	513	1	1	9.0	1	1	500	1	1	1046	1	1
2	281	1.83	0.91	5.1	1.74	0.88	287	1.74	0.87	596	1.75	0.88
3	203	2.53	0.84	3.9	2.23	0.76	202	2.48	0.83	432	2.42	0.81
4	160	3.20	0.80	2.9	3.16	0.79	145	3.43	0.86	334	3.13	0.78
4 (HT)	137	3.75	0.47	2.2	4.01	0.50	101	4.93	0.62	265	3.95	0.49

Table 6.3.: Scalability of Preprocessing Depending on the Number of Cores. The Transit Nodes Set has Size 10 000 in Each Experiment.

We see that the total preprocessing time is only about a factor of two larger than plain CH preprocessing. Most additional work is due to search space exploration from each node. We see that the different parts of the algorithm scale well with an increasing number of cores. The total efficiency is slightly lower than the efficiency of the individual parts, as it includes about 23.6 seconds of non-parallelized work due to the Voronoi computation. On average a non-local query takes 1.22 μs, while a local query takes 28.6 μs on average. This results in an overall average query time of 1.38 μs and the space overhead amounts to 147 Bytes per node. The rate of local queries is only 0.58 %. It does not reflect the performance of real cores, but HT comes virtually for free with modern commodity processors.

Comparison Against the Literature. We compare to previous approaches for our test instance. Some of these implementations were tested on an older AMD CPU [171], i.e. Machine F, that was still available for running the queries[1]. Table 6.4 shows *Original* values as given in the respective publications denoted by *From*, while columns *Compared* give preprocessing and running times either done on or normalized to the aforementioned AMD

[1]Note that a current off-the-shelf commodity machine is about 2–3 times faster.

		Preprocessing			Query		
		Reported	Compared	Overhead	Original	Scaled	
Method	From	[hh:mm]	[hh:mm]	[byte]	[μs]	[μs]	
CH	-	00:03	00:05	24	103	246	
Grid-TNR	[17]	≈20:00	≈20:00	21	63	63	
HH-TNR-eco	[171]	00:25	00:25	120	11	11	
HH-TNR-gen	[171]	01:15	01:15	247	4.30	4.30	
TNR+AF	[24]	03:49	03:49	321	1.90	1.90	
HL-0 local	[4]	00:03	00:35	1341	0.70	1.34	⋆
HL-∞ global	[4]	06:12	≈120:00	1055	0.25	0.49	⋆
HLC	[61]	00:30	00:59	100	2.99	5.74	⋆
CH-TNR	-	00:05	00:34	147	1.38	3.27	

Table 6.4.: Comparison of Various Distance Oracles to Our Algorithm.

Method	From	$\lvert\mathcal{T}\rvert$	Local [%]	False [%]
Grid-TNR	[17]	7 426	2.6	-
Grid-TNR	[17]	24 899	0.8	-
LB-TNR	[72]	27 843	-	-
HH-TNR-eco	[171]	8 964	0.54	81.2
HH-TNR-gen	[171]	11 293	0.26	80.7
CH-TNR	-	10 000	0.58	73.6
CH-TNR	-	24 000	0.17	72.1
CH-TNR	-	28 000	0.14	72.1

Table 6.5.: Comparison of Locality Filter Quality Against Several TNR Variants from the Literature.

machine using one core. Therefore, similar to the methodology in [24], a scaling factor of 1.915 is determined by measuring preprocessing and query times on both machines using a smaller graph (of Germany). Scaled numbers are indicated by a star symbol: $\star$. Values for CH were measured with our implementation.

The simplest TNR implementation is GRID-TNR that splits the input graph into grid cells and computes a distance table between the cells border nodes. Note that the numbers for GRID-TNR were computed on a graph of the USA, but the characteristics should be similar to our test instance. Preprocessing is prohibitively expensive while the query is about 20 times slower than ours. The low space consumption is due to the fact that it is trivial to construct a locality filter for grid cells. For HH-TNR [171] and TNR+AF [24], preprocessing is single-threaded. The corresponding scaling factor for preprocessing is 3.551 and the fastest HH based TNR variant is still slower by about a factor of two for preprocessing and queries. Note that the HH-based methods all implement a highly tuned TNR variant with multiple levels that is much more complex than our method. While TNR+AF has faster queries by about 25%, the (scaled) preprocessing is about an order of magnitude slower and the space overhead is twice as much. Also, TNR+AF requires a sophisticated implementation with a partitioning step and the computation of arc flags.

While the hub labeling based methods achieve superior query times, the reader should note the high space overhead incurred by these methods. Even the most space efficient HL needs more than seven times more space. HL-0 local reports faster preprocessing than our method with nine times higher space overhead. It should be noted that these experiments were done on three times as many cores with 20% faster clock speed of 3.2 GHz and 50% larger L3 cache of 16 MiB. Single core preprocessing for HL-O local takes 17.9 minutes while our approach is slightly faster with 17.4 minutes on a slower machine. We observe that even HL with the fastest preprocessing has faster queries than ours by about a factor of 2–3. On the other hand, only the space requirement of HLC has relevance in practice but it has slower queries than ours on average.

The quality of our locality filter is compared to other TNR implementations in Table 6.5. These variants differ in the number of transit nodes and in the graph used to determine them. Nevertheless, the graphs are all road networks that exhibit similar characteristics. The number of transit nodes for CH-TNR is chosen to resemble data from the literature. We see that the fraction of local queries of our variant is lower than or on par with the numbers from the literature. Also, the rate of false positives is much lower than previous work. Most noteworthy, the recent method of LB-TNR applies sophisticated optimization techniques, but does not produce a transit node set with superior locality as the rate of correctly computed global queries matches our locality filter.

6.2.4. Impact of $|\mathcal{T}|$ on Query Efficiency

Figure 6.1 gives a row-stacked plot that details the contributions of each part of the query to the average total query time depending on the transit node set size. We see that most of the query time is spent in table lookups and that this portion stays relatively stable over the entire parameter space. We attribute that to two reasons. First, the number of access nodes is relatively stable. It drops from roughly 8.5 to just below 7. Hence, the number of table lookups is also relatively stable. Second, the table lookups are mostly dominated by cache misses and it appears that our renumbering effort is successful in that it already minimizes the number of cache misses across the board.

In the following, preprocessing running times are reported for 4 threads. As reported before, the size of the transit node set is a tuning parameter. We look into the impact of varying the size of this set in the following experiments. Especially, we explore the effect of transit node set size on the fraction of local queries, space overhead and query time.

Table 6.6 reports on these experiments. Column $|\mathcal{T}|$ gives the size of the transit node set, while column *Prepoc.* reports on the duration of the preprocessing. Columns *(Non-)Local* give the fraction of (non-)local queries and the respective query times. Column *Amortized* reports amortized query times. The results of the experiment support the results from the previous section that the number of local queries decreases with an increasing transit node

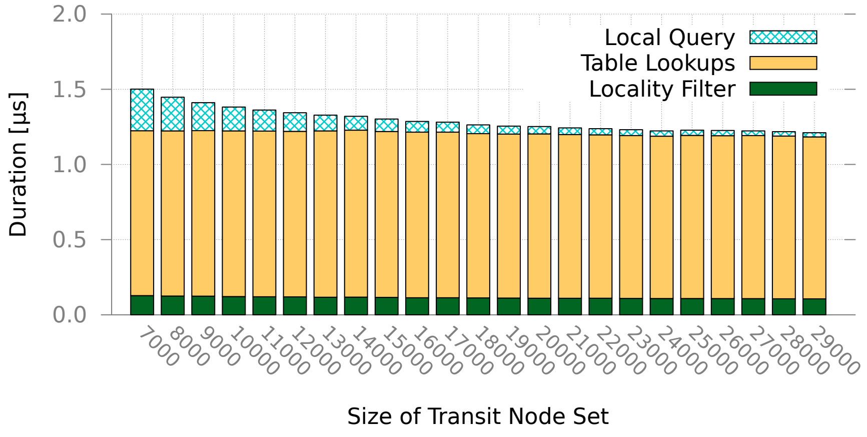


Figure 6.1.: Average Query Duration Depending on the Number of Transit Nodes. Time for Local Queries is Averaged over the Total Number of Queries.

	Preproc.	Non-Local		Local		Amortized
$\|\mathcal{T}\|$	[min]	[%]	[μs]	[%]	[μs]	[μs]
7000	6.1	99.12	1.225	0.88	32.661	1.501
14000	5.5	99.62	1.229	0.38	25.187	1.320
21000	5.2	99.79	1.198	0.21	21.977	1.243
28000	5.1	99.86	1.189	0.14	21.728	1.218

Table 6.6.: Average Query Efficiency, Preprocessing Time, and Space Overhead Depending on the Number of Transit Nodes using 4 cores. 100 000 Queries in Total.

set. Furthermore, we see a decrease in total preprocessing time. There is a trade-off between transit node set size and duration of search space exploration. Likewise, the amortized query time decreases as the number of local queries drops. This is also reflected in the absolute numbers of Table 6.7 in which 100 000 random queries are performed.

The decrease in local queries is not uniform across all Dijkstra ranks. There appears to be a threshold after which the fraction of performed local queries falls sharply. Table 6.8 reports on the fraction of local queries that are performed depending on the Dijkstra rank. Increasing the transit node set size effectively lowers the rank at which the locality filter detects roughly half of the queries as local queries. These values are given in **bold**. The threshold rank decreases by several orders of magnitude over the parameter space.

A closer look at the query performance according to Dijkstra rank is given in Figure 6.2. Note the log scale. We see that the query time is dominated by the rather expensive fall-back algorithm for short-range queries in all the experiments. Also, we see that the query time falls sharply for medium to long range queries once the shortest paths get covered by the transit node set. Beyond this threshold the time approaches the bare minimum needed for running the locality filter and the table lookups, which is constant in practice. Additionally, we see that increasing the transit node set size has an effect on the threshold when queries are not local any more. This is also supported by the results of Table 6.8. It is a tuning parameter that allows us to select a trade-off between transit node set size and query time.

	Local Searches		
		Time	Amortized
$\|\mathcal{T}\|$	#Performed	[μs]	[μs]
7 000	8 798	31.4	0.277
14 000	3 820	24.0	0.092
21 000	2 128	20.8	0.044
28 000	1 441	20.5	0.029

Table 6.7.: Rate of Local Queries and Corresponding Amortized Query Duration Depending on the Transit Node Set Size. 100 000 Queries in Total.

$\lvert\mathcal{T}\rvert$	$\leq 2^9$	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	$\geq 2^{21}$
7 000	100	100	99	98	96	88	74	**56**	34	15	5	1	0
14 000	100	99	98	95	86	68	**47**	28	12	4	1	0	0
21 000	100	99	96	89	73	**51**	29	14	5	1	0	0	0
28 000	100	97	93	81	**61**	38	19	8	2	1	0	0	0

Table 6.8.: Rate of Local Queries Depending on Dijkstra Rank. Bold Values Indicate the Approximate 50% Threshold.

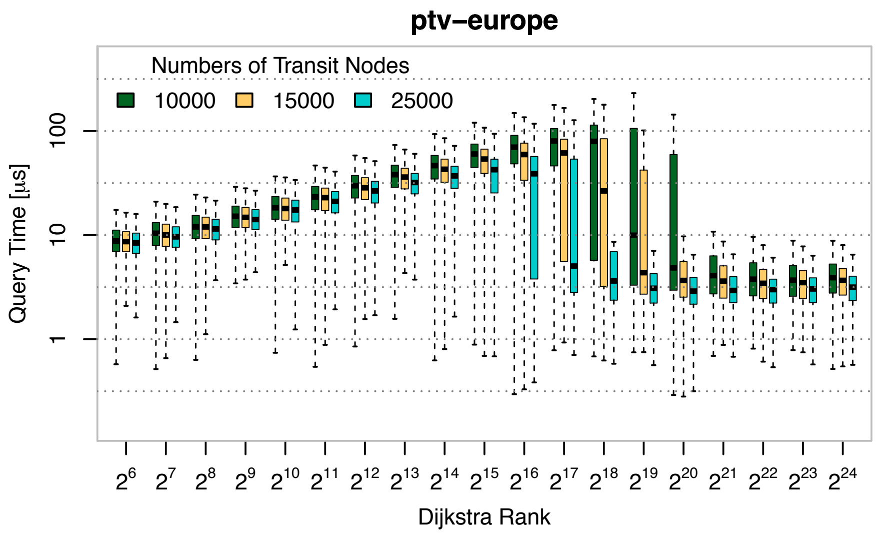


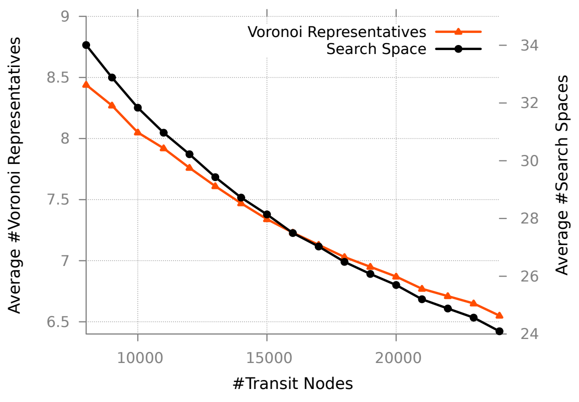
Figure 6.2.: Rank Plot Depending on Size of Transit Node Set for *ptv-europe*.

6.2.5. Impact of $\lvert\mathcal{T}\rvert$ on Space Overhead

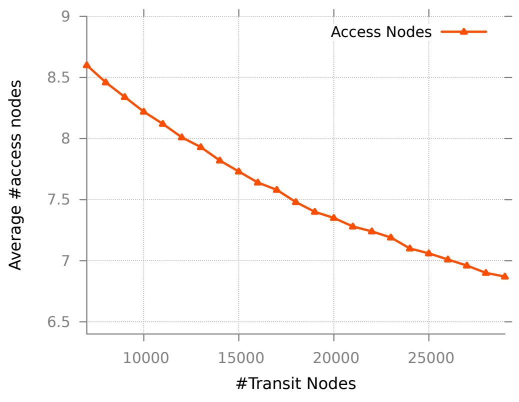
The effect of transit node set size on space requirements is examined next. Besides the underlying contraction hierarchy, the distance table, and access nodes, as well as the locality filter contribute to the space consumption.

The average numbers of nodes in the search space, Voronoi representatives and access nodes per node are plotted against varying sizes of $\mathcal{T}$. For all values, the average of the respective forward and backward sizes is given since the values are virtually identical. We observe that these numbers fall – as expected – the larger the transit node set gets. The results are plotted in Figure 6.3. Obviously, the raw size of the CH is independent of $\lvert\mathcal{T}\rvert$ while the distance table grows quadratically. Space for access nodes slowly *decreases* with $\lvert\mathcal{T}\rvert$ since the average number of access nodes decreases with smaller local search spaces. The same applies to the Voronoi locality filter – it needs less space and gets more effective at the same time.

Figure 6.4 shows the relation between memory requirements and transit node set size. Note that the implementation in this experiment does not merge (Voronoi) search spaces or access node sets to give a clearer picture of the memory consumption of each part of our



(a) Average Size of the Search Space and Corresponding Number of Voronoi Representatives.



(b) Average Number of Access Nodes.

Figure 6.3.: Depending on the Size of the Transit Node Set: Average Size of Search Space and Voronoi Filter (left), and Average Number of Access Nodes (right).

method. We see that the main driver here is the size of the distance table which depends quadratically on the size of the transit node set. Although, the average access node set decreases with an increasing transit node set size, it is not enough to compensate for the distance table. We note that the space requirement of the search spaces is more or less constant over the entire parameter space.

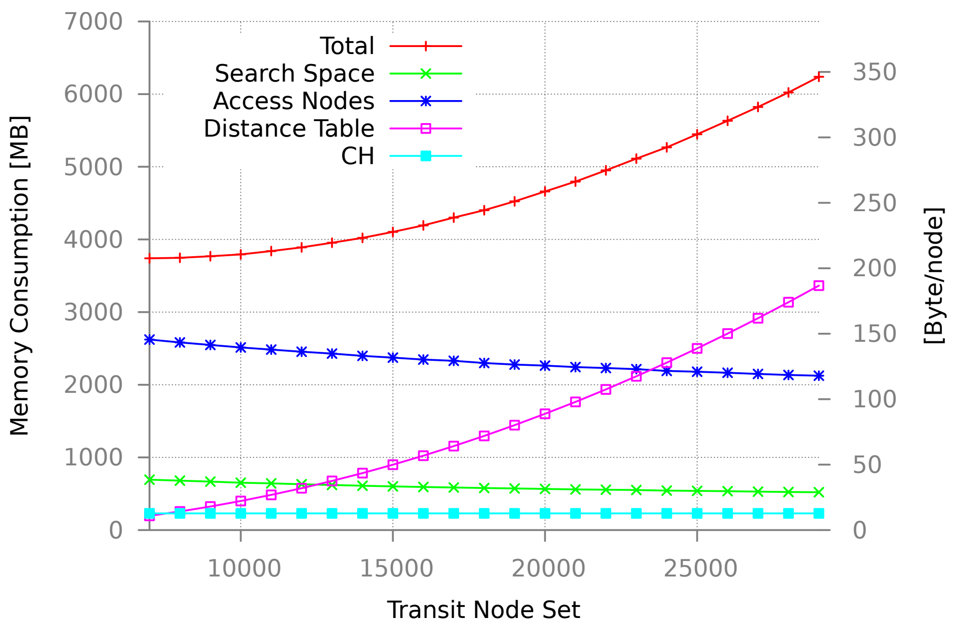


Figure 6.4.: Memory Consumption Depending on $|\mathcal{T}|$ Without Voronoi Compression.

6.2.6. Results for Further Instances

Further experiments are done with two additional instances with different characteristics and which are difficult for CH, *ptv-euro-dist* and *osm-germany-4*. The first one has the same topology as *ptv-europe* with distance metric edge-weights. The second one explicitly models turn restrictions from the data and uses travel-time metric. The experiments on *osm-germany-4* were performed on the slightly slower Machine I because it has more RAM. We determine scaling factors to compare the outcomes by running instance *ptv-europe* on that machine. The factors are 0.804 and 0.960 for preprocessing and query, respectively. The results of our experiments are scaled accordingly to the speed of the faster machine.

We see that the average number of access nodes as well as the average number of Voronoi representatives decrease with an increasing size of the transit node set. Although, it takes longer to preprocess the distance table, the impacted is minor when compared to CH graph preprocessing. Most nodes during a CH search are relaxed in the most upper portions of the hierarchy. The decrease in preprocessing duration is caused by likewise decreased search spaces in the sub-transit node portion of the hierarchy.

The same holds true for the query. The larger the transit node set, the faster the queries become. This is expected behavior because of two reasons. First, search spaces below the transit node set become smaller as argued above. Second, the number of local fallback queries decreases because even more of the queries can be answered by table lookups. Unfortunately, the distance table of the transit node set grows quadratically with the number of nodes. Therefore, the overall space consumption increases again at some point when the quadratic increase of the distance table can not be compensated by the falling average number of access nodes and Voronoi representatives. We attribute this behavior to the fact that most edge relaxations during CH query happen in the highest portions of the hierarchy. So, there is a point of diminishing returns when the transit node set covers this dense portion of the hierarchy. Note that the results shown in Table 6.9 are selected to reflect this observation.

Instance *osm-germany-4* is not strongly connected, it may be that we end up with an empty set of forward or backward access nodes for some nodes. In that case the minimum in Equation 6.1 minimizes over an empty set. We define this minimum as ∞ correctly indicating non-reachability in case of a non-local query. Similarly, non-existing paths between pairs of

Graph	$\lvert\mathcal{T}\rvert$	$\lvert A\rvert$	$\lvert S\rvert$	Byte / node	time [min]	query [μs]
ptv-euro-dist	10 000	18.0	22.1	440	32.4	6.717
	15 000	17.1	18.7	424	28.9	4.678
	25 000	14.5	14.7	456	26.1	3.317
osm-germany-4	20 000	9.11	14.66	278	18.9	2.669
	40 000	7.78	11.85	383	16.9	2.518
	50 000	7.37	11.01	476	16.6	2.678

Table 6.9.: Experiments on Further Graphs. Timings of the Second Graph are Scaled to Match the Same Hardware.

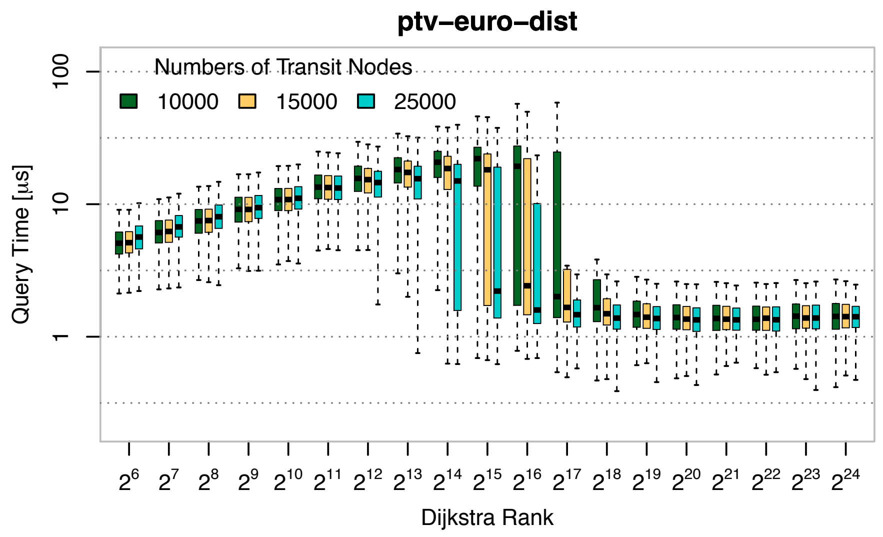


Figure 6.5.: Rank Plot Depending on Size of Transit Node Set for *ptv-europe-dist*.

transit nodes will be detected during the precomputation and are indicated by a distance of ∞, too. The search assigning Voronoi representatives may not reach all nodes and any unreached nodes are assigned to a dummy Voronoi region.

Next, we turn to *ptv-euro-dist*. As to be expected from previous work, we see that switching to distance metric is costly. The number of access nodes doubles and accordingly space overhead also doubles. Since the number of table lookups is quadratic in the number of access nodes, the query time nearly quadruples. On the positive side, the detailed model of *osm-germany-4*, which is perhaps closest to state of the art routing applications, behaves similar to euro-time. The number of access nodes increases only slightly, and considering the larger graph size, the preprocessing time also remains moderate. This is an important difference to plain CHs where switching to a detailed graph model leads to significantly increased query time.

Figure 6.5 shows the rank plot for the *ptv-euro-dist* instance for varying sizes of the transit node set. Again, we observe that the parameter influences the threshold from which on shortest paths are covered by access nodes. Also, we see the trade-off between query time and transit node set size. Figure 6.6 shows the result for the same experiment on the edge-expanded graph of Germany, instance *osm-germany-4*. Note that we experimented on a much higher number of transit nodes on this instance, because it is much larger than the other instances. Again, we observe a sharp cut-off from which on paths are covered by access nodes and that the value of $|\mathcal{T}|$ is a tuning parameter when this cut-off occurs. Most interestingly, the queries seem to have a greater variance in the sense that there are outliers that have rather low query time.

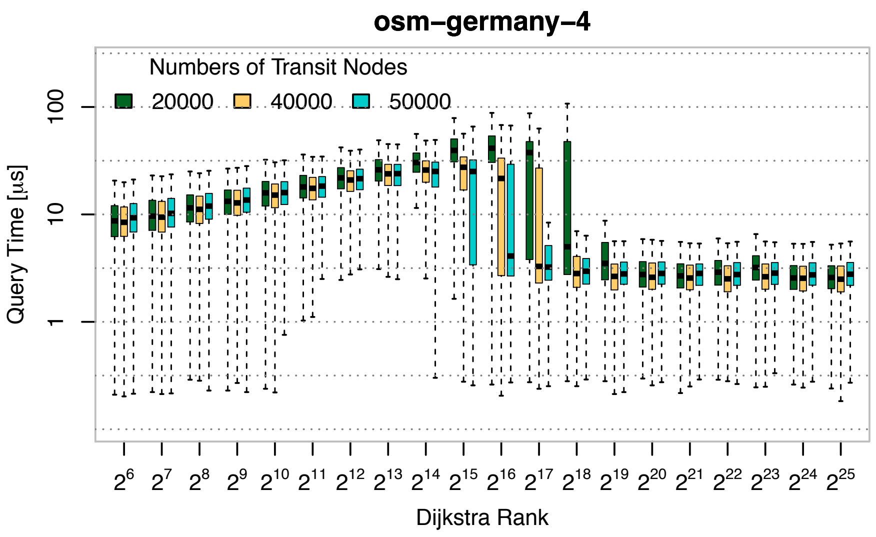


Figure 6.6.: Rank Plot Depending on Size of Transit Node Set for *osm-germany-4*.

6.3. Further Improvements

In addition to the previous experiments we identify a number of additional enhancements and use cases of our method. Each of the following sub-sections is work in progress at the time of writing. Thus, the results shall be treated as preliminary.

6.3.1. Pruning with Arc Flags

The experimental evaluation of Section 6.2.3 shows that the majority of the query time is spent in table lookups and only to a lesser extent in the locality filter. Here, we analyze how further pruning using arc flags could be applied to achieve a sub-microsecond distance oracle on a fictive 2 GHz CPU. Thus, while the following numbers are encouraging, they have to be taken with a grain of salt. First, we briefly explain the layout of the query and reach for previous work by Bauer et al. [24], Delling [55] and Abraham et al. [58] to conduct the preprocessing. Second, we note that the cost of a L1 cache miss is about 2–10 cycles and the cost of a L3 cache miss is about 100 cycles on a modern memory architecture. A L1 cache hit is accounted for by a single nano-second. On a 2 GHz machine this amounts to 5 and 50 nanoseconds, respectively. These numbers were determined experimentally by Luxen and Schieferdecker [128] when researching the cost associated with low-level memory accesses.

The TNR query can be sped up by using arc flags as previously reported, e.g. [24]. It is very similar to traditional arc flags. Instead of partitioning the entire input graph, only the core induced by the transit nodes is preprocessed. Before running the table lookups for each and every pairwise combination of access nodes, a small set of extra data in cache is queried, if a table lookup and the possibly associated expensive cache miss is needed at all.

If we partition the overlay network induced by the transit nodes into 48 regions, like [55], this would require 96 bits (uncompressed) for each node to store forward and backward flags.

This totals in less than 120 kBytes of additional information for the exemplary 10 000 transit nodes from Section 6.2.3 which easily fits into the L2 cache of any modern processor. The entire arc flag information cannot be scanned sequentially, but only a cache line of 32 bytes at a time. We assume that the access nodes for each node have been sorted previously. Since the data is sorted, it is fair to assume that six fetch operations into L2 cache suffice to get all transit nodes. This accounts to our fictive runtime with about 5 nanoseconds for each L1 cache miss and a single nanosecond for each *AND*-operation to get the information if a transit node can be discarded.

The following numbers are exemplary for the `euro-time` graph. If we have 6.1 Voronoi access nodes on average, we expect to check 6.1×6.1 pruning flags in total. As Bauer et al. [24] report, the remaining number of table lookups for TNR-AF is 3.1 which costs us a L3 cache miss in the worst case. Thus, we account for an additional 55 nanoseconds for each such access. If a local query CHASE query costs 6.1 microseconds on average [58] and is conducted for 0.581% of all queries, then the local searches account for roughly 100 nanoseconds on average. This amounts to an expected total query time on a fictive 2 GHz CPU of

$$250\text{ns} + 6 \cdot 5\text{ns} + (6.1)^2 \cdot 1\text{ns} + 3.1 \cdot 55\text{ns} + 100\text{ns} \simeq 590\text{ns},$$

assuming that a single Voronoi-Locality filter invocation costs about 250 nanoseconds. This is a conservative estimate since it does not account for any SIMD tuning opportunity.

For CHASE preprocessing, recent work of Abraham et al. [58] gives an efficient algorithm to compute arc flags on our test instance in mere minutes of preprocessing including CH construction. The additional memory overhead amounts to roughly 600 MB. Further, we make the (simplifying) assumption that computing the arc flags for the transit node overlay graph can be computed without much additional cost. Thus, we conclude that it is possible to construct a sub-microsecond distance oracle in less than 30 minutes.

6.4. Concluding Remarks

We have shown that a very simple implementation of CH-TNR yields a speedup technique for route planning with an excellent trade-off between query time, preprocessing time, and space consumption. In particular, at the price of twice the (quite fast) preprocessing time of Contraction Hierarchies, we get two orders of magnitude faster query times. Our purely graph theoretical locality filter outperforms previously used geometric filters. To the best of our knowledge, this eliminates the last remnant of geometric techniques in competitive speedup techniques for route planning. This filter is based on intersections of CH search spaces, and thus, exhibits an interesting relation to the hub labelling technique.

When comparing speedup techniques one can view this as a multi-objective optimization problem along the dimensions query time, preprocessing time, space consumption, and simplicity. Any pareto-optimal, i.e. non-dominated, method is worthwhile considering and good methods should have a significant advantage with respect to at least one measure without undue disadvantages for the other dimensions. In this respect, CH-TNR fares very well. Only hub labelling achieves significantly better query times but at the price of much higher space consumption, in particular when comparable preprocessing times are desired. Moreover, simple variants of hub labelling have even worse space consumption and less clear advantages in query time. When looking for clearly simpler techniques than CH-TNR, plain CHs come into mind but at the price of two orders of magnitude larger query time and a surprisingly small gain in preprocessing time.

CH-TNR also has significant potential for further performance improvements. Our variant of CH-TNR focuses on maximal simplicity except for the Voronoi filter which is needed for space efficiency. There are many further improvements that won't drastically change the position of CH-TNR in the landscape of speedup techniques but that could yield noticeable improvements with respect to query time, preprocessing time, or space at the price of a more complicated implementation. We now outline some of these possibilities:

Query time. Arz et al. [9] outline a number of improvements on top of CH-TNR. In [17, 171] local queries are handled fast by introducing additional *layers* of secondary and tertiary transit nodes. Our local queries are faster than Highway Hierarchies, but at least a secondary layer would be useful to further reduce the query times.

As given in Section 6.3.1, we expect about twice faster queries by combining CH-TNR with arc flags for an additional sense of goal direction. Preprocessing time could be much smaller than in [24] by new CH based methods for fast one-to-all shortest paths [58].

Furthermore, a fast many-to-one search can be based on CH-TNR. The idea is to precompute v–t-distances for all transit nodes $v \in \mathcal{T}$ and to store them in a separate array T. This can be done using $|A^{\downarrow}(v)| \cdot |\mathcal{T}|$ table lookups accessing only $|A^{\downarrow}(t)|$ rows of the distance table. Array T is likely to fit into cache. For a tuned locality filter, we also precompute $\mu(s,t)$ for all source nodes s which require a local query. The method looks promising. Unfortunately, reliable results are not yet available.

Preprocessing time. Besides CH construction the most time consuming part of our preprocessing is the exploration of the sub-transit node CH search spaces. This can probably be accelerated by a top-down computation as in [4]. Note that using post-search-stalling we still get optimal sets of access nodes. Finding Voronoi regions might be parallelizable to some extent since it explores a very low diameter graph.

Space. There are a number of relatively simple low level tuning opportunities here. For example, we can more aggressively exploit overlaps between forward/backward access nodes and search space representatives. These "dual use" nodes need to be stored only in the access nodes set together with a flag indicating that they are also a region representatives. We could also encode backward distances to access nodes as differences to forward distances. As in HH-TNR we could also encode access nodes of most nodes as the union of the access nodes of their neighbors. The region representatives stored by our graph Voronoi filter are virtually identical to the access nodes so that we only need to store a flag indicating whether an access node is also a region representative plus the few region representatives that are not access nodes also. In our experiments this would reduce space consumption by another ≈ 15 bytes per node.

CHAPTER 7

Discussion

We presented a number of building blocks for scalable mapping services. To do so, we systematically followed the paradigm of Algorithm Engineering. We described the design and the analysis of our algorithms in detail and conducted experimental evaluations on real-world data sets to emphasize the practical relevance of the solutions. Likewise, the experiments validated the efficiency of our algorithms.

We looked into the problem of geocode matching and provided an approximate dictionary index as a building block. We applied this approximate index to build an efficient and error-correcting geocoder. This was the first thorough algorithmic solution to the problem. Additionally, we looked at the sub-problem location disambiguation and introduced a neighborhood graph that is able to find important cities in the vicinity independent of any manually chosen distance thresholds.

We saw that the witness search of shared memory parallel Contraction Hierarchies preprocessing can be adapted to practically use only constant space per preprocessing core by applying tabulation hashing. Moreover, we saw that the approach can also be used to construct an efficient tie-breaking oracle that is used for independent set selection in the contraction phase. Then, we introduced distributed memory parallel preprocessing of Contraction Hierarchies on a cluster of machines. The problem of preprocessing a distributed graph is solved by introducing communication phases that synchronize graph changes among the nodes of a compute cluster. Besides that, we showed how to conduct queries in such a distributed cluster of machines.

To build further scalable services, we exploited the properties of the search data structure of Contraction Hierarchies. We gave a way to compute location scores in a road network by using an index data structure for points of interest. We introduced multi-hop ride sharing, in addition to single-hop ride sharing, as a way to enhance the features of a ride sharing platform. Here, passengers are allowed to possibly transfer between possible rides during a journey as opposed to sticking only with a single driver. We observed speedups that were an order of magnitude higher than what could have been expected from literature. Next, we looked at the efficient generation of alternative routes. An additional preprocessing step introduces an effective and efficient way to compute admissible alternative routes over a single via node. Also, we showed how to compute user equilibria on road networks that is an important step when doing traffic simulations.

We did not only introduce algorithm to advanced routing queries, we also looked into how to efficiently construct a distance oracle that is easy to implement. We presented the first instantiation of Transit Node Routing that solely builds on Contraction Hierarchies. Notably, our variant is the first to *not* rely on any geometric information (or otherwise embedding) to build a locality filter that identifies local queries. The additional preprocessing effort of the distance oracle takes about the same time as the construction of the underlying hierarchy. The oracle provides distances in the order of a single microsecond on average.

We see the applicability of our algorithms and data structures in practice. A number of features have already been implemented in the open source routing engine *Project OSRM*, e.g. the space efficient witness search for Contraction Hierarchies, and the computation of an admissible alternative route. There is considerable interest from outside the scientific research community in such scalable mapping services. This is reflected by the fact that the publicly available routing service[1] of Project OSRM is seeing 30-50 000 routing queries at the time of writing per day. Likewise, the municipality of Copenhagen, Denmark is planning to roll out an officially endorsed bicycle route planner based on Project OSRM by Summer of 2013.

Future Work. We see a number of directions for future work. Geographical data sets in general and road networks in particular moved away from static data that was compiled by a vendor every other quarter of the year. It has moved to crowd-sourced data sets that literally change by the minute. Most notably, OpenStreetMap adds around 250 000 road segment to its data base per day. Designing algorithms and data structures that are able to efficiently handle this dynamics is an interesting future challenge for research.

This thesis can be seen as a starting point to work on further building blocks of features of a scalable mapping service. As such, there is a number of further building blocks that are still waiting to be looked at. We are especially thinking of more personalized features and features that handle a stream of changes to the underlying data set. For example, the generation of flexible route descriptions that are able to increase (or decrease) the level of detail of the description for parts of the route on the fly without resorting to full re-computation is a challenging feature. This way, the description is able to reflect local knowledge of a user. Yet another interesting challenge of making services more dynamic is the integration of live traffic feeds into a mapping service. This integration, for example, shall deliver alternatives to a route that has smallest detour under given traffic conditions

[1]`http://map.project-osrm.org`

Bibliography

[1] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. HLDB: Location-Based Services in Databases. In *ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pages 339–348. ACM, 2012.

[2] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension and provably efficient shortest path algorithms. *submitted*, 2013.

[3] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *LNCS*, pages 230–241. Springer, 2011.

[4] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *European Symposium on Algorithms (ESA'12)*, volume 7501 of *LNCS*, pages 24–35. Springer, 2012.

[5] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013.

[6] Ritesh J. Agrawal and James G. Shanahan. Location Disambiguation in Local Searches Using Gradient Boosted Decision Trees. In *ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'10)*, pages 129–136, 2010.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, 1993.

[8] Julian Arz. Contraction-Hierarchy-Based Transit-Node Routing. Studienarbeit, Karlsruhe Institute of Technology, 2012.

[9] Julian Arz, Dennis Luxen, and Peter Sanders. Transit Node Routing Reconsidered. Technical report, Karlsruhe Institute of Technology, 2012. `http://arxiv.org/pdf/1302.5611v1.pdf`.

[10] Julian Arz, Dennis Luxen, and Peter Sanders. Transit Node Routing Reconsidered. In *International Symposium on Experimental Algorithms (SEA'13)*, LNCS. Springer, 2013. to appear.

[11] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative Route Graphs in Road Networks. In *International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11)*, volume 6595 of *LNCS*, pages 21–32. Springer, 2011.

[12] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 14–22. IEEE, 1998.

[13] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[14] Hillel Bar-Gera. Traffic assignment by paired alternative segments. *Transportation Research Part B: Methodological*, 44(8-9):1022–1046, 2010.

[15] H. Bast. Lecture slides: Efficient Route Planning. `http://ad-wiki.informatik.uni-freiburg.de/teaching/EfficientRoutePlanningSS2012`, 2012.

[16] Holger Bast, Stefan Funke, and Domagoj Matijevic. TRANSIT – Ultrafast Shortest-Path Queries with Linear-Time Preprocessing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[17] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.

[18] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[19] Gernot Veit Batz, Robert Geisberger, Dennis Luxen, Peter Sanders, and Roman Zubkov. Efficient Route Compression for Hybrid Route Planning. In *Mediterranean Conference on Algorithms (MedAlg'12)*, volume 7659 of *LNCS*, pages 93–107. Springer, 2012.

[20] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In *International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *LNCS*, pages 166–177. Springer, 2010.

[21] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 2013. To appear.

[22] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, Andrea Schumm, and Dorothea Wagner. The Shortcut Problem – Complexity and Algorithms. *Journal of Graph Algorithms and Applications*, 16(2):447–481, 2012.

[23] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, 2009.

[24] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, 2010.

[25] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 57:38–52, 2010.

[26] Martin Beckmann, C.B.McGuire, and C.B.Winsten. *Studies in the Economics of Transportation*. Yale University Press, New Haven, 1959.

[27] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[28] Mark De Berg, Marc van Kreveld, Rene van Oostrum, and Mark Overmars. Simple Traversal of a Subdivision without Extra Storage. *International Journal of Geographical Information Science*, 11(4):359–373, 1997.

[29] Bing Maps. http://maps.bing.com.

[30] Marcel Birn, Manuel Holtgrewe, Peter Sanders, and Johannes Singler. Simple and Fast Nearest Neighbor Search. In *Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 43–54. SIAM, 2010.

[31] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *International Conference on Data Engineering, 2001 (ICDE'01)*, pages 421–430. IEEE, 2001.

[32] David Branston. Link capacity functions: A review. *Transportation Research*, 10:223–236, 1976.

[33] Klaus Brieter, Claus Christoph Eicher, Verena Haart, and Mario Vigl. Mit dem Navi sicher in den Stau. *ADAC Motorwelt*, 3:56–59, 2010.

[34] M. Bruynooghe, A. Gilbert, and M. Sakarovitch. Une Methode d'Affectation du Trafic. In *International Symposium on the Theory of Road Traffic*, pages 198–204, 1969.

[35] Bundesministerium für Verkehr, Bau und Stadtentwicklung. Mobilität in Deutschland 2008. Case study, 2008. `http://www.mobilitaet-in-deutschland.de/pdf/MiD2008_Abschlussbericht_I.pdf`.

[36] Bureau of Public Roads. *Traffic Assignment Manual.* U.S. Dept. Of Commerce, Washingtion D.C., 1964.

[37] W. A. Burkhard and R. M. Keller. Some Approaches to Best-Match File Searching. *Communications of the ACM*, 16(4):230–236, 1973.

[38] Stefan Burkhardt and Juha Kärkkäinen. One-Gapped q-Gram Filters for Levenshtein Distance. In *Symposium on Combinatorial Pattern Matching (CPM'02)*, volume 2373 of *LNCS*, pages 225–234, 2002.

[39] Stefan Burkhardt and Juha Kärkkäinen. Better Filtering with Gapped q-grams. 56:51–70, 2003.

[40] Cambridge Vehicle Information Tech. Ltd. Choice Routing. `http://camvit.com/camvit-technical-english/Camvit-Choice-Routing-Explanation-english.pdf`, 2005.

[41] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[42] Ho-Leung Chan, Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. Compressed Indexes for Approximate String Matching. *Algorithmica*, 58(2):263–281, 2010.

[43] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and Efficient Fuzzy Match for Online Data Cleaning. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 313–324. ACM, 2003.

[44] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[45] Yanyan Chen, Michael G. H. Bell, and Klaus Bogenberger. Reliable Pretrip Multipath Planning and Dynamic Adaptation for a Centralized Road Navigation System. *IEEE Transactions on Intelligent Transportation Systems*, 8(1):14–20, 2007.

[46] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms. *Mathematical Programming, Series A*, 73:129–174, 1996.

[47] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. In *ACM–SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 83–92. IEEE, 1997.

[48] Peter Christen, Alan Willmore, and Tim Churches. A Probabilistic Geocoding System Utilising a Parcel Based Address File. In *Data Mining*, volume 3755 of *LNCS*, pages 130–145. Springer Berlin Heidelberg, 2006.

[49] Archie L. Cobbs. Fast Approximate Matching Using Suffix Trees. In *Symposium on Combinatorial Pattern Matching (CPM'95)*, volume 937 of *LNCS*. Springer, 1995.

[50] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In *ACM Symposium on Theory of Computing (STOC'04)*, pages 91–100. ACM, 2004.

[51] D.F. Cooke. *The History of Geographic Information Systems: Perspectives from the Pioneers*. Prentice Hall, 1998.

[52] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.

[53] K. B. Davidson. The Theoretical Basis of a Flow-Travel Time Relationship for use in Transportation Planning. *Australian Road Research*, 8:32–35, 1978.

[54] Brian C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[55] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 2009.

[56] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, 2011.

[57] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing Multimodal Journeys in Practice. In *International Symposium on Experimental Algorithms (SEA'13)*, LNCS, 2013. to appear.

[58] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 2012.

[59] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *LNCS*, pages 376–387. Springer, 2011.

[60] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE, 2011.

[61] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hub Label Compression. In *International Symposium on Experimental Algorithms (SEA'13)*, LNCS. Springer, 2013. to appear.

[62] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In *International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *LNCS*, pages 125–136. Springer, 2009.

[63] Daniel Delling and Renato F. Werneck. Faster Customization of Road Networks. In *International Symposium on Experimental Algorithms (SEA'13)*, LNCS. Springer, 2013. to appear.

[64] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[65] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[66] John F. Dillenburg, Ouri Wolfson, and Peter C. Nelson. The Intelligent Travel Assistant. In *International Conference on Intelligent Transportation Systems (ITSS'02)*, pages 691–696. IEEE, 2002.

[67] Ulrich Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat Inc., 2007. `http://people.redhat.com/drepper/cpumemory.pdf`.

[68] Florian Drews. Multi-Hop Ride Sharing. bachelor thesis, Karlsruhe Institute of Technology, Fakultät für Informatik, 2012.

[69] William J. Drummond. Address Matching: GIS Technology for Mapping Human Activity Patterns. *Journal of the American Planning Association*, 61(2):240–251, 1995.

[70] Matthias Duschl, Thomas Brenner, Antje Schimke, and Dennis Luxen. Contribution and Spatial Dimension of Geolocated External Factors to the Growth of Firms – Empirical Evidence for Germany. In *Geography of Innovation 2012*, 2012.

[71] Pierce Eichelberger. The Importance of Addresses: The Locus of GIS. In *Urban and Regional Information Systems Association Annual Conference (URISA'93)*, volume 1, pages 212–222, 1993.

[72] Jochen Eisner and Stefan Funke. Transit Nodes – Lower Bounds and Refined Construction. In *Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 141–149. SIAM, 2012.

[73] David Eppstein. Finding the k shortest paths. In *Symposium on Foundations of Computer Science (FOCS'94)*, pages 154–165. IEEE, 1994.

[74] ESRI. ArcGIS 9 Geocoding Rule Base Developer Guide, 2009.

[75] Alex Fabrikant, Christos Papadimitriou, and Kunal Talwar. The Complexity of Pure Nash Equilibria. In *ACM Symposium on Theory of computing (STOC'04)*, pages 604–612, 2004.

[76] Fletcher Foti, Paul Waddell, and Dennis Luxen. A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale. In *Transportation Research Board Conference on Innovations in Travel Modeling*. TRB, 2012.

[77] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[78] Kimmo Fredriksson. Engineering Efficient Metric Indexes. *Pattern Recognition Letters*, 28(1):75–84, 2007.

[79] Robert Geisberger. *Advanced Route Planning in Transportation Networks.* PhD thesis, Karlsruhe Institute of Technology, Fakultät für Informatik, 2011.

[80] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In *Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.

[81] Robert Geisberger, Dennis Luxen, Sabine Neubauer, Peter Sanders, and Lars Volker. Fast Detour Computation for Ride Sharing. Technical report, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

[82] Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker. Fast Detour Computation for Ride Sharing. In *Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*, volume 14 of *OASIcs*, pages 88–99, 2010.

[83] Robert Geisberger and Peter Sanders. Engineering Time-Dependent Many-to-Many Shortest Paths Computation. In *Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*, volume 14 of *OASIcs*, pages 74–87, 2010.

[84] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[85] Andrew V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.

[86] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

[87] Daniel W. Goldberg. Geocoding Best Practices Guide. Technical report, University of Southern California, GIS Research Laboratory, 2008.

[88] Daniel W. Goldberg, John P. Wilson, and Craig A. Knoblock. From Text to Geographic Coordinates: The Current State of Geocoding. *Journal of the Urban and Regional Information Systems Association*, 19:33–47, 2007.

[89] Sreenivas Gollapudi and Rina Panigrahy. A Dictionary for Approximate String Search and Longest Prefix Search. In *ACM International Conference on Information and Knowledge Management (CIKM'06)*, pages 768–775. ACM, 2006.

[90] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New Indices for Text: Pat Trees and Pat Arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice-Hall, 1992.

[91] Google Maps. http://maps.google.com.

[92] Leonidas Guibas, Donald Knuth, and Micha Sharir. Randomized Incremental Construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[93] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD international conference on Management of Data (SIGMOD'84)*, pages 47–57. ACM, 1984.

[94] P. Hansen. Bricriteria Path Problems. In *Multiple Criteria Decision Making – Theory and Application*, pages 109–127. Springer, 1979.

[95] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[96] Stephan Hartwig and Michael Buchmann. Empty Seats Travelling. Technical report, Nokia Research Center, 2007. `http://research.nokia.com/files/tr/NRC-TR-2007-003.pdf`.

[97] Monika R. Henzinger, Philip N. Klein, Satish Rao, and Ashok Subramanian. Faster Shortest-Path Algorithms for Planar Graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[98] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.

[99] Heikki Hyyrö, Kimmo Fredriksson, and Gonzalo Navarro. Increased Bit-Parallelism for Approximate String Matching. *ACM Journal of Experimental Algorithmics*, 10:1–27, 2005.

[100] H. Imai and Masao Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of Information Processing*, 9(3):159–162, 1987.

[101] Christian S. Jensen, Jan Kolářvr, Torben Bach Pedersen, and Igor Timko. Nearest Neighbor Queries in Road Networks. In *ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'03)*, pages 1–8. ACM, 2003.

[102] Tanuja Joshi, Joseph Joy, Tobias Kellner, Udayan Khurana, A Kumaran, and Vibhuti Sengar. Crosslingual Location Search. In *ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*, pages 211–218. ACM, 2008.

[103] Christian Jung, Daniel Karch, Sebastian Knopp, Dennis Luxen, and Peter Sanders. Engineering Efficient Error-Correcting Geocoding. In *ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS '11)*, pages 469–472. ACM, 2011.

[104] Daniel Karch. An efficient algorithm for fault-tolerant geocoding. Diplomarbeit, Karlsruhe Institute of Technology, Fakultät für Informatik, 2010.

[105] Daniel Karch, Dennis Luxen, and Peter Sanders. Improved Fast Similarity Search in Dictionaries. Technical report, Karlsruhe Institute of Technology, Fakultät für Informatik, 2010. `http://arxiv.org/pdf/1102.3306.pdf`.

[106] Daniel Karch, Dennis Luxen, and Peter Sanders. Improved Fast Similarity Search in Dictionaries. In *Symposium on String Processing and Information Retrieval (SPIRE'10)*, LNCS, pages 173–178. Springer, 2010.

[107] Hassan A. Karimi, Matej Durcik, and William Rasdorf. Evaluation of Uncertainties Associated with Geocoding Techniques. *Computer-Aided Civil and Infrastructure Engineering*, 19(3):170–185, 2004.

[108] Juha Kärkkäinen. Computing the Threshold for q-Gram Filters. In *Scandinavian Workshop on Algorithm Theory (SWAT'02)*, volume 2368 of *LNCS*, pages 348–357. Springer, 2002.

[109] Juha Kärkkäinen and Joong Chae Na. Faster Filters for Approximate String Matching. In *Meeting on Algorithm Engineering & Experiments (ALENEX'07)*. SIAM, 2007.

[110] Juha Kärkkäinen and Peter Sanders. Simple Linear Work Suffix Array Construction. In *Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[111] Tim Kieritz. Distributed Parallel Time Dependent Contraction Hierarchies. Master's thesis, Karlsruhe Institute of Technology, Fakultät für Informatik, 2009.

[112] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed Time-Dependent Contraction Hierarchies. In *International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *LNCS*, pages 83–93. Springer, 2010.

[113] Matthias Kirschner, Philipp Schengbier, and Tobias Tscheuschner. Speed-Up Techniques for the Selfish Step Algorithm in Network Congestion Games. In *International Symposium on Experimental Algorithms (SEA'09)*, pages 173–184. Springer, 2009.

[114] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.

[115] Donald E. Knuth. Notes on “Open” Addressing. Technical report, , 1963. `ftp://ftp.inria.fr/INRIA/Projects/algo/web/AofA/Research/src/first.ps.gz`.

[116] Donald E. Knuth. *The Art Of Computer Programming*, volume 3. Addison-Wesley, 1997.

[117] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Workshop on Experimental Algorithms (WEA'05)*, volume 3503 of *LNCS*, pages 126–138. Springer, 2005.

[118] Nancy Krieger. Overcoming the Absence of Socioeconomic Data in Medical Records: Validation and Application of a Census-Based Methodology. *American Journal of Public Health*, 82(5):703–710, 1992.

[119] Nancy Krieger, Jarvis T. Chen, Pamela D. Waterman, Mah-Jabeen Soobader, and Rosa Carson S. V. Subramanian. Geocoding and monitoring of US socioeconomic inequalities in mortality and cancer incidence: does the choice of area-based measure and geographic level matter? The Public Health Disparities Geocoding Project. *American Journal of Epidemiology*, 156(5):471–482, 2002.

[120] Harold William Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistic Quarterly*, 2:63–87, 1955.

[121] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[122] Ulrich Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.

[123] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[124] M. Law. *Guide to Worldwide Postal-Code and Address Formats: Practical Information About International Addressing, Including Country-By-Country Postal-Code and Address Formats*. WorldVu, LLC, 2004.

[125] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[126] Dennis Luxen and Peter Sanders. Hierarchy Decomposition for Faster User Equilibria on Road Networks. In *International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *LNCS*, pages 242–253. Springer, 2011.

[127] Dennis Luxen and Dennis Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 260–270. Springer, 2012.

[128] Dennis Luxen and Dennis Schieferdecker. Doing More for Less – Cache-Aware Parallel Contraction Hierarchies Preprocessing. Technical report, Karlsruhe Institute of Technology, Fakultät für Informatik, 2012. `http://arxiv.org/pdf/1208.2543v1.pdf`.

[129] Dennis Luxen and Christian Vetter. Real-Time Routing with OpenStreetMap data. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'11)*, pages 513–516. ACM, 2011.

[130] Moritz G. Maaß and Johannes Nowak. A new method for approximate indexing and dictionary lookup with one error. *Information Processing Letters*, 96(5):185–191, 2005.

[131] Moritz G. Maaß and Johannes Nowak. Text Indexing with Errors. *Journal of Discrete Algorithms*, 5(4):662–681, 2007.

[132] Udi Manber and Gene Myers. Suffix Arrays: a new Method for on-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[133] Frank Marguerite and Philip Wolfe. An Algorithm for Quadratic Programming. *Naval Research Logistics Quaterly*, 3:95–110, 1956.

[134] Bruno Martins, Mário J. Silva, Sérgio Freitas, and Ana Paula Afonso. Handling Locations in Search Engine Queries. In *ACM Workshop On Geographic Information Retrieval (SIGIR'06)*. ACM, 2006.

[135] Ernesto Queiros Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.

[136] Kurt Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27(3):125–128, 1988.

[137] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox.* Springer, 2008.

[138] Merriam Webster. Webster's Third New International Dictionary, 1985. `http://www.merriam-webster.com/` – accessed March 29th, 2013.

[139] Ulrich Meyer. Single-Source Shortest-Paths on Arbitrary Directed Graphs in Linear Average-Case Time. In *ACM–SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 797–806. ACM, 2001.

[140] Stoyan Mihov and Klaus U. Schulz. Fast Approximate Search in Large Dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.

[141] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11(2.8):1–29, 2006.

[142] M. Mor and A. S. Fraenkel. A Hash Code Method for Detecting and Correcting Spelling Errors. *Communications of the ACM*, 25(12):935–938, 1982.

[143] Girish Motwani and Sandhya G. Nair. Search Efficiency in Indexing Structures for Similarity Searching. Technical report, Indian Institute of Science, Department of Computer Science and Automation, 2004. `http://arxiv.org/pdf/cs/0403014v2.pdf`.

[144] John D. Murchland. Road Network Traffic Distribution in Equilibrium. In *Mathematical Models in the Social Sciences*, pages 145–183, 1979.

[145] Alan T. Murray, Tony H. Grubesic, Ran Wei, and Elizabeth A. Mack. A Hybrid Geocoding Methodology for Spatio-Temporal Data. *Transactions in GIS*, 15(6):795–809, 2011.

[146] Gene Myers. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *Journal of the*, 46(3):395–415, 1999.

[147] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search on Time-Dependent Road Networks. *Networks*, 59:240–251, 2012.

[148] Gonzalo Navarro and Ricardo Baeza-Yates. Improving an Algorithm for Approximate Pattern Matching. *Algorithmica*, 30:473–502, 1998.

[149] Noam Nisan, Tim Roughgarden, Éva Tardos, and Vijay V. Vazirani, editors. *Algorithmic Game Theory.* Cambridge University Press, 2007.

[150] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *Data Compression Conference (DCC '09)*, pages 193–202. IEEE, 2009.

[151] Masyuku Ohta, Kosuke Shinoda, Yoichiro Kumada, Hideyuki Nakashima, and Itsuki Noda. Is Dial-a-Ride Bus Reasonable in Large Scale Towns? — Evaluation of Usability of Dial-a-Ride Systems by Simulation. In *Multiagent for Mass User Support (MAMUS'03)*, volume 3012 of *LNCS*, pages 105–119. Springer, 2004.

[152] OpenStreetMap. `http://osm.org` – accessed March 29th, 2013.

[153] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.

[154] R. Thomas O'Reagan and Alan Saalfeld. Geocoding Theory and Practice at the Bureau of the Census. Technical report, United States Census Bureau, 1987.

[155] Mihai Patrascu and Mikkel Thorup. The Power of Simple Tabulation Hashing. In *ACM Symposium on Theory of computing (STOC '11)*, pages 1–10. ACM, 2011.

[156] Ira Pohl. Heuristic Search Viewed as Path Finding in a Graph . *Artificial Intelligence*, 1(3):193–204, 1970.

[157] Project Gutenberg. `http://www.gutenberg.org/` – accessed March 29th, 2013.

[158] PTV AG. `http://www.ptv.de` – accessed February, 25th 2013.

[159] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Experimental Comparison of Shortest Path Approaches for Timetable Information. In *Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 88–99. SIAM, 2004.

[160] Scott M. Ramming. *Network Knowledge and Route Choice*. PhD thesis, Massachusetts Institute of Technology, 2002.

[161] John A. Rice and Vassilis Tsotras. Exact Graph Search Algorithms for Generalized Traveling Salesman Path Problems. In *International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 344–355. Springer, 2012.

[162] Duangduen Roongpiboonsopit and Hassan A. Karimi. Comparative evaluation and analysis of online geocoding services. *International Journal of Geographical Information Science*, 24:1081–1100, 2010.

[163] R. W. Rosenthal. The Network Equilibrium Problem in Integers. *Networks*, 3:53–59, 1973.

[164] Tim Roughgarden. *Selfish Routing and the Price of Anarchy*. MIT Press, 2005.

[165] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest Neighbor Queries. *ACM SIGMOD Record*, 24(2):71–79, 1995.

[166] Luis M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and P. Morales. Approximate String Matching with Compressed Indexes. *Algorithms 2*, 3:1105–1136, 2009.

[167] Hanan Samet and Robert E. Webber. Storing a Collection of Polygons using Quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, 1985.

[168] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. *ACM Journal of Experimental Algorithmics*, 17(1):1–40, 2012.

[169] Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *European Symposium on Algorithms (ESA'11)*, volume 6942 of *LNCS*, pages 469–480, 2011.

[170] Mark De Berg Sanderson and Janet Kohler. Analyzing Geographic Queries. In *Workshop on Geographic Information Retrieval (SIGIR'04)*. ACM, 2004.

[171] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 2008.

[172] Vibhuti Sengar, Tanuja Joshi, Joseph Joy, Samarth Prakash, and Kentaro Toyama. Robust Location Search From Text Queries. In *ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'07)*, pages 24:1–24:8. ACM, 2007.

[173] Vibhuti Sengar, Tanuja Joshi, Joseph M. Joy, and Samarth Prakash. Building a Global Location Search Service. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, pages 1243–1246. ACM, 2008.

[174] Yosef Sheffi. *Urban Transportation Networks: Equlibrium Analysis with Mathematical Programming*. Prentice-Hall, Inc Englewood Cliffs, NJ 07632, 1982.

[175] Front Seat Software. Walk Score Methodology, 2010. `http://blog.walkscore.com/wp-content/uploads/2010/12/WalkScoreMethodology.pdf`.

[176] Christian Sommer. Shortest-Path Queries in Static Networks, 2012. submitted. Preprint available at `http://www.sommer.jp/spq-survey.htm`.

[177] Heinz Spiess. Technical Note–Conical Volume-Delay Functions. *Transportation Science*, 24(2):153–158, 1990.

[178] Yuki Sugiyama, Minoru Fukui, Macoto Kikuchi, Katsuya Hasebe, Akihiro Nakayama, Katsuhiro Nishinari, Shin ichi Tadaki, and Satoshi Yukawa. Traffic jams without bottlenecks – experimental evidence for the physical mechanism of the formation of a jam. *New Journal of Physics*, 10(3):033001, 2008.

[179] Burkhard Stiller Thomas Bocek, Ela Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, 2007. http://fastss.csg.uzh.ch/.

[180] Mikkel Thorup. Undirected Single Source Shortest Paths with Positive Integer Weights in Linear Time. *Journal of the ACM*, 46(3):362–394, 1999.

[181] Esko Ukkonen. Approximate String-Matching with q-grams and Maximal Matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[182] Esko Ukkonen. Approximate String Matching over Suffix Trees. In *Symposium on Combinatorial Pattern Matching (CPM'93)*, volume 684 of *LNCS*, pages 228–242. Springer, 1993.

[183] Unknown Authors. TMC-Stauumfahrung:Verkehrsprobleme durch Stauverlagerungen? Technical report, ADAC e.V., 2010.

[184] Christian Vetter. Parallel Time-Dependent Contraction Hierarchies. Technical report, Karlsruhe Institute of Technology, Fakultät für Informatik, 2009. `http://algo2.iti.kit.edu/download/vetter_sa.pdf`.

[185] Lars Volker. Route Planning in Road Networks with Turn Costs, 2008. Universität Karlsruhe, Fakultät für Informatik, Student Research Project. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/volker_sa.pdf`.

[186] John Glen Wardrop. Some Theoretical Aspects of Road Traffic Research. In *Institute of Civil Engineers: Engineering Divisions*, volume 1, pages 325–378, 1952.

[187] Wikipedia. `http://wikipedia.org` – accessed March, 29th 2013.

[188] W. John Wilbur and Karl Sirotkin. The Automatic Identification of Stop Words. *Journal of Information Science*, 18:45–55, 1992.

[189] J.W.J. Williams. Algorithm 232: Heapsort. *Journal of the ACM*, 7(6):347–348, 1964.

[190] William E. Winkler. Approximate String Comparator Search Strategies for very Large Administrative Lists. Technical report, Statistical Research Division, U.S. Census Bureau, 2005.

[191] Ho Chung Wu, Robert Wing Pong Luk, Kam F. Wong, and Kui L. Kwok. Interpreting TF-IDF Term Weights as Making Relevance Decisions. *ACM Transactions on Information Systems*, 26(3):1–37, 2008.

[192] Sun Wu and Udi Manber. Agrep – a Fast Approximate Pattern-Matching Tool. In *USENIX Technical Conference*, pages 153–162, 1992.

[193] Sun Wu and Udi Manber. Fast Text Searching: Allowing Errors. *Communications of the ACM*, 35(10):83–91, 1992.

[194] Yun Hui Wu, Lin Jie Guan, and Stephan Winter. Peer-to-Peer Shared Ride Systems. In *GeoSensor Networks*, volume 4540 of *LNCS*, pages 252–270. Springer, 2006.

[195] Xin Xing, Tobias Warden, Tom Nicolai, and Otthein Herzog. SMIZE: A spontaneous Ride-Sharing System for Individual Urban Transit. In *Multiagent System Technologies (MATES'09)*, volume 5774 of *LNCS*, pages 165–176. Springer, 2009.

[196] Jin Y. Yen. Finding the k Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.

[197] Albert Lindsey Zobrist. A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Sciences Department, University of Wisconsin, Madison, 1969.

APPENDIX A

Pseudo-Code of Geocoding Query

Listing A.1: Error-Correcting Geocoding Query

```
function geocode(street, city, d)
   street_tokens = tokenize(normalize(street))
   city_tokens = tokenize(normalize(city))
   street_candidates = match(street_tokens, street_index, d, $\infty$)
   partial_city_candidates = match(city_tokens, city_index, 0, $\infty$)
   for s in street_candidates do
    matched_city = (cities(s) $\cap$ partial_city_candidates)
    if matched_city $\neq \emptyset$ do
      if rating((s, matched_city)) $\geq$ threshold do
        return (s, matched_city) end
    end
   end
   for c in partial_city_candidates do
    periphery_city_candidates = periphery(c)
    matched_city = (cities(s) $\cap$ periphery_city_candidates)
    if (matched_city $\neq \emptyset$) and (rating((s, matched_city)) $\geq$ threshold) do
      return (s,matched_city) end
   end
   city_candidates = match(city_tokens, city_index, 2, $\infty$)
   street_cities = $\emptyset$
   $C_{\mathcal{T}\times\mathcal{S}} := \{(t,s) \in$ city_candidates $\times$ street_candidates: $t \in \text{cities}(s) \neq \emptyset\}$
   for candidate in $C_{\mathcal{T}\times\mathcal{S}}$ do
    matched_city = (cities(s) $\cap$ periphery_city_candidates)
    if rating(candidate) $\geq$ threshold do
      return candidate end
   end
end
```

APPENDIX B

Assembly Code of Tabulation Hash Based Tie-Breaker

Listing B.1: X86 Assembly Code of Tabulation Hash Bias Function

```
bias(unsigned int, unsigned int):
  mov rdx, QWORD PTR table1[rip]
  mov rax, QWORD PTR table2[rip]
  mov r8d, edi
  shr r8d, 16
  movzx ecx, di
  movzx r9d, si
  movzx r8d, r8w
  movzx ecx, WORD PTR [rdx+rcx*2]
  movzx edx, WORD PTR [rdx+r9*2]
  xor cl, BYTE PTR [rax+r8*2]
  mov r8d, esi
  shr r8d, 16
  movzx r8d, r8w
  xor dl, BYTE PTR [rax+r8*2]
  cmp edi, esi
  movzx ecx, cl
  setb  al
  movzx edx, dl
  cmp cx, dx
  setb  sil
  cmp cx, dx
  cmovne  eax, esi
  ret
```

This is the actual assembly code generated by GCC 4.6.3 and has only three instructions more than the code for a tie-breaker based on a simple linear congruence like $a \mod p + b$.

APPENDIX C

Combined Rankplot for Distributed Contraction

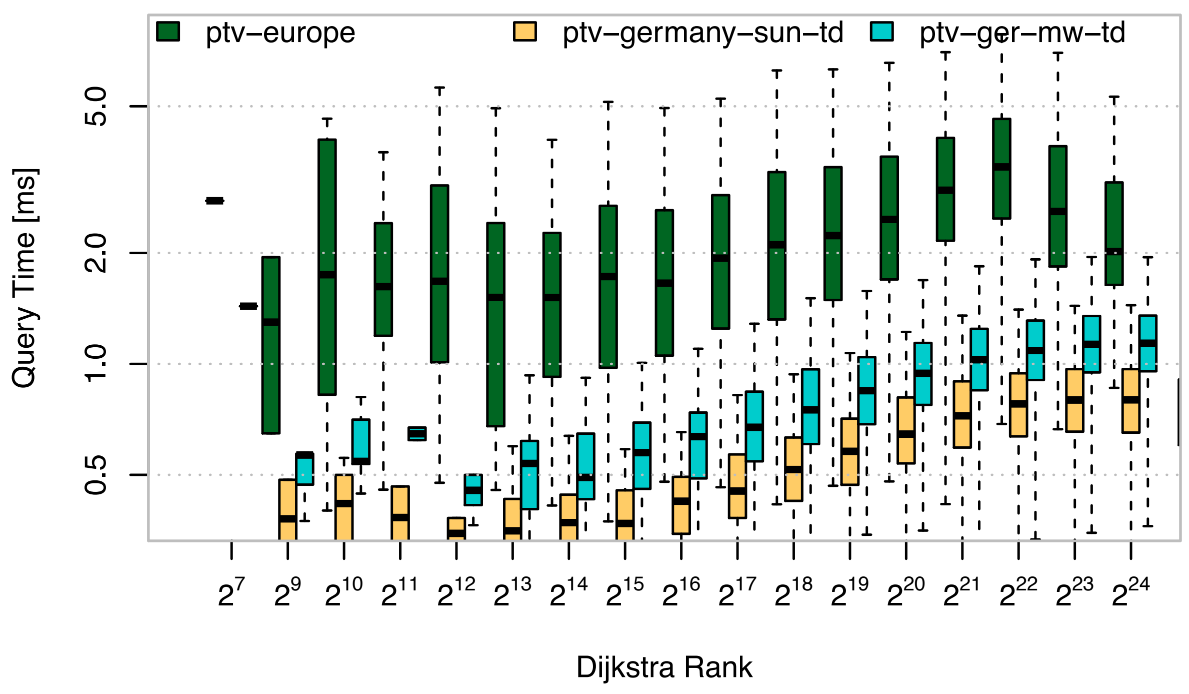


Figure C.1.: Combined Rankplot from [112] of Distributed Queries for all Three Test Instances on 16 PEs.

APPENDIX D

Machine Table

	CPU Type	Frequency [GHz]	Cores	RAM [GB]	Cache [MB]	Sections
A	Core i7-920	2.667	4	12	8, L3	3.2, 3.3, 5.5, 6
B	Xeon X5355	2.667	8	16	8, L3	4.3
	(200 cluster nodes with InfiniBand interconnection network)					
C	Opteron 6212	2.600	6	128	16, L3	4.2
D	Core i7-860	2.800	4	16	8, L3	3.4.1
E	Xeon X5690	3.470	12	256	12, L3	5.2
F	Opteron 270	2.000	1	8	2, L2	5.3
G	Opteron 6168	1.900	12	128	6, L3	5.3, 5.5
H	Opteron 8350	2.000	1	64	2, L2	5.6
I	Xeon X5550	2.667	8	48	8, L3	3.2

Table D.1.: List of Machines Used in the Experiments.

APPENDIX E

Extracting Graphs from OpenStreetMap

We explain how to generate the graphs from OpenStreetMap (OSM) that were used in thesis from raw data. The description is given for a Unix/Linux based operating system, but the steps should work very similarly on other operating systems. First, we show how to obtain the data, and second, we show how extract a graph. Third, the graph data file format is briefly explained.

The data comes in two different file format. One format is plain XML, the other is an equivalent packed binary format, called *pbf*. Both formats are supported by the tool chain used here. OSM Data literally changes by the minute and can be obtained from different sources on the Internet. Geofabrik[1] provides daily extracts of certain areas of the world, i.e. select continents, countries, or select federal states.

The data of OSM is also available as a full history dump[2], that contains every revision of each object in the database. Revisions that existed at a given timestamp can be generated from this dump with a tool set called *Osmium*[3] which is written in C++ and runs on all major platforms. The code is open source and freely available. The program in file `examples/osmium_range_from_history.cpp` handles the generation. It generates the history for a range of time and if start and end UNIX timestamp of that range equal, it generates the planet file for that particular timestamp. These timestamps are set in code by changing the values of variable `time1` and `time2` to valid UNIX timestamps:

```
Osmium::Handler::RangeFromHistory<Osmium::Output::Handler> range_handler
    (out, time1, time2);
```

After compiling the planet file can be extracted with:

[1]http://download.geofabrik.de
[2]http://planet.osm.org/planet/full-history/
[3]wiki.osm.org/wiki/Osmium

```
$ ./ osmium_range_from_history infile.osm.pbf outfile.osm.pbf
```

Once an *.osm.*{*xml/pbf*} file has been generated or downloaded, it can be processed with the routines of Project OSRM. Using the default car profile this is done by the following command line: The result of this step is a generated graph file (`file.osrm`) and a list of turn

```
$ ./osrm-extract file.osm.pbf
```

restrictions (`file.osrm.restrictions`). We now describe the graph format:T The first four bytes give the number of node, then a list of nodes is given. This is followed by four bytes that give the number of edge and then a list of edges.

```
struct Node {
  unsigned lat; //multiplied with 100.000
  unsigned lon; //multiplied with 100.000
  unsigned id;  //original OSM ID
  bool isBarrier; //cars cannot pass this node
  bool isTrafficLight;
};
```

```
struct Edge {
  unsigned sourceID; //original OSM ID
  unsigned targetID; //original OSM ID
  int distance;      // in [m]
  short direction;   // 0 = bidir, 1 = oneway
  unsigned category; //see profile
  short maxSpeed;    // for cars
  unsigned nameID;   //ID for street name
  bool isRoundabout; //indicates if edge is part of roundabout
  bool ignoreInGrid; //indicates of edge shall be ignored in NN lookup
  bool accessRestricted; //indicates if access is restricted
};
```

```
struct Restriction {
  unsigned viaNode;    //original OSM ID
  unsigned sourceNode; //original OSM ID
  unsigned targetNode; //original OSM ID
    struct Bits { //mostly unused
    Bits() : isOnly(false), unused1(false), unused2(false),
             unused3(false), unused4(false), unused5(false),
             unused6(false), unused7(false) {}
    char isOnly:1; //indicates if the restriction is a only-restriction
       or a no-restriction
    char unused1:1;
    char unused2:1;
    char unused3:1;
    char unused4:1;
    char unused5:1;
    char unused6:1;
    char unused7:1;
    } flags;

};
```

Notation

$[m]$	range of $[0, \dots, m-1]$	28
$\bar{G}$	reverse graph	24
$\mathcal{N}_d(w)$	d-(deletion-)neighborhood	40
$\mu(s,t)$	length (cost) of shortest path between nodes s and s	25
Σ	alphabet	29
τ	departure time	24
ε	empty string	29
$A^{(v)}$	Authoritative compute node	80
$d^{\downarrow}(u,v)$	backward distance	32
$d^{\uparrow}(s,v)$	forward distance	35
$deg(v)$	degree of node v	24
$deg_{in}(v)$	in-degree of node v	24
$deg_{out}(v)$	out-degree of node v	24
E	finite set of graph edges	24
$e = (u,v) \in E$	graph edge	24
$ed(s,t)$	edit distance between strings s and t	30
$G = (V,E)$	graph	24
$G^{\downarrow}$	backward search space	32
$G^{\uparrow}$	forward search space	32
$l(e)$	length (cost) of edge e	24
$l(P)$	length of a path P	25
$m := \|E\|$	number of edges in a graph	24
$n := \|V\|$	number of nodes in a graph	24
P	path in G	25
P_{st}	shortest path between nodes s and t	25
s_i	i-th character of string s	29
V	finite set of graph nodes	24
CH	Contraction Hierarchies	30
DAG	directed acyclic Graph	25

IDF	Inverse Document Frequency	53
PE	processing element, i.e. a core	38
PQ	priority queue	26

Curriculum Vitæ

Personal Data

Place and Date of Birth:	Frankfurt/Main, Germany — 03 July 1978
Address:	Blindstraße 36, 76187 Karlsruhe, Germany
email:	luxen@kit.edu

Work Experience

Current Apr 2008	Researcher at Karlsruhe Institute of Technology *Institute of Theoretical Informatics, Algorithms I* Academic research on algorithm engineering with emphasis on route planning and geocoding algorithms.
Jan 2008 Aug 2007	Research Associate at Frankfurt School of Finance & Management *Information Management Chair* Main Duty: Introductory courses to freshmen on fundamentals of data structures, algorithms and computability.
Oct 2007 Apr 2003	Student assistant at Goethe University, Frankfurt Worked for the chair of Theoretical Computer Science and supported teaching duties of the professor.
Jun 2007 May 2001	IT specialist at Dresdner Bank AG, Germany Worked part-time at infrastructure departement of the *Enabling Technologies Group*, Corporate Center IT.

Education

July 2007	**Goethe University**, Frankfurt/Main, Germany Diploma in Computer Science 1.1 – *Very Good* – Major: Theoretical Computer Science – Minor: Meterology Thesis: “Local Search in Variable Depth” Advisor: Prof. Dr. Georg Schnitger Top 3% of all graduates
May 2001	Dresdner **Dresdner Bank AG**, Frankfurt/Main, Germany State recognized apprenticeship as certified IT specialist 89% – Good, 7 % above average
July 1998	**Gymnasium Oberursel**, Oberursel/Ts., Germany Abitur, general qualification for university entrance

Certificates

June 2001	Siemens – IT specialist for integrated systems
June 2000	University of Cambridge, UK – First Certificate in English

List of Publications

Peer-Reviewed Conference Papers

Multi-Hop Ride Sharing
Florian Drews, and Dennis Luxen
6th Annual Symposium on Combinatorial Search (SoCS 2013)

Transit Node Routing Reconsidered
Julian Arz, Dennis Luxen, and Peter Sanders
12th International Symposium on Experimental Algorithms (SEA 2013)

Efficient Route Compression for Hybrid Route Planning
G. Veit Batz, Robert Geisberger, Dennis Luxen, Peter Sanders, and Roman Zubkow
1st Mediterranean Conference on Algorithms (MedAlg 2012)

Candidate Sets for Alternative Routes in Road Networks
Dennis Luxen, and Dennis Schieferdecker
11th International Symposium on Experimental Algorithms (SEA 2012)

A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale
Fletcher Foti, Paul Waddell, and Dennis Luxen
4th Transportation Research Board Conference on Innovations in Travel Modeling (ITM'12)

Robust Mobile Route Planning with Limited Connectivity
Daniel Delling, Moritz Kobitzsch, Dennis Luxen, and Renato Werneck
Meeting on Algorithm Engineering & Experiments (ALENEX 2012)

Contribution and Spatial Dimension of Geolocated External Factors to the Growth of Firms – Empirical Evidence for Germany
Matthias Duschl, Thomas Brenner, Antje Schimke, and Dennis Luxen
Geography of Innovation 2012

Engineering Efficient Error-Correcting Geocoding (short paper)
Christian Jung, Daniel Karch, Sebastian Knopp, Dennis Luxen, and Peter Sanders
19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2011)

Real-Time Routing with OpenStreetMap data (short paper)
Dennis Luxen, and Christian Vetter
19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2011)

Hierarchy Decomposition and Faster User Equilibria on Road Networks
Dennis Luxen, and Peter Sanders
International Symposium on Experimental Algorithms (SEA 2011)

Improved Fast Similarity Search in Dictionaries
Daniel Karch, Dennis Luxen, and Peter Sanders
String Processing and Information Retrieval Symposium (SPIRE 2010)

Fast Detour Computation for Ride Sharing
Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker
Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2010)

Distributed Time-Dependent Contraction Hierarchies
Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter
International Symposium on Experimental Algorithms (SEA 2010)

Submitted Papers and Articles

Journal Articles

Candidate Sets for Alternative Routes in Road Networks
Dennis Luxen, and Dennis Schieferdecker
Invited Submission – ACM Journal on Experimental Algorithmics (JEA)

Industry-specific growth and agglomeration
Matthias Duschl, Thomas Brenner, Antje Schimke, and Dennis Luxen
Submitted – Journal of the Regional Studies Association (CRES)

Deutsche Zusammenfassung

Kartendienste sind in den vergangenen Jahren im Internet allgegenwärtig geworden. Diese Dienste erfreuen sich großer Beliebtheit. Dabei wird allerdings häufig übersehen, dass es nur einige wenige global agierende Anbieter gibt, die den Löwenanteil des Markts unter sich aufteilen. Dazu kommt, dass auch nur einige wenige Anbieter in der Lage sind im sprichwörtlichen Sinne Millionen an Benutzern am Tag effizient bedienen zu können. Leider ist die verfügbare Literatur über diese Lösungen dünn gesät. Die vorliegende Arbeit stellt einzelne Bausteine vor, wie verschiedene Feature eines Kartendienst effizient zu Designen und zu Implementieren sind.

Die offensichtlichsten Kernkomponenten eines Kartendienstes sind die Anzeige von Karten, die Routenplanung und weitere sogenannte Location Based Services, die komplexere Benutzeranfragen beantworten. Sie tun dies indem sie bestehende Algorithmen und Datenstrukturen nutzen oder neue zu diesem Zwecke entworfen werden. In der Regel ist es so, dass einfache Lösungen, die ad-hoc entworfen und implementiert werden, schnell Ergebnisse zeitigen. Allerdings kommen diese Ad-Hoc-Lösungen schnell an die Grenze ihrer Leistungsfähigkeit. Es kann sein, dass diese Verfahren für kleine Datensätze und eine niedrige Nutzerzahl zufrieden stellend funktionieren.

Das echte Problem bei Kartendiensten im Internet ist allerdings die Skalierbarkeit des Dienstes. Es ist daher wichtig Algorithmen und Datenstrukturen zu nutzen, die auf zweierlei Art skalieren. Zum einen müssen sie mit der Menge der zugrunde liegenden Daten skalieren und zum anderen mit der Zahl der Benutzer. Damit eine Dienst wirklich skaliert, muss er folgende Probleme effizient lösen:

- Anfragen müssen so schnell wie möglich beantwortet werden.
- Die Algorithmen des Dienstes müssen robust gegen etwaige Ungenauigkeiten der Eingabe sein.
- Vorberechnungen müssen gegebenfalls schnell sein, dabei aber mit moderaten Speicherverbrauch zu recht kommen. Zudem soll es möglich sein sich ändernde Daten regelmäßig neu vorauszuberechnen.

Die vorliegende Arbeit untersucht Datenstrukturen und Algorithmen für Teilservices eines Kartendienstes, die die oben genannten Punkt erfüllen.

Ergebnisse

Diese Arbeit untersucht neue und erweiterte Algorithmen und Datenstrukturen, die verschiedene Aspekte eines Kartendienstes skalierbar implementieren. Diese Entwürfe folgen dabei dem Paradigma des Algorithm Engineering. Neben dem Entwurf und er theoretischen Analyse wird auch experimentell die Leistungsfähigkeit Lösungsansätze untersucht. Dies heißt, dass die hier vorgestellten Verfahren implementiert und experimentell ausgiebig mit realistischen Daten untersucht werden. Die vorliegende Arbeit untersucht Datenstrukturen und Algorithmen für die folgenden Probleme:

Geokodierung. Ein Kartendienst ist immer nur so gut wie sein Lokalisierungsmechanismus. Es kann sein, dass ein Benutzer nicht die Geokoordinaten zu einem Ort, aber dafür eine Textbeschreibung, bspw. eine Adresse, kennt. Eine exzellente Routingkomponente wird in diesem Sinne nur als exzellent wahr genommen, wenn Start- und Zielorte sinnvoll zugewiesen werden. Beschreibungen, also Adressformate im weiteren Sinne, sind allerdings nicht einheitlich in allen Regionen der Welt [124] und selbst dann können Ortsangaben auch durchaus mehrdeutig sein. Schreibfehler das Problem nicht einfacher.

Zuerst wird ein fehlertoleranter Textindex untersucht. Dieser ist in der Lage Anfragen an ein Wörterbuch approximativ zu beantworten. Anschaulich gesprochen werden einige Schreibfehler toleriert. Der Schwerpunkt liegt hier auf der Analyse eines Tuning-Parameters, der es erlaubt Rechenzeit gegen Speicherplatz zu tauschen und umgekehrt. Im weiteren Verlauf wird dieser Textindex als Baustein benutzt um einen effizienten Geokodierer zu entwerfen. Dieser Geokodierer kann in Echtzeit Textbeschreibungen von Orten in sinnvolle Ortsreferenzen auflösen. Auch hier werden verschiedene Parameter experimentell untersucht.

Routenplanung. Der Teildienst mit der größten Außenwirkung ist zweifelsfrei die Routenplanung. Die Arbeit untersucht wie ein bestehendes und erfolgreiches Verfahren der schnellen Routenplanung, Contraction Hierarchies, effizient auf modernen Rechnern vorberechnet werden kann. Dabei werden zwei Ansätze verfolgt. Zum einen wird gezeigt, wie sich die Vorberechnung auf mehreren CPU-Kernen eines Computers mit wenig Speicherplatz implementieren lässt. Dazu werden bestehende Heuristiken angepasst und sogenannte Speicherhierarchien ausgenutzt. Zum Anderen wird die Vorberechnung auf einen Cluster vieler Computer mit mittlerer Leistungsfähigkeit verteilt. Es wird gezeigt, wie die Vorberechnung von Contraction Hierarchies auf solch einem Cluster durchgeführt wird. Zusätzlich wird untersucht, wie sich auch Anfragen der Benutzer auf einen Cluster verteilen lässt.

Weitere Location Based Services. In diesem Teil der Arbeit werden grundlegende Eigenschaften der Suchdatenstruktur von Contraction Hierarchies ausgenutzt um mehrere Location Based Services im weiteren Sinne zu konstruieren. Zum einen werden bestehende Techniken soweit adaptiert, dass Punkten in einem Straßennetzwerk eine Bewertung zugewiesen werden kann, bspw. wie gut die Dinge des täglichen Lebens von diesen Punkten aus zu erledigen sind. Anschließend wird untersucht wie bei Mitfahrgelegenheiten die Zuweisung von Angeboten und Nachfragen algorithmisch gelöst werden kann. Beispielsweise werden nicht nur perfekte Zuweisungen gesucht, sondern auch sinnvolle, die einen kleinen Umweg enthalten. Analog dazu wird untersucht, wie sinnvolle Alternativrouten in einem Straßennetzwerk schnell und effizient berechnet werden können. Dies ist wichtig um eventuelle Staus zu umfahren. Im letzten Teil dieses Abschnitts der Arbeit wird ein Verfahren vorgestellt, dass schnell und effizient eine Verkehrsumlegung berechnen kann. Dies ist wichtig bei der Verkehrssimulation um aus gegebenen Verkehrmustern Voraussagen ableiten zu können. Alle diese Verfahren werden experimentell untersucht und der Einfluss gegebener Parameter herausgearbeitet.

Distanzorakel. Im letzten und innovativsten Teil dieser Arbeit wird ein sehr schnelles Distanzorakel auf Basis von Contraction Hierarchies konstruiert. Es wird gezeigt, dass es möglich ist in (nahezu) konstanter Zeit Distanzanfragen auf einem Straßennetzwerk zu beantworten ohne auf geometrische Informationen zurückzugreifen. Dies war ein länger offen stehendes Problem. Die Korrektheit des Verfahrens wird theoretisch gezeigt. Zudem gibt es eine ausgiebige experimentelle Analyse. Diese zeigt, wie sich der Aufwand für die Konstruktion des Orakels in Abhängigkeit von den vorhandenen Parameter verhält und welche Parameterkombinationen beste Ergebnisse liefern.