Timetable Information and Shortest Paths

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Frank Schulz

aus Biberach an der Riß

Tag der mündlichen Prüfung: 25. Januar 2005

Erste Gutachterin: Frau Prof. Dr. Dorothea Wagner

Zweiter Gutachter: Herr Prof. Dr. Peter Widmayer

Contents

Acknowledgements

The author would like to thank the reader for the interest, and, in particular, the first two readers Dorothea Wagner and Peter Widmayer for refereeing the thesis. Thanks to my supervisor Dorothea Wagner I was able to start working for this thesis in September 2000, at that time at the university of Konstanz. All these years she supported me at the best: The working conditions were always perfect, embedded in a great "algo" work group; I was able to attend many workshops and conferences, and could also spend half a year abroad; and among many other things I want to thank for the (sometimes necessary) advice to cut my hair. Through the EU-project AMORE, which was focused on algorithms and methods for optimising the railways in Europe, I got to know Peter Widmayer, who was along with Dorothea Wagner one of the project leaders. I was very glad that he agreed to referee this thesis.

Many thanks also to all the people that contributed to this thesis—by doing joint research or by just being there. In particular, I want to thank: my officemate Thomas, with whom I guess I have spent more time than with any other person the last four years; my friend Grigoris, aka my Hellenic accountant and Karelia supplier; Christos, my Hellenic supervisor, who gave me great support not only during the time I was in Patras; Karsten, who laid the foundation of this thesis by inspiring the preceding work we did in Konstanz; and Evangelia, Martin, and Ulrik, without whose joint work this thesis wouldn't exist as it is.

Vielen Dank an alle die mich die letzten Jahre begleitet und unterstützt, und so am Entstehen dieser Arbeit mitgeholfen haben!

Frank Schulz
Karlsruhe, February 2005

Chapter 1

Introduction

Since the first railway lines have been constructed, mobility gained importance in modern life, and public transportation systems have grown to huge and complex networks. Until the late eighties itineraries had to be planned manually using printed railway guides. In small railway networks (e.g., the timetable in the year 2001 of the Greek railways spans a few tens of pages) optimal itineraries can be quite easily determined manually. However, in larger timetables (e.g., in the year 1957, the "Kursbuch" published by the German railways had already 1 272 pages), manual planning gets difficult and time-consuming, especially if also local traffic shall be included and optimal connections are wanted. In the late eighties of the last century, the first electronic timetable information systems were established in Germany. Current systems are for example HAFAS [HAF], which is used by many European railway companies, or EFA [EFA], which is mainly used for local traffic limited to smaller regions in Europe. Empirically, the resulting connections are in the majority of cases satisfying. There are cases, however, for which the suggested itineraries are clearly not optimal. The main reason for such non-optimal connections is that the algorithms behind the systems employ heuristic methods to reduce the search space (in order to achieve an acceptable response time) that do not always guarantee optimal solutions.

In the 15 or 20 years that passed by since the first timetable information systems have been developed, the performance of computers has been drastically improved. State-of-the-art computers—with clock rates of several gigahertz and many gigabytes of main memory—have reached (or will reach soon) the point where the running time of *optimal* algorithms is feasible for application in a productive timetable information system. In this thesis, we investigate two models that map a timetable information query to *a single shortest-path problem* in an appropriately defined graph (in contrast to most algorithms behind existing systems that split the problem in two or more parts). Provided an optimal shortest path is calculated, also the optimality of the resulting itinerary can be guaranteed. Two issues are crucial for the modelling: On the one hand, many details concerning feasible itineraries have to be included in the graphs, for example, rules for train transfers. On the other hand,

the notion of optimality has to be clearly defined. Different criteria like earliest arrival at the destination or minimisation of train transfers are modelled as edge weights. Besides optimal modelling, the average running time of the algorithms is crucial. We evaluate the algorithms by conducting extensive experimental analyses using real-world data.

Not only the brute force of the modern high-performance computers can be utilised to compute optimal connections, but also algorithmic improvements of standard shortest-path algorithms are fruitful by reducing the running time. Since we translated the timetable information problem to shortest path problems, we can use variants of Dijkstra's classical shortest-path algorithm. The running time of these algorithms applied to real-world instances can be considerably improved by speedup-techniques while still optimal solutions can be guaranteed—a typical approach in the domain of Algorithm Engineering. We introduce such a speedup-technique, called the *multi-level graph approach*, which is based on a hierarchical decomposition of the underlying graph. Also, the combination with other known techniques of improving the running time of Dijkstra's algorithm is an issue.

The final part of the thesis is motivated by the hierarchical decompositions needed for the previously mentioned speedup-technique, and is not directly related with timetable information anymore. Generally, it cannot be guaranteed to find a "good" decomposition of a given graph. However, Lipton and Tarjan [LT79] introduced the first linear-time algorithm for determining good *separators in planar graphs*, and many improvements have been suggested later on. These algorithms compute a set of separator nodes, whose removal decomposes the graph into at least two components, and guarantee upper bounds on the size of the separator and on the size of the components. For theoretical analyses, these worst-case bounds are enough, but for practical application, separator algorithms that are optimised towards finding possibly the best separators (i.e., having a small number of separator nodes and balanced components) are desired. We are engineering the classical algorithms and we will see that a subprocedure is suited as separator algorithm on its own: even though the upper bound is worse than that of the classical algorithms, on almost all of the graphs we have considered, surprisingly good results are achieved.

The outline of the thesis is as follows: After introducing fundamentals regarding shortest path algorithms, graph models incorporating time, and Pareto-optimal shortest paths (Chapter 2), we deal with the modelling of timetable information problems as shortest path problems (Chapters 3 and 4). The second part investigates how these shortest path problems can be solved efficiently (Chapter 5). During the second part, we will come across the problem of decomposing a given graph, which leads us to the topic of the third and final part: Separators in planar graphs (Chapter 6).

Chapter 2

Fundamentals

This chapter introduces basic concepts and algorithms that will be needed later. We review some shortest-path algorithms, address the problem of finding optimal paths when multiple criteria are involved, and finally discuss how to deal with edge lengths that depend on time.

2.1 Single-source Shortest Path Algorithms

We are going to model timetable information problems as shortest path problem in an appropriately defined *graph*. Such a graph is usually referred to as G, and consists of a set of *nodes* V (also called vertices) and a set of directed *edges* $E \subseteq V \times V$ (also called arcs). The number of nodes is denoted by n and the number of edges by m, respectively. An edge has the form (u, v), where $u, v \in V$ are two distinct nodes (i.e., edges are ordered pairs of nodes). Sometimes, we also deal with *undirected* graphs, if the direction of the edges is not important. In that case, an edge has the form $\{u, v\}$ and is a subset of the nodes containing exactly two nodes.

Each edge is assigned a *length*, which will mostly be a natural number for our applications. A path is a sequence $v_1, e_1, v_2, \ldots, e_{k-1}, v_k$ of nodes and edges such that for every i $(1 \leq i < k)$ the edge e_i connects v_i and v_{i+1}: $e_i = (v_i, v_{i+1})$. For simple graphs (i.e., graphs that neither contain loops nor parallel edges), the path is already uniquely defined by the sequence of nodes. The *length* of a path is the sum of the lengths of all edges in the path. A s-t-path from a source node s to a target node t is a path whose first node is s and whose last node is t. If among all such s-t-paths there is none of smaller length, the path is called *shortest path* from the source s to the target t.

Since for the problems we are considering the source node of the shortest path queries will always be given, we only consider *single-source* shortest path problems: find shortest paths from the given source node to all other nodes. In some of our models, also the target will be fixed, and in that case we refer to the problem as *single-pair*. The case of a limited set of target nodes also occurs, referred to as *single-source some-targets* problem: among all paths from the source to one of the target nodes a path with minimal length is wanted (note that only *one* path has to be computed in this case).

These restrictions on the shortest path problems to be solved limit the choice of reasonable algorithms. Since the target is more or less fixed, a shortest path algorithm can be terminated once the shortest path to the target is found. Such a termination can easily be done if the algorithm is *label-setting* (i.e., in each step of the algorithm the shortest path to one node is determined and the label of that node is definitely set). Usually, such an algorithm can be terminated after only a small fraction of all nodes (also referred to as the *search space*) have been processed. Several speedup techniques, for example the goal-directed search, which we describe later in detail (cf. Section 3.4 on pages 24 et seq.), try to improve the running-time by further reducing the search space. In contrast, *label-correcting* algorithms improve in each step the currently known shortest path to every other node, and the algorithm cannot just be aborted as in the label-setting case above. Thus, we focus on label-setting algorithms.

2.1.1 Dijkstra's Algorithm

The classical label-setting algorithm is due to Dijkstra [Dij59], and most of the timetable-information problems are going to be solved by this algorithm and variants of it. The given edge lengths are required to be non-negative.

Each node is assigned a distance label and a marker with the states *unvisited, visited*, and *finished*. The algorithm initialises all nodes to be *unvisited*, sets the distance label of the source node to 0, and marks the source node to be *visited*. Further, the algorithm maintains a *priority queue* that stores the *visited* nodes, and adds the source node to the priority queue at the beginning. The priority in this queue is the distance label of the nodes (the smaller the better). The remainder of the algorithm is a loop that terminates when the priority queue is empty: The node u with the smallest distance label is removed from the queue, and it is marked to be *finished*. Then, all outgoing edges of u are considered one by one and "*relaxed*": Let v be the other node of such an edge. If, on the one hand, v is still *unvisited* or, on the other hand, v is already *visited* but the distance label of u plus the edge length from u to v is smaller than the distance label of v, the distance label of v is set to the distance label of u plus the edge length of the edge from u to v. The node v has to be marked *visited* and inserted in the priority queue unless it was already in that state. If v was already *visited*, it was already in the queue and the priority may have to be updated.

If the algorithm is executed more than once for the same graph, but for a different source node, the initialisation of the marker does not have to be repeated always. Instead, a timestamp technique can be used: Let us assume we are executing the algorithm the i-th time. The marker is encoded by an integer value m as follows: a node is *unvisited* if $m < 2 \cdot i$, it is *visited* if $m = 2 \cdot i$, and the node is marked *finished* if $m = 2 \cdot i + 1$.

2.1.2 Priority Queues for Dijkstra's Algorithm

In the original description of Dijkstra [Dij59], there is not specifically mentioned how to choose the node with the smallest distance label among all *visited* nodes. The naive approach to just scan all *visited* nodes results in a running time quadratic in the number of nodes. For graphs that have quadratic many edges in the number of nodes, this running time is optimal, since also every edge is processed by the algorithm. However, for "sparse" graphs with less edges, the running time can be improved by implementing a better priority queue.

Dial's Implementation

If the edge lengths are natural numbers bounded by a small constant C, a bucket technique suggested by Dial [Dia69] leads to a very efficient algorithm. We maintain an array of C buckets, each of the buckets being able to store a set of nodes. Inserting of a node with distance d is done by adding the node to the (d mod C)-th bucket. Further, we have a counter that reflects the bucket that contained the last node that has been removed. Removing a node is simple: we increase the bucket counter until a non-empty bucket is reached (if we reach the last bucket we continue at the first bucket in a circular way), and take a node from this bucket. If the distance of a node has to be updated, we remove the node from the bucket it is currently stored in, and insert it with the new distance in another bucket.

The running time of Dial's implementation of Dijkstra's algorithm is $\mathcal{O}(m+n\cdot C)$, because it is possible that for each removal of a node from the queue, the bucket counter has to be increased $C-1$ times. In practice, especially if C is small, Dial's implementation is one of the most efficient implementations of Dijkstra's algorithm with integral edge lengths [HD88].

Fibonacci and Pairing Heaps

In the pointer model of computation, the use of a Fibonacci heap [FT87] as priority queue gives the theoretically best bound for Dijkstra's algorithm of $\mathcal{O}(m+n\log n)$. With this heap the insert and remove operations can be done in $\mathcal{O}(\log n)$ time, and the decrease-key operation (update of priority) is possible in amortised constant time.

The pairing heap [FSST86] is a relative of the Fibonacci heap. The focus of the pairing heap is an easy and efficient implementation. The disadvantage with respect to Fibonacci heaps is that for pairing heaps, the decrease-key operation cannot be guaranteed to run in constant time [Fre99]. However, experimental studies [SV87, Lia92] indicate that pairing heaps are in practice very efficient, and a good choice to be used in Dijkstra's algorithm, especially in the general case of non-integral edge lengths.

A pairing heap is implemented as a single tree, where each tree node contains one element of the heap. It respects the heap order, i.e., the priority of the children is not smaller than the priority of the parent node. The element with the smallest priority is stored in the root of the tree. The operations insert, remove, and decrease-key

are implemented as a series of linking operations. See [FSST86] for the details.

2.1.3 Other algorithms

Being a fundamental topic in computer science, there are many different shortest path algorithms for a variety of different versions of the problem. Besides various textbooks presenting the classical algorithms [AMO93, CLRS01, Len90, Meh84], surveys on more recent work and experimental comparisons as well as engineering aspects can be found in [Zwi01, Gol01a, CGR96].

An important issue in recent research about single-source shortest path algorithms is the design of algorithms with a better running time than $\mathcal{O}(m+n\log n)$ in a RAM model of computation with word operations. Some of these new algorithms are of more theoretical interest, like Thorup's linear-time algorithm for undirected graphs [Tho99], but also from a practical point of view there are interesting results, like the average-case linear-time algorithms proposed by Meyer [Mey01] and Goldberg [Gol01b]. The latter algorithms use hierarchically organised buckets as data structures to store already visited nodes.

Another direction of recent work is the development of speedup techniques for single-pair shortest path algorithms applied to large and sparse graphs (see [WW] for a survey). These techniques compute in a preprocessing step information about shortest paths. The space consumption to store this information is usually linear in the number of nodes (must be less than quadratic, since otherwise the complete distance matrix could be stored). Then, to answer a query, the pre-computed information can be used to limit the search space in Dijkstra's algorithm or other shortest-path algorithms. In Chapter 5 we will discuss such a technique in detail and present an experimental study with data from timetable information, and also consider other known speedup techniques of that kind.

We conclude the discussion of shortest path algorithms with a linear time shortest-path algorithm for a very special graph class that will occur in the course of this thesis.

Shortest paths in dags. A directed graph G that doesn't contain any directed cycle is called a directed acyclic graph, or simply *dag*. In dags, the single-source shortest path problem can be solved in linear time: The nodes of the dag are first topologically ordered. Such an ordering of the nodes requires that every edge points forward (i.e., all neighbours of each node appear always later in the ordering). It can be constructed by a depth-first search in linear time. Then, the shortest path can be found by processing the nodes of the graph in the topological order and relaxing the outgoing edges as in Dijkstra's algorithm. Since the algorithm doesn't need to maintain a priority queue and every edge is touched at most once, the total running time is linear in the size of G. For a more detailed description of this algorithm we refer the reader to [CLRS01].

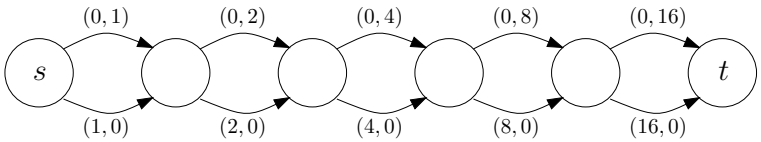

Figure 2.1: Example of a graph with $p(t) = 2^{n-1}$ many Pareto-optimal s-t-paths (here $n = 6$ and $p(t) = 32$; example taken from [Möh99]).

2.2 Multi-Criteria Shortest Paths

Later on we will also have to deal with more than one different edge lengths, also referred to as different *optimisation criteria* (e.g., the travel time and the number of transfers in a graph representing a railway network): Let $l_1(e), \dots l_k(e)$ be the k different edge lengths. The length of a path P is defined to be the sum of the tuples $(l_1(e), \dots, l_k(e))$ over all edges e on the path P. It is not clear how to define a single "shortest path" anymore, especially if all different length measures are considered to be equally important. Instead, one can introduce the notion of *dominating paths*: A path P dominates another path Q if for one length l_i the path P is better than Q, and for all other lengths l_j the path P is at least as good as Q. More formally, we define path domination as follows:

Definition 2.1 *A s-t-path P with length $(p_1, \dots p_k)$* dominates *another s-t-path Q with length $(q_1, \dots, q_k)$ if $p_i < q_i$ for some i $(1 \leq i \leq k)$, and $p_j \leq q_j$ for all other j $(1 \leq j \leq k, i \neq j)$.*

Dominated paths are not optimal in the sense that there is a better path which improves one criterion, while all other criteria remain at least as good. Hence, we are interested in all non-dominated paths:

Definition 2.2 *A s-t-path is called* Pareto-optimal[1] *if it is not dominated by any other s-t-path.*

It is long known that the problem to determine all Pareto-optimal paths is (weakly) $\mathcal{NP}$-hard in general [War87]. We denote by $p(v)$ the number of Pareto-optimal s-v-paths. In general, $p(v)$ may be exponential in the number of nodes of the graph, even if $k = 2$, as Figure 2.1 shows. In the following we outline some algorithms for computing Pareto-optimal paths, which run for special cases in polynomial time, and for the general case in exponential time. As we will see later, the latter algorithms can be efficient enough in practice, since often the number of Pareto-optimal paths is quite small (see for example [MHW01] for an experimental study).

[1] The notion of a *Pareto optimum* dates back to Vilfredo Pareto. In [Par06] he writes: "We will say that the members of a collectivity enjoy maximum ophelimity in a certain position when it is impossible to find a way of moving from that position very slightly in such a manner that the ophelimity enjoyed by each of the individuals of that collectivity increases or decreases. That is to say, any small displacement in departing from that position necessarily has the effect of increasing the ophelimity which certain individuals enjoy, and decreasing that which others enjoy, of being agreeable to some, and disagreeable to others."

2.2.1 Lexicographically First Solution

For every order of the criteria (assume the criteria $l_1, \ldots, l_k$ are in such an order), there is one special Pareto-optimal path: the one that is *lexicographically optimal* (i.e., minimal with respect to the lexicographical order defined on the path lengths). The lexicographically optimal path can be computed by Dijkstra's algorithm by maintaining tuples $(d_1, \ldots, d_k)$ as node labels and using the lexicographical order when the edges are relaxed. It is clear that the lexicographically optimal path is Pareto-optimal, since a dominating path would be lexicographically smaller.

2.2.2 All Pareto-optimal Paths

All Pareto-optimal paths from a node s to all other nodes can be determined by a *labelling algorithm* similar to Dijkstra's algorithm (cf. [Möh99, The95, Zie01]). Instead of maintaining one label per node, the labelling algorithm maintains a set of labels for each node. A single label of a node v is, as above for the lexicographically first solution, a tuple $(d_1, \ldots, d_k)_v$. The set of labels at a node v at some stage of the algorithm represents the non-dominated paths from s to v found so far. The differences to Dijkstra's algorithm are: (i) initially, the set of labels for each node is empty, and the label for s is set to 0^k; (ii) instead of storing nodes in the priority queue, labels $(d_1, \ldots, d_k)_v$ are stored (along with the corresponding node v) and lexicographically ordered; (iii) instead of just updating the labels for the outgoing edges $e = (u, v)$, a new label $(d_1 + l_1(e), \ldots, d_k + l_k(e))_v$ is added to the labels of v, and non-dominated labels are removed from the set of labels of v.

The labelling algorithm considers for every Pareto-optimal s-v-path the outgoing edges of the end-node v, which dominates the running time of the algorithm. This may require exponential running time in the number of nodes, but if the number of Pareto-optimal paths per node is bound by a constant, the asymptotic running time is the same as the one of Dijkstra's algorithm.

In the single-pair case (i.e., the destination node t is known), the labelling algorithm can of course be terminated when all Pareto-optimal paths at t have been found. In particular, in the bicriterion case (i.e., $k = 2$), the shortest s-t-path P' with respect to the second criterion l_2, can be computed beforehand using Dijkstra's algorithm, and the labelling algorithm can be terminated once a Pareto-optimal s-t-path of length (d_1, d_2) has been found with $d_2 = l_2(P')$.

Similar techniques are also used in labelling approaches for the (resource) constrained shortest path problem (cf. [Zie01]), which is also weakly $\mathcal{NP}$-hard [GJ79]. As in the normal shortest-path problem, there is one edge length l per edge. Additionally, edges are assigned k resources, which are also additive along a path, and there is a resource limit for each resource. The goal is to find a shortest path with respect to l that satisfies all resource constraints. During the labelling algorithm, only paths are considered that are *promising*: a path P is non-promising if a resource exceeds the given limit, or if the length $l(P)$ exceeds an upper bound on an optimal solution.

2.3 Graphs and Time

Time is obviously an important issue in timetable information. First, we define the discrete time values used in the following chapters, and after that review models for time-dependent shortest paths introduced by Orda and Rom. We also throw a first glance at the concept of time-expansion.

2.3.1 Time Values

Time will be modelled in this thesis always as discrete time values representing minutes. Sometimes, only the time within one day is of interest and the time values are in the interval $[0, 1439]$; in other occasions also the day is encoded in the time value by adding 1440 times number of days between a given reference day and the actual day. The formal definition is as follows.

Definition 2.3 *A* time value $t \in \mathbb{N}$ *represents minutes past midnight of a specific reference day. Such a time value* t *is of the form* $t = a \cdot 1440 + b$*, where* $a \in \mathbb{N}$ *and* $b \in [0, 1439]$*. Hence, the actual* time within a day *is* $t \pmod{1440}$ *and the actual* day *is* $\lfloor t/1440 \rfloor$*.*

Given two time values t and t' we often need to refer to the time that passes starting at the time within the day represented by t until the time within the day represented by t' is reached (regardless of the actual day that t and t' represent). We refer to this time difference as *day-diff*(t, t'). For example, if t represents two o'clock in the afternoon and t' ten o'clock in the morning, *day-diff*(t, t') is 20 hours. Note that this time difference is not symmetric, for example, using the above sample values the *day-diff*(t', t) is 4 hours.

Definition 2.4 *Given two time values* t *and* t'*, the* day-diff(t, t') *is the smallest nonnegative integer* l *such that* $l \equiv t' - t \pmod{1440}$*.*

2.3.2 Time-Dependent Shortest Paths

In [OR91] Orda and Rom investigate an approach of modelling shortest paths in graphs whose edge weights depend on time. Let T denote a set representing time (in the case of timetable information usually $T = \mathbb{N}$). Whenever an edge is passed, the *delay* on this edge depends on time and is modelled as a function $d_e : T \to T$. Further, edge *weights* are also functions of time, for example $w_e : T \to \mathbb{R}$. Generally, waiting is allowed at the nodes, but there are also waiting costs at the nodes, which can be used to disallow waiting by using high waiting costs. Given a path and the waiting times at the intermediate nodes, the weight of the path can be calculated using the delay and weight functions. The goal is to find, given a departure node and time, a path (along with waiting times at the nodes) of smallest weight. It turns out that, in the general case, the problem is difficult to solve, even infinite shortest paths are possible, as Figure 2.2 shows.

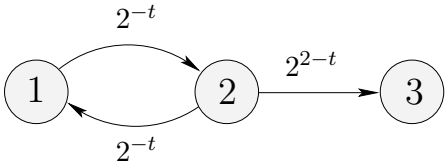

Figure 2.2: Sample time-dependent graph (cf. [OR91]) with an infinite shortest 1-3-path. The edge labels denote the time-dependent weight, while the delay is equal to 1 for each edge.

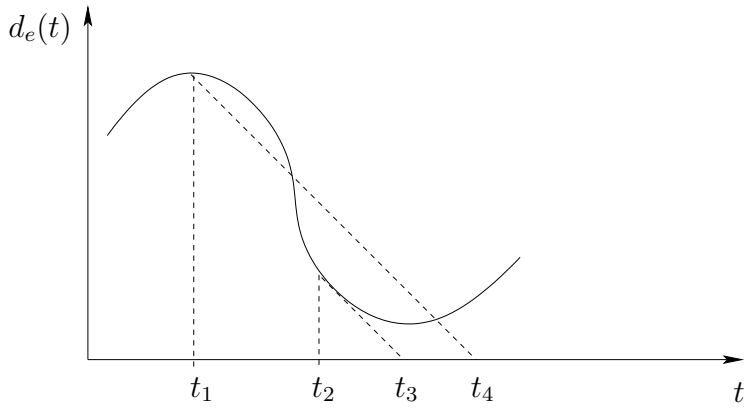

Figure 2.3: Waiting pays off for an edge $e = (u, v)$ and the shown delay function d_e. If t_1 is the arrival time at u, no waiting implies an arrival time of t_4 at the node v, whereas waiting until t_2 yields an arrival at $t_3 < t_4$.

However, Orda and Rom have further shown that in restricted cases, a Dijkstra-like algorithm can be used to determine time-dependent shortest paths [OR90]. In that, there are no weights anymore, and the objective is to minimise the arrival time (i.e., the problem corresponds to finding a *fastest* path). If waiting is forbidden, the problem can be shown to be $\mathcal{NP}$-hard, but if unrestricted waiting is allowed and *all delays are non-negative* (i.e., for all t, $d_e(t) \geq 0$), a modification of Dijkstra's algorithm solves the problem: The distance labels represent arrival times at the nodes, and whenever an edge $e = (u, v)$ is to be relaxed and t_u is the arrival time at the node u, the optimal waiting time τ_e for e can be determined by minimising $t_u + d_e(t_u + \tau_e)$. The latter time value is the tentative arrival time at the node v in Dijkstra's algorithm.

Waiting at nodes may pay off: Consider for example the delay function d_e of the edge $e = (u, v)$ shown in Figure 2.3. In that case, even if the earliest arrival at u is t_1, a waiting time of $\tau_e = t_2 - t_1$ yields the earliest arrival time at v. The possibility of waiting at nodes involves the calculation of optimal waiting times which may be not very efficient. However, for a restricted class of delay functions, the *FIFO* (first in first out) delays, waiting never pays off and the calculation of waiting times is easy. A delay function is *FIFO* if it satisfies the following condition:

$$t_1 \leq t_2 \Rightarrow t_1 + d_e(t_1) \leq t_2 + d_e(t_2).$$

For a FIFO delay function d_e of an edge $e = (u, v)$, the optimal arrival time t_v at v can simply be determined by evaluating $d_e(t_u)$, where t_u is the arrival time at u. We summarise the above discussion by the following lemma.

Lemma 2.1 *Given a time-dependent network with edge delays $d(e)$. If d is* non-negative *and* FIFO*, a shortest path with respect to d (also referred to as* fastest path*) can be computed efficiently by the above mentioned adaption of Dijkstra's algorithm.*

2.3.3 Time-Expansion

Instead of modelling time using delay and weight functions depending on time as described in the previous section, the approach of *time-expansion* translates the delays and weights that depend on time into a static time-expanded graph. This approach requires discrete time values, and in the general case for every time value a copy of the node is maintained in the time-expanded version of the graph. A crucial problem with time-expansion is the huge size of the resulting graph; it may even be impossible to explicitly store such a graph.

In the field of dynamic flow problems time-expansion is widely used: already Ford and Fulkerson [FF58, FF62] applied this approach, and in several recent studies of variants of dynamic flow problems time-expansion was considered (see for example [FET98, FS02, KLS02, FS03]). Since in timetable information time values are discrete, time-expansion can be applied directly. Actually, only for certain time values involving an *event* in the timetable (e.g., a departure or an arrival of a train), a copy of the node has to be maintained in the time-expanded graph. Hence, the resulting graph is not as huge as in the general case. We will discuss time-expansion applied to timetable information in more detail in Section 3.3.1 (pages 18 et seq.).

Chapter 3

The Basic Timetable Information Problem

After discussing previous work, we start the investigation of timetable information with the introduction of the basic timetable information problem: the *earliest arrival problem*. The goal is to find a train connection from a departure station A to an arrival station B that departs at A later than a given departure time and arrives at B as early as possible. We consider throughout this chapter a *simplified version* of the problem in assuming that transfers within a station take negligible time.

We review two approaches of modelling the simplified earliest arrival problem as shortest path problem: the *time-expanded approach* that we have investigated in [SWW00], and the *time-dependent approach* suggested by Brodal and Jacob [BJ04]. New issues concerning modelling and algorithms provided in this chapter are: (i) the concept of the fully time-expanded graph, which will be a useful abstraction when the models are extended in the next chapter; and (ii) an extension of the goal-directed search technique applied to the time-dependent approach. Comparing the two approaches, it is argued theoretically in [BJ04] that the time-dependent approach is more efficient than the time-expanded one. We address the same question from an experimental point of view and provide, as main contribution of this chapter, an experimental comparison of the time-expanded and the time-dependent approaches with respect to their performance, and show that—at least in the simplified version—the time-dependent approach is clearly superior to the time-expanded approach.

Parts of this and the following chapter have been published as "*Experimental Comparison of Shortest Path Approaches for Timetable Information*" [PSWZ04a]. The subsequent chapter discusses on the one hand realistic variants of the earliest arrival problem involving transfer times, and on the other hand also other optimisation criteria than the earliest arrival, for example the number of transfers and multi-criteria problems.

3.1 Previous work

Two main approaches have been proposed for modelling timetable information as shortest path problem: the *time-expanded* [SWW00, MHW01, PS98, Möh99], and the *time-dependent* approach [BJ04, Nac95]. The common characteristic of both approaches is that a query is answered by applying some shortest path algorithm to a suitably constructed graph.

The *time-expanded approach* constructs the time-expanded graph in which every node corresponds to a specific time event (departure or arrival) at a station and edges between nodes represent either elementary connections between the two events (i.e., served by a train that does not stop in-between), or waiting within a station. Depending on the optimisation criterion, the construction assigns specific fixed weights to the edges. This naturally results in the construction of a very large (but usually sparse) graph. The simplified version of the earliest arrival problem—where details like transfer rules and traffic days are neglected—has been extensively studied: In [SWW00] (see also [Sch00]), we explicitly used the time-expanded approach to model the simplified earliest arrival problem as shortest path problem in a static graph and solve the problem optimally, in contrast to the approaches used in practice, which usually cannot guarantee optimal solutions. We conducted an extensive experimental study and were able to demonstrate that—at least in the simplified scenario—the running time of the time-expanded approach on state-of-the-art computers is acceptable. To achieve this result, we applied several speedup techniques, which guarantee optimal solutions, to Dijkstra's algorithm for computing the shortest path.

An extension of the time-expanded approach able to count train transfers is presented by Müller-Hannemann and Weihe along with an experimental study focused on multi-criteria problems in [MHW01]. The results of this study are quite promising: they show that in practice (among other data also the time-expanded graph was considered) the number of Pareto-optimal paths is often very small, and labelling approaches are feasible. In [MHSW02], the same authors together with Schnee investigate the issue of space consumption when more complex real-world scenarios shall be modelled. In a subsequent study, Müller-Hannemann and Schnee extensively investigate multi-criteria optimisation in the time-expanded graph by a labelling approach [MHS]; they relax the notion of Pareto-optimal connections in order to find all attractive train connections. Möhring suggests the time-expanded model as graph-theoretic concept for timetable information in [Möh99]. He further discusses algorithms for solving multi-criteria problems, and focuses on a distributed approach for timetable information, which is also the topic of the recent projects DELFI [DEL] and EU-Spirit [EUS]: the railway network is considered as consisting of several (overlapping) subnetworks (e.g., each subnetwork is operated by a different company or institution), and a global solution is constructed from several subqueries to the conventional timetable information systems operated on the respective subnetworks. In a sense such new systems operate like meta search engines for the web.

The idea of the *time-dependent approach* is to avoid the maintenance of a node

per event. Instead, the time-dependent graph is used in which every node represents a station, and two nodes are connected by an edge if the corresponding stations are connected by an elementary connection. The weights on the edges are assigned "on-the-fly": the weight of an edge depends on the time in which the particular edge will be used by the shortest path algorithm to answer the query. The general idea of time-dependent networks dates back to Orda and Rom [OR90, OR91]; see Section 2.3 (pages 9 et seq.). Brodal and Jacob argued in [BJ01, BJ04] that in the simplified case of the earliest arrival problem, Dijkstra's algorithm considers many redundant edges in the time-expanded approach. They suggest to use a time-dependent network instead and proved by a detailed theoretical analysis of operation counts in both approaches that the time-dependent approach is more efficient than the time-expanded approach. This was the starting point of our experimental comparison of the two approaches that we provide in this chapter. The work of Nachtigal [Nac95] can also be classified as a time-dependent approach to timetable information, but the problem specification is different to the one we consider: given a source station, for all other stations arrival functions depending on the departure time shall be computed. In contrast, we always consider the departure time as part of the query.

Finally, we want to mention two predecessors of "real" timetable information systems in use. Around the year 1988, the Dutch and German train companies started to use electronic timetable information systems. We want to mention two algorithms behind these early systems. At that time, the computers were too slow and had too few memory to dare implementing the whole problem by a shortest path search in a huge time-expanded graph with something like a million of nodes. Heuristics, as they were for example known from AI, were used instead to keep the search spaces small enough. Tulp and Siklòssy describe in [TS88] the TRAINS system, which was used by the Dutch railways (NS) at that time as a prototype: It is based on a graph where nodes represent cities. They distinguish two levels of the network, a "static" level which consists of arcs between nodes representing distances, and a "dynamic" level where the arcs include information about the departure and arrival times of trains. The dynamic level can be regarded as a time-dependent network as described above. The algorithm uses the static level to cut out the "interesting" part of the network, without considering any information about time. Note that this cutting is heuristic in the sense that optimal connections may be lost by that step, which we don't want to tolerate in the models investigated later in this thesis. Then, a train connection is calculated by a modification of Dijkstra's algorithm trying to incorporate time for train changes at stations. Once a connection to the destination station is found, a backward search tries to improve the result (e.g., to find a connection that departs later and has the same arrival time).

Baumann and Schmidt outline in [BS88] an algorithm called ARIADNE, which can be regarded as ancestor of HAFAS [HAF], the timetable information system that is nowadays used by the German railway company (Deutsche Bahn) and many other railway companies worldwide. As in TRAINS, the algorithm considers two different networks: a static graph representing the topographic railway network, and a dynamic network including time, traffic days, train classes etc. The ARIADNE

```
*Z 00461 80____    01
*G IR  8000096 8000112
*A VE 8000096 8000112 000934
8000096 Stuttgart Hbf              1201
8000302 Plochingen           1214 1215
8000127 Goeppingen           1225 1226
8002218 Geislingen(Steige)   1237 1238
8000170 Ulm Hbf              1301 1311
8000943 Biberach(Riss)       1334 1335
8000746 Bad Schussenried     1348 1349
8000014 Aulendorf            1354 1355
8004965 Ravensburg           1409 1410
8000112 Friedrichshafen St.  1425
```

Figure 3.1: Sample raw timetable data: a train on the famous "Schwäbsche Eisebahne" line from Stuttgart via Ulm, Biberach, Meckenbeuren and Durlesbach to Friedrichshafen [Sch] (Meckenbeuren and Durlesbach are not anymore stops of long-distance trains). The first three lines represent general information about the train, followed by one line per station visited by the train.

algorithm works in two phases: The first phase ("Wegesuche") searches feasible paths in the static network by a bidirectional version of Dijkstra's algorithm and outputs a subgraph of the network to be considered in the second phase. Note that again—as in the TRAINS algorithm—optimal solutions may be lost by this step. The second phase ("Zeitsuche") computes on the dynamic, time-dependent version of the network, limited by the subgraph computed in the first phase, several feasible train connections. These are rated according to measures like travel time, quality of trains, direct connection, etc. The authors also mention caching of frequently asked queries to improve the average running time, which shall be less than 6 seconds on average. Nowadays a timetable information server should be able to guarantee average running times in the magnitude of a hundredth of a second.

3.2 Problem Specification

In this section, we provide a mathematical formulation of the basic timetable data we are using in this chapter, and further define the first and most fundamental timetable problem, the *earliest arrival problem (EAP)*. We use terminology from railway transport, but, of course, the problem definitions, models, and algorithms can as well be applied to other domains with similar characteristics.

3.2.1 Data

A *timetable* consists of data concerning: *stations* (or bus stops, ports, etc.), *trains* (or busses, ferries, etc.) connecting stations, and *departure* and *arrival times* of trains at stations. More formally, we are given a set of *trains* $\mathcal{Z}$, a set of stations $\mathcal{B}$, and a set of *elementary connections* $\mathcal{C}$ whose elements c are 5-tuples of the form $c = (Z, S_1, S_2, t_d, t_a)$. Such a tuple (elementary connection) is interpreted as train Z leaves station S_1 at time t_d, and the *immediately next* stop of train Z is station S_2 at time t_a. If x denotes a tuple's field, then the notation $x(c)$ specifies the value of x in the elementary connection c.

The *departure* and *arrival times* $t_d(c)$ and $t_a(c)$ of an elementary connection $c \in \mathcal{C}$ within a day are integers in the interval $[0, 1439]$ representing time in minutes after midnight. The *length* of an elementary connection c, denoted by $\mathit{length}(c)$, is $\mathit{day\text{-}diff}(t_d(c), t_a(c))$, and reflects the time that passes between the departure and the arrival (see the definition of *day-diff* provided in Section 2.3.1 on page 9). Figure 3.1 shows a sample data set in the raw input format.

3.2.2 The Earliest Arrival Problem

Let $P = (c_1, \ldots, c_k)$ be a sequence of elementary connections together with departure times $dep_i(P)$ and arrival times $arr_i(P)$ for each elementary connection c_i, $1 \leq i \leq k$. We assume that the times $dep_i(P)$ and $arr_i(P)$ include data regarding also the departure/arrival day by counting time in minutes from the first day of the timetable. A time value t is of the form $t = a \cdot 1440 + b$, where $a \in [0, 364]$ and $b \in [0, 1439]$. Hence, the actual time within a day is $t \pmod{1440}$ and the actual day is $\lfloor t/1440 \rfloor$ (cf. the definition of time values in Section 2.3.1)

Such a sequence P is called a *consistent connection* from station $A = S_1(c_1)$ to station $B = S_2(c_k)$ if it fulfils the following consistency conditions: the departure station of c_{i+1} is the arrival station of c_i, and the time values $dep_i(P)$ and $arr_i(P)$ correspond to the time values t_d and t_a, resp., of the elementary connections (modulo 1440), and the departure of the each elementary connection (except the first one) is later than the previous arrival. More formally, P is a *consistent connection* if the following conditions are satisfied:

$$\begin{aligned} S_2(c_i) &= S_1(c_{i+1}) \\ dep_i(P) &\equiv t_d(c_i) \pmod{1440} \\ arr_i(P) &= \mathit{length}(c_i) + dep_i(P) \\ dep_{i+1}(P) - arr_i(P) &\geq 0 \end{aligned}$$

Note that by this definition we assume that it is possible to transfer at any station to any other train that departs later than or equal to the arrival time at the specific station. We further assume that all trains are operated daily. Because of these simplifications, we refer to the problem defined here also as the *simplified* earliest arrival problem. In the next chapter we discuss how more realistic transfer rules as well as traffic days specifying operation days for trains can be incorporated.

We are additionally given a large, on-line sequence of *queries*. A query generally

defines a set of valid connections, and an optimisation criterion (or criteria) on that set of connections. The problem is to find the optimal connection (or a set of optimal connections) w.r.t. the specific criterion or criteria. For the earliest arrival problem, a query (A, B, t_0) consists of a departure station A, an arrival station B, and a departure time t_0. The departure time is in $[0, 1439]$ for the simplified version. Connections are *valid* if they depart at least at the given departure time t_0, and the optimisation criterion is to minimise the difference between the arrival time and the given departure time.

3.3 Basic Modelling Approaches

We review the modelling of the simplified version of EAP for two approaches: the time-expanded and the time-dependent approach. In addition, we discuss some further insights and similarities of the approaches.

3.3.1 Time-Expanded Model

The time-expanded model, as it is described in [SWW00], follows the principle of time-expansion mentioned in Section 2.3.3 (page 11) and is based on the *time-expanded* graph which is constructed as follows. There is a node for every time *event* (departure or arrival) at a station, and there are two types of edges. For every elementary connection (Z, S_1, S_2, t_d, t_a) in the timetable, there is a *train-edge* in the graph connecting a *departure-node*, belonging to station S_1 and associated with time t_d, with an *arrival-node*, belonging to station S_2 and associated with time t_a. In other words, the endpoints of the train-edges induce the set of nodes of the graph. For each station S, all nodes belonging to S are ordered according to their time values. Let $v_1, \ldots, v_k$ be the nodes of S in that order. Then, there is a set of *stay-edges* (v_i, v_{i+1}), $1 \leq i \leq k-1$, and (v_k, v_1) connecting the time events within a station and representing waiting within that station. The length of an edge (u, v) is *day-diff*(t_u, t_v), where t_u and t_v are the time values associated with u and v, respectively. Figure 3.2 illustrates this definition. The following theorem states that an appropriate shortest path computation on the above graph solves the simplified version of EAP.

Theorem 3.1 *A shortest path in the time-expanded graph from the first departure-node s at the departure station A with departure time later than or equal to the given start time t_0 to one of the arrival-nodes of the destination station B constitutes a solution to the simplified version of EAP in the time-expanded model.*

Proof. To prove correctness, it suffices to show that: (i) there is a one-to-one correspondence between a valid and consistent A-B-connection and an A-B-path in the time-expanded graph; and (ii) the valid and consistent connection constructed from a shortest path is optimal in the EA sense (i.e., the difference of the arrival time and the given start time t_0 is minimal).

(i) Let Π be a path from node s of station A to some arrival-node of station B. We will show that Π corresponds to a valid and consistent A-B-connection P in the EAP setting. Define P to be the sequence of all elementary connections corresponding to the train-edges that occur in the path Π. Further, define the time value $dep_i(P)$ to be the length of the subpath from s to the tail of the i-th train-edge plus the time associated with s. Similarly, set $arr_i(P)$ to be the length of the subpath from s to the head of the i-th train-edge plus the time associated with s. It can be easily verified that P is a valid and consistent connection, whose duration equals the length of Π.

Conversely, let P' be a valid and consistent A-B-connection departing from A at t_0 or at the earliest possible time after t_0, and let the corresponding departure-node be s. We will show that P' corresponds to an A-B-path Π' starting from s and ending at some arrival-node of B in the time-expanded graph. Π' is constructed as follows. It contains all train-edges corresponding to the elementary connections in P'. Π' starts at s, which is connected by a *simple* path of stay-edges with the tail of the first train-edge. We can assume without loss of generality that all stays are less than a day and correspond to simple paths of stay-edges in the time-expanded graph, because when every train operates daily, an optimal connection never stays more than a day at a station (otherwise there would be a one day faster connection). When traffic days shall be modelled, this is not necessarily true anymore; see Section 4.2 (pages 43 et seq.). The head of the first train-edge is connected by a path of stay-edges to the tail of the second train-edge, and so on, until the tail of the last train-edge is reached. Clearly, the head of the last train-edge is an arrival-node of B. It can be easily verified that the length of Π' is equal to the difference of the arrival and the departure time of P'.

(ii) Let P be a valid and consistent connection (departing from A at t_0 or at the earliest possible time after t_0) that corresponds to a shortest A-B-path Π in the time-expanded graph (starting from departure-node s). Assume on the contrary that there is an A-B-connection (valid and consistent) P', departing from A at t_0 or at the earliest possible time after t_0, and arriving to B earlier than P. Let Π' be the A-B-path in the time-expanded graph corresponding to P'. The length of Π' is equal to the difference of the arrival and the departure time of P', which is by assumption less than the difference of the arrival and the departure time of P, and consequently also smaller than the length of Π. This, however, is a contradiction to the fact that Π is a shortest path. ∎

The Fully Time-Expanded Graph

Instead of introducing stay-edges from the last node to the first node at every station, the time-expanded graph can also be fully expanded by maintaining a copy of every event for each day in the timetable period. This reflects the general idea of time-expansion described in Section 2.3.3 more directly. An explicit construction of such a graph is usually not possible (it would be too large for practical instances), but it has the nice property that there is a one-to-one correspondence between all connections—not only those with stays of less than a day—and paths.

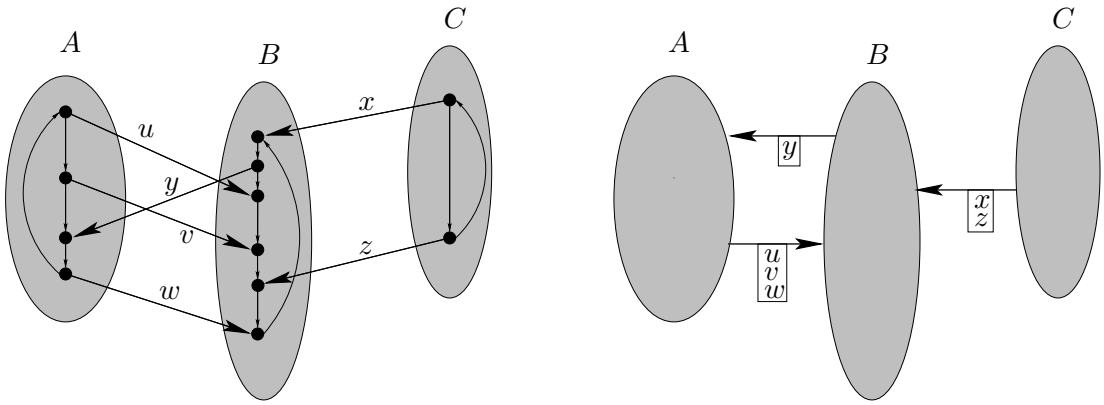

Figure 3.2: The time-expanded graph (left) and the time-dependent graph (right) of a timetable with three stations A, B, C. There are three trains that connect A with B (elementary connections u,v,w), one train from C via B to A (x,y) and one train from C to B (z).

If the timetable is valid for N days, the *fully time-expanded graph* is based on N copies of the time-expanded graph. Whenever there is an overnight edge in the i-th copy, the edge is redirected to the corresponding node in the $(i+1)$-st copy, for $i < N$; in the N-th copy, overnight edges are deleted. In the following, we assume that each node in the fully time-expanded graph is not only assigned the time of the day t_d in the interval $[0, 1439]$, but also the absolute time in the timetable, i.e., a node in the copy corresponding to day i is assigned time $t = t_d + i \cdot 1440$.

There is an obvious one-to-one correspondence between (valid and consistent) connections and paths in the fully time-expanded graph. Thus, it is also obvious that an appropriate shortest path in the fully time-expanded graph provides a solution to the earliest arrival problem. There are two further interesting properties of the fully time-expanded graph: The first observation is that the graph doesn't contain any directed cycles and therefor is a *dag* (a directed acyclic graph). The second property is that any two s-t-paths connecting the same two nodes s and t have the same length.

Algorithm

These two properties suggest that the shortest path can be found in linear time: In dags, one can use a topological order of the nodes as described in Section 2.1.3 (page 6). Further, if two s-t-paths have always the same length, any graph search in the fully time-expanded graph, like breadth first search for example, would suffice to determine the first reachable node at the target station.

However, applying directly such procedures would require the huge fully time-expanded graph. Instead, we suggest to apply Dial's implementation of Dijkstra's algorithm to the time-expanded graph and argue below that it benefits from the special properties: Since edge lengths are non-negative, the actual shortest path in the time-expanded graph can be found by Dijkstra's algorithm (cf. Section 2.1.1).

All edge lengths are natural numbers less than or equal to 1440, so we can use Dial's bucket implementation of the priority queue (cf. Section 2.1.2, page 5) and thus the running time is linear in the number of nodes. Actually, only a fraction of the nodes is processed because the main loop can be aborted when a node at the destination station is reached.

Dijkstra's algorithm processes the nodes in the order of distance from the source node. In fact, all reachable nodes from the start node are processed ordered by time. On the one hand, ordering all nodes of the fully time-expanded graph by time is obviously a topological order, since there is no edge backward in time. Thus, Dijkstra's algorithm processes the nodes of the fully time-expanded graph in a topological order. On the other hand, searching the reachable nodes of the graph by increasing time is a very good idea, since we can abort the search once we reach the very first node at the destination station, which is exactly what Dijkstra's algorithm applied to the time-expanded graph does. Hence, applied to the fully time-expanded graph with its special structure, Dial's implementation of Dijkstra's algorithm is a linear-time graph search in topological order that can be terminated when the first node at the destination station is reached.

3.3.2 Time-Dependent Model

The time-dependent model suggested in [BJ04] uses the time-dependent shortest paths investigated by Orda and Rom; see Section 2.3.2 (page 9). It is also based on a directed graph, called *time-dependent* graph. In this graph, there is only one node per station, and there is an edge e from station A to station B if there is an elementary connection from A to B. The set of elementary connections from A to B is denoted by $\mathcal{C}(e)$. The definition is illustrated in Figure 3.2. The length of an edge $e = (v, w)$ depends on the time at which this particular edge will be used by an algorithm which solves EAP. In other words, if T is a set denoting time, then the length of an edge $e = (v, w)$ is given by $\ell_e(t) = f_e(t) - t$, where t is the departure time at v, $f_e : T \to T$ is a function such that $f_e(t) = t'$, and $t' \geq t$ is the earliest possible arrival time at w. The time-dependent edge length l_e is a delay function in the terminology of time-dependent shortest paths introduced in Section 2.3.2. A sample delay function l_e as defined above is depicted in Figure 3.3.

As we have already seen in Lemma 2.1 (page 11), a time-dependent shortest path can be found easily when the delay functions are non-negative and FIFO: a modification of Dijkstra's algorithm can be used, where the delay functions have to be evaluated. In our case each time-dependent edge length ℓ_e is non-negative by construction; we have to ensure that it is FIFO:

$$t \leq t' \Rightarrow t + \ell_e(t) \leq t' + \ell_e(t').$$

Using the earliest arrival functions f_e, a simpler and equivalent condition is that f_e is *non-decreasing*:

$$t \leq t' \Rightarrow f_e(t) \leq f_e(t').$$

The meaning of the condition is that overtaking of trains on an edge is not allowed, and from now on we will assume that the condition is always fulfilled, which is

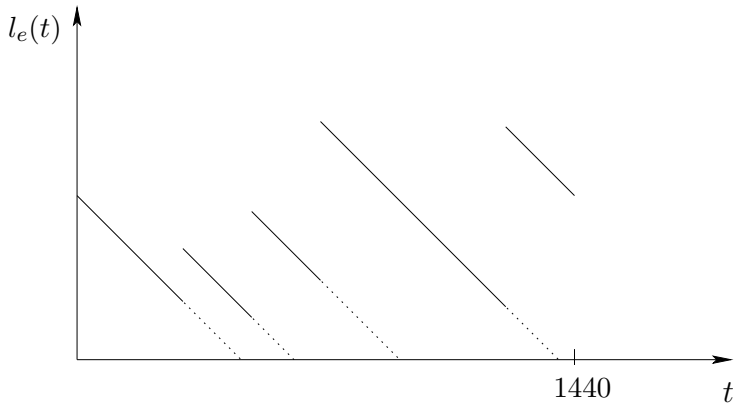

Figure 3.3: Sample delay function l_e used in the time-dependent model for the earliest arrival problem. The dotted lines intersecting with the abscissa indicate the arrival times $f_e(t)$ at the arrival station, and jumps occur whenever a train departs. After one day (1440 minutes) the function is repeated, i.e., $l_e(t + 1440) = l_e(t)$.

true for our data. It is also reasonable, since if ever a train overtakes another train somewhere between two stations, the first train is probably a fast high-speed train and the second a slower train; these two trains usually don't stop at the same stations (the slower train stops more frequently), and thus they don't belong to the same edge.

Assumption 3.1 *For any two given stations A and B, there are no two trains leaving A and arriving to B such that the train that leaves A second arrives first at B.*

Algorithm

Next, we explain the modification of Dijkstra's algorithm that can be used to solve the earliest arrival problem in the time-dependent model in more detail (see also [BJ04]). Let D denote the departure station and t_0 the earliest departure time. The differences, with respect to Dijkstra's algorithm, are: Set the distance label $\delta(D)$ of the starting node corresponding to the departure station D to t_0 (and not to 0), and calculate the edge lengths on-the-fly. The edge lengths are calculated as follows. Since Dijkstra's algorithm is a label-setting shortest-path algorithm, whenever an edge $e = (A, B)$ is considered, the distance label $\delta(A)$ of node A is optimal. In the time-dependent model, $\delta(A)$ denotes the earliest arrival time at station A. In other words, we indeed know the earliest arrival time at station A whenever the edge $e = (A, B)$ is considered, and therefore we know at that stage of the algorithm (by Assumption 3.1) which train has to be taken to reach station B via A as early as possible: the first train that departs later than or equal to the earliest arrival time at A. Let $t = \delta(A)$ and let $c^* \in \mathcal{C}(e)$ be the connection minimising $\{\mathit{day\text{-}diff}(t, t_d(c)) \mid c \in \mathcal{C}(e)\}$. The particular connection c^* can be easily found by binary search if the elementary connections $\mathcal{C}(e)$ are maintained in a sorted array. The edge length of e, $\ell_e(t)$, is then defined as $\ell_e(t) = \mathit{day\text{-}diff}(t, t_d(c^*)) + \mathit{length}(c^*)$.

Consequently, $f_e(t) = t + \ell_e(t)$. Note that actually the lengths $\ell_e(t)$ don't have to be explicitly calculated, since in Dijkstra's algorithm only the distance (resp. arrival time) $f_e(t)$ at the target node (resp. station B) is needed, which we can get directly by the arrival time $t_a(c^*)$ of the corresponding connection.

The following theorem is proved explicitly in [BJ04] and can also be derived from the general results on time-dependent shortest paths (cf. Section 2.3.2). Its correctness is based, as we have discussed already above, on the following two facts: (i) the edge lengths are non-negative, which means in other words that all f_e have *non-negative delay* (for all t, $f_e(t) \geq t$), and is true by definition; and (ii) the edge lengths are FIFO, which means in our case that all f_e are *non-decreasing* ($t \leq t' \Rightarrow f_e(t) \leq f_e(t')$), and follows directly from Assumption 3.1. Because of the nature of the investigated application, we can safely assume that all functions we are considering later on have non-negative delay, too.

Theorem 3.2 *The above modified Dijkstra's algorithm solves the simplified version of EAP in the time-dependent model, provided that Assumption 3.1 holds.*

3.3.3 Comparison of the Approaches

In the simplified scenario we are investigating in this chapter, the graphs that are used in the two approaches are strongly related: Contracting all nodes that belong to the same station in the time-expanded graph and deleting parallel edges afterwards yields the time-dependent graph. Further, the algorithm used in the time-dependent approach can be viewed as an improved implementation of the simple shortest-path search by Dijkstra's algorithm in the time-expanded approach: If the first edge from some station A to another station B has already been processed by Dijkstra's algorithm in the time-expanded graph, all other edges e'_{AB} from station A to station B do not have to be considered anymore. The reason is that such an edge doesn't provide an improvement since the path through the first edge extended by some stay-edges to the head of the edge e'_{AB} has the same length. In a sense, the time-dependent algorithm implements this observation.

However, it is not immediately clear whether the time-dependent approach is superior. On the one hand, the edge lengths have still to be computed in the time-dependent algorithm, which consumes running time as well. In [BJ04] it is argued theoretically that in the time-dependent model less elementary operations have to be executed. We are going to approve this theoretical observation by means of an experimental analysis in Section 3.5. On the other hand, the similarity of the graphs and the algorithms in the two approaches is disturbed when the realistic specifications are incorporated into the models in the following chapter, which has also implications on the running time of the algorithms: the time-dependent algorithm is not anymore that faster than the time-expanded algorithm.

3.4 Basic Speedup Techniques

Now that we have seen how the basic timetable information problem can be modelled, we turn our focus on how the running time of the algorithms can be improved by basic speedup techniques. Note that by the notion of *speedup technique* we always refer to a strategy to improve the running time of a shortest-path algorithm that *always guarantees the optimality of the solution*. On the one hand, a well-known speedup technique for shortest path algorithms called goal-directed search is applied to both models (for the time-dependent model we suggest a modification of it), and on the other hand basic improvements of the models themselves and the model-specific algorithms are introduced. Later, in Chapter 5 (pages 61 et seq.), more sophisticated techniques are discussed. In the rest of the section we briefly review the goal-directed search for the time-expanded model and present our further modifications and model-specific techniques in more detail.

3.4.1 Time-Expanded Model

Goal-Directed search

A commonly used speedup technique is the goal-directed search, also referred to as the A^* algorithm or the method of potentials, introduced originally by Hart, Nilsson and Raphael [HNR68] in the field of artificial intelligence (see also [Len90]). The intuition behind this method is to direct the search carried out by Dijkstra's algorithm towards the destination. The length of every edge is modified in a way that if the edge points towards the destination its length gets smaller, while if the edge points away from the destination node, then its length gets larger. More precisely, for an edge (u, v) with length $\ell(u, v)$, its new length $\ell'(u, v)$ becomes $\ell'(u, v) = \ell(u, v) - p[u] + p[v]$, where $p[\cdot]$ is a potential function associated with the nodes of the graph. A potential function $p[\cdot]$ is called *valid* if $\ell'(u, v)$ is non-negative. The optimality of the goal-directed search with valid potentials is shown by the next lemma.

Lemma 3.1 *Let $p[\cdot]$ be a valid potential. A s-t-path P is a shortest path with respect to edge lengths $\ell'(u, v)$ if and only if P is a shortest path in the original graph with edge lengths $\ell(u, v)$. Dijkstra's algorithm applied to the graph with edge lengths $\ell'(u, v)$ can be used to compute P.*

Proof. Let $P = (s = v_1, \ldots, v_k = t)$ be a s-t-path. We can proof that $\ell'(P) = \ell(P) + c$ for a constant c depending only on s and t.

$$\begin{aligned}
\ell'(P) &= \sum_{i=1}^{k-1} \ell'(v_i, v_{i+1}) \\
&= \ell(v_1, v_2) - p[v_1] + p[v_2] + \ell(v_2, v_3) - p[v_2] + p[v3] + \ldots + p[v_k] \\
&= \ell(P) - p[v_1] + p[v_k] \\
&= \ell(P) - p[s] + p[t].
\end{aligned}$$

Hence, a shortest path with respect to ℓ' is also a shortest path with respect to ℓ (otherwise a shorter path P' with $\ell(P') < \ell(P)$ would imply $\ell'(P') < \ell'(P)$), and vice versa. Since all edge lengths $\ell'(u, v)$ are non-negative, Dijkstra's algorithm can be applied to the graph with lengths $\ell'(u, v)$ to compute the shortest path P. ∎

Usually, lower bounds on the destination station are used to obtain valid potentials. In our case, we exploit the geometric information about stations, which is available in our data: If $D(u)$ denotes the Euclidean distance of the station of u to the destination station B, and v_{max} is the maximum speed of the timetable[1], the potential function in the time-expanded model is defined as $p[u] = D(u)/v_{max}$. This scaling of Euclidean distances guarantees valid (i.e., non-negative) potentials: Let $D(u, v)$ denote the Euclidean distance of an edge (u, v). Since $\ell(u, v)$ denotes the travel time of some train, $\ell(u, v) \geq D(u, v)/v_{max}$ (for stay-edges the validity is clear since the Euclidean distance is 0), and the triangular inequation for Euclidean distances yields $D(u, v) + D(v) \geq D(u)$. We obtain

$$\ell(u, v) - p[u] + p[t] \geq \frac{D(u, v) - D(u) + D(v)}{v_{max}} \geq 0.$$

Omitting Nodes

There is a simple way to reduce the size of the graphs in the time-expanded model, based on the fact that there are a lot of nodes with out-degree one. Any path through such a node must continue to the head of the single outgoing edge. Thus, we can safely delete such nodes from the graph and re-direct the incoming edges to the head of the single outgoing edge. The weight of a re-directed edge is the sum of the incoming edge plus the weight of the outgoing edge.

In the time-expanded graph, we omit arrival-nodes except for those belonging to the destination station. The arrival-nodes at the destination station are important since the algorithm has to be terminated once such a node is processed in the main loop of Dijkstra's algorithm. Since the destination is only known when a query is issued, arrival nodes at the destination station are dynamically inserted on demand during the algorithm.

The technique yields a graph of roughly half the original size, since all arrival-nodes have out-degree one. Figure 3.4 illustrates the construction.

3.4.2 Time-Dependent Model

Goal-Directed search

The goal-directed search heuristic described in Section 3.4.1 can be also applied in the time-dependent model. However, preliminary experiments with exactly the same technique showed no improvement of the running time in the time-dependent model. Therefore, we tried to improve the goal-directed search method. Our new

[1]The speed of an elementary connection is the Euclidean distance of the two stations involved divided by the travel time. The maximum speed v_{max} of the timetable is the maximum over all speeds of elementary connections.

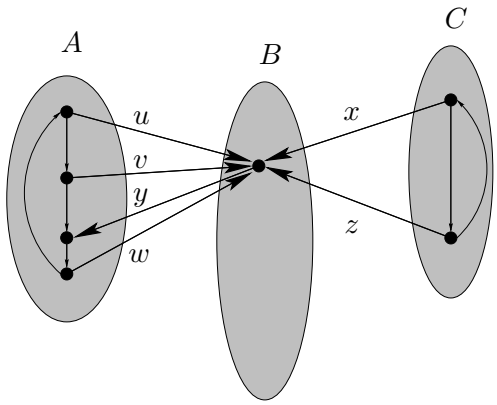

Figure 3.4: The sample time-expanded graph from Figure 3.2 (page 20), with redundant nodes omitted. Station A is assumed to be the destination station, i.e. at station A none of the nodes is omitted.

variant uses: (i) a different potential function, which depends on the destination station; (ii) Manhattan, besides Euclidean, distances between stations; and (iii) integral potentials to avoid expensive floating-point operations. The details are as follows.

As in Section 3.4.1, for an edge $e = (u, v) \in E$ with time-dependent length $\ell_e(t)$, the new length function $\ell'_e(t)$ is defined as $\ell'_e(t) = \ell_e(t) - p[u] + p[v]$, where $p[\cdot]$ is the potential function. Let B denote the destination station. Then, define

$$p[u] = D(u, B)\lambda_B, \quad u \in V, \quad \lambda_B \geq 0,$$

where $D(u, B)$ is the Euclidean or Manhattan distance between the station of node u and the destination station B. The parameter λ_B is called the *scaling factor* and is a non-negative number (which can be different for each destination station B) that is used to scale the distances so that $\ell'_e(t)$ is non-negative. The scaling factor is defined as

$$\lambda_B = \min_{(u,v)\in E, D(u,B)-D(v,B)>0} \frac{\min_t \ell_{(u,v)}(t)}{D(u, B) - D(v, B)}.$$

Note that λ_B corresponds to the inverse of the maximum speed considered in Section 3.4.1; however, in Section 3.4.1 the maximum speed is the same for all destinations, while λ_b depends on the destination station. The basic idea behind the modification we propose here is that the maximum speed is calculated only over all edges pointing geographically towards the destination station. Thus, for edges pointing towards the destination, the condition $\ell'_e \geq 0$ is fulfilled because of the scaling. For the other edges pointing away from the destination, the length anyway gets bigger and the condition is also fulfilled: $\ell'_e \geq \ell_e \geq 0$. The next lemma shows more formally that the potentials $p[\cdot]$ as defined above are valid.

Lemma 3.2 *The function $p[\cdot]$ is a valid potential function for the goal-directed search, i.e., ℓ'_e is non-negative for each edge e.*

Proof. Let $(\alpha, \beta) \in E$ be the edge yielding the minimum in the definition of λ_B, i.e., $\lambda_B = \ell_{(\alpha,\beta)}(t_0) = \min_t \ell_{(\alpha,\beta)}(t)$. Note that there is an edge $(u, v) \in E$ such that $D(u, B) - D(v, B) > 0$, since otherwise there would be no way reaching B—the distance to get there from every node $u \in V$ could never be reduced. Then, for every edge $e = (u, v) \in E$ with $D(u, B) > D(v, B)$ we have

$$\frac{\min_t \ell_e(t)}{D(u,B) - D(v,B)} \geq \frac{\ell_{(\alpha,\beta)}(t_0)}{D(\alpha,B) - D(\beta,B)} \tag{3.1}$$

Let t' be an arbitrary time value. Then, from the definition of ℓ'_e we get

$$\ell'_e(t') \geq \min_t \ell_e(t) - \ell_{(\alpha,\beta)}(t_0)\frac{D(u,B) - D(v,B)}{D(\alpha,B) - D(\beta,B)}$$

and from Equation (3.1) we obtain that

$$\min_t \ell_e(t) \geq \ell_{(\alpha,\beta)}(t_0)\frac{D(u,B) - D(v,B)}{D(\alpha,B) - D(\beta,B)}$$

which consequently implies that

$$\ell'_e(t') \geq 0.$$

Now, if $D(u, B) - D(v, B) \leq 0$, then $-[D(u, B) - D(v, B)] \geq 0$ and $-\lambda_B[D(u, B) - D(v, B)] \geq 0$, which means that $\ell'_e = \ell_e - \lambda_B[D(u, B) - D(v, B)] \geq \ell_e \geq 0$. ∎

Even if all edge weights are integers, the use of the Euclidean or Manhattan distances as potentials forces us to use floating point numbers in the priority queue, and this may result in an additional time overhead. In order to avoid this, we can transform the floating-point potentials to integers without invalidating the potentials as the following lemma shows.

Lemma 3.3 *If for the non-negative numbers $w \in \mathbb{N}$, $a, b \in \mathbb{R}^+$ it holds that $w - a + b \geq 0$, then it will also hold that $w - \lfloor a \rfloor + \lfloor b \rfloor \geq 0$.*

Proof. If $a = b$, the proposition holds trivially. Let $fr(a) = a - \lfloor a \rfloor$ and $fr(b) = b - \lfloor b \rfloor$. Consider first the case $a < b$. Then, $a - b < 0 \Rightarrow \lfloor a \rfloor - \lfloor b \rfloor < fr(b) - fr(a)$. Assume that $w - \lfloor a \rfloor + \lfloor b \rfloor < 0$. Since $w \geq 0$, we must have that $\lfloor a \rfloor - \lfloor b \rfloor > 0$, and hence $0 < \lfloor a \rfloor - \lfloor b \rfloor < fr(b) - fr(a)$. But, $fr(b) - fr(a) < 1$ and $\lfloor a \rfloor - \lfloor b \rfloor$ is an integer, a contradiction.

We turn now to the case $a > b$. Assume again that $w - \lfloor a \rfloor + \lfloor b \rfloor < 0$. Then, $w < \lfloor a \rfloor - \lfloor b \rfloor$. We also have $\lfloor a \rfloor - \lfloor b \rfloor \leq w + fr(b) - fr(a)$. Combining the last two inequalities and the fact that $fr(b) - fr(a) < 1$, we get that $w < \lfloor a \rfloor - \lfloor b \rfloor < w + 1$, which is again a contradiction. ∎

Consideration of the integral parts of the floating-point potentials turned out to be rather beneficial in certain cases as our experiments show.

Avoiding Binary Search

We also considered the heuristic that avoids the binary search as described in [BJ04]. The idea is as follows. Let k be the out-degree of a node v in the time-dependent graph and let $(v, u_1), \ldots, (v, u_k)$ be its outgoing edges. Construct a table D_v by sorting all events of v's outgoing edges with respect to their departure time. Place the first k such events in the first k entries (primary segment) of D_v, leave the next k entries empty (secondary segment), place the next k events in the next k entries of D_v, and so on. Let t_0 be the last event in D_v before some secondary segment. For every (v, u_i), $1 \leq i \leq k$, find the first event with departure time $t \geq t_0$ and put it into the i-th entry of the next secondary segment. For an edge (w, v), let t_1 be the arrival time of a primary event $P^w \in D_w$ (event belonging to some primary segment of D_w). Create a pointer from P^w to the immediately next primary event $P^v \in D_v$ with timestamp $t_2 \geq t_1$. The above construction avoids binary search, because when node v is extracted from the priority queue, we simply follow the pointer of the event P^w that caused v's extraction from the priority queue, and which leads to a primary entry $P^v \in D_v$. Hence, to find the next outgoing event of node v it suffices to scan the rest of the primary segment containing P^v and its next secondary segment.

3.5 Experimental Comparison of the Models

The main goal of our experimental study is to compare in practice the performance of the time-expanded and the time-dependent approach. For both models we also investigate the speedup techniques described in the previous section.

Given two different implementations and a timetable, we define the *relative performance* or *speedup* with respect to a measured performance parameter as the ratio of the value obtained by the first implementation and the value obtained by the second one. When one time-expanded and one time-dependent implementation is compared, we always divide the time-expanded value by the time-dependent value, i.e., we consider the speedup achieved when the time-dependent approach is used instead of the time-expanded approach.

For the experiments described in this section, the code is written in `C++` and compiled with the GNU `C++` compiler version 3.2. The experiments were run on a PC with an AMD Athlon XP 1500+ processor at 1.3 GHz and 512MB of memory running Linux (kernel version 2.4.19). The implementation of the time-dependent model for the earliest arrival problem uses the graph data structure of LEDA [NM99] version 4.4.

3.5.1 Data

We have used real-world data from the German and French railways. More precisely, the following five timetables have been used (ordered by size). The first timetable contains French long-distance railway traffic (`france`) from the winter period 1996/97. The remaining four are German timetables from the winter period

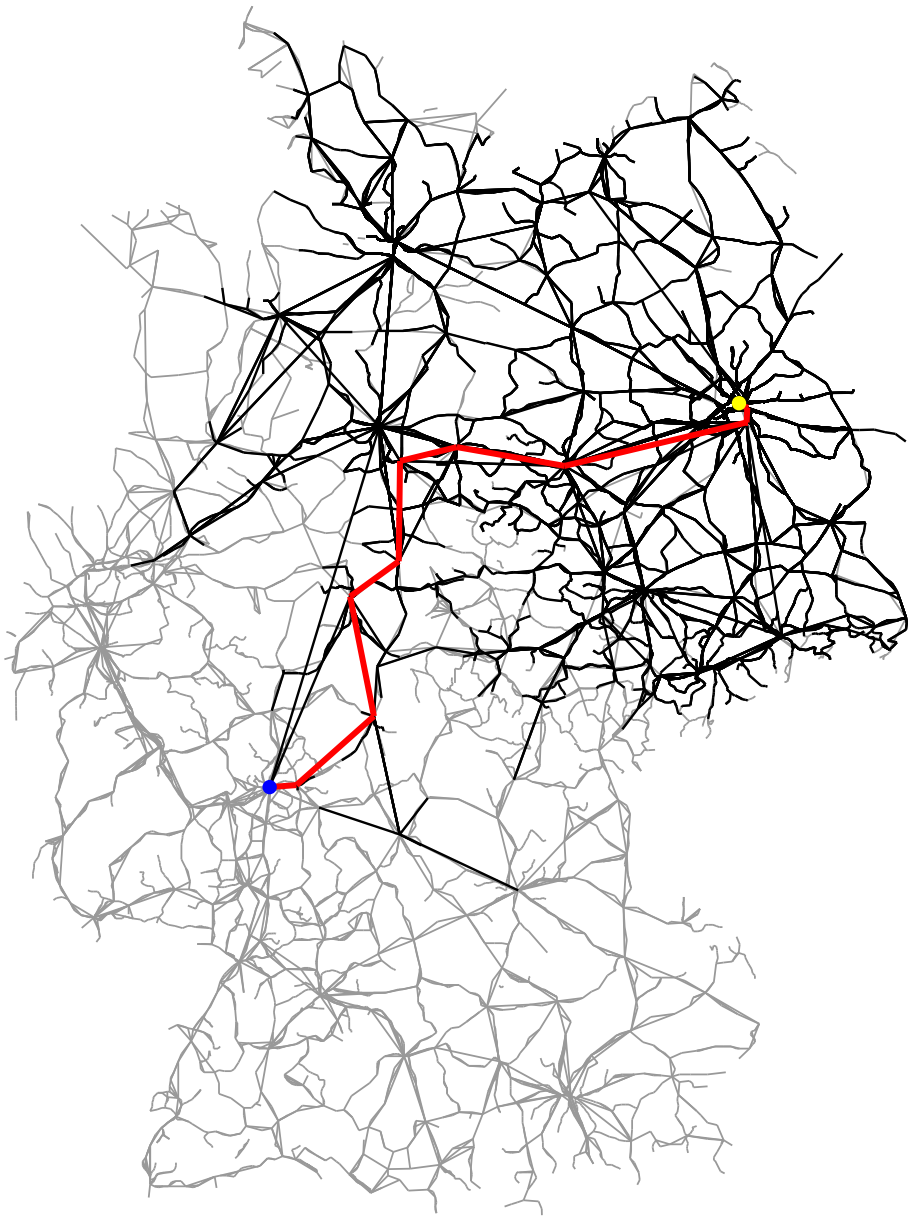

Figure 3.5: A visualisation of the timetable data `ger-longdist` (gray), the search space corresponding to a sample query from Berlin to Frankfurt (black), and the optimal train connection (thick line).

	Time-Expanded		Time-Dependent			
Timetable	Nodes	Edges	Nodes	Edges	Elem. conn. per node	Elem. conn. per edge
`france`	166085	332170	4578	14791	36	11
`ger-longdist`	480173	960346	6817	18812	70	26
`ger-local1`	691541	1383082	13460	37315	51	19
`ger-local2`	1124824	2249648	13073	36621	86	31
`ger-all`	2295930	4591860	32253	92507	71	25

Table 3.1: Parameters of the considered graphs for each of the timetables. The two columns on the left show the size of the graph used in the time-expanded model with the optimisation described in Section 3.4.1. The number of nodes equals the total number of elementary connections in the timetable. The remaining columns show the parameters of the graph used in the time-dependent model.

2000/01; one resembles the long-distance railway traffic in Germany (`ger-longdist`), two contain local traffic in Berlin/Brandenburg (`ger-local1`) and in the Rhein/Main region (`ger-local2`), and the last is the union of all these three German timetables (`ger-all`). We would like to note that HAFAS [HAF], the timetable information system used by the German railway company Deutsche Bahn, is based on data of the same format. Table 3.1 shows the characteristics of the graphs used in these models for the above mentioned timetables.

Real-world queries were available only for the timetables `ger-longdist` and `ger-all`: we took 50 000 queries from a snapshot of a central HAFAS server [HAF] in Germany that originally has been used in [SWW00] and consists of over half a million of real-world queries. We additionally generated random queries for every timetable. Each query-set consists of 50 000 queries of the form departure station, destination station, and earliest departure time.

3.5.2 Implementation Environment and Performance Parameters

For the time-expanded model the implementation is based on that used in [SWW00]; the optimisation technique to omit the arrival-nodes is included. For the time-dependent model, we have implemented both the plain version that uses binary search as well as the "avoid binary search" technique. For both models we also used the goal-directed search. (See Section 3.4, for the description of the speedup techniques.) Thus, for the time-expanded model we have two different implementations (goal-directed search with Euclidean distances or not), while for the time-dependent model we have several implementations depending on the use of: binary search or the "avoid binary search" version, the goal-directed search heuristic with Euclidean or Manhattan distances, and whether floating-point or integral potentials are used.

For each possible combination of timetable and implementation variant we per-

	Timetable	Real	Time [ms]	El. Conn.	Nodes	Edges
Expanded	`france`		100.4	30824	33391	61649
	`ger-longdist`		169.6	44334	48094	88668
	`ger-local1`		608.7	176720	182717	353443
	`ger-local2`		840.1	226027	232511	452056
	`ger-all`		1352.8	326186	342917	652378
	`ger-longdist`	✗	66.7	18891	20853	37783
	`ger-all`	✗	392.1	96943	104369	193888
Dependent	`france`		8.2	8539	2269	4463
	`ger-longdist`		10.7	20066	3396	5129
	`ger-local1`		19.7	26792	6535	9835
	`ger-local2`		20.7	31698	6524	10075
	`ger-all`		76.6	79981	16145	26333
	`ger-longdist`	✗	5.5	11173	1711	2682
	`ger-all`	✗	37.3	40808	6926	11647

Table 3.2: Average CPU-time and operation counts for solving a single query for the time-expanded (upper part) and the time-dependent model (lower part). The arrival-nodes are omitted in the time-expanded model (see Section 3.4.1), and in the time-dependent model binary search was used. Goal-directed search was not applied in both cases. The column *Real* indicates whether real-world or random queries have been used.

formed the corresponding set of random queries (for `ger-longdist` and `ger-all` we additionally performed the corresponding real-world queries) and measured the following performance parameters as *mean values* over the set of performed queries: CPU-time in milliseconds, number of nodes, number of edges, and number of elementary connections touched by the algorithm. For the time-expanded model, the number of touched elementary connections is the number of train-edges touched by the algorithm, while for the time-dependent model it is the total number of elementary connections that have been used for calculating the edge lengths. More precisely, when binary search is used in the time-dependent model, for a single edge the number of steps needed by the binary search is the number of touched elementary connections.

3.5.3 Results and Discussion

Our experimental results are reported in Figures 3.6 and 3.7, and Tables 3.2, 3.3, and 3.4. The reported results clearly show that the time-dependent model solves the simplified earliest arrival problem considerably faster than the time-expanded model, for every considered data set. Regarding CPU-time, the speedup ranges between 12 (`france`) and 40 (`ger-local2`) when the basic implementations are used (see Fig. 3.6 and Table 3.2), and between 17 (`france`) and 57 (`ger-local2`)

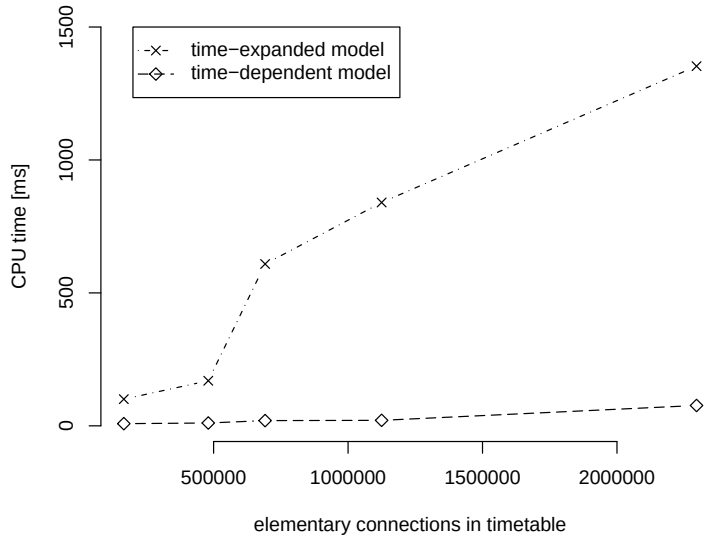

Figure 3.6: Performance of the basic implementations of the time-expanded and time-dependent models for the simplified earliest arrival problem (no goal-directed search, binary search in the time-dependent model) regarding the five timetables and the random queries. Each point represents the average over these queries of one measured performance parameter. On the abscissa the size of the timetable in number of elementary connections is shown, and the ordinate represents the average CPU-time for answering a query in the time-expanded model (top curve) and the time-dependent model (lower curve), respectively.

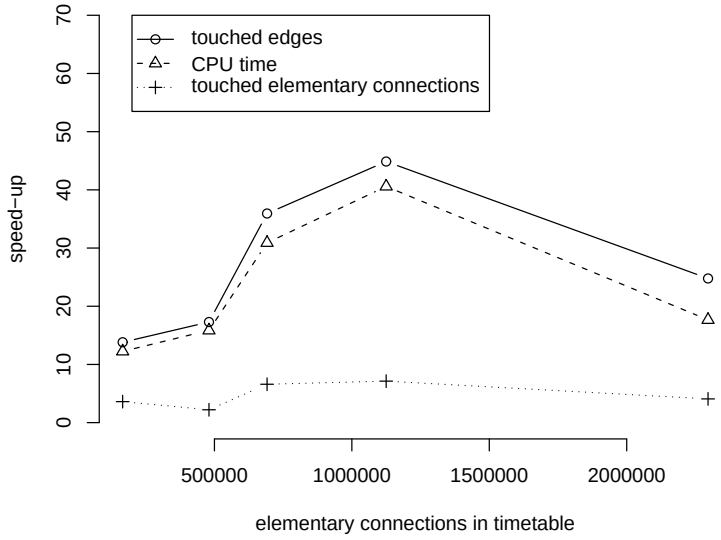

Figure 3.7: Like Figure 3.6, but now the ordinate represents the speedup of using the time-dependent model instead of the time-expanded model, with respect to the number of touched edges, CPU-time, and number of touched elementary connections (from top to bottom).

when comparison concerns the best implementations (including heuristics) in both models (see Tables 3.3 and 3.4).

Concerning the time-dependent model, we observe that is is better to use the "avoid binary search" technique (see Tables 3.3 and 3.4). Compared to the binary search implementation the speedup was between 1.39 (`ger-local1`) and 1.86 (`ger-all` with real-world queries). The goal-directed search technique always reduces the search space of Dijkstra's algorithm, i.e., the number of touched nodes and edges. However, this reduction payed off only in a few cases in the sense that it could not also decrease the CPU-time. In most cases the CPU-time was increased due to the additional computations required to calculate the edge lengths, and this is the main reason why this technique appears slower in the results. Another reason is that in the timetables used in our experiments the maximum speed over all elementary connections is high. A high maximum speed yields small potential functions[2], and thus bad performance of the goal-directed search technique.

[2]In both models the potentials are inversely proportional to the maximum speed; for the time-expanded case this is clear by definition, and for the time-dependent case see the definition of the factor λ_B.

Time-Dependent Model, Binary Search

	Timetable	Real	Time [ms]	El. conn.	Nodes	Edges
Goal Eucl. int	france		9.4	7072	1593	3415
	ger-longdist		13.5	16597	2737	4217
	ger-local1		28.6	26008	6257	9434
	ger-local2		30.4	31196	6398	9895
	ger-all		100.3	74525	14568	24030
	ger-longdist	✗	6.3	8349	1238	1991
	ger-all	✗	43.1	33676	5551	9420
Goal Eucl. float	france		9.4	7062	1590	3410
	ger-longdist		13.6	16560	2730	4208
	ger-local1		28.9	25975	6249	9422
	ger-local2		30.7	31152	6389	9882
	ger-all		103.6	74394	14538	23983
	ger-longdist	✗	6.4	8318	1233	1984
	ger-all	✗	44.5	33565	5532	9388
Goal Manh. int	france		7.9	7225	1647	3511
	ger-longdist		11.2	16975	2807	4316
	ger-local1		23.2	26086	6284	9473
	ger-local2		24.7	31235	6407	9908
	ger-all		86.4	74822	14639	24138
	ger-longdist	✗	5.3	8555	1272	2041
	ger-all	✗	38.0	33994	5615	9524
Goal Manh. float	france		7.7	7214	1644	3505
	ger-longdist		11.1	16938	2800	4306
	ger-local1		23.1	26053	6276	9461
	ger-local2		24.6	31189	6398	9894
	ger-all		88.9	74689	14608	24091
	ger-longdist	✗	5.2	8524	1267	2034
	ger-all	✗	39.0	33880	5594	9491

Table 3.3: Comparison of the time-dependent implementations that use binary search and four different versions of goal-directed search: Euclidean distance with integer and float potentials, and Manhattan distance with integer and float potentials. Columns are as in Table 3.2.

Time-Expanded Model

	Timetable	Real	Time [ms]	El. Conn.	Nodes	Edges
Goal Eucl. int	france		84.0	22259	24179	44517
	ger-longdist		175.0	34259	37453	68517
	ger-local1		684.3	170369	176243	340741
	ger-local2		953.0	219992	226386	439986
	ger-all		1392.6	285440	300788	570885
	ger-longdist	✗	54.3	13384	14931	26768
	ger-all	✗	341.9	74069	80229	148140

Time-Dependent Model, Avoid Binary Search

Plain Dijkstra	france		5.9	8942	2262	4386
	ger-longdist		7.5	9216	3396	5129
	ger-local1		14.2	18312	6541	9814
	ger-local2		14.6	18435	6524	10075
	ger-all		47.4	48520	16146	26333
	ger-longdist	✗	3.8	4773	1711	2682
	ger-all	✗	20.1	20993	6927	11648
Goal Eucl. int	france		6.6	6711	1614	3406
	ger-longdist		9.2	7553	2737	4217
	ger-local1		20.5	17656	6301	9499
	ger-local2		21.5	18088	6398	9895
	ger-all		63.0	44010	14568	24030
	ger-longdist	✗	4.2	3527	1238	1991
	ger-all	✗	23.7	16898	5552	9421
Goal Manh. int	france		5.1	6926	1669	3505
	ger-longdist		7.1	7733	2807	4316
	ger-local1		15.3	17621	6289	9480
	ger-local2		16.1	18113	6407	9908
	ger-all		50.5	44214	14639	24138
	ger-longdist	✗	3.2	3618	1272	2041
	ger-all	✗	19.1	17088	5615	9524

Table 3.4: Comparison of goal-directed search in the time-expanded case (upper part) and the technique to avoid binary searches in the time-dependent case (lower part). In the time-dependent case two different distance measures for the goal-directed search are reported, the Euclidean and the Manhattan distances with integral potentials, which were the fastest. Columns are as in Table 3.2.

Chapter 4

Towards Realistic Timetable Information

The previous chapter introduced basic modelling techniques exemplarily for a version of the earliest arrival problem which is, for application in a real timetable information system, too restrictive. In reality, several issues regarding realism of the problems and models have to be addressed: First, we have to deal with realistic transfer rules, and waive the assumption that transfers within a station take negligible time. Furthermore, in reality trains are operated only on certain *traffic days*, so we cannot assume—as we did so far—that every day in the timetable is the same. Another issue is to consider different optimisation criteria besides the earliest arrival: In the *minimum number of transfers* problem, the goal is to find a connection that minimises the number of train transfers when considering an itinerary from A to B. In the *latest departure* problem, among all optimal solutions to an instance of the earliest arrival problem, the one departing the latest is wanted. Finally, a timetable information system should also allow combinations of different criteria.

We show how both approaches to model timetable information introduced previously can be extended to cope with the realistic problem specifications outlined above. We also conducted extensive experiments comparing the extended approaches. This comparison is important, since the described extensions are mandatory for real-world applications, and (to the best of our knowledge) nothing is known about the relative behaviour of realistic versions of the two approaches. The extensions used in the time-dependent model have been published as "*Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach*" [PSWZ04b].

4.1 Realistic Transfer Rules

In contrast to the previous chapter, where transfers were always allowed, we investigate now how non-zero transfer times at stations can be included in the models introduced in Sections 3.3.1 and 3.3.2 (pages 18 et seq.). Also, more sophisticated transfer rules are under consideration. We refer to the earliest arrival problem with realistic transfer rule as the *realistic* earliest arrival problem, or *realistic EAP*.

4.1.1 Specification

At a station $S \in \mathcal{B}$ it is possible to transfer from one train to another only if the time between the arrival and the departure at that station S is larger than or equal to a given, station-specific, *minimum transfer time*, denoted by *transfer*(S). To include this transfer rule in our formal specification, we extend the definition of a *consistent connection* in Section 3.2.2 (page 17) by the following condition:

$$dep_{i+1}(P) - arr_i(P) \geq \begin{cases} 0 & \text{if } Z(c_{i+1}) = Z(c_i) \\ transfer(S_2(c_i)) & \text{otherwise.} \end{cases}$$

There may also be more detailed transfer rules, for example the transfer time can be smaller for trains that depart from the same platform. Transfer times can be given between train routes (also referred to as train lines) instead of specifying one single transfer time per station. Another more general way is to specify a station-specific minimum transfer time, and exceptions in the form of a set of additional feasible transfer trains for each arrival of a train at a station.

4.1.2 Time-Expanded Model

To solve the realistic earliest arrival problem, we extend the time-expanded model by constructing the *realistic time-expanded graph* as follows. Based on the time-expanded graph, we keep, for each station, a copy of all departure- and arrival-nodes in the station which we call *transfer-nodes*; see Figure 4.1. The stay-edges are now introduced between the transfer-nodes. For every arrival-node there are two additional outgoing edges: one edge to the departure-node of the same train; and a second edge to the transfer-node with time value greater than or equal to the time of the arrival-node plus *transfer*(S). The edge lengths are defined as in the definition of the original model in Section 3.3.1 (page 18).

More detailed transfer rules can also be easily integrated into the realistic time-expanded graph. Each exceptionally allowed transfer from train Z_1 to train Z_2 can be modelled by an additional edge from the arrival-node of Z_1 at the specific station to the departure-node of Z_2.

The correctness of the modelling is established by the following theorem.

Theorem 4.1 *A shortest path in the realistic time-expanded graph from the first departure-node at the departure station A with departure time later than or equal to the given start time t_0 to one of the arrival-nodes of the destination station B constitutes a solution to the realistic EAP in the extended time-expanded model. The actual path can be found by Dijkstra's algorithm.*

Proof. The construction of paths from connections and vice versa used in the proof of Theorem 3.1 can be easily adapted to the realistic EAP, and thus that proof applies analogously. ∎

The technique of omitting redundant nodes described in Section 3.4.1 (page 25) can also be applied to the realistic time-expanded graph. However, here the graph can only be reduced by about one third (instead of one half in the simple version of the graph), since only the departure-nodes have out-degree one.

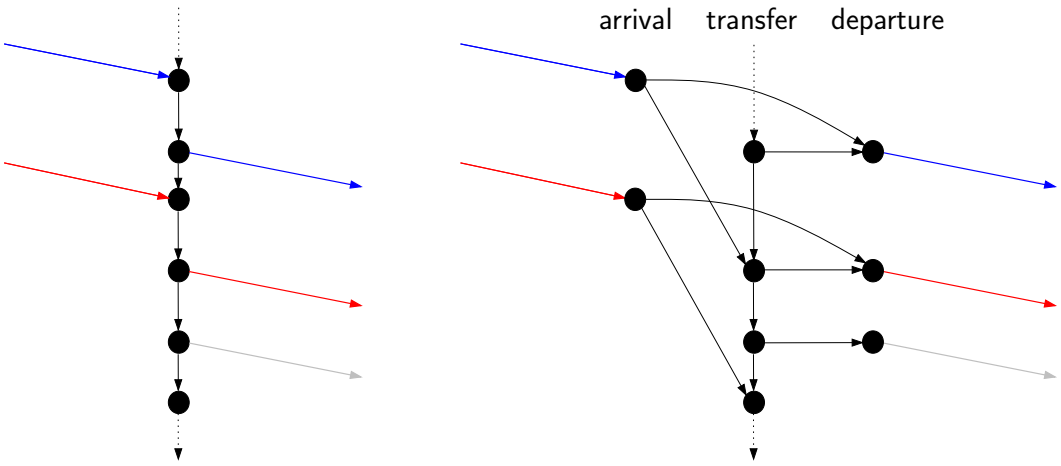

Figure 4.1: Modelling train transfers in the time-expanded model for a sample station. On the left the original modelling from the previous chapter is shown, and on the right the realistic time-expanded graph with three types of nodes: arrival, transfer and departure nodes.

4.1.3 Time-Dependent Model

To model non-zero train transfers, we extend the original time-dependent model using information on the routes that trains may follow. Hence, we assume that we are given a set of train routes and their respective time schedules. We examine both the cases of a constant transfer time per station and detailed transfer rules which we also refer to as *variable* transfer time here. A somehow similar idea for the constant transfer time case was very briefly mentioned in [BJ04], but without providing details. In the following, we describe the construction of a graph $G = (V, E)$ that models these two cases. We shall refer to G as the *train-route graph*.

Let $\mathcal{S}$ be a set of nodes representing stations. For $u \in \mathcal{S}$, we denote by $station(u)$ the actual station which u represents. We say that nodes $s_0, s_1, ..., s_{k-1}$, $k > 0$, form a *train route* if there is some train starting its journey from $station(s_0)$ and visiting consecutively $station(s_1), ..., station(s_{k-1})$ in turn. If there are more than one trains following the same schedule (with respect to the order in which they visit the above nodes), then we say that they all belong to the same train route P. Note that it can be $station(s_i) = station(s_j)$, $i \neq j$, $s_i \neq s_j$, $0 \leq i, j \leq k-1$, for example when the train performs a loop.

For $u \in \mathcal{S}$, let Σ_u be the set of different train routes that stop at $station(u)$, and let $\mathcal{P}_u$ be a set that contains exactly one node for each $P \in \Sigma_u$ that passes through $station(u)$. Also, let $P_u = |\mathcal{P}_u|$, and $\mathcal{P} = \bigcup_{u \in \mathcal{S}} \mathcal{P}_u$. Then, the node set V of G is defined as $V = \mathcal{S} \bigcup \mathcal{P}$. For $u \in \mathcal{S}$, we denote by $p_i^u, 0 \leq i < P_u$, the node representing the i-th train route that stops at u.

The edge set $E = A \bigcup D \bigcup \overline{D} \bigcup R$ of G consists of four types of edges which are defined as follows.

- $A = \bigcup_{u \in \mathcal{S}} A_u$, where $A_u = \bigcup_{0 \leq i < P_u} \{(p_i^u, u)\}$.
- $D = \bigcup_{u \in \mathcal{S}} D_u$, where $D_u = \bigcup_{0 \leq i < P_u} \{(u, p_i^u)\}$.

- $\overline{D} = \bigcup_{u \in \mathcal{S}} \overline{D}_u$, where $\overline{D}_u = \emptyset$, if the time needed for a transfer is the same for all trains that stop to $station(u)$; and $\overline{D}_u = \bigcup_{0 \leq i,j < P_u, i \neq j} \{(p_i^u, p_j^u)\}$, otherwise.
- $R = \bigcup_{u,v \in \mathcal{S}, 0 \leq i < P_u, 0 \leq j < P_v} \{(p_i^u, p_j^v) : station(u)$ and $station(v)$ are visited successively by the same train route and p_i^u, p_j^v are the corresponding route nodes$\}$.

An edge e is called a *route* or *timetable edge* if $e \in R$, and it is called a *transfer edge* if $e \in A \bigcup D \bigcup \overline{D}$. The modelling with train routes is based on two additional assumptions.

Assumption 4.1 *Let u, v be any two nodes in $\mathcal{S}$ and $p_i^u \in \mathcal{P}_u$, $p_j^v \in \mathcal{P}_v$ such that $(p_i^u, p_j^v) \in R$. If d_1, d_2 are departure times from p_i^u and a_1, a_2 are the respective arrival times to p_j^v, then $d_1 \leq d_2 \Rightarrow a_1 \leq a_2$.*

This assumption is the equivalent of Assumption 3.1 (page 22) in the simplified time-dependent approach, and it states that there cannot be two trains that belong to the same train route, such that the first of them leaving a station is a slow train, while the following one is a fast train and it arrives to the next station before the first. When this assumption is violated, we can enforce it by separating the trains that belong to the same train route into different speed classes, and hence introduce new train routes, one for each different speed class, where all follow the same schedule as before.

Assumption 4.2 *For any $u \in \mathcal{S}$ and $p_i^u \in \mathcal{P}_u$ such that $(x, p_i^u) \in R$, for some $x \in V$, let $\delta_x^{u_i}$ be the smallest interval between two successive arrivals to p_i^u from $(x, p_i^u) \in R$ and τ_u be the maximum time needed for a transfer at $station(u)$. Then, it must hold that $\delta_x^{u_i} \geq \tau_u$.*

Assumption 4.2 serves the purpose of ensuring that waiting at stations to take the next train of the same train route cannot be beneficial. In other words, given that Assumption 4.1 holds, taking the first possible train from a station A to some station B will not result in missing some connection from B that could be used if we had followed some train (of the same train route) that departed later than the one we followed. It will be seen later that Assumption 4.2 will only be needed in the case where we consider variable transfer times within the same station.

In the following, we shall assume that $u, v \in \mathcal{S}$, $0 \leq i, i' < P_u$, $0 \leq j < P_v$, and that T is a set representing time.

Constant Transfer Time

In this case, the edge costs of the train-route graph are defined as follows (see Figure 4.2). Let $transfer(u) = transfer(station(u))$ denote the transfer time of the station a node u belongs to.

- An edge $(p_i^u, u) \in A$ has zero cost.
- An edge $(u, p_i^u) \in D_u$ has cost determined by a function $g_u : T \to T$, such that $g_u(t) = t + transfer(u)$.

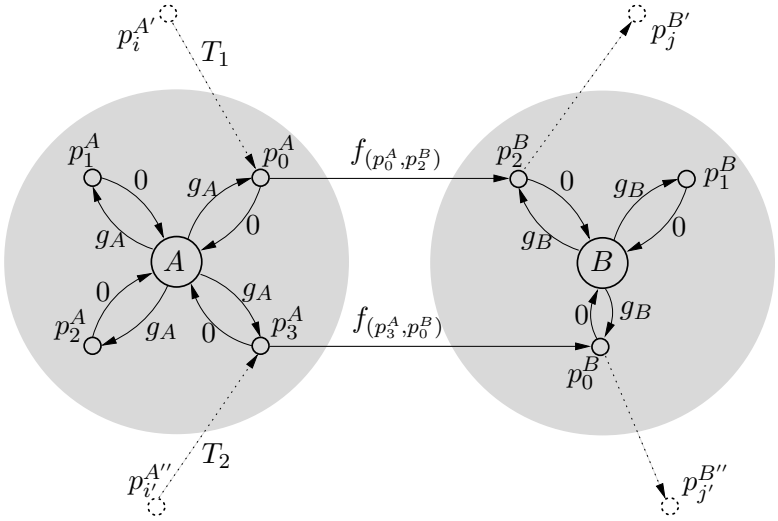

Figure 4.2: An example of modelling through train routes with constant transfer time per station.

- An edge $(p_i^u, p_j^v) \in R$ has cost determined by a function $f_{(p_i^u, p_j^v)} : T \to T$ such that $f_{(p_i^u, p_j^v)}(t)$ is the time at which p_j^v will be reached using the edge (p_i^u, p_j^v), given that p_i^u was reached at time t.

To solve the realistic EAP for a given query (α, β, t_0), we need to find a shortest path from station α to β using the modified Dijkstra's algorithm introduced in Section 3.3.2 (pages 22 et seq.), where for each edge we use its associated cost function as described above. We actually need to find the shortest path in the graph $G = (V, E_{\alpha,\beta})$ from s_α to s_β starting at time t_0, where $\alpha = station(s_\alpha)$, $\beta = station(s_\beta)$, and $E_{\alpha,\beta} = A \bigcup D \bigcup R$ (see Figure 4.2). Note that all edges $e \in D_{s_\alpha}$ must have zero cost.

Theorem 4.2 *The above algorithm solves the realistic EAP with constant transfer times at stations in the extended time-dependent model, provided that Assumption 4.1 holds.*

Proof. In view of the discussion in Section 3.3.2 (page 21) regarding the correctness of Theorem 3.2, the correctness of the above algorithm for solving the realistic EAP, when the transfer time in each station is constant, follows from the non-negative delay assumptions of functions f and g, and by the fact that both functions are non-decreasing (f by Assumption 4.1, and g by construction). ∎

Variable Transfer Time

The edge costs of the train-route graph in this case are defined as follows (see also Figure 4.3).

- An edge $(p_i^u, u) \in A$ has zero cost.

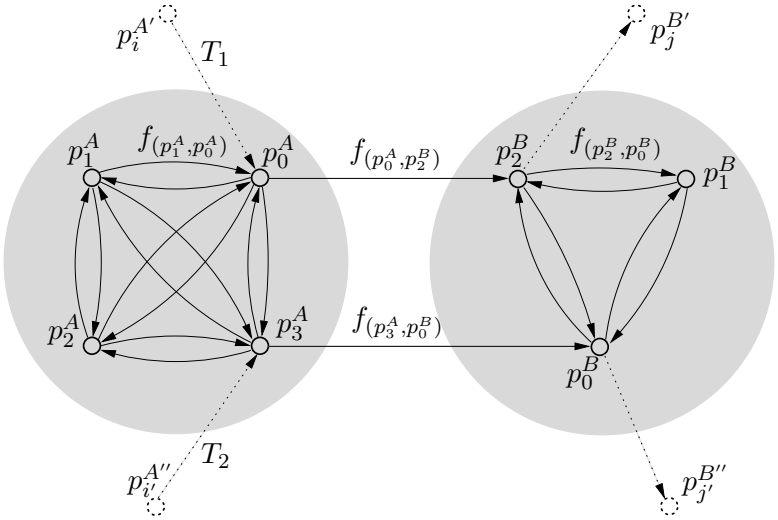

Figure 4.3: An example of modelling through train routes with variable transfer times at the stations.

- An edge $(u, p_i^u) \in D_u$ has zero cost. An edge $(p_i^u, p_{i'}^u) \in \overline{D}_u$ has cost determined by a function $f_{(p_i^u,p_{i'}^u)} : T \to T$ such that $f_{(p_i^u,p_{i'}^u)}(t)$ represents the time required by a passenger who reached $station(u)$ at time t with a train of the train route p_i^u, to be transferred to the first possible train of route $p_{i'}^u$. In particular, $f_{(p_i^u,p_{i'}^u)}(t) = t + \tau^u_{change(i,i')}(t)$, where $\tau^u_{change(i,i')}(t)$ is the function that, for each arriving time, returns the corresponding transfer time.
- An edge $(p_i^u, p_j^v) \in R$ has cost determined by a function $f_{(p_i^u,p_j^v)} : T \to T$ such that $f_{(p_i^u,p_j^v)}(t)$ is the time at which p_j^v will be reached using the edge (p_i^u, p_j^v), given that p_i^u was reached at time t.

As before, given a realistic EAP query (α, β, t_0), all we need is to find a shortest path from station α to station β in the above graph, where for each edge we use its associated cost function as described above. This is again accomplished by the modified Dijkstra's algorithm (cf. Section 3.3.2). We actually need to find a shortest path in the graph $G = (V, E_{\alpha,\beta})$ from s_α to s_β starting at time t_0, where $\alpha = station(s_\alpha)$, $\beta = station(s_\beta)$, but now $E_{\alpha,\beta} = D_{s_\alpha} \bigcup A_{s_\beta} \bigcup \overline{D} \bigcup R$ (see Figure 4.3). Note that all edges $e \in D_{s_\alpha}$ must have zero cost.

Let nodes $p_i^u, p_j^u \in \mathcal{P}_u$, $0 \leq i, j < P_u$, $i \neq j$, such that $(p_i^u, p_j^u) \in \overline{D}$. In addition let node $p_{i'}^v \in \mathcal{P}_v$, $0 \leq i' < P_v$ such that $(p_{i'}^v, p_i^u) \in R$. In order to be able to apply the above algorithm and solve EAP in this case, we have to ensure that the functions are non-decreasing. Assumption 4.1 ensures that $f_{(p_{i'}^v,p_i^u)}(t)$ is non-decreasing. What we need to prove is that $f_{(p_i^u,p_j^u)}(t)$ is also non-decreasing when $t \in T_{u_i}$, where T_{u_i} denotes the set of the arrival time values to p_i^u from the edge $(p_{i'}^v, p_i^u) \in R$. (Note that $p_{i'}^v$ is the only in-neighbor, i.e., tail of an in-coming edge, of p_i^u that is not in $\mathcal{P}_u$.)

Lemma 4.1 *The function $f_{(p_i^u,p_j^u)}(t)$ is non-decreasing when $t \in T_{u_i}$, where $T_{u_i} = \{t | t$ is the arrival time to p_i^u from the edge $(p_{i'}^v, p_i^u) \in R\}$.*

Proof. Let τ_u be the maximum time needed for a transfer at $station(u)$, let $\delta^{u_i}_{v_{i'}}$ be the minimum interval between two successive arrivals to p^u_i from $(p^v_{i'}, p^u_i)$, and let $t_1, t_2 \in T_{u_i}$ where $t_1 < t_2$. Since t_1, t_2 are two distinct arrival times at p^u_i from $(p^v_{i'}, p^u_i)$, it holds that $t_2 - t_1 \geq \delta^{u_i}_{v_{i'}}$. By Assumption 4.2, we have that $\delta^{u_i}_{v_{i'}} \geq \tau_u$. Also, $f_{(p^u_i, p^u_j)}(t) - t \leq \tau_u$, $t \in T$, since $f_{(p^u_i, p^u_j)}(t) - t$ is the time needed for a transfer from p^u_i to p^u_j on time t. Consequently,

$$t_2 - t_1 \geq \delta^{u_i}_{v_{i'}} \geq \tau_u \geq f_{(p^u_i, p^u_j)}(t_1) - t_1$$

$$\Rightarrow t_2 - t_1 \geq f_{(p^u_i, p^u_j)}(t_1) - t_1$$

$$\Rightarrow f_{(p^u_i, p^u_j)}(t_1) \leq t_2 \leq f_{(p^u_i, p^u_j)}(t_2)$$

$$\Rightarrow f_{(p^u_i, p^u_j)}(t_1) \leq f_{(p^u_i, p^u_j)}(t_2)$$

which completes the proof of the lemma. ∎

Lemma 4.1 and the above discussion establish the next theorem whose proof follows that of Theorem 4.2.

Theorem 4.3 *The above algorithm solves the realistic EAP with variable transfer times at stations in the extended time-dependent model, provided that Assumptions 4.1 and 4.2 hold.*

4.2 Traffic Days

For all models described so far we have assumed that every train operates daily, i.e., the timetable is identical every day. In this section we discuss how different traffic days can be integrated in the models. For each train we are given the information on which day of the timetable it is valid.

4.2.1 Specification

A timetable is valid for a number of N *traffic days*, and every train is assigned a bit-field of N bits determining on which traffic day the train operates (for overnight trains the departure of the first elementary connection counts). A connection was defined in Section 3.2.2 (page 17) to be a sequence of elementary connections c_i together with departure times dep_i and arrival times arr_i. Since every elementary connection c_i is assigned one train, the validity of c_i at a given day can be verified through the traffic days of the respective train. The day within the timetable of one of the elementary connections c_i is encoded in the time value dep_i and can be calculated by $\lfloor dep_i(P)/1440 \rfloor$; see also Section 2.3.1 (page 9). Hence, we have to extend the definition of a *consistent connection* by the following condition:

$$c_i \quad \text{is valid on day } \lfloor dep_i(P)/1440 \rfloor$$

4.2.2 Time-Expanded Model

When traffic days are used, an optimal connection may stay for more than a day at an intermediate station (e.g., assume that on a holiday no trains are operated at all). Such connections do not correspond to simple paths in the time-expanded graph, and the problem cannot be solved directly using that graph. Therefore, we make use of the fully time-expanded graph introduced in Section 3.3.1 (page 19) to tackle the problem in the time-expanded approach. As we will see below, we do not have to explicitly maintain this graph. The execution of an algorithm on the fully time-expanded graph can be simulated on the original (Section 3.3.1) or on the realistic time-expanded graph (Section 4.1.2).

As described earlier for the simplified case of the earliest arrival problem, the fully time-expanded graph is based on N copies of the time-expanded graph, if the timetable period consists of N days. The construction can be done analogously for the realistic version including the modelling of realistic transfer rules. To incorporate the traffic day information, in the i-th copy all train-edges that correspond to elementary connections that are not valid on day i are deleted from the graph. Again, there is an obvious one-to-one correspondence between connections (consistent also with traffic days) and paths in the fully time-expanded graph, which immediately yields the following theorem.

Theorem 4.4 *A shortest path in the original (resp. realistic) fully time-expanded graph constitutes a solution to the simplified (resp. realistic) version of EAP when elementary connections are valid on specific traffic days only.*

Since the size of the fully time-expanded graph is huge (N times the size of the time-expanded graph), we consider now how we can avoid to maintain this huge graph explicitly. By construction, all edge lengths in the fully time-expanded graph are less than a day. Assume an application of Dijkstra's algorithm to the fully time-expanded graph, and let t be the time associated with the first node in the priority queue. All other nodes in the priority queue have an associated time larger than or equal to t and less than $t + 1440$. This observation allows the simulation of the algorithm by applying a modification of Dijkstra's algorithm to the time-expanded graph: the time-expanded graph reflects the subgraph of the fully time-expanded graph induced by the nodes with associated time in the interval $[t, t + 1440]$, and by ignoring all train-edges that correspond to trains which are not operated on the considered day. In the simulation, the day under consideration has to be determined by dividing the current absolute departure time[1] by 1440. Whenever a node is settled, it is not marked permanent in Dijkstra's algorithm. It is marked to be untouched again, by setting its distance label to infinity, because from that point on it is considered to be the copy of the node in the next day. This can safely be done since the fully time-expanded graph is a dag and Dijkstra's algorithm processes the nodes in topological order: no edge is pointing backward, and thus the node can be "re-used".

[1]The current absolute departure time is equal to the absolute time associated with the source node s plus the current distance of the considered node.

4.2.3 Time-Dependent Model

The model as defined in Section 3.3.2 (pages 21 et seq.) or in Section 4.1.3 is inherently able to handle traffic days. For route edges, the length is determined by the first elementary connection on that edge leaving later than the arrival time. Now, the length is determined by the first edge leaving later than the arrival time that is valid on the day under consideration (the day is computed by dividing the arrival time by 1440).

Note, however, that the speedup technique to avoid binary search in the calculation of edge lengths described in Section 3.4.2 (pages 28 et seq.) cannot be applied directly anymore.

4.3 The Minimum Number of Transfers Problem

Besides the earliest arrival (EA), the minimum number of transfers (MNT) is an important criterion in timetable information. The problem of minimising the number of transfers is formally specified and it is shown how it can be solved using the graphs constructed in both the time-expanded and the time-dependent models for the earliest arrival problem.

4.3.1 Problem Specification

In the *minimum number of transfers problem* (MNTP), a query consists only of a departure station A and an arrival station B. Trains are assumed to operate daily, and there is no restriction on the number of days a timetable is valid[2]. All connections from A to B are valid, and the optimisation criterion is to minimise the number of train transfers. More precisely, let $P = (c_1, \ldots, c_k)$ be a connection from A to B and let $trans_i(P) \in \{0, 1\}$ be a variable denoting whether a transfer is needed from elementary connection c_i to c_{i+1}, $1 \leq i < k$. Then, $trans_i(P) = 1$, if $Z(c_{i+1}) \neq Z(c_i)$, and $trans_i(P) = 0$, otherwise. Consequently, the objective of MNTP is to minimise, among all P, the quantity $\sum_{i=1}^{k-1} trans_i(P)$.

4.3.2 Modelling

The graphs defined for the realistic earliest arrival problem in both the time-expanded (Section 4.1.2) and the time-dependent (Section 4.1.3) approach can be used to solve the minimum number of transfers problem with a similar method: edges that model transfers are assigned a weight of one, and all the other edges are assigned weight zero, as shown exemplarily in Figure 4.4. In the time-expanded case all edges from

[2]This assumption can be safely made since time is not minimised in MNTP, and thus in a MNT-optimal connection one can wait arbitrarily long at a station for some connection that is valid only on certain days.

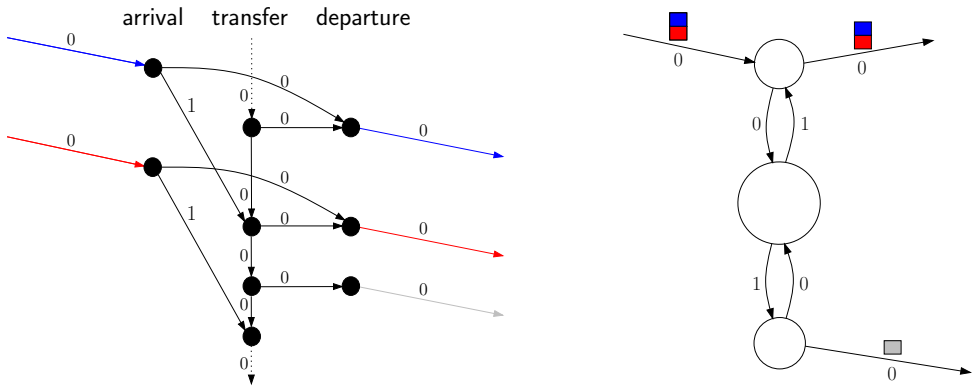

Figure 4.4: Edge weights for solving the minimum number of transfers problem. On the left the edge weights used in the MNTP modelling are shown for the sample time-expanded graph from Figure 4.1 (page 39), and on the right for the train-route graph of the same instance, assuming that the upper two (blue and red) trains form a train route, and the lower (gray) train another train route.

arrival-nodes to transfer-nodes have weight one, whereas in the time-dependent case the edges in the set D (i.e., edges from nodes representing stations to route-nodes), except those belonging to the departure station, are assigned weight one, and all other edges have weight zero. Note that the edge costs in the time-dependent graph are all static here. The correctness of this modelling for finding MNT-optimal connections as shortest paths can be easily verified as in the proof of Theorem 3.1. This establishes the following theorem.

Theorem 4.5 *A shortest path in the realistic time-expanded (resp. train-route) graph, with the edge costs given above, from a node belonging to (resp. representing) the departure station to a node belonging to (resp. representing) the arrival station is a solution to the MNTP.*

4.4 Bicriteria Problems

We consider also bicriteria problems with the earliest arrival (EA) and the minimum number of transfers (MNT) as the two criteria. In the following, we parameterise the bicriterion problem we consider by (X,Y), where X (resp. Y) is the first (resp. second) criterion we want to optimise and X,Y $\in$ {EA,MNT}. The general problem and algorithms for multi-criteria shortest paths have been described in Section 2.2 (pages 7 et seq.). Concerning timetable information, Möhring [Möh99] provides an overview on methods to be used when multiple criteria are involved. Müller-Hannemann and Weihe [MHW01] conducted experiments computing all Pareto-optimal paths in a time-expanded graph for timetable information (they also considered other graphs). They found that, for several multi-criteria problems in timetable information graphs, there are on average only very few Pareto-optimal paths per node, which implies that the labelling algorithm is feasible despite its exponential running

time in general. Also Müller-Hannemann and Schnee apply a labelling algorithm to solve multi-criteria problems in the time-expanded approach [MHS]. In that work, they found that the notion of Pareto-optimality is sometimes too strict, and they suggest to use a relaxed variant of Pareto-optimality in order to find all attractive train connections.

Given a departure station A, a destination station B, and a departure time t, we are interested in three problem variants: First and most generally, we want to find all *Pareto-optimal solutions* (i.e., the set of feasible solutions where the attribute-vector of one solution is not dominated by the attribute-vector of another solution; cf. Definition 2.2). Second, a resource constrained solution for (X, Y) shall be calculated: a solution that minimises the first criterion X while retaining the second criterion (also referred to as resource) Y below a given threshold (cf. the constrained shortest path problem in the bicriterion case described on page 8). We are mainly interested in the (EA,MNT) case here and refer to that problem as the *Earliest Arrival problem with Bounded number of Transfers* (EABT), which is defined to be the problem of finding a valid and consistent connection from A to B such that the arrival time at B is the earliest possible, and subject to the additional constraint that the total number of transfers performed in the path is not greater than an additionally given threshold k. Finally, the third problem deals with the *lexicographically first solution*: among all solutions that minimise X the one with minimal Y.

4.4.1 Time-Expanded Model

We use the realistic time-expanded graph (cf. Section 4.1.2) to find the lexicographically first Pareto-optimum as well as all Pareto-optimal solutions. Surprisingly, it will turn out that also all Pareto-optimal solutions can be computed by the normal version of Dijkstra's algorithm in the bicriterion case we consider here. Note that in the general case (i.e., either more than two criteria or two criteria that don't include travel time), this method doesn't work and a general multi-criteria algorithm like the labelling algorithm has to be used.

Lexicographically First Pareto-Optimum

We first consider the (EA,MNT) case. Every edge e of the realistic time-expanded graph is now associated with a pair of costs $(a, b) = (EA(e), MNT(e))$, where $EA(e)$ is the cost of e when solving the realistic EAP (Section 4.1.2) and $MNT(e)$ is the cost of e when solving MNTP (Section 4.3). Define on these cost-pairs (a, b) the canonical addition, i.e., $(a, b) + (a', b') = (a + a', b + b')$, and the lexicographic comparison, i.e., $(a, b) < (a', b') \Leftrightarrow (a < a')$ or $(a = a'$ and $b < b')$. To find the lexicographically first (EA,MNT) Pareto-optimal solution, as discussed already in Section 2.2.1 (page 8), it then suffices to run Dijkstra's algorithm by maintaining distance labels as pairs of integers and by initialising the distance label of the start-node s to $(0, 0)$. The optimal solution is found when a node at the destination station is considered for the first time during the execution of the algorithm. The (MNT,EA) case is symmetric

to the above and can be solved similarly. The proof of Theorem 3.1 can be easily adopted to establish the following.

Theorem 4.6 *A shortest path in the realistic time-expanded graph using cost-pairs associated with its edges as defined above constitutes a solution to the problem of finding the lexicographically first Pareto-optimal connection (among all connections that minimise the first criterion, the one with minimum value in the second criterion).*

Note that in the same way the latest-departure problem can be solved by minimising the difference between arrival time and actual departure time as second criterion.

All Pareto-Optima

As already mentioned, finding all Pareto-optimal solutions is generally a hard problem, since there can be an exponential number of them. However, the realistic fully time-expanded graph used in Section 4.2 has a very interesting and useful property: Given a start node s, for every node v in the time-expanded graph there is at most one Pareto-optimal s-v-path. Since the lexicographically first s-v-path is a Pareto-optimal path (cf. Section 2.2.1), this lexicographically first s-v-path is the desired Pareto-optimal s-v-path. Hence, we can compute for each node of the destination station the shortest path according to the cost-pairs $(a, b) = (EA(e), MNT(e))$ with the canonical addition and the lexicographic comparison. When the first node of the destination station is settled, we have found the first (EA,MNT)-Pareto-optimal connection. We let Dijkstra's algorithm continue; whenever a node of the destination station is settled with a smaller number of transfers than in any of the already found Pareto-optimal solutions, a new Pareto-optimal connection (which corresponds to the shortest path to that node) is found. The algorithm can be stopped when the Pareto-optimal solution with the lowest possible number of transfers (i.e., the solution of MNTP) is found.

Theorem 4.7 *All Pareto-optimal solutions for the two criteria EA and MNT can be enumerated during one run of Dijkstra's algorithm in the fully time-expanded graph using cost-pairs associated with its edges as defined above.*

Proof. It suffices to prove that each node of the destination station in the fully time-expanded graph has at most one Pareto-optimal solution. In that graph, every node v is associated with an absolute time value $t(v)$ (not only the time of the day). Thus, every v_1-v_2 path has length $l = t(v_2) - t(v_1)$ according to the EA criterion, and if k is the minimum number of transfers from v_1 to v_2, then (l, k) is the only Pareto-optimal solution for v_2, since all other solutions have the same EA length l and an equal or larger number of transfers. ∎

Now, similarly to Section 4.2.2, the realistic fully time-expanded graph does not need to be maintained explicitly. The above algorithm can again be simulated on the realistic time-expanded graph. The simulation is identical to that described in Section 4.2.2.

Earliest Arrival with Bounded Number of Transfers

As we will see in this section, the above algorithm for finding all Pareto-optimal solutions can be terminated earlier if only a solution is wanted that minimises the earliest arrival (EA) while retaining the number of transfers (MNT) below a given threshold k. By Theorem 4.7, all Pareto-optimal solutions are enumerated in lexicographical (EA,MNT) order during the algorithm, and thus the solution to the earliest arrival problem with bounded transfers (EABT) is found when the first node at the destination is processed by the algorithm that has a number of transfers less than or equal to k. The running time can be further improved by considering during the algorithm only *promising* paths with a number of transfer less than or equal to k (cf. also the labelling algorithm to solve the constrained shortest path problem outlined on page 8): all node labels with a number of transfers larger than k are ignored. We formulate this result as a corollary to Theorem 4.7:

Corollary 4.1 *The EABT problem can be solved in the time-expanded approach by applying a modification of the algorithm used in Theorem 4.7. The modified algorithm only considers promising paths and stops when the first feasible solution is determined.*

Note that in the case of minimising MNT and retaining EA less than or equal to t, the problem can be tackled similarly. Again, the algorithm described above for all Pareto-optimal solutions can be used. Now, the algorithm is terminated when a node at the destination is processed that has an arrival time larger than t. The solution is obtained by the last node with an arrival time less than or equal to t.

4.4.2 Time-Dependent Model

We use the train-route graph that models the realistic EAP (Section 4.1.3) to solve the three variants of the bicriteria optimisation problems.

Lexicographically First Pareto-Optimum

We present an approach similar to that described for the time-expanded case, but which finds only the lexicographically first (MNT,EA) Pareto-optimal solution. Later, we will explain why it fails to do the same for the (EA,MNT) case.

A solution to the lexicographically first (MNT,EA) Pareto-optimum problem can be easily achieved, if instead of using a single value for the cost of an edge, we use a pair (a, b) of values, and define the canonical addition and lexicographic comparison to these pairs (exactly as in Section 4.4.1). The attribute values a, b of the pairs are updated separately. For the (MNT,EA) case, a is the MNT cost defined on $\mathbb{N}$ (non-negative integers), and b is the EA cost defined on T (set representing time). An edge $e \in E$ has cost determined by a function $h_e : (\mathbb{N}, T) \to (\mathbb{N}, T)$, such that each attribute value is determined by the corresponding edge function.

Theorem 4.8 *A shortest path in the train-route graph using cost-pairs associated with its edges as defined above constitutes a solution to the problem of finding the lexicographically first (MNT,EA) Pareto-optimal connection.*

Proof. To prove the correctness, it suffices to show that the new edge cost function h_e is non-decreasing and with non-negative delay. Consider the modelling of the constant transfer cost case, and let $\tau, \tau_1, \tau_2 \in \mathbb{N}$ and $t, t_1, t_2 \in T$.

- If $e \in A$, then $h_e(\tau, t) = (\tau, t)$. If $(\tau_1, t_1) \leq (\tau_2, t_2)$, then $h_e(\tau_1, t_1) \leq h_e(\tau_2, t_2)$.
- If $e \in D$, then $h_e(\tau, t) = (\tau + 1, t + x) \geq (\tau, t)$, where $x \geq 0 \in T$ is the transfer time at the station that e belongs to. If $(\tau_1, t_1) \leq (\tau_2, t_2)$, then $h_e(\tau_1, t_1) = (\tau_1 + 1, t_1 + x) \leq (\tau_2 + 1, t_2 + x) = h_e(\tau_2, t_2)$.
- If $e \in R$, then $h_e(\tau, t) = (\tau, f_e(t)) \geq (\tau, t)$, since f_e has non-negative delay. If $(\tau_1, t_1) \leq (\tau_2, t_2)$, then
  - if $\tau_1 < \tau_2$, then $h_e(\tau_1, t_1) = (\tau_1, f_e(t_1)) < (\tau_2, f_e(t_2)) = h_e(\tau_2, t_2)$, and
  - if $\tau_1 = \tau_2 = \tau$ and $t_1 \leq t_2$, then $h_e(\tau_1, t_1) = (\tau, f_e(t_1)) \leq (\tau, f_e(t_2)) = h_e(\tau_2, t_2)$.

The case with variable transfer cost can be proved similarly. ∎

In contrast to the time-expanded case, the symmetric problem of finding a lexicographically first (EA,MNT) Pareto-optimal solution cannot be solved by just using appropriate edge cost-pairs on the train-route graph, as in Section 4.4.1. To show that such an approach may fail, consider the train-route graph shown in Figure 4.5 with a query to find an (EA,MNT) Pareto-optimal connection from A to D. Assume there is a train T_1 on the first route from A to C via B, and another train T_2 on the second route from A to D via C. Further assume that both T_1 and T_2 depart at A later than the given departure time, and let train T_1 arrive earlier than train T_2 at station C (let $C_1 < C_2$ be the corresponding arrival times at C, and D_2 be the arrival time of train T_2 at station D.) Consider the edge e from station C to station D. The time-dependent shortest-path algorithm with edge cost-pairs (a, b), where a is the EA cost and b is the MNT cost, will fail to find an (EA,MNT) Pareto-optimal solution, because the time-dependent function h_e in this case is *decreasing*. It holds that $(C_1, 1) < (C_2, 0)$, and

$$h_e(C_1, 1) = (D_2, 1) > (D_2, 0) = h_e(C_2, 0).$$

Hence, when the algorithm relaxes edge e will assign the label $(D_2, 1)$ to the head of e, and consequently output a connection with that destination label, i.e., a connection using train T_1 and T_2 with one change at station C. However, this is not the lexicographically first (EA,MNT) Pareto-optimal connection, since the connection that uses only train T_2 yields a lexicographically smaller destination label $(D_2, 0)$.

Earliest Arrival with Bounded Number of Transfers

In this section, we describe two algorithms for solving the EABT problem. The first one is an adaptation of the graph-copying method proposed in [BJ04] to our realistic time-dependent model (train-route graph). The second one is an adaptation of the labelling approach outlined in Section 2.2.2 (page 8; see also [Zie01]) for solving

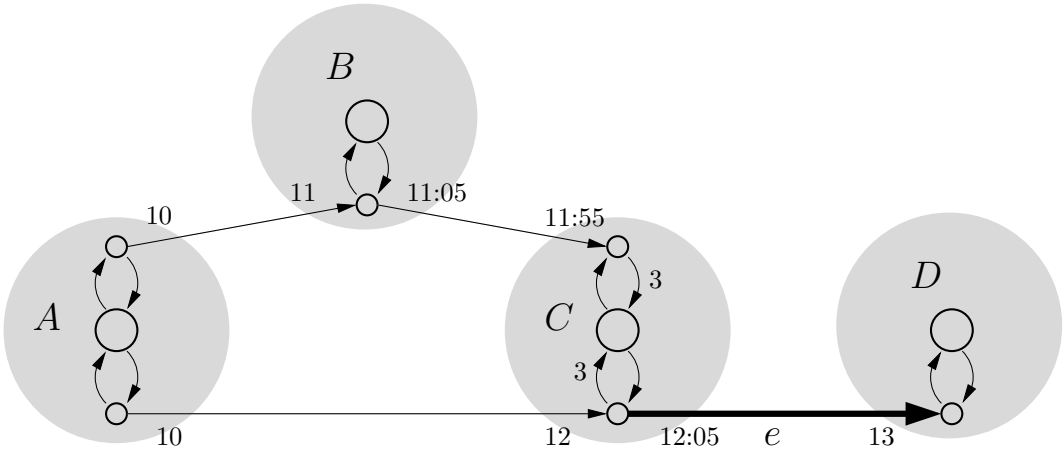

Figure 4.5: Lexicographically first (EA,MNT) connections cannot be found by simply using pairs as edge costs in the train-route graph: Assume that a connection from A via B arrives earlier in C (including transfer time) than a train from A to C, and both connections end with the same train from C to D. The label of the source of e becomes $(11{:}58, 1)$, and the time-dependent algorithm outputs the connection with one transfer via B, while the optimal connection (A-C-D) involves no transfer with the same arrival time.

resource constrained shortest paths to our realistic time-dependent model. Let A be the source station, B the destination station, t the departure time, and k the bound on the number of transfers.

The idea of [BJ04] adapted to our realistic time-dependent model is as follows. We construct a new graph $G' = (V', E')$ consisting of $k+1$ levels. Each level contains a copy of the train-route graph $G = (V, E)$, where $E = A \bigcup D \bigcup \overline{D} \bigcup R$) (cf. Section 4.1.3). For node $u \in V$, we denote its i-th copy, placed at the i-th level, by u_i, $0 \leq i \leq k$. For each edge $(u, v) \in A \bigcup R$, we place in E' the edges (u_i, v_i), for all i with $0 \leq i \leq k$. For each edge $(u, v) \in D \bigcup \overline{D}$, we place in E' the edges (u_i, v_{i+1}), for all i with $0 \leq i \leq k$. These edges, which connect consecutive levels, indicate transfers. With the above construction, it is easy to verify that a path from some node s_0 (copy of s at the 0-th level) to a node x_l (copy of x at the l-th level) represents a path from the station of s to the station of x with l transfers. In other words, the EABT problem can be solved by performing a time-dependent shortest path computation in G' aiming to find a shortest path from the node s_0 in level 0, where s represents the source station A in G, to the first possible u_i at level i, $0 \leq i \leq k$, where u is the node of the train-route graph representing the destination station B. Let $n = |V|$ and $m = |E|$. Since G' consists of $k+1$ copies of G, an application of Dijkstra's algorithm on G' for solving EABT takes $\mathcal{O}(mk + nk \cdot \log(nk))$ time (assuming that the computation of a time-dependent edge length can be done in constant time).

The adaptation of the labelling approach to our train-route graph $G = (V, E)$ is as follows. We use the time-dependent modification of Dijkstra's algorithm (cf. Section 4.1.3), where now we maintain up to $k+1$ (instead of one) labels. Each label is

of the form $(t_i, l_i)_u$, $0 \leq i \leq k$, representing the currently best time t_i to reach node u by performing exactly l_i transfers.

Let s be the node representing the source station A in the train-route graph. Initially, we insert to the priority queue the label $(t, 0)_s$. The priority queue is ordered according to time, aiming at computing the earliest arrival path. In the main loop of the algorithm, a label $(t_l, l)_u$ is extracted from the priority queue. If u represents the destination station (i.e., $station(u) = B$), the algorithm is terminated and t_l is the earliest arrival at B with less than or equal to k transfers. Otherwise, we relax the outgoing edges of u considering that u is reached on time t_l and with l transfers. In addition, if $(t_{l'}, l')$ was the last label of u that has been extracted, then we delete from the priority queue all labels of the form $(t_r, r)_u$ for $l < r < l'$, setting $l' = k$ in the case where $(t_l, l)_u$ was the first of the labels of u to have been extracted. In this way, we discard the labels dominated by $(t_l, l)_u$ from the priority queue, since for all such $(t_r, r)_u$ it holds that $t_l \leq t_r$ (as $(t_l, l)_u$ was extracted before $(t_r, r)_u$) and $l < r$. Clearly, such labels are no longer useful as $(t_l, l)_u$ corresponds to an s-u path at least as fast as the one suggested by $(t_r, r)_u$, and with less transfers than the latter. Exactly for the same reasons, when we relax an edge $(u, v) \in E$ having found a new label $(t_{l_1}, l_1)_v$ for v, we will actually update the label of v only if there has been so far no label of v extracted from the priority queue, or if the last label of v that was extracted had a number of transfers greater than l_1. The edge (u, v) is ignored if $l_1 > k$, and thus only promising paths (i.e., paths with less than or equal to k transfers) are considered during the algorithm.

Concerning now the complexity of the labelling algorithm, we need to see that for each node the total number of labels that is scanned in order to find those that are in the priority queue and can be safely deleted is $\mathcal{O}(k)$, while the total number of deletions is $\mathcal{O}(nk)$, where $n = |V|$. This is due to the fact that we only check the labels from the last known (by a *delete-min* operation) number of transfers, until the previous one. In this way, each label is checked at most once throughout the execution of the algorithm. Since each edge will be relaxed at most $k + 1$ times, the total number of relaxations will be $\mathcal{O}(mk)$, where $m = |E|$. We can also see that the total number of labels that is in the priority queue is at most $\mathcal{O}(nk)$. Because of this, the time for a *delete-min* or a *delete* operation is $\mathcal{O}(\log(nk))$. This means that the total time needed for the algorithm is $\mathcal{O}(mk + nk \cdot \log(nk))$, which is asymptotically the same with the previous one. The discussion in this section establishes the following.

Theorem 4.9 *The Earliest Arrival with Bounded Number of Transfers problem can be solved in the extended time-dependent model in time $O(mk + nk \cdot \log(nk))$, where n (resp. m) is the number of nodes (resp. edges) of the train-route graph, and k is the bound in the number of transfers.*

All Pareto-Optima

Of course, all Pareto-optimal solutions could be calculated by the standard labelling algorithm applied to the train-route graph. In order to consider fewer paths and to be able to terminate the algorithm when all Pareto-optimal solutions at the

	Time-Expanded		Time-Dependent			
Graph	Nodes	Edges	Station Nodes	Route Nodes	Timetable Edges	Transfer Edges
Simplified	289432	578864	6685	–	17577	–
Realistic	578864	1131164	6685	79784	72779	159568

Table 4.1: Graph parameters for the realistic models, applied to the same input timetable `ger-longdist1`, and compared to the simplified models used in the previous chapter.

destination have been found, we propose to apply the modified labelling algorithm introduced above for the EABT problem with beforehand calculated bound on the number of transfers: First, solve EAP and count the number of transfers found, say M. Then, run the labelling algorithm of above for solving EABT. Recall that the algorithm maintains for each node $M+1$ labels, where label i, for $0 \leq i \leq M$, stores the best EA solution performing exactly i transfers (provided that such a solution exists), and discards dominated paths. Hence, instead of stopping the algorithm when a label belonging to the destination station is processed in the main loop of the algorithm, we can just continue with the execution of the algorithm to produce the next solution with at most $M-1$ transfers, considering the bound k being equal to $M-1$, and so on, until a path solving MNTP is found (or until no new path is found). The above discussion is summarised as corollary of the previous theorem.

Corollary 4.2 *The modified labelling algorithm of Theorem 4.9 can enumerate all Pareto-optimal solutions for the two criteria EA and MNT in the extended time-dependent model.*

4.5 Experimental Comparison of the Models

In this section we investigate how the time-expanded and the time-dependent approaches compare when the realistic versions of the problems and models are considered. The general experimental setup is the same as in Section 3.5 (pages 28 et seq.). As input data we use a modification of the `ger-longdist` timetable which we refer to as `ger-longdist1`. Since train changes are considered here, we only use the trains of the timetable that operate on one specific day, whereas in the simplified case discussed before we assumed that all trains operate daily. The real-world and random queries as described in Section 3.5.1 (page 28) are used. Table 4.1 shows the parameters of the graphs used in the realistic models compared to the original models.

	Time-Expanded			Time-Dependent		
Problem	Nodes	Edges	Time [ms]	Nodes	Edges	Time [ms]
EA-realistic	40624	73104	78	44731	83662	50
MNT	101731	138417	125	26680	83173	38
Lex-F(MNT,EA)	99061	137075	161	28272	83363	83
All Pareto-Opt	123943	236887	287	78412	145444	181

Table 4.2: Main results for the realistic models, applied to the input timetable `ger-longdist1`: the average number of nodes and edges touched by the algorithms, and the CPU time in milliseconds for the different problems solved.

4.5.1 Implementation Environment

For both approaches we implemented the described solutions for the realistic earliest arrival (EA-realistic) problem (Section 4.1), the minimum number of transfers (MNT) problem (Section 4.3), the all Pareto-optima (All Pareto-Opt) problem involving EA and MNT as the two criteria (Section 4.4), the lexicographically first (MNT,EA) Pareto-optimum (Lex-F(MNT,EA)), and the lexicographically first (EA,MNT) Pareto-optimum (Lex-F(EA,MNT))—the latter only for the time-expanded model; see Section 4.4.2. Moreover, for the time-dependent model, we have also considered the two algorithms for solving the earliest arrival with bounded number of transfers problem (Section 4.4.2): the one based on the graph-copying approach (EABT-BJ) and the one based on the labelling approach (EABT-L).

In the time-expanded implementations we again reduced the node set by omitting the departure-nodes in the realistic time-expanded graph (see Section 3.4.1, page 25). Also in the realistic time-dependent implementations we applied heuristics similar to the "avoid binary search" method as described in Section 3.4.2 (pages 28 et seq.).

4.5.2 Results

Table 4.2 reports on the comparison between the problems solved in both realistic versions of the time-expanded and the time-dependent approaches. The key parameters—number of touched nodes, edges, and average running time—are displayed for the real-world queries. More details, as well as results on random queries and other problems, are shown in Tables 4.3 and 4.4.

Concerning CPU time, the results show that the time-dependent approach still performs better than the time-expanded approach in all cases considered. However, the gap for the realistic EAP is not as huge as it was for the simplified EAP; in fact, for real-world queries the speedup is now only 1.5. Moreover, if one considers the number of touched edges, then for real-world queries the speedup is even smaller than 1 (i.e., the time-expanded approach is better). For the other problems considered, the time speedup ranges from 1.5 to 3.3, while the touched-edges speedup is 1.6 in all cases.

In particular, for the MNT problem, the extended time-dependent model (train-

Time-Expanded Model

Problem	Real Queries	Time [ms]	Average Nodes	Average Edges
EA-simplified	✗	70	20760	41519
EA-realistic	✗	78	40624	73104
MNT	✗	125	101731	138417
Lex-F(EA,MNT)	✗	82	40628	73123
Lex-F(MNT,EA)	✗	161	99061	137075
All Pareto-Opt	✗	287	123943	236887
EA-simplified		106	34469	61955
EA-realistic		122	61159	111301
MNT		212	169299	239841
Lex-F(EA,MNT)		129	61195	111386
Lex-F(MNT,EA)		259	163438	234297
All Pareto-Opt		405	170946	330150

Time-Dependent Model

Problem	Real Queries	Time [ms]	Average Touched	Average Edges
EA-simplified	✗	10	2967	4365
EA-realistic	✗	50	44731	83662
MNT	✗	38	26680	83173
Lex-F(MNT,EA)	✗	83	28272	83363
EABT-L	✗	79	39070	71127
EABT-BJ	✗	77	46127	82749
All Pareto-Opt	✗	181	78412	145444
EA-simplified		11	3315	
EA-realistic		54	48200	89953
MNT		47	33455	96646
Lex-F(MNT,EA)		106	35262	97833
EABT-L		104	49179	90406
EABT-BJ		107	60493	109298
All Pareto-Opt		219	92378	172514

Table 4.3: Detailed results for the realistic problems. The upper table concerns the time-expanded implementations, and the lower table the time-dependent ones. All results are based on the `ger-longdist1` timetable. For comparison with the original models, we have included the results for the simplified version of the earliest arrival problem (EA-simplified). Table 4.4 shows more detailed results for the time-dependent implementation.

Problem	Real Queries	Timetable Edges touched	Transfer Edges touched	El. conn. per timetable edge
EA-simplified	✗	4365	–	3.126
EA-realistic	✗	38168	45494	0.319
MNT	✗	21558	61615	–
Lex-F(MNT,EA)	✗	22901	60462	0.178
EABT-L	✗	32093	39034	0.247
EABT-BJ	✗	37499	45250	0.261
All Pareto-Opt	✗	65753	79691	0.211
EA-simplified		4811	–	3.005
EA-realistic		41011	48942	0.311
MNT		27235	69411	–
Lex-F(MNT,EA)		28779	69054	0.172
EABT-L		40638	49768	0.242
EABT-BJ		49392	59906	0.254
All Pareto-Opt		77610	94904	0.204

Table 4.4: More detailed results for the time-dependent implementation, distinguishing the number of edges between timetable edges (which have time-dependent lengths) and transfer edges (which have static lengths). Also shown is the average number of elementary connections that are touched during the calculation of edge lengths. Note that a value of less than one makes sense here, since for all transfer edges this number is zero: no elementary connection has to be touched at all.

route graph) is clearly superior to the extended time-expanded model (realistic time-expanded graph); the speedup with respect to CPU time is 3.3 for real-world queries and 4.5 for random queries. This is clear since both graphs are static, and the realistic time-expanded graph is much larger and contains a lot of redundant information that is not needed for solving the MNT problem. A similar observation holds for the lexicographically first (MNT,EA) Pareto-optimum problem, where the CPU speedup is 1.9 for real-world queries and 2 for random queries, and for the all Pareto-optima problem, where the CPU speedup is 1.5 for real-world queries and 1.8 for random queries.

We would like to note that the solution to the problem of finding all Pareto-optima exhibits a quite stable behaviour compared to EAP in both approaches. It is 3.6 times slower than EAP for real-world queries in both models, and about the same factor slower when random queries are considered (3.3 in the time-expanded model and 4 times slower in the time-dependent model).

Further observations regarding each model separately are as follows.

Time-Expanded Model.

The graph used in the realistic EAP (Table 4.3) has less than twice as many nodes and edges as the graph used in the simplified EAP, and is of very similar structure. Thus, it needs only slightly more time to solve the realistic EAP than to solve the simplified EAP (11% more time using real-world queries and 16% more using random queries). The lexicographically first (EA,MNT) Pareto-optimal problem is solved in a very similar way as the realistic EAP. It is interesting to observe that the CPU-time as well as the average number of nodes and edges touched are almost identical. In contrast, the MNT, the lexicographically first (MNT,EA) Pareto-optimum, and the all Pareto-optima problems require more CPU time than the realistic EAP, since a much bigger part of the graph has to be explored.

Time-Dependent Model.

Although the train-route graph has approximately 13 times more nodes and edges than the graph of the original time-dependent model (see Table 4.1), the experimental results reported in Table 4.3 show that the time required for solving the realistic EAP is only 5 times slower than that of the simplified EAP. This is due to the fact that the structure of the two graphs is quite different. While the original time-dependent graph has only timetable edges, almost 70% of the train-route graph edges have constant cost and the timetables of the other edges have much less events than their corresponding edges in the original time-dependent graph.

The MNT problem is solved faster than the realistic EAP, since in this case all edge lengths in the train-route graph are static and hence no binary search is needed.

The solution of the lexicographically first (MNT,EA) Pareto-optimum problem involves (again) time-dependent edge lengths and is slower than both MNTP and the realistic EAP, even though its performance parameters (number of nodes and edges touched) are similar to those of MNTP. This is due to the fact that in this problem more computations are needed in order to maintain the priority heap. It

is interesting to observe that the algorithm for solving the lexicographically first (MNT,EA) Pareto-optimum problem touches on average roughly half of the elementary connections than when the realistic EAP is solved. This can be explained as follows. While solving the Lex-F(MNT,EA) problem, the first connections to be considered are the ones with no transfers (i.e., those that belong to the train routes that pass through the departure station), the next connections will be those with only one transfer, and so on, thus avoiding to perform a transfer as long as this is possible. Now, the implementation does not use binary search at a timetable edge, unless the source of the edge has been reached through a transfer edge of the same station. In addition, the time of this problem is greater than the time of EAP, since in order to compare two node labels there will exist quite often the need to compare the second cost parameter (time), as the first one (number of transfers) is the same for much more nodes.

Both algorithms for solving the EABT problem require practically the same time. It is interesting to observe though that the labelling approach (EABT-L) touches much less nodes and edges than the graph-copying approach (EABT-BJ).

The algorithm for enumerating all Pareto-optimal solutions uses the computation of the EABT problem as a sub-procedure. It needs, however, only double the time it is required for the solution of EABT. The same applies when it is compared to the algorithm for solving the lexicographically first (MNT,EA) Pareto-optimum problem.

4.5.3 Discussion

We have discussed time-expanded and time-dependent models for several kinds of single-criteria and bicriteria optimisation problems on timetable information systems. In the time-expanded case, extensions that model more realistic requirements (like modelling train transfers) could be integrated in a more-or-less straightforward way and the central characteristic of the approach is that a solution to a given optimisation problem could be provided by solving a shortest path problem in a static graph, even for finding all Pareto-optimal solutions in the considered bicriteria optimisation problems. In the time-dependent case, the central characteristic of having one node per station had to be violated when more realistic requirements (like the integration of minimum transfer times at stations) were considered, and more sophisticated techniques in the bicriteria optimisation problems had to be used for their effective solution. Nevertheless, all the problems under consideration could be efficiently modelled in an extension of the time-dependent model introduced here. Moreover, it turned out that such modelling was more compact than in the extended time-expanded model, thus resulting in better performance in practice.

Our experimental study showed that the time-dependent approach is clearly superior with respect to performance when the original version of the models is considered, and speedup factors in the range from 10 to 40 were observed. When considering extensions of the models for the solution of realistic versions of optimisation problems, the time-dependent approach still performs better, but with a much smaller difference though (the speedup is now reduced in the range of 1.5 to

3.3). The time-expanded approach benefits in this case from the straight-forward modelling that allows more direct extensions and effective solutions. When other optimisation criteria shall be integrated, it is more likely that it can be modelled directly as edge lengths in the time-expanded model than in the time-dependent model. In case the criterion can be expressed as additive costs for elementary connections, these costs induce edge lengths in the time-expanded graph, while it is not clear if the costs can be mapped to feasible edge lengths in the time-dependent approach, since only the first elementary connection per edge is considered.

Chapter 5

Towards Efficient Timetable Information

In the previous two chapters several timetable information problems have been reduced to the problem of finding a shortest path in an appropriately defined graph. The focus had been on the modelling, and the shortest paths have been computed by *standard algorithms*, namely Dijkstra's algorithm and a variant of Dijkstra's algorithm for time-dependent graphs. As already discussed previously, the crucial point for the core of a timetable information system is to minimise the average running-time per query. Not only in timetable information the scenario arises where a large number of on-line shortest path queries in a huge graph have to be processed as fast as possible: also in many other practical applications, including route planning for car traffic [IOAI91, CF94, JMN99, SWN92, JP02], integrated travel information systems [SKC93], database queries [SFG97, GSVGM98], and Web searching [BJM00], the algorithmic core problem consists in solving shortest-path queries as fast as possible.

In practice, the usual approach to tackle the shortest path problems arising in scenarios like the above is to use heuristic methods, which in turn implies that there is no guarantee for an optimal answer. On the contrary, we are interested in *distance-preserving* algorithms (i.e., shortest path algorithms that produce an optimal answer for any input instance). Distance-preserving algorithms were not in wide use in traffic information systems, mainly because the average response time was perceived to be unacceptable. However, the results in [SWW99, SWW00] showed that distance-preserving variants of Dijkstra's algorithm are competitive in the sense that they do not constitute the bottleneck operation in the above scenario.

This chapter deals with further improving the running-time of the shortest-path algorithms by investigating distance-preserving speedup techniques. First, an approach using so-called *"multi-level graphs"* is introduced and investigated in general, and we will see that it is especially suited for application in our timetable information models. The extensive experimental analysis we conducted confirms this observation and moreover yields the rather surprising result that several hierarchical levels are beneficial in practice. Parts of this work appeared as "*Using Multi-Level Graphs for Timetable Information in Railway Systems*" [SWZ02]. Further, a brief review of other speedup techniques is provided including the discussion whether the tech-

niques can or cannot be applied to our timetable information models. We are also interested whether it makes generally sense to combine these speedup techniques, and present an experimental study using several generated and real-world graphs, which has been published as "*Combining Speed-Up Techniques for Shortest-Path Computations*" [HSW04].

5.1 The Multi-Level Graph Approach

Several of the approaches used so far in traffic engineering introduce speedup techniques based on hierarchical decomposition. For example, in [IOAI91, AJ94, CF94, JP02] graph models are defined to abstract and store road maps for various routing planners for private transport. In her PhD thesis [Fli04], Flinsenberg successfully used an hierarchical approach similar to the HiTi graphs introduced by Jung and Pramanik [JP02] in the area of car navigation. Similarly, Siklóssy and Tulp [ST91] introduce a space reduction method for shortest paths in a transportation network, and Buchholz [Buc00] investigates in his PhD thesis more generally how precomputed information stored as hierarchical graphs can help in—approximately or optimally—finding shortest paths. The idea behind such techniques is to reduce the size of the graph in which shortest path queries are processed by replacing precomputed shortest paths by edges. The techniques are hierarchical in the sense that the decomposition may be repeated recursively. Several theoretical results on shortest paths, regarding planar graphs [Fre91, Fre95, KPSZ96] and graphs of small treewidth [CZ00, CZ98], are based on the same intuition.

So far, however, there exist few systematic studies of hierarchical decomposition techniques, especially when concrete application scenarios are considered. In [SWW00], we made a first attempt to introduce and evaluate a speedup technique based on decomposition, called selection of stations. Based on a small set of selected nodes an auxiliary graph is constructed, where edges between selected nodes correspond to shortest paths in the original graph. Consequently, shortest path queries can be processed by performing parts of the shortest path computation in the much smaller and sparser auxiliary graph. In [SWW00], we extensively studied this approach for one single choice of selected nodes, and the results are quite promising.

We introduce a hierarchical decomposition technique called the *multi-level graph model* that generalises the approach of [SWW00]. A multi-level graph $\mathcal{M}$ of a given weighted (directed) graph $G = (V, E)$ is a graph which is determined by a sequence of subsets of V and which extends E by adding multiple levels of edges. This allows to efficiently construct a subgraph of $\mathcal{M}$ which is substantially smaller than G and in which the shortest path distance between any of its nodes is equal to the shortest path distance between the same nodes in G. Under the new framework, the auxiliary graph used in [SWW00]—based on the selection of stations—can be viewed as adding just one level of edges to the original graph. The crucial difference of the multi-level graphs we are investigating and HiTi-graph like approaches [JP02,

Fli04], which are mainly applied in the area of car navigation, is that in the latter approaches the underlying graphs are decomposed by *edge separators*, whereas our technique is based on *node separators*. We believe that for application in timetable information, a decomposition of the railway network by stations (represented by nodes) is better suited than a decomposition by edges, since intuitively the network is easily decomposable by removing important hubs (i.e., important stations). We follow this intuition in the decompositions shown in Section 5.2.3.

5.1.1 Definition of the Multi-Level Graph

Let $G = (V, E)$ be a weighted (directed) graph with non-negative edge weights. The *length* of a path is the sum of the weights of the edges in the path. The *multi-level* graph $\mathcal{M}$ of G is, roughly speaking, a graph that extends G in two ways: On the one hand, it extends the edge-set of G by multiple *levels* of edges. On the other hand, it provides the functionality to determine for a pair of nodes $s, t \in V$ a subgraph of $\mathcal{M}$ such that the length of a shortest path from s to t in that subgraph is equal to the shortest path length in G. To achieve this, we use a special data structure called the *component tree* (a tree of connected components).

The objective is that the resulting subgraph of $\mathcal{M}$ is substantially smaller than the original graph G. Then, single-pair shortest path algorithms can be applied to the smaller graph, improving the performance. The multi-level graph is built on the following input: (i) a weighted (directed) graph $G = (V, E)$; and (ii) a sequence of l subsets of nodes S_i $(1 \leq i \leq l)$. The subsets S_i are decreasing with respect to set inclusion: $V \supset S_1 \supset S_2 \supset \ldots \supset S_l$.

To emphasise the dependence on G and the sets $S_1, \ldots, S_l$, we shall refer to the multi-level graph by $\mathcal{M}(G; S_1, \ldots, S_l)$. The node-sets S_i will determine the levels of the multi-level graph. In the following, we shall discuss the construction of the multi-level graph and of the component tree.

Level Construction

Each level of $\mathcal{M}(G; S_1, \ldots, S_l)$ is determined by a set of edges. The endpoints of these edges determine the node set of each level. For each set S_i $(1 \leq i \leq l)$, we construct three sets of edges:

- *level edges* $E_i \subseteq S_i \times S_i$;
- *upward edges* $U_i \subseteq (S_{i-1} \setminus S_i) \times S_i$; and
- *downward edges* $D_i \subseteq S_i \times (S_{i-1} \setminus S_i)$.

We call the triple $L_i := (E_i, U_i, D_i)$ the *level* i of the multi-level graph. We further say that $L_0 := (E, \emptyset, \emptyset)$ is the *level zero*, where E are the edges of the original graph G. With the level zero there are totally $l + 1$ levels, so we say that $\mathcal{M}(G; S_1, \ldots, S_l)$ is an $l + 1$-level graph. Figure 5.1 illustrates a 3-level graph.

The construction of the levels is iterative, so we assume that we have already constructed the level L_{i-1}. The iteration begins with $i = 1$. For each node u in S_{i-1}

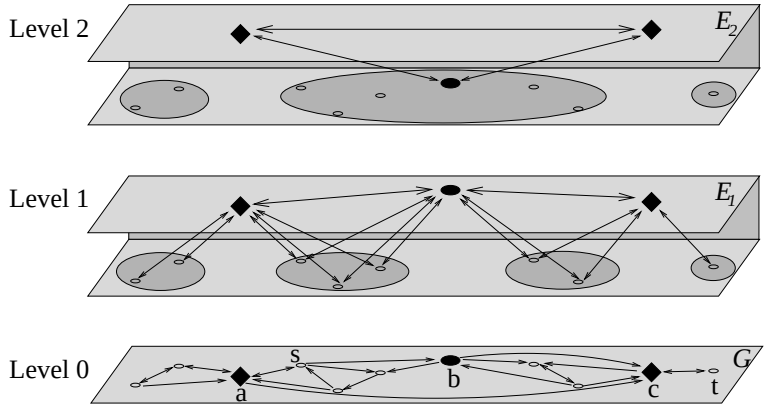

Figure 5.1: A simple example of a 3-level graph. Level zero consists of the original graph G. The sets of nodes that define the 3-level graph are $S_1 = \{a, b, c\}$ and $S_2 = \{a, c\}$. In order to show the levels, we draw copies of each node for the levels one and two, but actually there is only one occurrence of them in the 3-level graph. The levels one and two are each split into two planes, where the upper plane contains the edges E_i, and the lower plane shows the connected components in the graph $G - S_i$. The edges U_i and D_i connect nodes in different planes of one level.

consider a shortest-path tree T_u (rooted at u) in the graph (S_{i-1}, E_{i-1}). Candidates for edges in level L_i are all the edges $S_i \times S_i$ for level edges, $(S_{i-1} \setminus S_i) \times S_i$ for upward edges, and $S_i \times (S_{i-1} \setminus S_i)$ for downward edges. The condition to decide whether one candidate edge (u, v) is actually taken for the sets E_i, U_i and D_i is the following:

> L_i contains an edge (u, v) if and only if no internal node of the u-v path in T_u belongs to S_i.

In other words, if the u-v path contains no node of S_i except for the two endpoints u and v, the edge (u, v) is added to L_i. The weight of a new edge (u, v) is the shortest path length from u to v in G. Note that the level L_i is not uniquely determined by this construction, since the shortest-path trees are not unique. Now, we can define the multi-level graph as

$$\mathcal{M}(G; S_1, \ldots, S_l) := (V, E \cup \bigcup_{i=1\ldots l} (E_i \cup U_i \cup D_i)).$$

Connected components

Consider the subgraph of G that is induced by the nodes $V \setminus S_i$. We will use the following notation:

- the set of connected components is denoted by $\mathcal{C}_i$, and a single component is usually referred to by C;
- $V(C)$ denotes the set of nodes of a connected component C of $\mathcal{C}_i$;

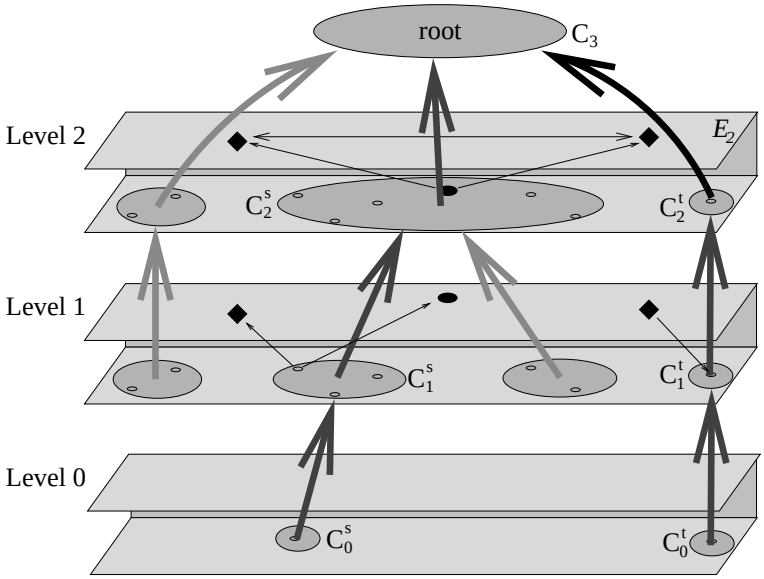

Figure 5.2: The component tree for the 3-level graph in Figure 5.1. Only the leaves for the nodes s and t are shown. The thin black edges are the edges E_{st} that define the subgraph with the same shortest path length as G.

- for a node $v \in V \setminus S_i$, let C_i^v denote the component in $\mathcal{C}_i$ that contains v;
- a node $v \in S_i$ is called *adjacent* to the component $C \in \mathcal{C}_i$, if v and a node of C are connected by an edge (ignoring direction);
- the set of adjacent nodes of a component C is denoted by $Adj(C)$.

The edges E_i, U_i and D_i can be interpreted in terms of connected components as follows (see Figure 5.1). The edges E_i resemble the shortest paths between nodes of S_i that pass through a connected component (i.e., if two nodes x and y are adjacent to the same component, and the shortest path from x to y is inside that component, then there is an edge from x to y representing that shortest path). This includes edges in G that connect two nodes of S_i. Notice that for a pair of nodes in S_i, the subgraph of $\mathcal{M}$ induced by E_i suffices to compute a shortest path between these nodes.

In the same way, the edges U_i represent shortest paths from a node inside a connected component to all nodes of S_i adjacent to that component, and the edges D_i represent the shortest paths from the adjacent nodes of a component to a node of the component.

Component tree

The data structure to determine the subgraph of $\mathcal{M}$ for a pair of nodes $s, t \in V$ is a tree with the components $\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_l$ as nodes. Additionally, there is a root C_{l+1}, and for every node $v \in V$ a leaf C_0^v in the tree (we assume that $Adj(C_0^v) := \{v\}$ and $Adj(C_{l+1}) := \emptyset$). The parent of a leaf C_0^v is determined as follows: Let i be the largest i with $v \in S_i$. If $i = l$, the parent is the root C_{l+1}. Otherwise, the smallest

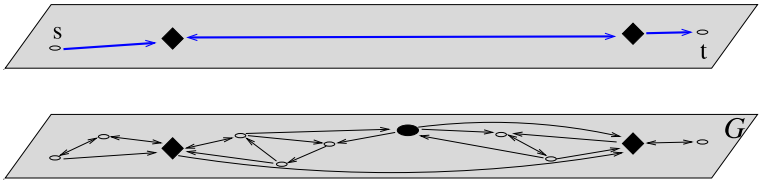

Figure 5.3: The subgraph $\mathcal{M}_{st}(G)$ of a sample query (top), and the original graph G (bottom).

level where v is contained in a connected component is level $i+1$, and the parent of C_0^v is the component $C_{i+1}^v \in \mathcal{C}_{i+1}$.

The parent of the components in $\mathcal{C}_l$ is also the root C_{l+1}. For one of the remaining components $C_i \in \mathcal{C}_i$, the parent is the component $C'_{i+1} \in \mathcal{C}_{i+1}$ with $V(C_i) \subseteq V(C'_{i+1})$. Figure 5.2 illustrates the component tree of the 3-level graph in Figure 5.1.

Subgraph

For the given pair of nodes $s, t \in V$ we consider the C_0^s-C_0^t path in the component tree. Let L be the smallest L with $C_L^s = C_L^t$ (i.e., $C_L^s = C_L^t$ is the lowest common ancestor of C_0^s and C_0^t in the tree). Then, with our notation for the components, the C_0^s-C_0^t path is

$$(C_0^s, C_k^s, C_{k+1}^s, \ldots, C_L^s = C_L^t, \ldots, C_{k'+1}^t, C_{k'}^t, C_0^t),$$

where $k > 0$ and $k' > 0$ are the levels of the parents of C_0^s and C_0^t as defined above (cf. darker tree edges in Figure 5.2). The subgraph with the same s-t shortest-path length as G is the subgraph $\mathcal{M}_{st}$ of $\mathcal{M}$ induced by the following edge set:

$$\begin{aligned} E_{st} := \quad & E_{L-1} \\ & \cup \bigcup_{i=k,\ldots,L-1} \{(u,v) \in U_i | u \in Adj(C_{i-1}^s), v \in Adj(C_i^s)\} \\ & \cup \bigcup_{i=k',\ldots,L-1} \{(u,v) \in D_i | u \in Adj(C_i^t), v \in Adj(C_{i-1}^t)\}. \end{aligned}$$

A sample subgraph $\mathcal{M}_{st}$ is illustrated in Figure 5.3. The correctness of the multi-level graph approach is shown by the following theorem.

Theorem 5.1 *The length of a shortest s-t path is the same in the graphs G and $\mathcal{M}_{st}(G; S_1, \ldots, S_l)$.*

Proof. Let $s, t \in V$ be a pair of nodes G for which a s-t path in G exists, and let $C_0^s, C_k^s, \ldots, C_L^s = C_L^t, \ldots, C_{k'}^t, C_0^t$ be the corresponding graph in the component tree. By definition, every edge (u, v) in $\mathcal{M}_{st}$ has a weight that is at least as large as the shortest-path length from u to v in G. Hence, the length of a shortest s-t path in $\mathcal{M}_{st}$ can never be smaller than the one in G. It remains to prove that there is a s-t path in $\mathcal{M}_{st}$ with the same length as a shortest s-t path P in G. To prove this, it suffices to prove the following claims, where $1 \leq x \leq l$:

1. For each pair of nodes $u, v \in S_x$ such that there exists a u-v path in G, the graph (S_x, E_x) contains a path with the same length as a shortest u-v path in G.

2. For the subgraph $\mathcal{M}'$ of $\mathcal{M}$ induced by the edge set

$$E_x \cup \bigcup_{i=k,\ldots,x} \{(u,v) \in U_i | u \in Adj(C_{i-1}^s), v \in Adj(C_i^s)\}$$

   it holds that for each node $w \in S_x$ that is reachable from s in G there exists a path from s to w in $\mathcal{M}'$ with the same length as a shortest s-w path in G.

3. For the subgraph $\mathcal{M}'$ of $\mathcal{M}$ induced by the edge set

$$E_x \cup \bigcup_{i=k',\ldots,x} \{(u,v) \in D_i | u \in Adj(C_i^t), v \in Adj(C_{i-1}^t)\}$$

   it holds that for each node $w \in S_x$ from which t is reachable in G there exists a path from w to t in $\mathcal{M}'$ with the same length as a shortest w-t path in G.

We first show how the proof is completed using the above claims, and then give the proofs of the claims. The value L is the level of the lowest common ancestor of C_0^s and C_0^t in the component tree. Because of this, s and t are in different components of the subgraph induced by $V - S_{L-1}$, and therefore at least one node of a shortest s-t path in G has to be in S_{L-1}. Let w (resp. z) be the first (resp. last) node of P that belongs to S_{L-1}. Then, nodes w and z split P into three (not necessarily non-empty) parts P_1, P_2 and P_3. By Claim 1, it follows that there is a w-z path in $\mathcal{M}_{st}$ with the same length as P_2. Similarly, by Claim 2, it follows that there is a path in $\mathcal{M}_{st}$ from s to w with the same length as P_1, and by Claim 3 that there is a path in $\mathcal{M}_{st}$ from z to t with the same length as P_3. The concatenation of these three paths is an s-t path in $\mathcal{M}_{st}$ with the same length as P.

We now turn to the proofs of the claims. The proofs are by induction on x. We give the proof of Claim 1; the proofs of the other claims follow similarly. We start with the basis of the induction ($x = 1$).

Let u and v be two nodes of S_1 and $P = (u = v_1, \ldots, v_z = v)$ be the shortest u-v path in the shortest-path tree T_u in G considered in the definition of the levels. If no internal node of that path belongs to S_1, by the definition of E_1, there is an edge $(u, v) \in E_1$ whose weight is the length of P, and we are done. Otherwise, some of the internal nodes of P belong to S_1, and we consider all the subpaths P_j of P, where P_1 is the part from u to the first node belonging to S_1, then P_2 is the part from the latter node to the second node in P belonging to S_1, and so on. The end-nodes of each subpath P_j are connected by an edge in E_1, because for these subpaths there is no internal node in S_1, and the weight of such an edge is exactly the length of P_j in G. The combination of all these edges is the path in (S_1, E_1) we are looking for.

Now, assume that the claim is true for any value smaller than x. Then, the induction step for x is proved in exactly the same way as for the basis, by replacing G by (S_{x-1}, E_{x-1}), S_1 by S_x, and E_1 by E_x. ∎

5.1.2 Regular Multi-Level Graphs

The basic idea of multi-level graphs is that the subgraph $\mathcal{M}_{st}$ is small and thus a shortest path can be found faster using that subgraph instead of the original graph G. In general, this is not necessarily true, actually the subgraph $\mathcal{M}_{st}$ may even be larger than G for a bad selection of the sets $S_1, \dots S_l$, for example if $l = 1$ and S_1 doesn't decompose the graph G at all. However, for *regular multi-level graphs*, under certain assumptions concerning the decomposition, we are able to prove a bound on the total number of edges in the multi-level graph itself and on the size of the subgraphs $\mathcal{M}_{st}$. The latter size depends crucially on the level L determining the subgraph $\mathcal{M}_{st}$, so we are also interested in the probability that for a random query at least a given level L is reached.

We use the following notions: The original graph $G = (V, E)$ consists of $n = |V|$ nodes and $m = |E|$ edges. Further, the set S_i $(1 \le i \le l)$ decomposes the graph G in connected components. Let the set of these components be denoted by $\mathcal{C}_i$, and $C_{i,j}$ be the j-th component in C_i $(1 \le j \le |\mathcal{C}_i|)$. Further, let A be the maximum number of adjacent nodes of a component over all components $C_{i,j}$.

Number of Edges in a Regular Multi-Level Graph

First, we count the total number of upward (U) and downward (D) edges: In level i $(1 \le i \le l)$, there are only upward/downward edges from/to nodes in $S_{i-1} \setminus S_i$. For each of these nodes there are at most A upward and A downward edges. If we sum over all levels we get:

$$\begin{aligned} |U| + |D| &\le \sum_{i=1}^{l} 2A(|S_{i-1}| - |S_i|) && (5.1)\\ &= 2A(n - |S_1| + |S_1| - |S_2| \dots |S_{l-1}| - |S_l|) && (5.2)\\ &= 2A(n - |S_l|) && (5.3)\\ &\le 2An. && (5.4) \end{aligned}$$

The level edges E_i in level i are on the one hand the edges induced by the nodes S_i in G (denoted by $E_G(S_i)$), and on the other hand the newly constructed edges. The latter edges can be at most $A(A-1)$ for each component: the worst case occurs when a component is substituted with a complete directed graph on all the adjacent nodes of that component. Hence,

$$|E_i| \le |E_G(S_i)| + A(A-1)|\mathcal{C}_i|. \tag{5.5}$$

With these observations, the total number of level edges (LE) can be obtained by taking the sum over all levels i, $1 \le i \le l$, and we get the following bound:

$$|LE| \le \sum_{i=1}^{l} |E_G(S_i)| + A(A-1) \sum_{i=1}^{l} |\mathcal{C}_i|. \tag{5.6}$$

In general, both addenda can become large if the decomposition by the sets $S_1, \dots, S_l$ is bad. However, if the following two assumptions—which intuitively characterise a

"good decomposition"—are true, we can show that also the number of level edges $|LE|$ is relatively small.

Assumption 5.1 *For each i, $1 \leq i < l$, the part of $E_G(S_i)$ that is not in $E_G(S_{i+1})$ is at most half as big as the same part in the previous level:*

$$|E_G(S_i) \setminus E_G(S_{i+1})| \leq |E_G(S_{i-1}) \setminus E_G(S_i)|/2.$$

Assumption 5.2 *For the decomposition of the graph G into components holds that $\sum_{i=1}^{l} |\mathcal{C}_i| \leq n$.*

A multi-level graph $\mathcal{M}(G)$ is called *regular* if it complies with these assumptions. Because of Assumption 5.1, for the first part of the above bound (Inequality 5.6) on $|LE|$, the following holds:

$$\begin{aligned}
\sum_{i=1}^{l} |E_G(S_i)| &= l|E_G(S_l)| + \\
&\quad (l-1)|E_G(S_{l-1}) \setminus E_G(S_l)| + \\
&\quad \ldots + \\
&\quad 2|E_G(S_2) \setminus E_G(S_3)| + \\
&\quad 1|E_G(S_1) \setminus E_G(S_2)| \\
&\leq |E_G(S_1) \setminus E_G(S_2)| + |E_G(S_1) \setminus E_G(S_2)| \left(\sum_{i=1}^{l-1} \frac{i}{2^i} \right) \\
&\leq |E_G(S_1) \setminus E_G(S_2)|(1 + \sum_{i=1}^{l-1} \frac{i}{2^i}) \\
&\leq 3|E_G(S_1) \setminus E_G(S_2)| \\
&\leq |E_G(S_1) \setminus E_G(S_2)| + |E_G(S_0) \setminus E_G(S_1)| \\
&\leq m.
\end{aligned}$$

Further, by applying Assumption 5.2 to the second part of the bound in Inequation 5.6, we obtain as bound on the number of level edges in regular multi-level graphs:

$$|LE| \leq m + A(A-1)n, \tag{5.7}$$

which yields together with Inequation 5.4 as final estimate for the total number of additional edges in the regular multi-level graph:

$$|U| + |D| + |LE| \leq m + [A^2 + A]n. \tag{5.8}$$

Random Queries

Let $(s,t) \in V \times V$ be a query selected randomly, where $P(s,t) = 1/n^2$. We are interested in the probability that the lowest common ancestor of s and t in the component tree, denoted as $level(s,t)$, is at least L ($1 \leq L \leq l+1$). This is the case if either one of the nodes s and t is in the set S_{L-1}, or if both s and t belong to different components in $\mathcal{C}_{L-1}$. Therefore, the number of pairs (s,t) that fulfil the following two conditions have to be counted:

1. If s belongs to S_{L-1}, then t can be any node: $|S_{L-1}|n$ pairs. The same holds if t belongs to S_{L-1}, but then we have counted the pairs where s and t belong both to S_{L-1} twice and have to subtract these terms: $-|S_{L-1}|^2$. In total we get $2|S_{L-1}|n - |S_{L-1}|^2$.

2. If s belongs to component $C_{L-1,j}$, then t belongs to any of the other components: $|C_{L-1,j}| \sum_{k \neq j} |C_{L-1,k}|$. The sum over all components in level $L-1$ yields the number of all (s,t)-pairs for the second condition:

$$\sum_{j=1}^{|\mathcal{C}_{L-1}|} \left[|C_{L-1,j}| \sum_{k \neq j} |C_{L-1,k}| \right]$$

The probability can now be calculated by the following formula:

$$\begin{aligned} P(level(s,t) \geq L) &= P(s \in S_{L-1} \text{ or } t \in S_{L-1}) + \\ &\quad P\left(s \in C_{L-1,j}, t \in C_{L-1,k}, \text{ and } j \neq k\right) \\ &= 1/n^2 \left(2|S_{L-1}|n - |S_{L-1}|^2 + \sum_{j=1}^{|\mathcal{C}_{L-1}|} \left[|C_{L-1,j}| \sum_{k \neq j} |C_{L-1,k}| \right] \right). \end{aligned}$$

Assuming that all components in $\mathcal{C}_{L-1}$ have the same size C the formula can be simplified to

$$P(level(s,t) \geq L) = \frac{2|S_{L-1}|n - |S_{L-1}|^2 + (C-1)(n - |S_{L-1}|)^2/C}{n^2}.$$

Finally, let us consider the highest possible level (i.e., $L = l+1$), and further assume that C and the number of nodes in the smallest subset of nodes $|S_l|$ are constant, and $n \to \infty$. Then, for the probability that the highest level is used in the subgraph $\mathcal{M}_{st}$ it holds

$$P(level(s,t) \geq l+1) \longrightarrow (C-1)/C. \tag{5.9}$$

Size of the Subgraph

The number of edges and nodes of the subgraph $\mathcal{M}_{st}$ used for solving a shortest-path query shall be estimated, depending on $L = level(s,t)$, the level of the lowest

common ancestor of s and t in the component tree. There are 2 nodes in level 0, at most A in the levels from 1 to $L-2$, and $|S_{L-1}|$ nodes in level $L-1$:

$$|V(\mathcal{M}_{st})| \leq 2 + (L-2)A + |S_{L-1}|.$$

The number of edges is the sum of all upward and downward edges used in levels 1 to $L-1$, plus the edges induced by S_{L-1} in G and the newly constructed level-edges for level $L-1$. In level 0 there are at most A upward edges. For the next level, from the A endpoints, there may be A^2 upward edges, which all lead to at most A endpoints in the next level. Because of this, for the following level also at most A^2 upward edges can occur. For downward edges the same argument holds. That means in total there are at most $2(A + (L-2)A^2)$ upward and downward edges. For the level-edges Inequation 5.5 can be used. Altogether for the total number of edges in the subgraph $\mathcal{M}_{st}$ holds that

$$|E(\mathcal{M}_{st})| \leq 2[A + (L-2)A^2] + |E_G(S_{L-1})| + |\mathcal{C}_{L-1}|A(A-1). \tag{5.10}$$

It turns out that A is a crucial parameter for the size of $\mathcal{M}_{st}$, and assuming A being a small constant, this size can be guaranteed to be small as well.

Assumption 5.3 *We assume that A, the maximum number of adjacent nodes of a component over all components $C_{i,j}$ in the regular multi-level graph $\mathcal{M}(G)$, is smaller than or equal to a constant α. Such a multi-level graph is called* α-regular multi-level graph.

For the majority of all queries $level(s,t) = l+1$ (as we have seen above for random queries). The following lemma shows that for such queries in α-regular multi-level graphs the search space (the number of edges in $\mathcal{M}_{st}$) is small. More precisely, the search space is asymptotically dominated by the number of levels l, which is usually in $\mathcal{O}(\log n)$, and by E_l, the number of level edges in level l.

Lemma 5.1 *Let $\alpha \in \mathbb{N}$ be a constant and $\mathcal{M}(G; S_1, \ldots, S_l)$ an α-regular multi-level graph. Then, given a shortest-path query (s,t) with* $\text{level}(s,t) = l+1$*, for the number of edges in the subgraph $\mathcal{M}_{st}$ holds that*

$$|E(\mathcal{M}_{st})| \in \mathcal{O}(l + |S_l|^2).$$

Proof. Inequation 5.10 was established by summing up the upward and downward edges to the first addend; the last two terms constitute a bound on the number of the level edges. Obviously, the number of level edges E_l in level l is also limited by the number of edges in a complete graph with $|S_l|$ nodes. Summing up the first addend of the bound 5.10 and the edges of a complete graph with $|S_l|$ nodes yields

$$|E(\mathcal{M}_{st})| \leq 2[A + (l-1)A^2] + |S_l|^2, \tag{5.11}$$

which completes the proof since $A \leq \alpha$. ∎

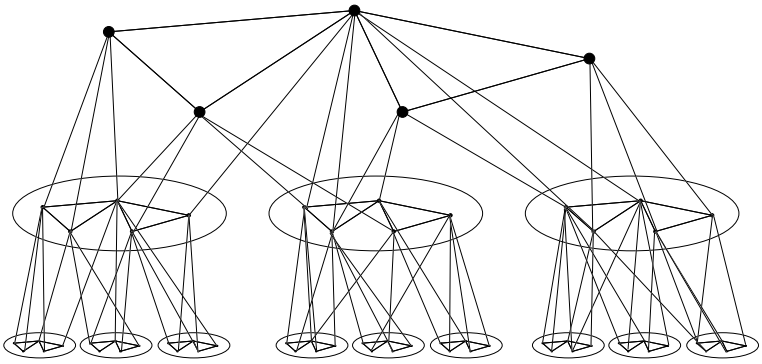

Figure 5.4: A component-induced random graph with $l = 3$, $C = 3$, $N = 5$, $M = 6$ and $A = 3$.

Discussion

We want to conclude the theoretical analysis of (α-regular) multi-level graphs with a discussion of the achieved results. Assumptions 5.1 and 5.2 are not very strict, and should be fulfilled by a reasonable decomposition of the graph. Then, the total number of additional edges is less than $m + [A^2 + A]n$ (cf. Inequation 5.8). (A was defined to be the maximum number of adjacent nodes of a component over all components $C_{i,j}$.) Hence, the parameter A is crucial for the amount of additional space needed. It is also important for the efficiency of the technique: for queries that use the highest level the subgraph contains at most $2[A + (l-1)A^2 + |S_l|^2$ edges (cf. Inequation 5.11). Note that the bounds are not very tight in practice: in the experiments with time-expanded graphs presented later, the total number of additional edges, for example, was always roughly equal to m, number of edges in the original graph.

Another important issue is the similarity of the components. If there is one large component in the highest level, the probability that two nodes belong to that component is quite high that the highest level is not used for the subgraph. In contrast, if all components have the same size C, the probability that the highest level is used is approximately $C - 1/C$ (if the graph is large enough; cf. Inequation 5.9).

In summary, decompositions should be used for which the components $C_{i,j}$ have few adjacent nodes in the corresponding set S_i. Additionally, the components in each level should be preferably of equal size.

5.1.3 Component-Induced Random Graphs

Motivated by the previous analysis of regular multi-level graphs, we introduce a random graph model with a regular hierarchical structure. Using information about this structure, we get multi-level graphs that comply with all the assumptions made in the previous section. Experiments with such graphs confirm the achieved theoretical results.

Recursive Construction

A *component-induced random graph* $G_c(l, C, N, M, A)$ is constructed recursively depending on the following parameters: (i) the number of levels l; (ii) the number of components per level C; (iii) the number of new nodes N and new edges M per level; and (iv) the number of adjacent nodes per component A. Feasible values are $C > 1$, $M \leq N(N-1)$, and $A \leq N$.

The recursive procedure takes as argument a level index i, which is l at the beginning, and first computes a classical Erdös-Réyni random graph R with N nodes and M edges, (i.e., R contains N nodes and M edges selected uniformly at random from all possible edges; see also [Bol85]). If R is not connected we repeat the construction. By setting $\mu_N = \log_2 N + 3$ and $M = \mu_N N$, the probability that R is connected is greater than 95% (cf. [Bol85]). If the level index is 0 the procedure ends at that point. Otherwise, C components are constructed by applying the recursive procedure C times with a level index of $i-1$. Finally, for each of these components A nodes from R are selected randomly and two edges between each of these nodes and randomly selected nodes in the respective component are introduced.

Size

The number of nodes n in the resulting graph G_c can be calculated by summing up the nodes generated in each level i. In the i-th level C^{l-i} random subgraphs R are generated which contain N nodes each, and in total we get

$$n = \sum_{i=0}^{l} C^i N. \tag{5.12}$$

Similarly, the number of edges can be calculated. The first random subgraph R in level l contains $m_l = M$ generated edges. In the i-th level the random subgraphs contain in total $C^i M$ edges, and additionally there are $2C^i A$ edges connecting the components to nodes from the previous level. Hence, the total number of edges in level i $(0 \leq i < l)$ is

$$m_i = C^{l-i} M + C^{l-i} 2A = C^{l-i}(M + 2A), \tag{5.13}$$

and the total number of edges are the edges of all levels 0 to l amounts to

$$m = M + \sum_{i=0}^{l-1} c^{l-i} M + C^{l-i} 2A = M + \sum_{i=1}^{l} C^i (M + 2A).$$

Bringing together the number of nodes n and the number of edges m we can obtain now the following bound on the number of edges:

$$\begin{aligned} m &= N\mu_N + \sum_{i=1}^{l} C^i (N\mu_N + 2A) \\ &= \left[\sum_{i=0}^{l} C^i N \right] \mu_N + 2 \sum_{i=1}^{l} C^i A \end{aligned}$$

$$\begin{aligned} &= n\mu_N + 2\sum_{i=1}^{l} C^i A \\ &\leq n\mu_N + 2n \qquad \text{(since } A \leq N\text{)} \\ &= n(\mu_N + 2). \end{aligned}$$

If we set $\mu_N = \log_2 N + 3$ as mentioned above, the graph G_c is sparse: for the number of edges in G_c it holds that

$$m \leq n(\log_2 N + 5).$$

Multi-Level Graph

A regular multi-level graph $\mathcal{M}(G_c; S_1, \ldots, S_l)$ can now be obtained by setting S_i $(1 \leq i \leq l)$ to be the nodes generated at levels greater than or equal to i. To show that $\mathcal{M}(G_c)$ is a *regular multi-level graph* (cf. Section 5.1.2), we have to prove that $\mathcal{M}(G_c)$ complies with Assumptions 5.1 and 5.2. The number of components induced by removing S_i amounts to $|C_i| = C^{l-i+1}$. Summing up all components and applying Equation 5.12 yields

$$\begin{aligned} \sum_{i=1}^{l} |C_i| &= \sum_{i=1}^{l} C^{l-i+1} = \sum_{i=0}^{l-1} C^i \\ &\leq \sum_{i=0}^{l} C^i N = n, \end{aligned}$$

which means that Assumption 5.1 is fulfilled. Further, the edges induced by the nodes S_i in G_c are all edges introduced in levels i to l (cf. Equation 5.13):

$$\begin{aligned} |E_{G_c}(S_i)| &= M + \sum_{j=i}^{l-1} m_j \\ &= M + \sum_{j=i}^{l-1} C^{l-j}(M + 2A) \\ &= M + \sum_{j=1}^{l-i} C^j (M + 2A). \end{aligned}$$

Now the number of edges in S_i that are not anymore in S_{i+1} can be calculated, and for $C \geq 2$ we can show that the condition of Assumption 5.2 is also fulfilled for our multi-level graph:

$$\begin{aligned} |E_{G_c}(S_i) \setminus E_{G_c}(S_{i+1})| &= |E_{G_c}(S_i)| - |E_{G_c}(S_{i+1})| \\ &= C^{l-i}(M + 2A) \\ &= \frac{1}{C}\left[C^{l-(i-1)}(M + 2A)\right] \\ &\leq \frac{1}{2}|E_{G_c}(S_{i-1}) \setminus E_{G_c}(S_i)|. \end{aligned}$$

Hence, our multi-level graph is *regular*, and the statements obtained above for regular multi-level graphs apply. Finally, we want to investigate the crucial parameter, namely the size of the subgraph $\mathcal{M}_{st}$ for a query (s, t). By Inequality 5.11 and the fact that $|S_l| = N$ in our case, the size of the subgraph is

$$|E(\mathcal{M}_{st})| \leq 2[A + (l-1)A^2] + N^2.$$

Given that $A \leq \alpha$ for a constant $\alpha \in I\!N$, Assumption 5.3 follows directly from the construction of the component-induced random graphs, and also Lemma 5.1 can be applied: In our case, the number of levels l is logarithmic in the number of nodes in G_c, and thus the size of the subgraph $\mathcal{M}_{st}$ is in $\mathcal{O}(\log n + N^2)$.

Concluding, we consider the speedup factor $s = m/|E(\mathcal{M}_{st}|$ indicating how much faster a shortest-path algorithm applied to the subgraph $\mathcal{M}_{st}$ can be compared to the same algorithm applied to G_c. By the latter observation concerning the size of the subgraph, the factor s increases with the number of nodes n in G_c, given the parameter N is kept at a constant size. This implies that the speedup of a shortest-path algorithm by using the multi-level approach can become arbitrarily high for component-induced random graphs that are sufficiently large (e.g., for fixed parameters C, N, M, and A, increasing the number of levels l yields arbitrary large graphs, where the other parameters remain constant).

Experiments

In [Hol03], Holzer shows various experiments with component-induced random graphs that confirm the theoretical results above. The number of levels considered in these experiments range from one to five, and also the impact of the other parameters is considered. In contrast to the definition of component-induced random graphs, the parameters N, C, and A may differ from level to level in [Hol03]. Besides the size of $\mathcal{M}_{st}$ for a given query, which was the crucial parameter to evaluate the approach in the theoretical analysis above, also the real speedup is measured. The real speedup is defined to be the ratio of CPU-times t_1/t_2, where t_1 is the CPU-time needed to apply Dijkstra's algorithm to the original graph G_c, and t_2 measures the CPU-time to apply Dijkstra's algorithm to the subgraph $\mathcal{M}_{st}$. The values of speedup are average values for a statistically significant set of random queries.

One crucial result is that with increasing size of the graph, also an increasing speedup can be observed, up to a speedup of approximately 1000 for a graph with roughly 100 0000 nodes and 5 levels. In addition, it turns out that, considering only component-induced random graphs G_c with a fixed number of nodes, the speedup differs greatly when the parameters for construction of the graph are varied (cf. Figure 5.5). For more details about the experiments we refer the reader to [Hol03].

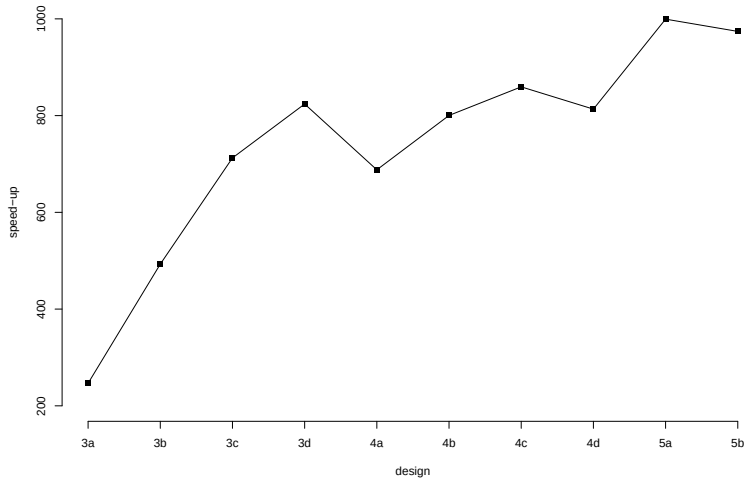

Figure 5.5: Speedup for component-induced random graphs G_c with roughly 100 000 nodes. The abscissa represents the design of the graphs: the number of levels (3, 4, and 5), and different labels (a-d) imply a different setting for the number of nodes N_i generated in level i.

5.2 Applying the Multi-Level Graph Approach

We adapted the multi-level graph approach introduced in the previous section for application to a timetable information system. Actually, we focus on the time-expanded model for the simplified earliest arrival problem timetable information described in Section 3.3.1 (page 18). For the realistic version and the time-dependent model the multi-level graph approach can be applied analogously. We conducted an extensive experimental study based on all long-distance train data (winter period 1996/97) of the German railways consisting of time-table information and real-world queries. Based on the time-expanded graph, we considered various numbers l of levels and sequences of subsets of nodes. For each of these values, the corresponding multi-level graphs are evaluated.

Our study was concentrated in measuring the improvement in the performance of Dijkstra's algorithm when it is applied to a subgraph of $\mathcal{M}$ instead of being applied to the original time-expanded graph. Our experiments demonstrate a clear speedup of the hierarchical decomposition approach based on multi-level graphs. More precisely, we first considered various selection criteria for including nodes on the subsets which determine the multi-level graphs. This investigation revealed that random selection (as e.g., proposed in [UY91]) is a very bad choice. After choosing the best criteria for including nodes in the subsets, we analysed their sizes and demonstrated the best values for these sizes. It turns out that the dependence of the multi-level graphs on the subset sizes is also crucial. Finally, for the best choices of subsets and their sizes, we determined the best values for the number of levels. For the best choice of all parameters considered we obtained a speedup of about 11 for CPU time and of about 17 for the number of edges hit by Dijkstra's algorithm.

Before turning to the experiments, we have to discuss a necessary customisation of the multi-level approach to the time-expanded graphs.

5.2.1 Station Graph

The earliest arrival problem is translated into a single-source some-target shortest path problem in the time-expanded graph (referred to by EG), where the targets for one query are the set of nodes belonging to one station. Since the multi-level graph approach is tailored to single-pair shortest path queries, we will need a second graph, the *station graph* (referred to by SG), which is identical with the graph used in the time-dependent model for the simplified earliest arrival problem introduced in Section 3.3.2 (page 21). It contains one node per railway station R, and there is an edge between two stations R_1 and R_2 if and only if there is an edge (v_1, v_2) in the time-expanded graph, with v_1 belonging to station R_1 and v_2 belonging to R_2. The station graph is simple and unweighted. With $T(R)$ we denote the set of all arrival and departure nodes in the time-expanded graph that belong to the station R. Note that the station graph is the graph minor of the time-expanded graph obtained by contracting all stay-edges in the time-expanded graph and by removing all but one of multiple edges. The following lemma follows directly by the definition of SG and EG.

Lemma 5.2 *Consider a subset Σ of nodes in SG, and let $T(\Sigma)$ be the set of all arrivals and departures of the stations in Σ. Then, if the stations $R_1, \ldots, R_k$ belong to one connected component of $SG - \Sigma$, the nodes $T(R_1), \ldots, T(R_k)$ belong to one connected component of $EG - T(\Sigma)$, and vice versa.*

5.2.2 Customisation of the Multi-Level Graph Model

If we define the multi-level graph $\mathcal{M}(EG)$ of the time-expanded graph EG according to the definition given in Section 5.1.1, then we would get a subgraph of $\mathcal{M}(EG)$ for a pair s, t of nodes on which we could solve a single-pair shortest path problem in order to determine an s-t shortest path in EG. The time-expanded modelling of the earliest arrival problem, however, requires to solve a single-source *some*-targets problem, and hence the direct application of the multi-level graph approach is not suitable for this case. Instead, we need a subgraph that guarantees the same shortest-path length between every pair of nodes belonging to two *stations* (i.e., sets of nodes of EG). Therefore, we define on EG a slightly modified version of a multi-level graph:

1. The first modification is to start with a sequence of l sets of stations of the station graph, Σ_i $(1 \leq i \leq l)$, which are decreasing with respect to set inclusion. Then, the l sets of nodes of the time-expanded graph are defined to be $S_i := \cup_{R \in \Sigma_i} T(R)$, all departures and arrivals of all the stations in Σ_i. The levels of the multi-level graph $\mathcal{M}$ are then defined using the S_i as described in Section 5.1.1 (page 63), yielding $\mathcal{M}(EG; S_1, \ldots, S_l)$. To emphasise the de-

pendence of S_i on Σ_i and in order to facilitate notation, we shall refer to this multi-level graph as $\mathcal{M}(EG; \Sigma_1, \ldots, \Sigma_l)$.

2. The component tree is computed in the station graph. There is one leaf C^R per station R, and $Adj(C^R) := T(R)$ (i.e., the arrivals and departures belonging to R).

3. We define a node v of the time-expanded graph to be adjacent to a component C of the station graph, if v and any node belonging to a station of C are connected by an edge in the time-expanded graph. With this definition, and s and t being the departure and arrival *stations*, the definition of the subgraph $\mathcal{M}_{st}$ is exactly the same as for general multi-level graphs in Section 5.1.1 (page 66).

Given a query with departure station s, arrival station t, and a departure time, the subgraph $\mathcal{M}_{st}$ of $\mathcal{M}(EG; \Sigma_1, \ldots, \Sigma_l)$ depends now on the *stations* s and t. The departure time determines the departure node in EG belonging to station s. To solve the query, we have to compute the shortest-path length from the departure node of EG to one of the nodes belonging to station t. Based on Theorem 5.1 and Lemma 5.2, we are able to show (next lemma) that it is sufficient to perform such a shortest path computation in $\mathcal{M}_{st}$.

Lemma 5.3 *For each departure node v in the time-expanded graph belonging to station s, the shortest-path length from v to one of the nodes belonging to station t is the same in the graphs EG and $\mathcal{M}_{st}(EG; \Sigma_1, \ldots, \Sigma_l)$.*

Proof. Using Lemma 5.2, the proof of Theorem 5.1 can be adopted to the customisations that were made for the time-expanded graph.

The proof for Claim 1 is exactly the same here. Claims 2 and 3 are modified in the way that now s and t are sets of nodes of EG, namely the sets of all arrivals and departures belonging to the stations s and t, respectively. Then, Claims 2 and 3 hold for each of these nodes, because of Lemma 5.2.

Let P be a shortest path in EG from the departure node v to one of the nodes belonging to station t. Then, similarly to the proof of Theorem 5.1, we can show that there is a path with the same end-nodes and of the same length in $\mathcal{M}_{st}(EG; \Sigma_1, \ldots, \Sigma_l)$. ∎

5.2.3 Experiments

As mentioned above, we will consider different multi-level graphs that are all based on one single graph. This original graph is the time-expanded graph EG_{DB} as defined in Section 3.3.1 (page 18), which is based on the winter 1996/97 train timetables of the German railroad company Deutsche Bahn (DB). It consists of 6960 stations, 931 746 nodes, and 1 397 619 edges. The format of the timetable data is the same as the one used in the comparison of the time-expanded and time-dependent model in Section 3.5.1 (page 28).

The second input to the multi-level graph for time-expanded graphs is the sequence of sets of stations $\Sigma_1, \ldots, \Sigma_l$, which determines the multi-level graph, referred to by $\mathcal{M}(EG_{DB}; \Sigma_1, \ldots, \Sigma_l)$. In the following we will omit the graph EG_{DB} in the notation of the multi-level graph. The goal of this experimental study is to investigate the behaviour of the multi-level graph with respect to the sequence $\Sigma_1, \ldots, \Sigma_l$. The experiments to measure the raw CPU time were run on a Sun Enterprise 4000/5000 machine with 1 GB of main memory and four 336 MHz UltraSPARC-II processors (of which only one was used). The preprocessing time to construct a multi-level graph (i.e., the additional edges and the component tree) varies from one minute to several hours.

Parameters

First of all, we want to measure the improvement in performance of shortest path algorithms if we compute the shortest path in the subgraph of $\mathcal{M}$ instead of the original graph G. From the snapshot of over half a million of realistic timetable queries, which has been used also in the experiments comparing the time-expanded and the time-dependent approach in Section 3.5.1 (page 28), we take a subset of 100 000 queries. Then, for each instance of a multi-level graph $\mathcal{M}$ that we consider we solve the queries by computing the corresponding shortest path in the subgraph of $\mathcal{M}$ using Dijkstra's algorithm. From these shortest path computations we consider two parameters to evaluate the improvement of the performance:

- *CPU-speedup*: the ratio between the average CPU time needed for answering a single query in the original time-expanded graph (103 milliseconds) and the average CPU time when the subgraph of $\mathcal{M}$ is used;

- *edge-speedup*: the same ratio when the average number of edges hit by Dijkstra's algorithm is used instead of the average raw CPU time.

Note that the time needed to compute the subgraph for a given query is only included in the CPU-speedup, not in the edge-speedup. Another issue is the space consumption, and therefore we define

- the *size of a level* of $\mathcal{M}$ to be the number of edges that belong to that level;

- the *size* of $\mathcal{M}$ to be the total number of edges in all levels of $\mathcal{M}$ (including the original graph G);

- the *relative size* of $\mathcal{M}$ to be the size of $\mathcal{M}$ divided by the number of edges in the original graph G.

Finally, to compare the improvement in performance and the space consumption, we consider the (CPU-, edge-) *efficiency* of $\mathcal{M}$, being the ratio between (CPU-, edge-) speedup and the relative size of $\mathcal{M}$.

Two Levels

In the following we define the sequences of sets of stations used in our experiments with 2-level graphs.

We define three sequences $A = (A_1, \ldots, A_{10})$, $B = (B_1, \ldots, B_{10})$, and $C = (C_1, \ldots, C_{10})$ of sets of stations, which are decreasing with respect to set inclusion. The first set in each sequence is identical for all the three and consists of all the stations that have a degree greater than two in the station graph; this yields a set of 1974 stations. The last set of each sequence contains 50 stations, and the sizes of the remaining 8 sets of stations are such that the sizes are equally distributed in the range $[50, 1974]$. The difference between A, B, and C is the criterion on the selection of stations:

A: In the timetable data, each station is assigned a value that reflects the *importance* of that station with respect to changing trains at that station. The sets A_i contain the stations with the highest importance values.

B: The sets B_i contain stations with the highest degrees in the station graph.

C: The set C_1 is a random set of stations. Then, for C_k $(2 \leq k \leq 10)$, stations are randomly selected from C_{k-1}.

These criteria for selecting stations are crucial for the multi-level graph approach. For criteria A and B we use additional information from the application domain: they reflect properties of important hubs in the railroad network. Removing these hubs yields intuitively a "good" decomposition of the network. The experimental results confirm this intuition.

Using each set of stations A_i, B_i, and C_i $(i = 1, \ldots, 10)$ as the set Σ_1, we compute the 2-level graph (i.e., consisting of the original graph being level zero and level one) $\mathcal{M}(\Sigma_1)$. Figure 5.6 shows the sizes (i.e., the number of edges) of the level one. For the sequences A and B these sizes are similar, and for the randomly selected sets C, the size grows dramatically as the number of stations decrease. In the following we will focus on the sequence A, since B shows similar but slightly worse results, and the multi-level graphs using sets of stations of C are too big.

For $i = 1, \ldots, 9$ with decreasing number of stations in A_i, the speedup and efficiency of $\mathcal{M}(A_i)$ is growing, and from 9 to 10 it is falling drastically, as Figure 5.7 shows. Figure 5.8 reveals one reason for this behaviour: While the number of stations in A_i is big enough, for almost all queries ($> 96\%$) the level one is used (i.e., the subgraph of $\mathcal{M}(A_i)$ used for the shortest path computation consists of the corresponding upward and downward edges of level one, and of the edge-set E_1). But for $i = 10$, for only about 60% of the queries the level one is used, and the remaining 40% of the queries have to be solved in level zero (i.e., the original graph). The queries for which level one is used still profit from level one as Figure 5.8 shows, but for the rest of the queries the speedup equals one. In total, this reduces the average speedup over all queries.

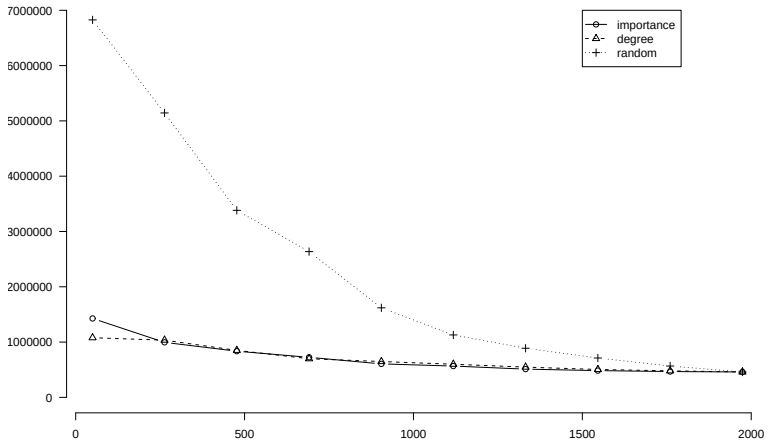

Figure 5.6: For each sequence A, B, and C, there is one curve. Each point corresponds to one set Σ_1 of stations in these sequences. The diagram shows the size of level one of the 2-level graph $\mathcal{M}(\Sigma_1)$ according to the number of stations in Σ_1.

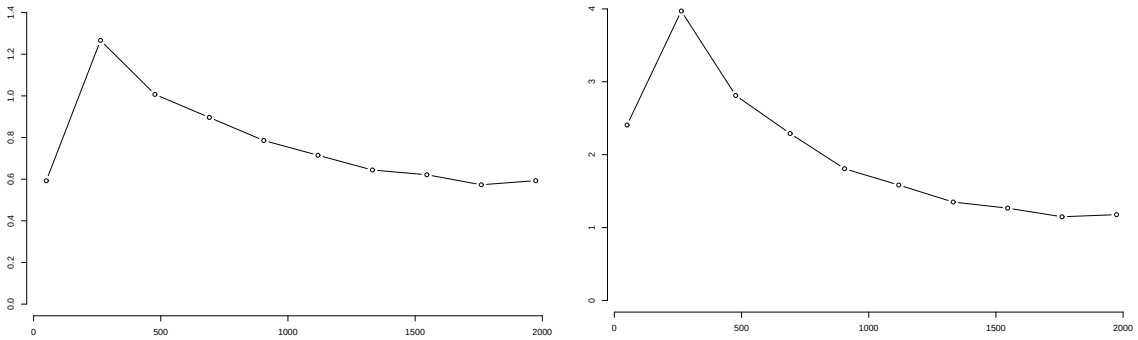

Figure 5.7: Each point corresponds to one 2-level graph $\mathcal{M}(A_i)$ for each set of stations in A. The left diagram shows the CPU-efficiency of $\mathcal{M}(A_i)$ according to the number of stations in A_i, and in the right diagram the ordinate is the average CPU-speedup for the 2-level graphs.

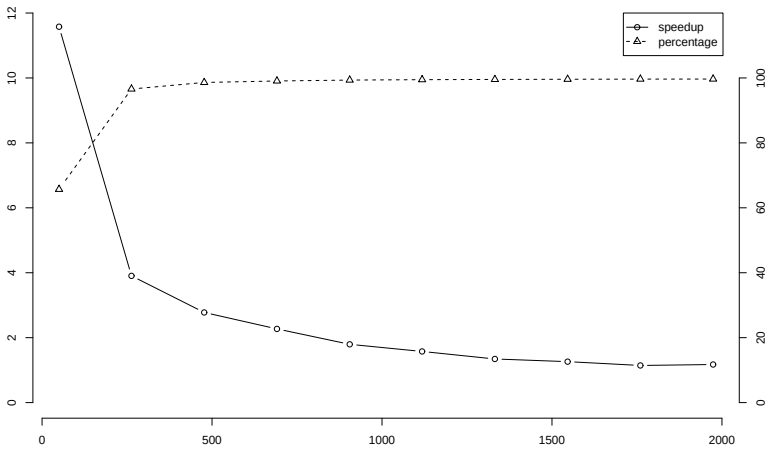

Figure 5.8: Like Figure 5.7, the points refer to sets of stations A_i, and the abscissa denotes the number of stations in A_i. For the curve that is growing with respect to the number of stations, the ordinate on the right shows the percentage of queries *for which the second level is actually used* (i.e., the lowest common ancestor in the component tree is the root), while for the descending curve the average CPU-speedup over all these queries is shown on the left ordinate.

Multiple Levels

The experiments with two levels show, that the set of stations A_9 with 263 stations yields the best performance, and (according to Figure 5.8) that the most interesting cases to investigate is to consider subsets with less than $|A_9|$ stations. In our test sequence, there is only the set A_{10} with less stations. Consequently, we included in the sequence A for our investigation with more than two levels also the subsets of stations A_{9a} (225 stations), A_{9b} (156 stations), A_{9c} (100 stations), and A_{10a} (30 stations).

Three Levels For every pair Σ_1, Σ_2 of sets of stations in A with $\Sigma_1 \supset \Sigma_2$, we consider the 3-level graph $\mathcal{M}(\Sigma_1, \Sigma_2)$. For fixed Σ_1, we investigate the behaviour of the 3-level graph with respect to Σ_2. Figure 5.9 shows this behaviour for $\Sigma_1 \in \{A_1, A_7, A_8, A_9\}$. With $\Sigma_1 = A_1$, we see the same drop of speedup and efficiency when Σ_2 gets too small as in the 2-level case. However, when the size of Σ_1 decreases (e.g., $\Sigma_1 = A_9$), we observe that the suitable choices for Σ_2 are the subsets A_{10} (50 stations) and A_{10a} (30 stations) which improve both speedup and efficiency. This also shows that different levels require different sizes of subsets.

More Levels For more than three levels, we do not investigate every possible combination of sets of stations in A, but follow an iterative approach. To get initial sequences $\Sigma_1, \ldots, \Sigma_{l-1}$ for the l-level graph, we take the sequences $\Sigma_1, \ldots, \Sigma_{l-2}$ that were the basis for the best $l-1$-level graphs, and combine these sequences with the sets of stations Σ_{l-1} in A with $\Sigma_{l-1} \supset \Sigma_{l-2}$. Then, subsequences of A are used as input for the l-level graph that are similar to the initial sequences.

Table 5.1 as well as Figure 5.10 show the results for the best l-level graph for $2 \leq l \leq 6$. The gap between CPU- and edge-speedup reveals the overhead to compute the subgraph $\mathcal{M}_{st}$ for a query using the multi-level graph, since the average CPU-

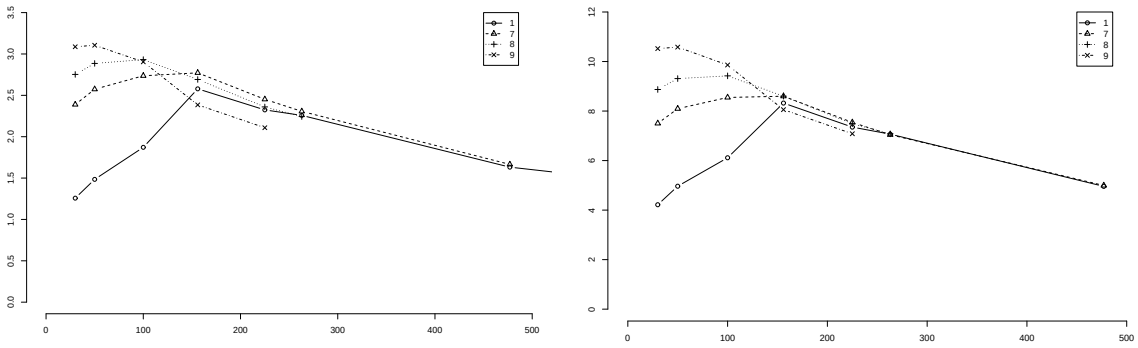

Figure 5.9: The equivalent of Figure 5.7 for 3-level graphs $\mathcal{M}(\Sigma_1, \Sigma_2)$. For each set of stations $\Sigma_1 \in \{A_1, A_7, A_8, A_9\}$ there is one curve, which is obtained by varying Σ_2 (abscissa). On the left hand, the ordinate shows the CPU-efficiency, while on the right hand the CPU-speedup is shown.

		speedup		efficiency	
l	$\mathcal{M}(\cdot)$	CPU	edge	CPU	edge
2	A_9	3.97	4.89	1.37	1.56
3	A_9, A_{10}	10.58	14.06	3.11	4.12
4	A_7, A_{9b}, A_{10a}	**11.18**	16.63	**3.48**	5.18
5	A_7, A_{9b}, A_{9c}, A_{10a}	9.91	**17.52**	3.06	**5.41**
6	A_7, A_9, A_{9a}, A_{9c}, A_{10a}	8.58	17.01	2.55	5.06

Table 5.1: Results for the best l-level graph. Note that the one-level graph is the original graph, and that the speedup and efficiency are ratios comparing the results for multi-level graphs with the original graph, so for the original graph the speedup and efficiency are one.

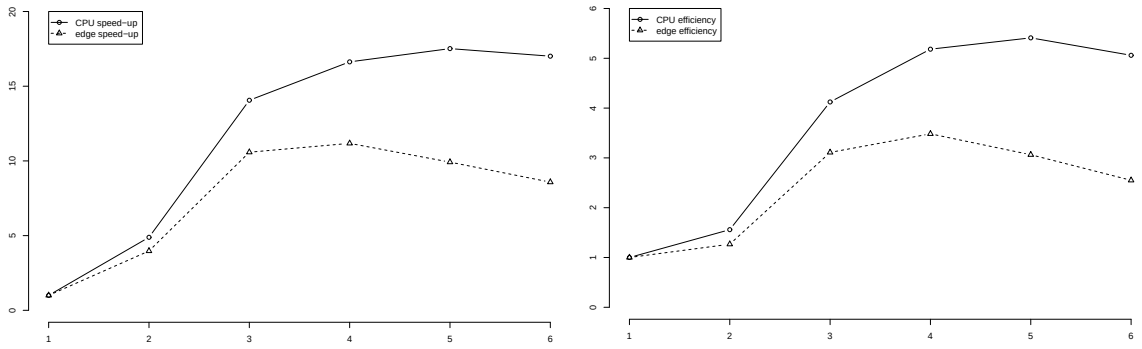

Figure 5.10: For different numbers l of levels the results of the best l-level graph is shown: the CPU- and edge-efficiency in the left diagram, and the CPU- and edge-speedup in the right one.

time includes this computation, but the average number of edges hit by Dijkstra's algorithm does not. Considering levels four and five, because of this overhead the CPU-speedup is decreasing while the edge-speedup is still increasing with respect to the number of levels. Experiments with larger values of l revealed that there is no further improvement in the speedup and/or in the efficiency.

5.2.4 Discussion

In this section, we empirically investigated a hierarchical decomposition approach based on multi-level graphs. The experiments were focused on a specific application scenario motivated by the timetable information problems discussed earlier.. Given the complexity of the recursive construction of the multi-level graph (or of similar models proposed in the literature), this concept might appear to be more of theoretical interest than of practical use, especially when more than one hierarchical level is introduced. To our surprise, our experimental study with multi-level graphs for this specific scenario exhibited a considerable improvement in performance regarding the efficient computation of on-line shortest path queries. The best results have been achieved with 4-level and 5-level graphs.

In defining the multi-level graphs, we considered three simple criteria A (importance of stations), B (highest degrees), and C (random choice) to select the stations. The latter criterion turned out to be a very bad choice. It further turned out that—given a reasonable criterion to select nodes—the size of the subsets of nodes defining the decomposition have a high influence of the achieved speedup. These results indicate that a careful selection of nodes is crucial for the success of the multi-level graph approach.

Further improvements could be possibly achieved by using more sophisticated versions of criteria A and B. For example, in [SWW00] we used a more sophisticated version of criterion A for 2-level graphs. This criterion adds new stations to a set of stations for which the 2-level graph is already known and hence is not applicable to generate sets of stations with fixed sizes for more than two levels. Consequently, it could not directly be used here. However, based on the similarities of the results for the sequences A and B, we believe that if a criterion is chosen which yields a better

performance for 2-level graphs, the performance of the multi-level graphs with more than two levels could be improved as well.

5.3 More Speedup Techniques

There is a multitude of speedup techniques known for Dijkstra's algorithm in the single-pair case.[1] We don't provide a complete overview of these techniques (see [WW] for a survey), but want to discuss a few such techniques with different characteristics. There are also methods that improve the running time but that don't guarantee optimal solutions, but we focus only on techniques guaranteeing optimal solutions.

We distinguish techniques that use preprocessing—like the multi-level graph approach above—and those that don't. Naturally, techniques with preprocessing are more effective when they are applied to a query, since in an expensive preprocessing step additional information has been computed already beforehand. Usually, these techniques only make sense in a setting when many queries have to be solved. Another distinctive feature is whether geometric information via coordinates is given or not. This information can be utilised to direct the search towards the destination. In the case that some coordinates of nodes are missing in a timetable (e.g., small stations or bus stops of a timetable), or no coordinates are given at all, we have successfully applied methods from graph drawing to generate the missing coordinates in [BSWW04]. The goal of that study was to provide coordinates yielding the best speedup of the respective technique.

5.3.1 Techniques Without Preprocessing

The *goal-directed search* introduced previously and the so-called *bidirected search* are examples of methods to improve the running time but don't require a preprocessing step.

Goal-directed Search

In Section 3.4 (pages 24 et seq.), the goal-directed search has been introduced already in detail. It is based on a potential function on the nodes, which is often a lower bound on the distance to the destination node. When coordinates are given, such a potential function can be obtained by using a linear function depending on the Euclidean distance to the destination, as we did. Good potential functions lead the search carried out by Dijkstra's algorithm towards the goal, and thus the search space (i.e., the number of edges touched by the algorithm) is reduced. In graphs with Euclidean distances as edge lengths we have observed an improvement in running time by a factor two or more, as Figure 5.15 shows (page 92; see also [SV86] for a

[1] We focus on techniques improving *Dijkstra's algorithm*. Note, however, that some of the techniques are more general in the sense that they can be applied to any shortest-path algorithm.

theoretical discussion of goal-directed search in Euclidean graphs).

However, our experiments in Section 3.5 (pages 28 et seq.) have revealed that applied to timetable information, the goal-directed search performs quite badly. The main reason is that the used scale factor needed to guarantee optimal solutions leads to bad potentials, and in the end the overhead to compute the potentials nearly cancels out the obtained reduction in search space.

Bidirected Search

The bidirected search simultaneously applies the "normal", or forward, variant of the algorithm, starting at the source node, and a so-called reverse, or backward, variant of Dijkstra's algorithm, starting at the destination node. With the reverse variant, the algorithm is applied to the reverse graph: a graph with the same node set V as that of the original graph, and the reverse edge set $\overline{E} = \{(u, v) \mid (v, u) \in E\}$. Let $d_f(u)$ be the distance labels of the forward search and $d_b(u)$ the labels of the backward search, respectively. The algorithm can be terminated when one node has been designated to be permanent by both the forward and the reverse algorithm. Then the shortest path is determined by the node u with minimum value $d_f(u) + d_b(u)$ and can be composed of the one from the start node to u, found by the forward search, and the edges reverted again on the path from the destination to u, found by the reverse search. (See also [AMO93] and [Len90].)

In timetable information, however, the bidirected search cannot be applied directly, neither in the time-expanded model nor in the time-dependent model. The reason is that the arrival time at the destination station is not known, and thus the starting node (and starting time in the time-dependent model) for the backward search is not specified. In [SWW00], we experimented with an idealised version of the bidirected search for the simplified earliest arrival problem, and used as initial node for the backward search the arrival node at the destination station computed by a normal run of Dijkstra's algorithm. By this idealised version we get a bound on the reduction of the search space achievable with any "real" applicable modification of the bidirected search—using, for example, estimates for the arrival time. Experiments with the idealised bidirected search showed that the search space (i.e., the number of touched edges) could be reduced to roughly one third of the search space of Dijkstra's algorithm.

5.3.2 Preprocessing Techniques

In a scenario like timetable information, where a multitude of shortest-path queries have to be solved in a large, sparse graph, preprocessing techniques are reasonable: On the one hand, a rather expensive preprocessing pays off since many queries can be answered much faster than with standard algorithms, which is especially important in an on-line setting. On the other hand, the complete distance matrix requires quadratic space in the number of nodes which means that it usually cannot be stored when graphs contain one million or more nodes. But, additional information in the size of the input graph can be precomputed and stored. For solving the

queries, this information is used to speedup the shortest-path algorithm.

Hierarchical Decomposition Techniques

The multi-level graph approach introduced in Section 5.1 is the first example of such a preprocessing technique. The input graph is hierarchically decomposed and additional edges representing shortest paths in the input graph are computed. The search space is reduced since the shortest path search can be done in a small subgraph. Speedup factors of 1000 for regular component-induced random graphs (cf. Section 5.1.3) and up to 11 for timetable information (cf. Section 5.2.3) show the potential of the technique. Similar approaches are the HiTi graphs introduced by Jung and Pramanik [JP02], and methods used in the area of car navigation systems [CF94, Fli04].

Geometric Shortest-Path Containers

This technique [WW03] requires a layout of the graph given by coordinates of the nodes. In the preprocessing, all shortest path trees have to be computed. For each edge $e \in E$, a set $S(e)$ of those nodes to which a shortest path starts with edge e is computed. Using a given layout, for each edge $e \in E$ the bounding box of $S(e)$ is stored in an associative array BB with index set E. Then, Dijkstra's algorithm can be performed on the subgraph induced by the edges $e \in E$ with the target node included in $BB[e]$. This subgraph can be determined on the fly, by excluding all other edges in the search.

We have introduced a variation of this technique in [SWW00], where as geometric objects we used angular sectors instead of bounding boxes. Experiments have been conducted with the time-expanded graph (for the simplified earliest arrival problem), and speedup factors in the range from 6 to 10 have been observed. Willhalm and Wagner showed in an extensive study [WW03] that bounding boxes are the fastest geometric objects in terms of running time (in particular, bounding boxes are more effective than the angular sectors), and competitive with much more complex geometric objects in terms of visited nodes. We refer to the specific case of bounding boxes as *shortest-path bounding boxes*.

5.3.3 Combination of Speedup Techniques

We have combined the above described preprocessing techniques using geometric containers (in this case angular sectors) and a preliminary version of multi-level graphs with one additional level in [SWW00] using the time-expanded graph. The result is promising: concerning the average CPU time (when Dial's priority queue was used), the single speedup factors observed were 6 and 9, and the speedup of the combination of both techniques was 34 (the best speedup one can hope for the combination would be $6 \cdot 9 = 54$). Figure 5.11 illustrates the reduction of search space achieved by the latter combination of techniques, compared to the search space of Dijkstra's algorithm shown in Figure 3.5 (page 29). In the following, we present an experimental study investigating further combinations of the four speedup

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

Table 5.2: Number of nodes and edges for all test graphs used for the experiments on combining speedup techniques.

techniques goal-directed search, bidirected search, multi-level graph approach, and shortest-path bounding boxes, applied to several different graph classes.

Graphs

The graphs under investigation are: (i) street graphs, taken from the publicly available data bases [Bar, Esr]; (ii) public transport graphs, where nodes are stations and edge weights are average travel times; (iii) random planar graphs generated by LEDA [NM99]; and (iv) a random graph model introduced by Waxman [Wax88], where coordinates are chosen uniformly at random in the unit square and the probability that an edge exists is higher for smaller edges than for larger edges. For each of the above four graph classes 10 graphs of different sizes have been studied; Table 5.2 shows the number of nodes and edges of all graphs. Node coordinates, which are needed by the speedup techniques goal-directed search and geometric containers, are available for all of the graphs.

Experiments

All 16 possible combinations of the four techniques have been implemented (we refer to [HSW04] for details on how the techniques can be combined) in `C++`, using the graph and priority queue data structures of the LEDA library [NM99] (version 4.4). The code was compiled with the GNU compiler (version 3.3), and experiments were run on an Intel Xeon machine with 2.6 GHz and 2 GB of memory, running Linux (kernel version 2.4).

For each graph and combination, we computed for a set of queries shortest paths, measuring two types of performance: the mean values of the running times (i.e., CPU time in seconds) and the number of nodes inserted in the priority queue. Finally, the parameters of interest are the speedup factors *CPU-speedup* and *node-speedup*

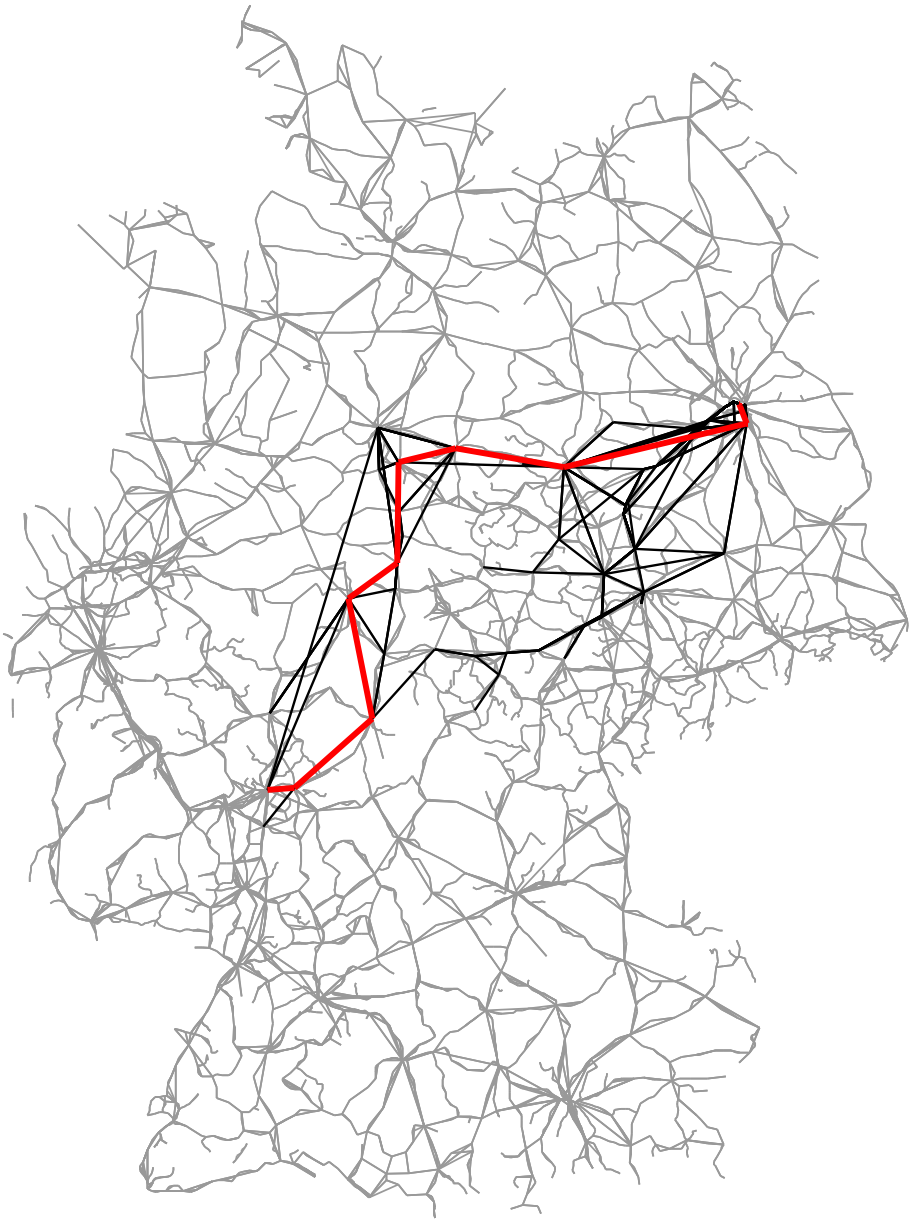

Figure 5.11: The search space (black edges) of a combination of the multi-level graph approach and geometric containers (angular sectors) for the sample query from Berlin to Frankfurt used in Figure 3.5.

obtained by dividing the performance of Dijkstra's algorithm by the performance of the respective combination of techniques. The queries were chosen at random and the amount of them was determined such that statistical relevance could be guaranteed (see also [WW03]).

Results

The outcome of the experimental study is shown in Figures 5.12–5.15. Each combination is referred to by a 4-tuple of shortcuts: `go` (goal-directed), `bi` (bidirected), `ml` (multi-level), `bb` (bounding box), and `xx` if the respective technique is not used (e.g., `go-bi-xx-bb`). In all figures, the graphs are ordered by size, as listed in Table 5.2.

On first sight it is striking that the multi-level graph approach performs generally much worse than in the experiments shown previously (cf. Section 5.2.3) when we applied the multi-level graph approach to our timetable information problem. This is due to the fact that here, for the implementation of the multi-level graph approach in combination with the other techniques, we use a relatively simple method to derive the hierarchical decomposition of the graph. It is based on the results in [Hol03] and basically removes a certain fixed number of nodes of high degree. However, as the experiments in Section 5.2.3 suggest, the performance of the multi-level graph approach highly depends on the various parameters of the decomposition. We believe that considerable improvements of the presented results are possible if these parameters are chosen carefully for every single graph.

The results indicate that there are speedup techniques that combine well and others where speedup does not scale. Good combinations are generally the goal-directed search with the multi-level graph approach, and also the bidirected search with the shortest-path bounding boxes, which complement each other very well. For real-world graphs, a combination including bidirected search, multi-level graph approach, and shortest-path bounding boxes is the best choice as to the number of visited nodes, and in terms of running time, the bidirected search in combination with shortest-path bounding boxes is the best. For generated graphs, the best combination is goal-directed search, bidirected search, and shortest-path bounding boxes for both the number of nodes and running time.

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

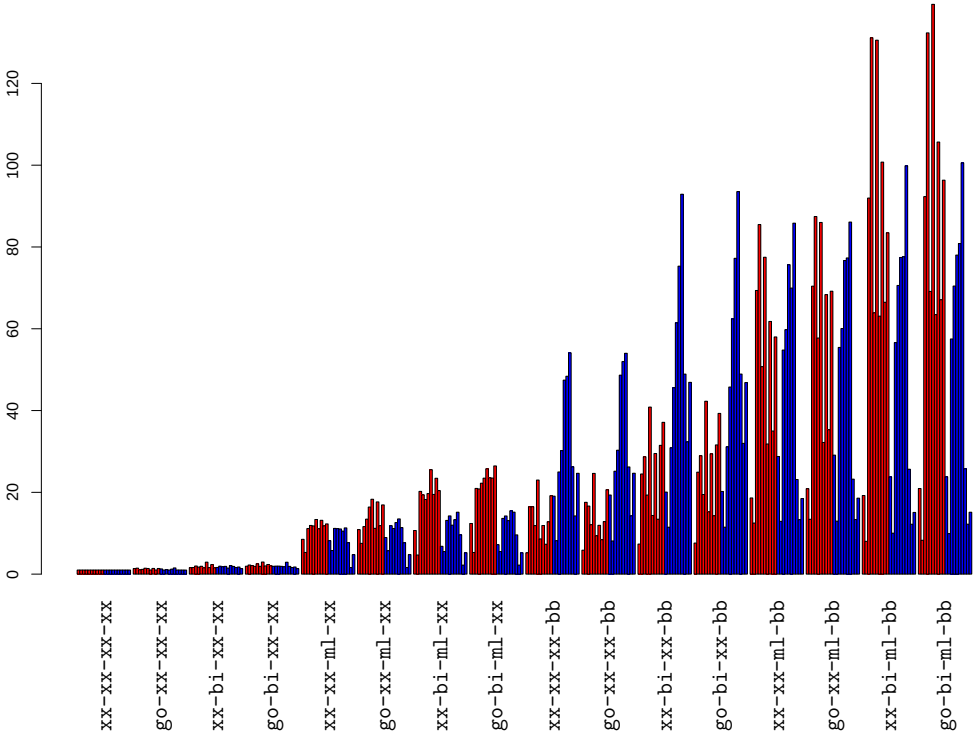

Figure 5.12: Speedup relative to Dijkstra's algorithm in terms of visited nodes for real-world graphs (in this order: street graphs in red and public transport graphs in blue)

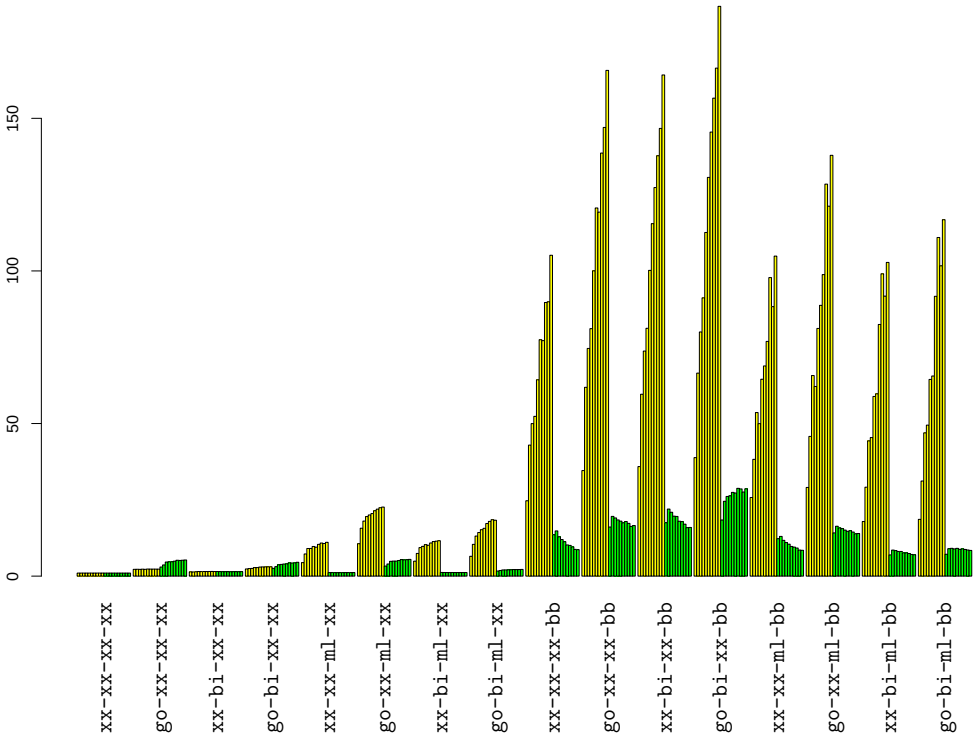

Figure 5.13: Speedup relative to Dijkstra's algorithm in terms of visited nodes for generated graphs (in this order: random planar graphs in yellow and random Waxman graphs in green)

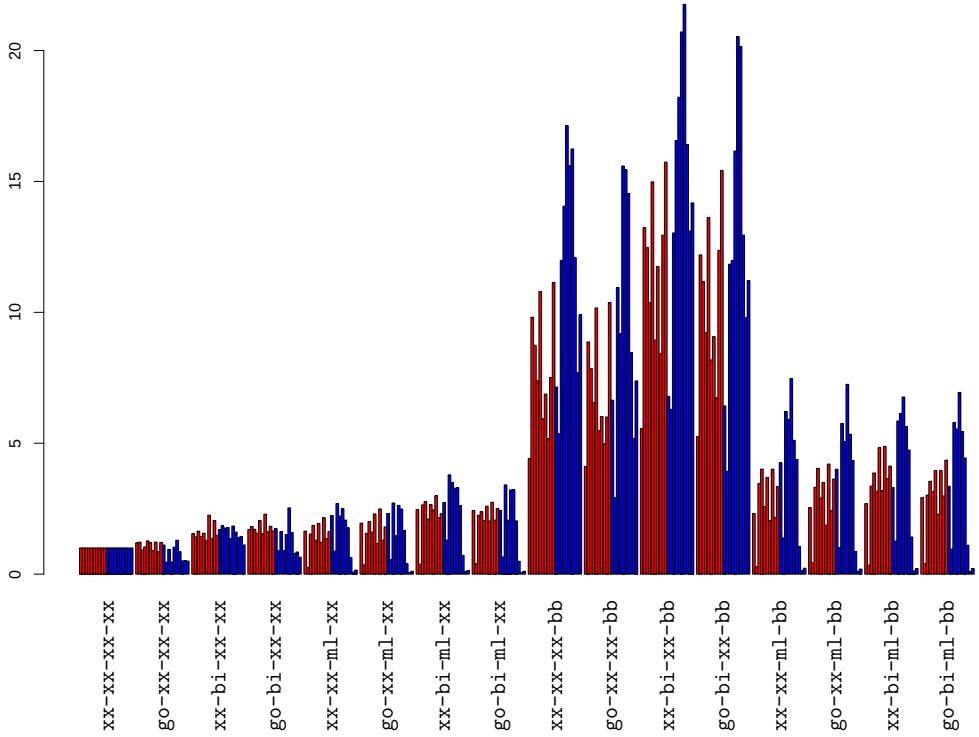

Figure 5.14: Speedup relative to Dijkstra's algorithm in terms of running time for real-world graphs (in this order: street graphs in red and public transport graphs in blue)

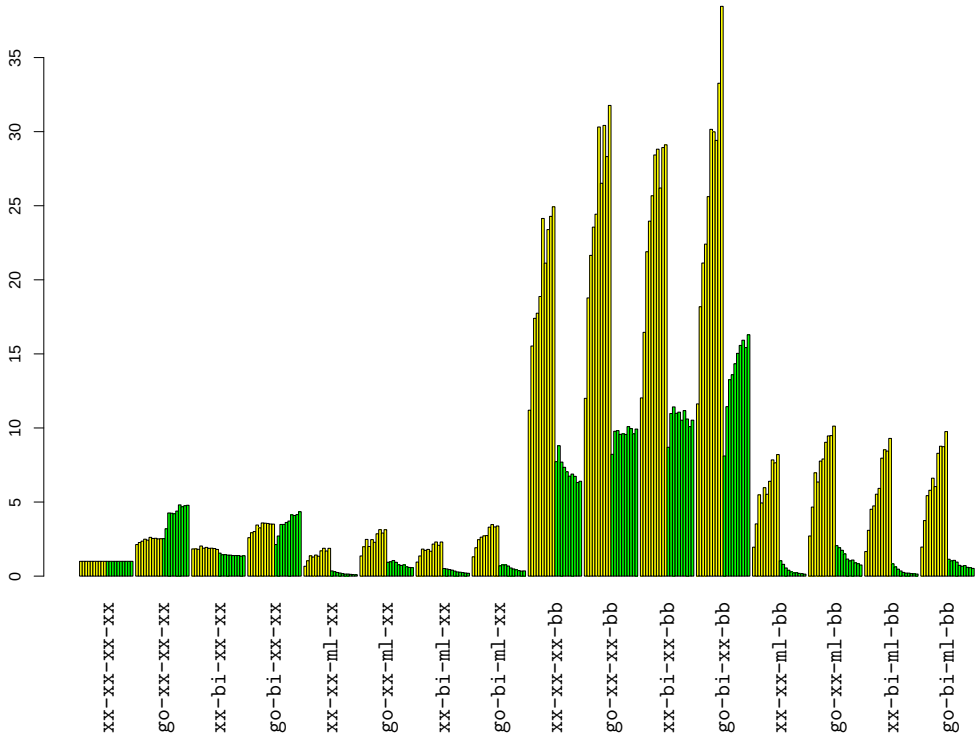

Figure 5.15: Speedup relative to Dijkstra's algorithm in terms of running time for generated graphs (in this order: random planar graphs in yellow and random Waxman graphs in green)

Chapter 6

Separators in Planar Graphs

The hierarchical decompositions needed in the multi-level graph approach introduced in the previous chapter leads us now to a topic that is not directly related to the timetable information problems. For planar graphs, there are efficient algorithms to compute relatively small node separators that decompose the input graph into balanced components: The *Planar Separator Theorem* was introduced by Lipton and Tarjan in [LT79], where they give a linear-time algorithm for determining a set of nodes (separator) of size smaller than $2\sqrt{2n}$ that separates a given planar graph with n nodes into two components of size smaller than $2n/3$. Djidjev [Dji82] improved the bound on the separator size to $\sqrt{6n}$, and also proved a lower bound of $1.55\sqrt{n}$, which is still the best known. The algorithms behind these two classical results share a common core algorithm, which determines an appropriate fundamental cycle in a planar graph that contributes to the sought separator.

Since then, a lot of generalisations and extensions have been made, and the best upper bound on separator size currently known is $1.97\sqrt{n}$ due to Djidjev and Venkatesan [DV97], where the aforementioned core algorithm is used as a subroutine, too. Recently, an experimental study [ADGM02] considered a variant of a planar separator algorithm where the bound on the component size can be chosen arbitrarily, which typically requires the graph to be separated into more than two components.

We implemented these classical algorithms, and consider several new algorithmic aspects regarding: (i) the optimisation of separator size and balance, instead of just guaranteeing upper bounds; (ii) the consideration of fundamental cycle separator algorithms in their own right; and (iii) the application of postprocessing techniques to improve the quality of the separators. These issues are the subject of a comprehensive experimental study that constitutes our second contribution. For our experiments, we used a large variety of planar graphs, both from real-world and synthetic inputs with different characteristics (e.g., size of diameter, size of minimum separator, etc).

A surprising outcome of our experimental investigation is that fundamental cycle separator algorithms always provide the best solutions. Another important issue of our experimental analysis concerns the arbitrary choices that have to be made during the course of an algorithm (e.g., the choice of a node as the root of a BFS tree). It turns out that such choices influence significantly the quality of the separators

found.

The contents of this chapter have appeared as technical report entitled *"Engineering Planar Separator Algorithms"* [HPS+04]. We start (Section 6.1) with a brief review of the planar separator algorithms and their implementations, give our optimisation criteria, and introduce the fundamental cycle separator (FCS) algorithm. Moreover, we present the postprocessing techniques applied and address implementation details of the FCS algorithm. Section 6.2 describes the graphs used for our experiments, while the results of our experimental study are reported in Section 6.3. We discuss the obtained results in Section 6.4.

6.1 Separating Planar Graphs

In this section, we consider classical linear-time planar separator algorithms implementing the Planar Separator Theorem as stated below. The node separators computed by the different algorithms fulfil different upper bounds $\beta\sqrt{n}$, for some constant β, on the separator size, while each of the remaining components contains less than two thirds of all nodes. The first theorem of this kind (for $\beta = 2\sqrt{2}$), which constitutes the foundation of our work, was introduced by Lipton and Tarjan [LT79]. For simplicity, we state the theorem and its related algorithms for the case of an unweighted planar graph. The algorithms and our implementations work for the weighted case, as it is introduced in [LT79], as well.

Theorem 6.1 (Planar Separator Theorem) *Given a planar graph G, the n nodes of G can be partitioned into three sets A, B, and S such that no edge joins a node in A with a node in B, neither A nor B consists of more than $2n/3$ nodes, and S contains no more than $\beta\sqrt{n}$ nodes, where β is a constant.*

The proof of the theorem for $\beta = 2\sqrt{2}$ provided in [LT79] is constructive and the running time of the resulting algorithm is linear in the number of nodes of G. We outline the algorithm in Section 6.1.2. It is based on so-called fundamental cycles, which can be computed in linear time. Given a spanning tree of the graph G, every non-tree edge $\{u, v\}$ induces a unique *fundamental cycle* consisting of the path in the spanning tree connecting u and v together with the edge $\{u, v\}$. The following lemma represents the crucial part of the proof; we describe the fundamental cycle algorithm behind the lemma in Section 6.1.4, and provide details about our implementation.

Lemma 6.1 (Fundamental Cycle Lemma) *Let G be a connected planar graph. Suppose G has a spanning tree T of height h. Then, the nodes of G can be partitioned into three sets A, B, and C such that no edge joins a node in A with a node in B, neither A nor B consists of more than $2n/3$ nodes, and C contains no more than $2h+1$ nodes, including the root of the tree. The nodes in C represent a fundamental cycle of G with respect to the spanning tree T.*

6.1.1 Optimisation Criteria

In practical applications of planar separator algorithms, requirements as to "good separations" may vary a lot. We therefore provide three optimisation criteria: *separator size*, *balance*, and *separator ratio*. Let A be the smaller and B the larger of the two components, then *balance* is defined as A/B, and the *separator ratio* as S/A (cf. [LR99]). What is desirable are small separator size and high balance at the same time; the separator ratio, which is to be minimised, represents a trade-off between the two targets. Note that if one of the simple criteria, separator size or balance, is to be optimised and there are several optimal solutions, then the separator ratio criterion becomes relevant.

6.1.2 The Algorithms

We investigate two classical linear-time algorithms, by Lipton and Tarjan (LT) [LT79] and by Djidjev (Dj) [Dji82]. Both work in three phases. First, a breadth-first search (BFS) tree is computed, partitioning the nodes into levels. If one of the BFS levels constitutes a separator fulfilling the size and balance requirements, then the algorithm returns that level. In case there are several feasible levels and an optimisation criterion is applied, the respective algorithm selects a level that is optimal with respect to that criterion. In the second phase, separators consisting of two levels of the BFS tree are considered, yielding a separation of the tree into a lower, middle, and upper part of the graph. Finally, if these separators do not fulfil the requirements, the third phase applies Lemma 6.1 to the middle part of the previous two-level separation. In this case, the separator consists of those two levels and the fundamental cycle found through the lemma. The algorithms differ in the selection of the levels, as described in more detail below. We also consider fundamental cycle separations computed by applying (the algorithmic version of) Lemma 6.1 directly to the graph.

Note that there are several parts in all the above algorithms, where certain arbitrary (in a sense "random") decisions have to be made: (i) the choice of the BFS root and the search itself; (ii) the triangulation of the graph, which is needed in phase 3 of the algorithms; and (iii) the choice of the fundamental cycle from among several feasible ones—the so-called choice of the *non-tree edge*, as explained in Section 6.1.4. We will thoroughly discuss the influence of the choices of the BFS root and the non-tree edge in Section 6.3.

Lipton and Tarjan (LT). First, the middle level in the BFS tree is considered: the first level, starting from the root, that covers together with the lower levels more than half of the nodes. If this level is too large, the levels above and below are scanned until in each direction a level of size less than $2(\sqrt{n} - D)$ is found, where D is the distance to the middle level. If these two levels do not meet the bound, then Lemma 6.1 is used to separate the part between these two levels and in this case the separator consists of the two levels plus a fundamental cycle. We consider a textbook version [Meh84, Koz92] of the algorithm guaranteeing a separator of size less than $4\sqrt{n}$, i.e., $\beta = 4$.

Djidjev (Dj). Already in [LT79], Lipton and Tarjan give an even better bound, and in [Dji82] Djidjev further improves the selection of levels to $\beta = \sqrt{6} \approx 2.45$. In a similar but more sophisticated way than that in LT, the algorithm tries to find a separator consisting of one or two levels of the BFS tree (which have to be smaller than in LT), and as final option also determines a fundamental cycle.

Fundamental Cycle Separation (FCS). During the experimental phase of this study, we observed that it is very effective to omit the selection of levels and directly consider fundamental cycles as separators. Simple cycle separators are promising from a theoretical point of view, since an upper bound on the separator size of $1.97\sqrt{n}$, which is (to our knowledge) the best one currently known [DV97], is achieved by a simple cycle. From a practical point of view, the algorithm used in the proof of the latter bound seems too complicated. Instead, we compute a simple cycle separator by applying Lemma 6.1 directly to the input graph. Note that we can only guarantee the separator to be smaller than or equal to $2d + 1$, where d denotes the diameter of the input graph.

Optimised versions. We have also implemented *optimised versions* of LT, Dj, and FCS that select an optimal separator according to a specific optimisation criterion (cf. Section 6.1.1).

6.1.3 Postprocessing

To the above algorithms we provide two optional postprocessing steps, which may help to improve the separation found by the specific algorithm in terms of separator size and/or balance. The first one, called *node expulsion*, consists of moving separator nodes that are not connected to both components A and B (and hence do not actually separate two nodes from different components) to one of the components, thus decreasing the size of the separator. If a node can be moved to either component, then it is assigned to the smaller one, so that imbalance does not deteriorate. The idea behind the other postprocessing step, called the *Dulmage-Mendelsohn optimisation* [AL96], is to detect a subset of the separator, $\emptyset \neq S' \subset S$, such that the subset $B' \subset B$, consisting of nodes that are adjacent to S' and belong to the larger component B, is smaller than S'. Then, the separator is modified by removing the nodes in S' and adding the nodes in B'. The size of the new separator is smaller than the original one, and the balance may be improved as well. For details on the construction of S' we refer to [AL96].

6.1.4 Implementing the Fundamental Cycle Lemma

The proof of Lemma 6.1 given by Lipton and Tarjan in [LT79] is constructive in the sense that it delivers an algorithm for computing the desired fundamental cycle. This algorithm provides the core to all planar separator algorithms under investigation. Briefly, the algorithm, which will be referred as the *Fundamental Cycle Separator (FCS) algorithm*, is as follows. The graph is triangulated and a spanning tree of the appropriate height is constructed. Every non-tree edge forms together with some

tree edges a cycle. Each of these cycles is a candidate fundamental cycle separating the graph into two parts, so the goal is to find a cycle such that these two parts have the appropriate sizes. The optimised version of the FCS algorithm selects a cycle that induces an optimal separator according to a specific optimisation criterion.

The FCS algorithm proceeds by examining each non-tree edge and the corresponding cycle, keeping track of the nodes on the cycle as well as of those inside it and their weight. Depending on whether the two edges, which form together with the current non-tree edge the triangular face lying *inside* the cycle, belong to the tree, the algorithm combines information computed for previous cycles until the desired cycle is found (e.g., in the case that one of the other two edges is non-tree and its corresponding cycle lies inside the current cycle). It is relatively straightforward to show that this strategy will compute a fundamental cycle.

Although the description of the algorithm is clear, there is a subtle problem of how to deal with the notions *inside* and *outside*. To compute correctly the information about cycles, there must be a sense of direction so that the non-tree edges are examined in a meaningful order. Our approach provides this sense of direction. We make use of the dual of the planar graph. The following well-known lemma provides an interesting property linking the spanning trees of a planar graph and its dual.

Lemma 6.2 *Let $G = (V, E)$ be a connected planar graph with dual $G^* = (V^*, E)$, and let $E' \subseteq E$. Then, $T = (V, E')$ is a spanning tree of G iff $T^* = (V^*, E - E')$ is a spanning tree of G^*.*

The given planar graph G is triangulated and a spanning tree $T = (V, E_T)$ of appropriate height is found using a simple breadth-first search, and the dual G^* is constructed. By Lemma 6.2, the edges $E - E_T$ form a spanning tree $T^* = (V^*, E - E_T)$ of G^*. A node of T^* is chosen (arbitrarily) to be the root of T^* and all edges of T^* are directed away from this root. It can be easily proven that all these constructions require linear time.

Since there is a one-to-one correspondence between the non-tree edges in the original planar graph and the tree edges of the dual, there is a correspondence between the cycles in the original and the tree edges of the dual graph. By first examining the cycles corresponding to the edges that lead to the leaf nodes of T^* and continuing towards the root, the properties of all the cycles can be computed inductively. The direction of the edges in T^* ensures that when examining a cycle, all information needed (from cycles that lie inside it) will have been already computed.

Evidently, our strategy provides the necessary order of cycle examination. It should be noted that in the actual implementation of the algorithm the construction of the dual graph and its spanning tree can be avoided. Instead, the construction of the spanning tree is simulated by performing a "breadth-first" traversal on the *faces* of the original graph using the non-tree edges. We keep track of the edges that are used to "enter" a face and then examine them in the reverse order of their discovery. The result is a correct and reasonably efficient implementation.

6.2 Data Sets

In the following, we give a brief description of the graph classes used in our experiments and illustrated in Figures 6.1-6.6. The first five categories consist of synthetically generated graphs, whilst the data sets in the last stem from real world.

Grid-like graphs. This category encompasses three classes of regular-structured graphs, namely `grid`, `rect(angular)`, and `sixgrid`. The `grid` and `rect` graphs can be regarded as an $x \times x$ or $x \times y$ raster of nodes, respectively, where adjacent nodes of the same row or column are connected by an edge. A `sixgrid` graph is composed of $x \times y$ hexagons in a honeycomb-like fashion. In a `grid` graph with n nodes a separator with minimal size consists of approximately $\sqrt{2n/3} \approx 0.82\sqrt{n}$ nodes. If $x \ll y$, then the smallest separator of a `rectangular` graph has x nodes, and a `sixgrid` graph has an optimal separator with $x + 1$ nodes.

Sphere approximation. In [Dji82] the currently best lower bound of $1.55\sqrt{n}$ on the separator size is proven by graphs that approximate the sphere. We consider two simple constructions of graphs that approximate the sphere, as *worst-case* examples concerning separator size. A `globe` graph is—simply speaking—the graph induced by (a specified number of) meridians and circles of latitude of a terrestrial globe. A `t-sphere` graph approximates the sphere by almost similar triangles, see e.g., [Bou92]. The iterative generation process starts with an icosahedron (consisting of 20 equilateral triangles with all nodes on the sphere). During an iteration each triangle is split into four smaller ones.

Graph with big diameter. Given a diameter d, we construct a maximal planar graph that consists of $3d + 1$ nodes and has diameter d. We refer to this class as `diameter`. By construction, such a graph has always a separator of size 3.

Random planar graphs. Random maximum planar graphs, denoted by `del-max` and `leda-max`, are generated such that the specified number of nodes are randomly placed in the plane and the convex hull of them is triangulated, the triangulation being a Delaunay triangulation (`del-max`) or a standard LEDA-triangulation [NM99] (`leda-max`), respectively. In addition, we have the `del` and `leda` graphs, which are obtained from `del-max` and `leda-max`, respectively, by deleting at random a specified number of edges. We will occasionally refer to `del` and `del-max` (`leda` and `leda-max`, resp.) as the Delaunay (LEDA, resp.) graphs.

Graphs with small separators. Given a planar graph, two copies of this graph are connected via a given small number of additional nodes, which constitute a perfectly balanced separator of a so constructed graph. The challenge of the algorithms is to re-determine these small separators. We consider four of the previous graph types and get the following new kind of generated graphs: `c-grid`, `c-globe`, `c-del-max`, and `c-leda-max` graphs.

Road map graphs. Regarding real-world data, we consider seven graphs representing the road networks of some U.S. cities and their surrounding areas (referred to as `city`), taken from the San Francisco Bay Area Regional Database (BARD) [Bar] and the Environmental Systems Research Institute (ESRI) info-page [Esr].

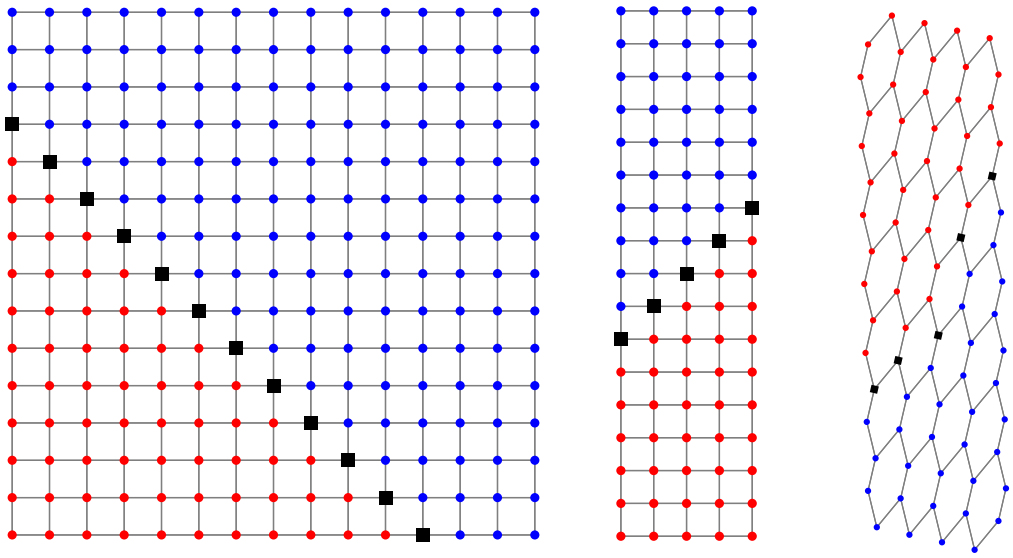

Figure 6.1: Sample grid-like graphs. From left to right: a 15x15 `grid`, a 17x5 `rectangular` graph, and a 7x4 `sixgrid` graph. Nodes belonging to a component are drawn as red (grey) and blue (black) circles, separator nodes as black squares.

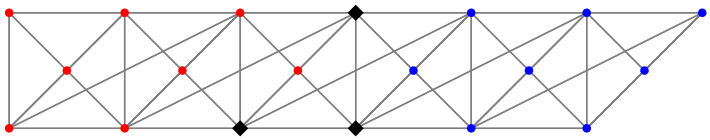

Figure 6.2: Sample `diameter` graph with diameter 6.

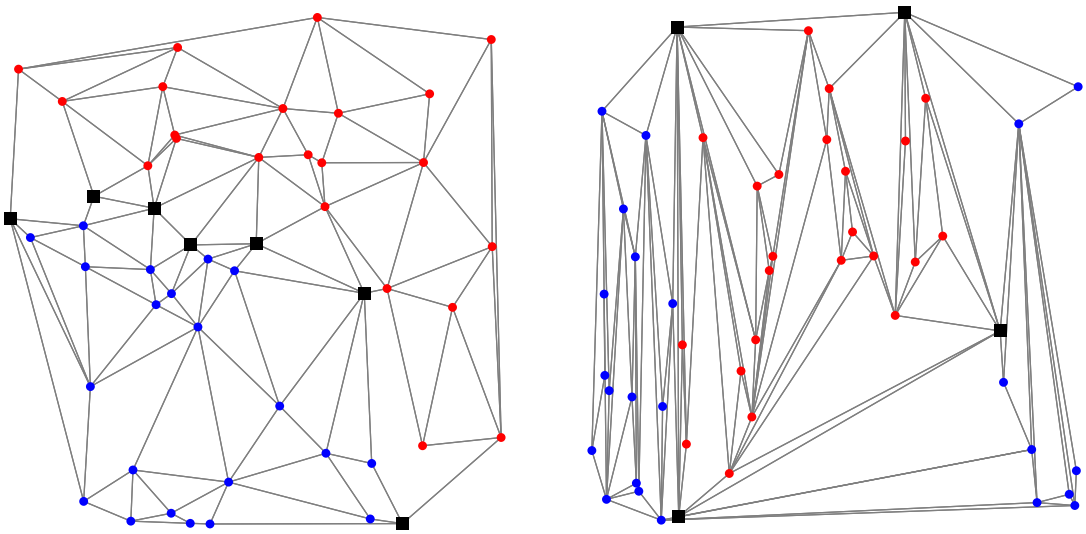

Figure 6.3: Sample random planar graphs with 50 nodes and 125 edges: `delaunay` (left) and `leda` (right).

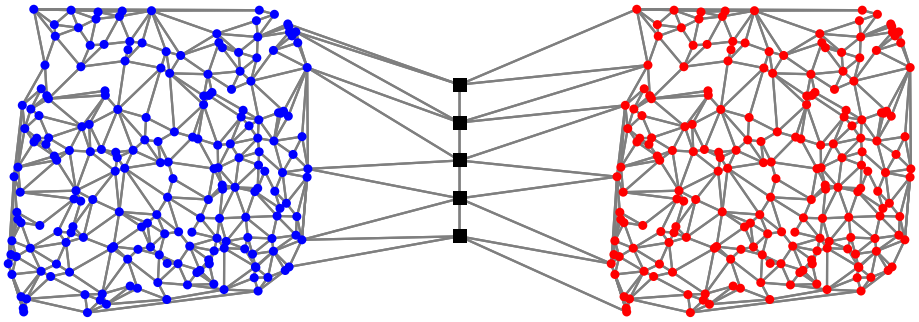

Figure 6.4: A `c-del-max` graph with 5 connecting nodes.

6.3 Experiments

Our experimental study is subdivided into three parts encompassing graphs of increasing size. The three algorithms LT, Dj, and FCS have been implemented in `C++` using the LEDA library [NM99] (version 4.5). The code is compiled with the GNU `C++` compiler (version 3.3.3) and the experiments were performed on a 2.8 GHz Intel Xeon machine running a Linux kernel (version 2.6.5).

6.3.1 Small Graphs

For each of the generated graph types `grid`, `rect`, `sixgrid`, `globe`, `del`, `leda`, `del-max`, and `leda-max`, we considered series of ten graphs containing between 50 and 1000 nodes. We take into account all algorithms, LT, Dj, and FCS, optimised on separator size, and each node was once chosen as BFS root. As already described above, if more than one smallest separators have been found, the one with best balance is selected.

Separator size and balance. Concerning the grid-like and `globe` graphs, the differences between the three algorithms are quite small. Due to the regular construction of these graphs, LT and Dj always succeed right after the first phase, and the smallest BFS level is almost optimum. The mean size of a fundamental cycle separator is always slightly smaller and yields better balance. For the randomly generated graphs, the results are different. Although for the Delaunay graphs, LT always terminates after the first phase with a smallest valid BFS level, Dj applies for around 15 per cent of the BFS roots the last phase of the algorithm.

The diagrams in Figure 6.7 show the mean values of separator size and balance for the case of Delaunay graphs. It can be clearly seen that FCS computes on average the best separators, while Dj is slightly better than LT. Considering the LEDA random graphs, the results are again different: With those, both LT and Dj always have to pass the third phase. The mean separator size of FCS is only slightly better than that of Dj, while LT is by far worse. The mean balance with LEDA graphs is similar for all three algorithms, in the range between 0.8 and 0.9.

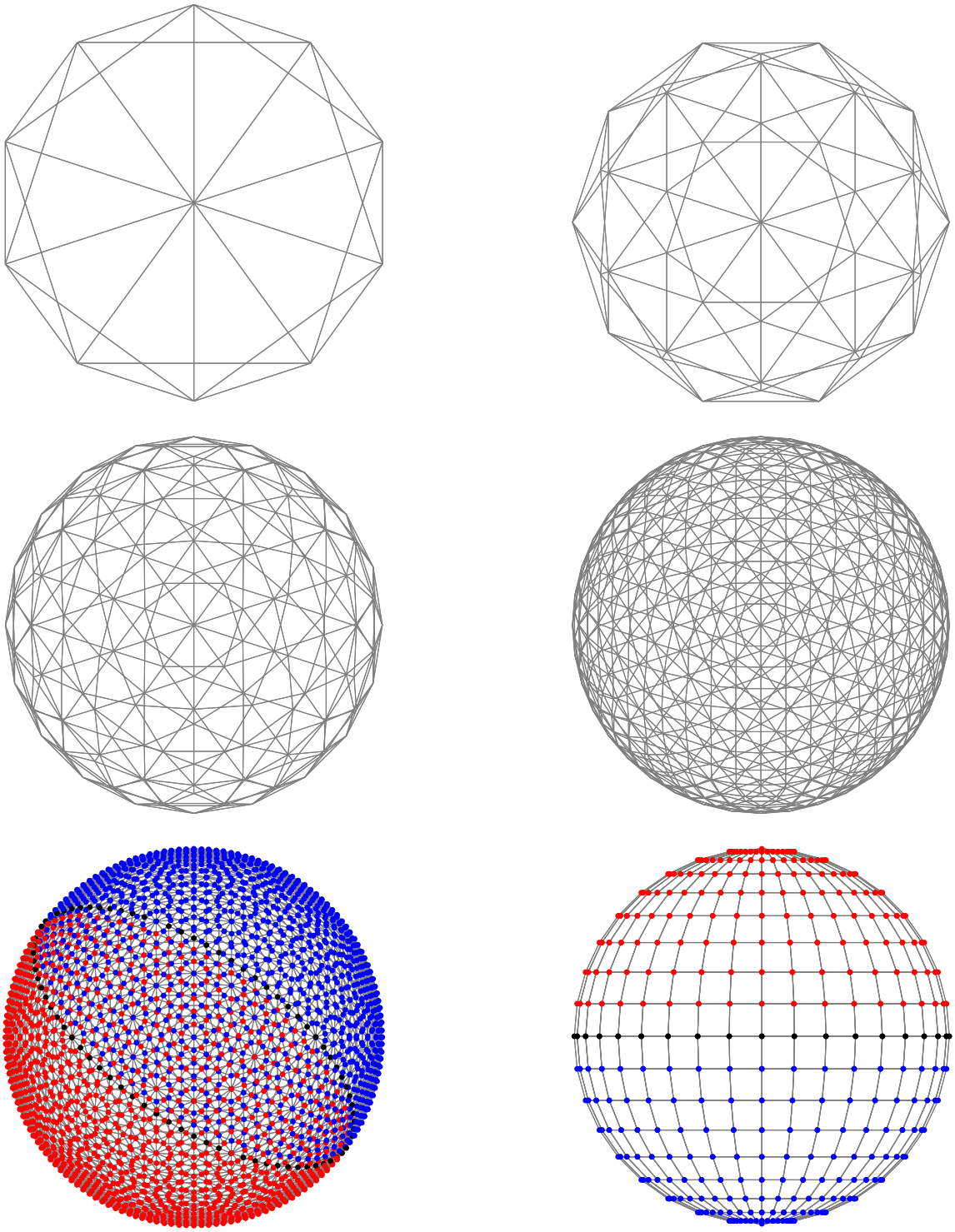

Figure 6.5: Approximation of the sphere: `t-sphere` graphs with 0 to 4 iterations, and a `globe` graph.

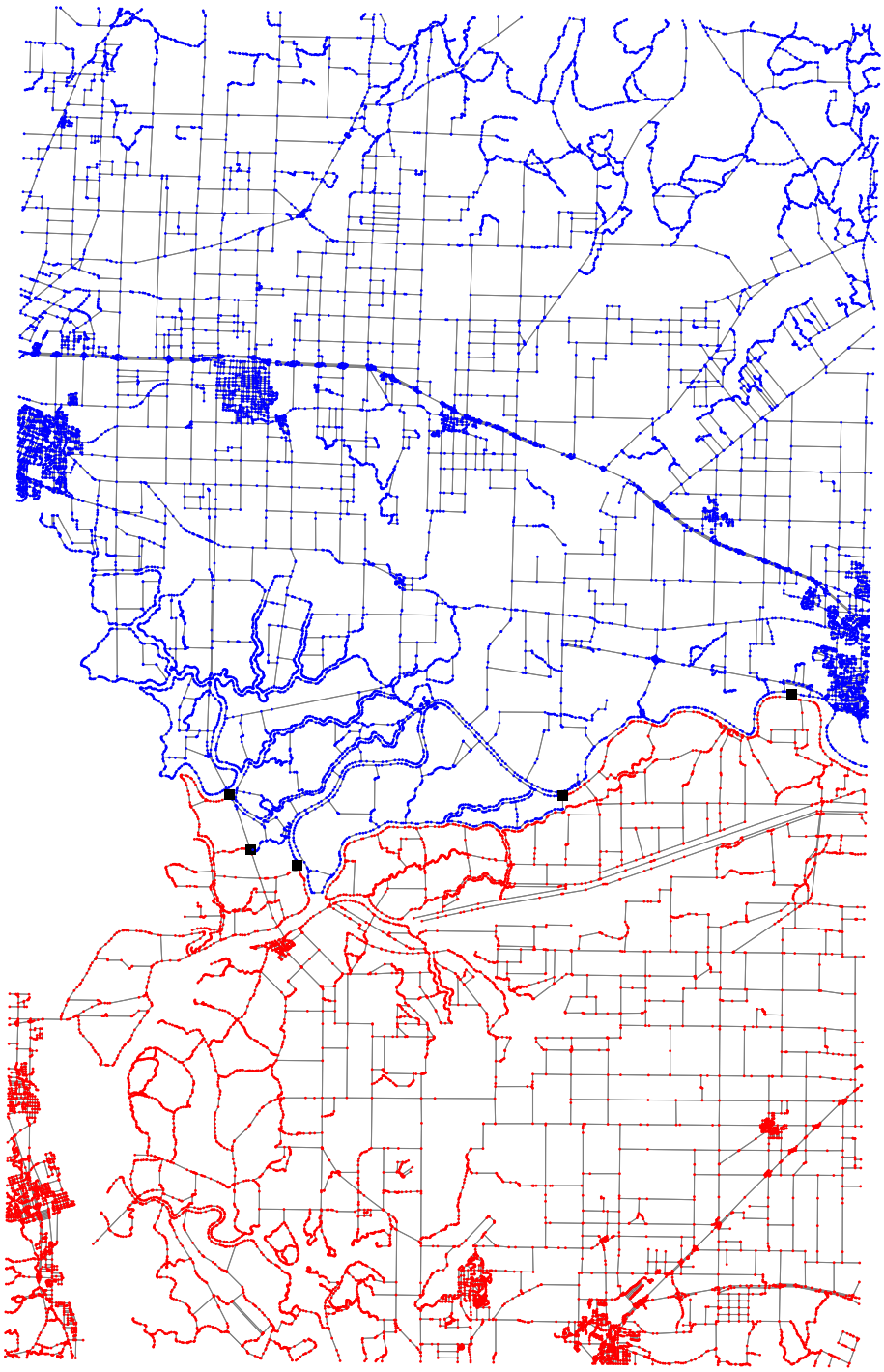

Figure 6.6: Sample real-world graph: `city5`. The separator has been computed with the randomised method suggested for very large graphs in Section 6.3.3.

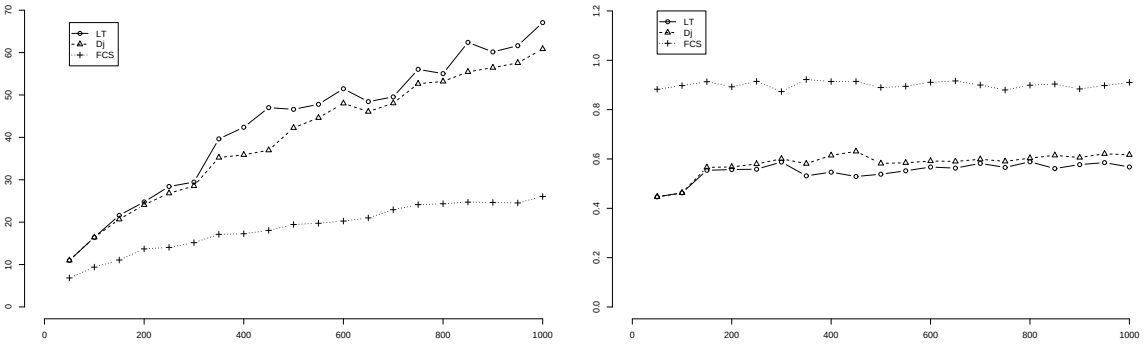

Figure 6.7: Experiments with a series of Delaunay graphs of sizes ranging from 50 to 1000 nodes: the mean separator size (left) and mean balance (right) with LT, Dj, and FCS.

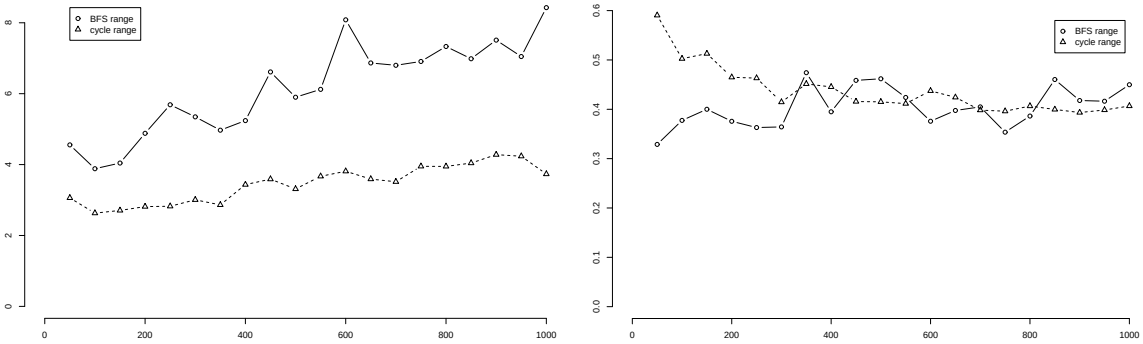

Figure 6.8: Same graph series as in Figure 6.7: for FCS the mean ranges of BFS root vs. non-tree edge selection concerning separator size (left) and balance (right).

graph	nodes	edges	diam	
			orig	triang
`grid`	10000	19800	198	67
`rect`	10000	19480	518	20
`sixgrid`	9994	14733	513	22
`globe`	10002	20100	101	101
`t-sphere`	10242	30720	96	96
`diameter`	10000	29994	3333	3333
`del`	10000	25000	56	45
`del-max`	10000	29971	52	48
`leda`	9989	25000	18	15
`leda-max`	10000	29975	15	14
`c-grid`	10087	19904	212	72
`c-globe`	10090	20325	144	142
`c-del-max`	10005	29972	65	58
`c-leda-max`	10005	29984	20	16
`city2`	2948	3564	131	14
`city3`	15868	16690	658	13

Table 6.1: Parameters for the graphs used in the experiments with large graphs. The table depicts the number of nodes and edges, the diameter (diam) of the original graph (orig) and the diameter of a triangulation of the graph (triang).

BFS root and non-tree edge selection. The diagrams in Figure 6.8 show the influence of BFS root selection and non-tree edge selection on separator size and balance. For FCS applied to the Delaunay graphs, the mean ranges of both the separator size and balance values are depicted, either over all possible BFS root nodes or over all possible non-tree edges. More precisely, the *mean range of the separator size over all possible BFS root nodes*, for example, is defined as follows: For every BFS root, determine the mean separator size over all possible non-tree edges. Then, the wanted *mean range* is the difference between the maximum and the minimum of these separator sizes among all BFS root nodes. The other mean ranges are defined similarly. The diagrams show that selection of the BFS root node is more decisive for the separator size than non-tree edge selection. Concerning balance, both selections are of similar importance.

6.3.2 Large Graphs

The second series of graphs that we experimented with, the *large graphs*, consists of 16 graphs of the categories mentioned in Section 6.2 of size roughly 10000 (except for the two real-world graphs, which have about 3000 and 16000 nodes, respectively). The `rectangular` graph represents a 20×500 raster, the `sixgrid` graph consists of 20×237 hexagons, the `globe` has 100 meridians and circles of latitude, and the `t-sphere` is constructed by 5 iterations. For `c-grid`, `c-del-max` and `c-leda-max`, the two copies of the respective graph are connected by 5 nodes, while for `c-globe`

graph	BFS level		LT		Dj		FCS	
	orig	triang	min	mean	min	mean	min	mean
grid	**82**	135	**82**	106	**82**	106	89	99
rect	20	931	20	27	20	27	**20**	20
sixgrid	21	839	21	28	21	28	**21**	21
globe	100	100	100	119	100	119	**100**	106
t-sphere	160	160	160	169	160	169	**160**	164
diameter	3	3	3	4	3	4	**3**	3.3
del	206	232	206	300	82	113	**65**	75
del-max	204	233	204	314	86	117	**74**	79
leda	783	1459	76	216	7	31	**5**	8
leda-max	1604	1619	56	205	7	26	**6**	10
c-grid	38	73	38	78	38	78	**5**	6.4
c-globe	4	4	4	96	4	96	**4**	12
c-del-max	74	215	74	318	19	65	**5**	8.3
c-leda-max	211	1050	78	209	7	32	**4**	4.5
city2	15	162	15	39	15	39	**4**	9.5
city3	28	2161	28	53	28	53	**4**	6.8

Table 6.2: Results for large graphs: BFS tree level for both the graph itself (orig) and a triangulation of it (triang); for each of the algorithms LT, Dj, and FCS, all of them optimised on separator size *with* postprocessing applied, the minimum and mean separator sizes; bold-face figures represent the best separator size achieved for the respective graph.

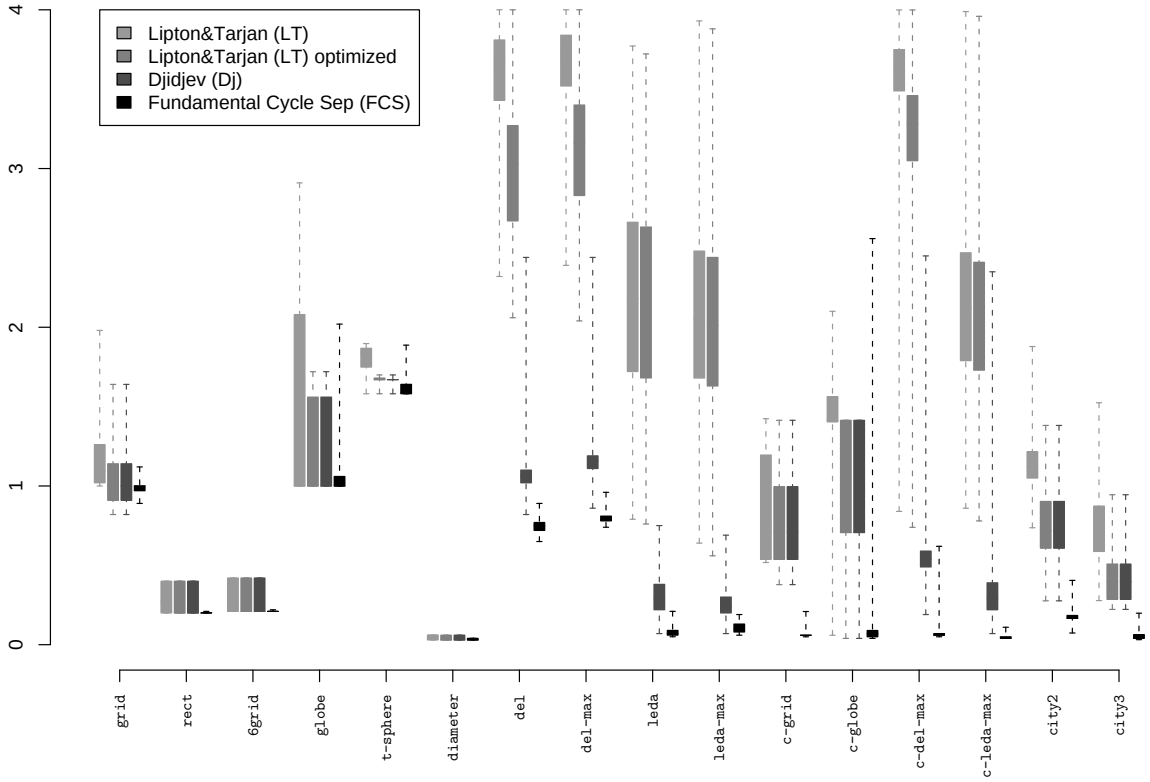

Figure 6.9: Box plots depicting the separator sizes relative to $\sqrt{n}$ obtained with unoptimised LT (light-gray) and LT (gray), Dj (dark-gray), and FCS (black), the latter three optimised on separator size. The dashed lines indicate the range of all separators *without* postprocessing applied.

only 4 nodes are used to connect the two `globe` graphs. A detailed synopsis of the graphs and some of their characteristics, such as the diameter and the minimum number of nodes in a level of a BFS tree forming a valid separator, for both the original (orig) and the triangulated graph (triang), respectively, are reported in Table 6.2.

In the first experiments that we carried out for large graphs, we investigated the performance in terms of separator size of LT, both unoptimised and optimised on separator size, Dj, and FCS, the latter ones also optimised on separator size. We ran each of these algorithms for each graph while once making each node the root of the BFS tree.

The subsequent experiments deal with the effect of postprocessing on the various algorithms. In order to appropriately choose the postprocessing variant to be performed with each optimisation criterion, we undertook a pre-study, performing for each graph and each combination of optimisation and preprocessing steps one run of each of the above algorithms.

To get an idea of the quality of the separators found by the algorithms, we

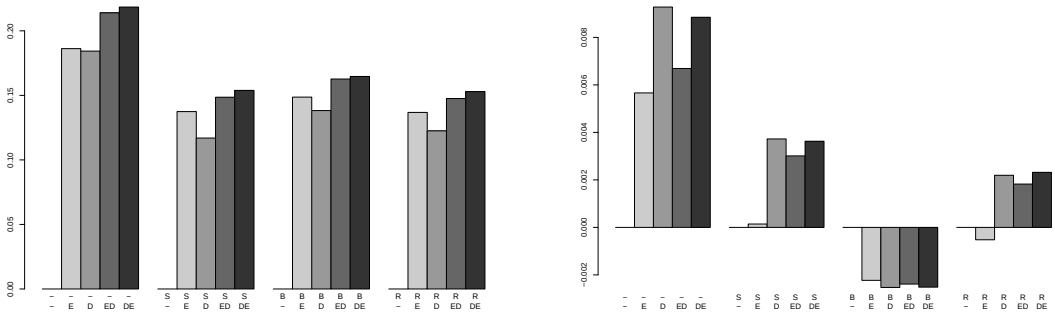

Figure 6.10: Pre-study: average separator reduction and balance improvement. The upper key in the x-axis labels denotes optimisation (-: none, S: separator, B: balance, R: separator ratio); the lower one stands for postprocessing (-: none, E: node expulsion, D: Dulmage-Mendelsohn decomposition), where double letters reflect the application order of the postprocessing steps.

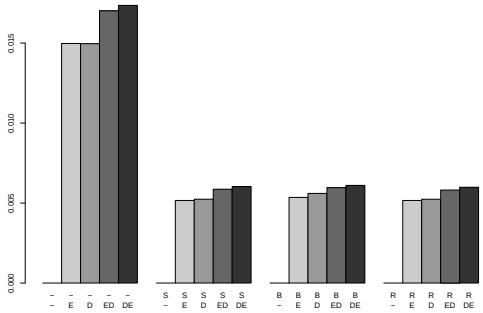

Figure 6.11: Pre-study: separator ratio improvement. The keys are as in Figure 6.10.

compare them against separators obtained with the help of *MeTiS* [Kar], a graph partitioning tool collection. Separators determined by MeTiS are a trade-off between separator size and balance, so for the sake of a meaningful comparison, we contrast the MeTiS results and our algorithms optimised on separator ratio.

Pre-study. Figures 6.10 and 6.11 depicts average values of the separator reduction relative to the former separator size and the absolute improvement in terms of balance and of separator ratio. These results suggest that optimisation of separator size and separator ratio should be accompanied by a combination of Dulmage-Mendelsohn optimisation as the first postprocessing step and node expulsion afterwards; the same holds when not optimising at all. With balance as optimisation criterion, no postprocessing step is on the average able to improve balance further.

Comparison of algorithms. The results of the experiments regarding the separator sizes achieved by the various algorithms is listed in Table 6.2, and illustrated in Figure 6.9 by means of box plots that represent the middle fifty per cent of the data series (note that the whiskers here span the whole range of outcomes). The data show that—except for the `grid` graph—the smallest separator is found by FCS. This, together with the fact that the boxes are clearly slender, and—except for `c-globe`—the ranges are minimal for FCS, suggests that FCS outperforms sig-

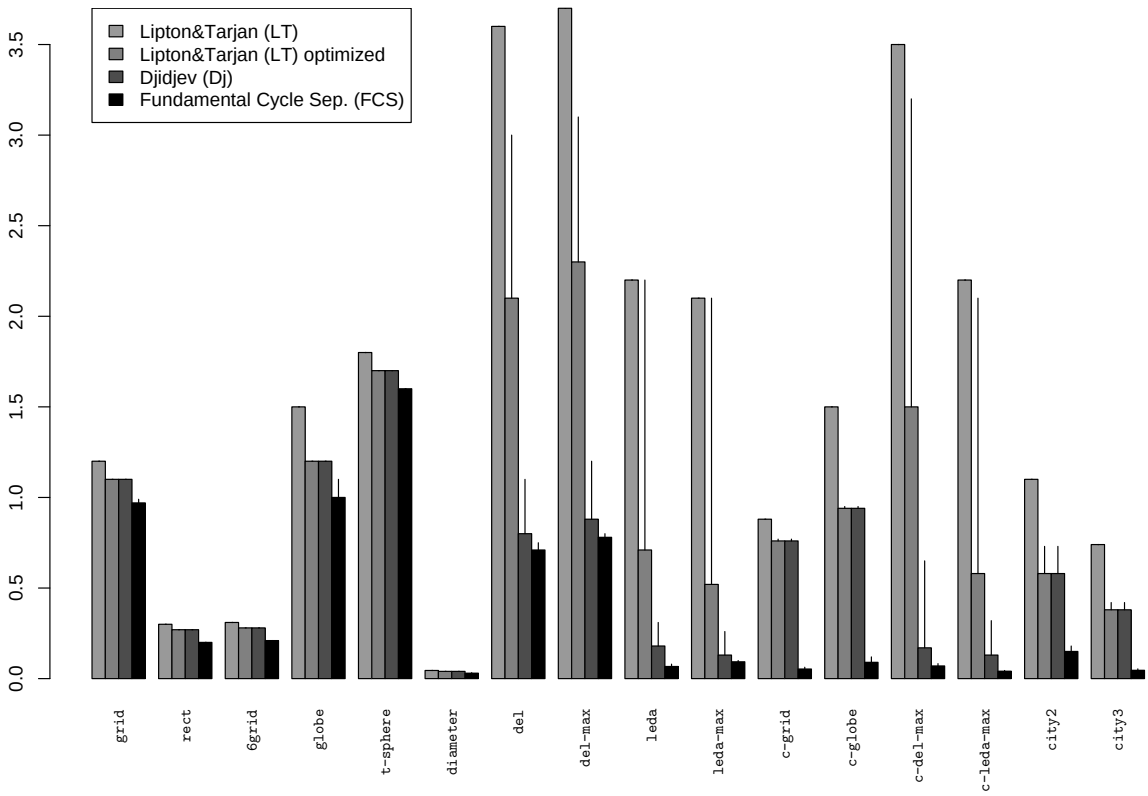

Figure 6.12: Postprocessing: average separator size for unoptimised LT (light-gray) and LT (gray), Dj (dark-gray), and FCS (black), the latter three optimised on separator size. The line on top of a bar shows the respective value before the postprocessing.

nificantly the other algorithms in terms of separator size.

The runtime is linear for all of our algorithms. However, for the algorithms LT and Dj, the constant crucially depends on the phase in which the algorithms terminate (cf., Section 6.1.2). The first two phases consist basically of a breadth-first search, while the computation of the fundamental cycle requires expensive operations like embedding, triangulation, and copying. FCS, of course, computes a fundamental cycle and always needs the expensive operations. LT and Dj terminate after phase 1 with all grid-like graphs, sphere-approximating graphs, and with the `diameter`, `c-grid`, `c-globe`, and `city` graphs. In contrast, the LEDA, `c-del-max`, and `c-leda-max` graphs in the majority of cases require phase 3. For the Delaunay graphs, LT mostly terminates after phase 1, but Dj needs phase 3. The mean running time for LT (applying only phase 1) in the `city3` graph, for example, is 0.04 seconds, while FCS involving a fundamental cycle computation needs 0.71 seconds.

Postprocessing. Figure 6.12 depicts the average separator size achieved with the diverse algorithms before and after the application of a postprocessing step. On the one hand, for the grid-like and sphere-approximating graphs as well as the `diameter`

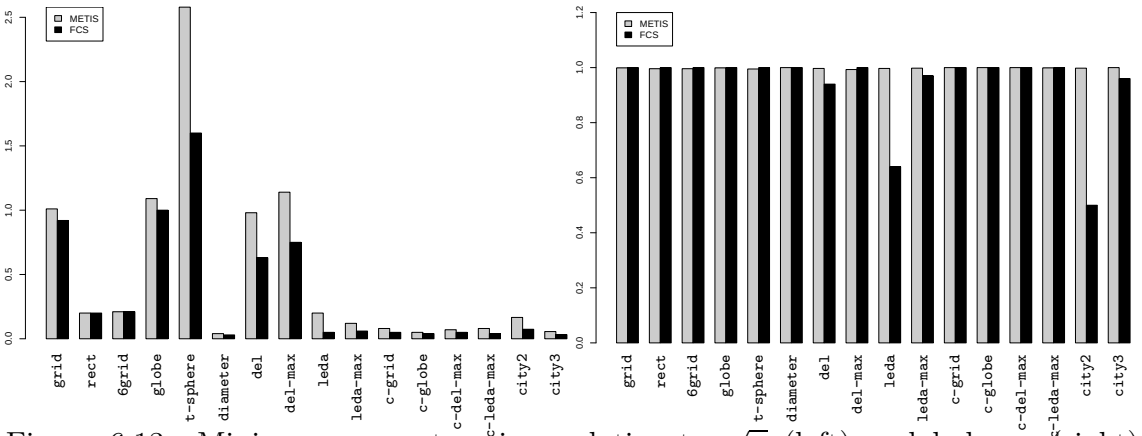

Figure 6.13: Minimum separator sizes relative to $\sqrt{n}$ (left) and balance (right) computed with MeTiS (light-gray) and with FCS (black) optimised on separator ratio.

graph, the separators found by the algorithms without postprocessing cannot be much improved. (Often the separators are already close to optimal solutions for these graphs.) On the other hand, for the remaining graphs, the separators computed by the algorithms Dj and LT are very large compared to an optimal solution, and in these cases the postprocessing greatly improves the separators. The separators computed by FCS can generally be improved only a little.

Benchmark. MeTiS provides—amongst others—implementations of two algorithms, *kmetis* and *pmetis*, for computing small-cardinality k-way edge partitions with balancing constraints. The application of both *kmetis* and *pmetis* on all of our instances yield 2-partitions of 'very high' balances (meeting the requirement that each part encompass at least one third of the graph's nodes) and with 'quite few' cut edges.

From an edge partitioning thus obtained we can get a node separator by choosing an appropriate subset of the end-nodes of the edges forming the cut. To achieve this, our implementation proceeds as follows: at each time, until all cut edges are covered, i.e., have one of their end-nodes included in the separator, we pick one end-node as a separator node, trying to cover with it as many cut edges not covered yet as possible.

Since among LT, Dj, and FCS optimised on separator ratio, the solutions computed by FCS were the best with respect to both separator size and balance, we compare MeTiS with FCS only. Figure 6.13 shows the best separator size and balance values. One may state that with FCS, the separator size is always at least as good as with MeTiS and balance is almost always comparable. Those graphs whose balance is considerably worse with FCS than with MeTiS (`leda` and `city2`) exhibit by far smaller separators with FCS, which suggests that the weighting between the two criteria, separator size and balance, seems to be more in favour of separator size with FCS, while MeTiS tends to prefer balance. Indeed, almost perfect balance can always be achieved with FCS optimised only on balance.

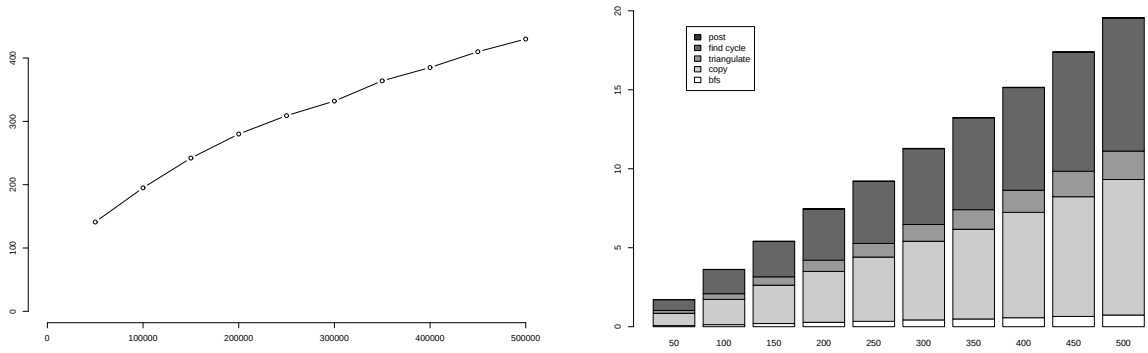

Figure 6.14: Delaunay graph series: Separator size (left) and computation time in seconds (right). The separators were computed with FCS for ten randomly selected BFS roots optimised on separator ratio. Shares in computation time for one BFS root (from bottom to top): computing the BFS tree, copying the graph, triangulating the graph, determining fundamental cycles, postprocessing.

graph	nodes	edges	size	balance
`city1`	1429	3034	5	0.871
`city2`	2948	3564	8	0.996
`city3`	15868	16690	7	0.869
`city4`	20036	41476	10	0.789
`city5`	24106	53826	5	0.740
`city6`	38823	79988	8	0.704
`city7`	44878	90930	7	0.547

Table 6.3: City graph series: number of nodes and edges, and best separator size and balance values achieved with the randomised fundamental-cycle heuristic optimised on separator ratio.

6.3.3 Very Large Graphs

Under the name of very large graphs, we consider two series of graphs increasing in size: a series of city graphs with numbers of nodes up to about 45,000 and a series of ten random `del` graphs of sizes between 50,000 and 500,000. For these graphs we computed separations by the following procedure: run FCS on ten BFS trees of a given graph, determined by a random node as root, and from among these separations take the one with best separator ratio.

The results of the experiments with the city graph series is depicted in the table aside. Obviously, all city graphs have extremely small separators, which are also found by our algorithm. The separators for the `city2` and `city3` graphs, which had already been included in the experiments of the previous section, are somewhat bigger than those of the preceding experiment (8 and 7 instead of 4, resp.; see Table 6.2). This is due to the fact that: (i) the separator ratio is now optimised instead of the separator size; and (ii) we do not longer take into account every node

as a BFS root.

Figure 6.14 shows the separator sizes and the running times with the Delaunay graph series. The balance is very high for all graphs, namely greater than 0.94, and the separator size divided by $\sqrt{n}$ is always in the range between 0.6 and 0.63. This suggests that the separator ratio criterion indeed represents a good trade-off between separator size and balance. The running times vary between approximately 2 and 20 seconds. Also, the diagram pronouncedly reflects linearity of the algorithms' time complexity.

6.4 Discussion

Our experiments have shown that, especially for graphs with small separators, there is a high potential for optimising the separators computed by the algorithms. Both the postprocessing and in particular the Fundamental Cycle Separation yielded almost-optimal separators with respect to separator size and balance (in contrast to the classical algorithms for many graphs). Selection of the non-tree edge in the fundamental cycle computation has a considerable influence on both criteria, and we are able to select the best during the respective algorithm. The choice of the BFS root also exhibits a great impact on separator quality, mainly on its size.

The only graph where FCS did not compute the best solution was the `grid` graph (cf. Table 6.2). This must be due to a "bad" triangulation of the graph, since one can specify a triangulation such that FCS finds the optimal solution. For `grids`, the optimal separator is not a cycle in the graph, but it is a cycle in a "good" triangulation. A strategy to improve the algorithm would be to compute the triangulation during the breadth first search to determine the spanning tree. The intuition behind this approach is to assume that the root belongs to the optimal separator that shall be computed; by triangulating during the breadth first search, edges are used to triangulate the graph such that the optimal separator becomes a fundamental cycle in the triangulated graph. Of course, the problem that also the breadth first search makes random choices, which can lead to non-optimal separators, remains.

For practical application, in case more runtime can be invested in order to achieve high quality separators, we suggest to select a small (randomly chosen) set of BFS roots and compute the separators using an optimised FCS algorithm with postprocessing.[1] Our experiments on very large graphs demonstrated that this strategy yields good results.

[1] We have unsuccessfully tried to classify "good" BFS roots. A better strategy to select the BFS roots remains as an open problem.

Chapter 7

Conclusion

We want to conclude the thesis by summarising the achieved results and giving some remarks about open problems for each of the three main parts.

7.1 Timetable Information

We have thoroughly investigated models and algorithms for the core of an timetable information system for public transportation. The crucial issues have been (i) the optimality of the solutions with respect to several criteria; (ii) realistic modelling (e.g., the integration of train transfers); and (iii) the efficiency of the algorithms. Both the time-expanded and the time-dependent approaches have been implemented and we evaluated the performance and experimentally compared the models. We were able to model the realistic itinerary specifications under consideration in both approaches as shortest path problems. For the simplified specification without transfer rules, the time-dependent approach clearly outperforms the time-expanded approach. Realistic transfer rules required extensions of the models: Additional nodes and edges had to be included such that a transfer could be modelled as edge cost. In the time-expanded model the number of edges is doubled, while in the time-dependent model the number of edges is more drastically increased (by a factor depending on the number of train lines per station). This different behaviour is also reflected in our experiments: for most of the considered problems, the running time is roughly the same in both realistic models.

When we integrated the additional criterion of optimising transfers (in combination with the earliest arrival), the time-expanded approach benefited from the straightforward modelling of itineraries as paths in a static graph. In the time-dependent approach a bigger change of the underlying graph was necessary. The crucial principle of the time-dependent approach is to only consider the first train leaving a station in some direction. However, when train transfers shall also be optimised, the first train is not always the best anymore. Hence, the edges had to be split into several edges, each representing the set of trains belonging to one train line, such that the first train is the best among all (fewer than before) trains on a new edge again: it doesn't make sense to transfer into a later train of the same line.

It may be necessary to include further optimisation criteria besides the two main

criteria earliest arrival and minimum number of transfers that we have discussed in this thesis. A criterion that can be modelled as edge length in the time-expanded or the time-dependent graph can be integrated easily into the approaches. The labelling algorithm to compute all Pareto-optimal paths can be used to find all Pareto-optimal connections with respect to earliest arrival, transfers, and the new criterion. It is more likely that an additional criterion can be modelled as edge lengths in the time-expanded graph than in the time-dependent graph, since in the latter graph—as discussed above—only the first train on an edge is considered, which is usually not the best with respect to the new criterion. In the time-expanded graph, each elementary train connection corresponds to a different edge, and thus every criterion that can be broken down to a cost per elementary connection can be modelled directly as edge length. In case the criterion cannot be modelled as edge lengths, it might be possible to extend the graphs similarly to the extensions we did for modelling transfers.

7.2 Speedup Techniques

We have investigated techniques to improve the performance of our algorithms for the single-criterion case (i.e., modifications of Dijkstra's algorithm). Standard approaches like bidirected or goal-directed search could either not be applied or did not yield a significant improvement. However, we demonstrated that preprocessing techniques, like the multi-level graph approach that we have extensively investigated, have the potential to drastically improve the running time by two digit factors. Also the combination of such techniques is viable, as we have observed that combinations of techniques yield much higher speedup factors as the single techniques for the time-expanded graph as well as for other graphs (randomly generated and road networks). The drawback of these techniques is, of course, the expensive preprocessing step. As open problem we want to mention the treatment of dynamic changes of the graph. When an edge length is changed or edges are deleted or added to the graph, the precomputed information (e.g., the multi-level graph) is not valid anymore and has to be updated accordingly. Such dynamic changes may be due to delayed trains, or accidents. Also, traffic days of trains can be regarded as dynamic changes of the underlying graph (i.e., some edges are removed on certain days).

The multi-level graph approach proved to be suited to improve the performance of Dijkstra's algorithm by factors from 11 to 17 when it was applied to the time-expanded graph. The best performance was achieved when 3 or 4 additional levels have been used, which shows that it is indeed useful to use more than one additional level, which was not clear before. Our experience with multi-level graphs is that the selection of nodes inducing the hierarchical decomposition needs to be done carefully. The resulting speedup crucially relies on (i) the number of additional levels (for larger graphs more levels are feasible); (ii) good decompositions; and (iii) on the number of selected nodes in each level. From the theoretical point of view we have shown that for graphs with an especially regular hierarchical structure the multi-

level graph approach yields search spaces that are asymptotically smaller than the search space of Dijkstra's algorithm. Hence, for sufficiently large graphs, arbitrary high speedup factors can be achieved. Such high speedup factors have also been observed in experiments with random graphs of regular hierarchical structure.

7.3 Separators in Planar Graphs

Good node separators of graphs can be characterised by a small set of separating nodes, and a balanced decomposition (i.e., the induced components are of similar size), as we have seen, for example, in our analysis of the multi-level graph approach. For planar graphs, we have investigated modifications of classical linear-time algorithms for determining such good separators: given a planar graph with n nodes, a separator size of $\mathcal{O}(\sqrt{n})$ can be guaranteed, while all induced components are smaller than $2n/3$. Our focus was to compute optimal separators rather than to just fulfil the worst-case bounds. In particular, the fundamental cycle separation turned out to provide excellent separators in practice. In the classical algorithms, these fundamental cycles are used as subprocedures, since better worst-case bounds can be achieved by additional steps before the fundamental cycle separation is applied. However, we have experimentally shown for many graph classes that the direct application of fundamental cycle separation yields the better results, which are close to optimal solutions. Also, the application of post-processing techniques proved to be very useful.

The algorithms rely on a spanning tree which is calculated by a breadth first search initiating from an arbitrary root node. Our experiments have shown that the choice of the root has a high influence on the solution found by the algorithms, especially on the number of nodes in the separator. Since the range of observed separator sizes is quite small for the fundamental cycle separation, it suffices to compute fundamental cycle separators for a small number of randomly chosen root nodes: with high probability a good separator is found, while the running time is still linear in the size of the graph. However, this strategy is a bit unsatisfactory, since no good worst-case bounds can be guaranteed. An interesting issue for future work would be whether a spanning tree can be computed deterministically that always guarantees a good solution.

Bibliography

[ADGM02] Lyudmil Aleksandrov, Hristo Nicolov Djidjev, Hua Guo, and Anil Maheshwari. Partitioning planar graphs with costs and weights. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX 2002)*, volume 2409 of *LNCS*, pages 98–110. Springer, 2002.

[AJ94] R. Agrawal and H. Jagadish. Algorithms for searching massive graphs. *IEEE Transact. Knowledge and Data Eng.*, 6:225–238, 1994.

[AL96] Cleve Ashcraft and Joseph W. H. Liu. Applications of the Dulmage-Mendelsohn decomposition and network flow to graph bisection improvement. Technical Report CS-96-05, Dept. of Computer Science, York University, North York, Ontario, Canada, 1996. http://www.cs.yorku.ca/techreports/1996/CS-96-05.html.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[Bar] Bay Area Regional Database. http://bard.wr.usgs.gov/.

[BJ01] Gerth S. Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. Technical Report ALCOMFT-TR-01-176, BRICS, University of Aarhus, Denmark, 2001. http://www.brics.dk/ALCOM-FT/TR/ALCOMFT-TR-01-176.html.

[BJ04] Gerth S. Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. In *Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2003)*, volume 92 of *Electronic Notes in Theoretical Computer Science*, pages 3–15. Elsevier, 2004. A previous version appeared as [BJ01].

[BJM00] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[Bol85] B. Bolobás. *Random Graphs*. London Academic Press, 1985.

[Bou92] Paul Bourke. Sphere generation, 1992. http://astronomy.swin.edu.au/~pbourke/modelling/sphere/.

[BS88] Norbert Baumann and Richard Schmidt. Buxtehude–Garmisch in 6 Sekunden. Die elektronische Fahrplanauskunft (EFA) der Deutschen Bundesbahn. *Die Bundesbahn. Zeitschrift für aktuelle Verkehrsfragen*, pages 929–931, 10 1988.

[BSWW04] Ulrik Brandes, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Generating node coordinates for shortest-path computations in transportation networks. *Journal of Experimental Algorithmics*, 9(1.1), 2004.

[Buc00] Friedhelm Buchholz. *Hierarchische Graphen zur Wegesuche*. PhD thesis, Fakultät für Informatik der Universität Stuttgart, 2000.

[CF94] A. Car and A. Frank. Modelling a hierarchy of space applied to large road networks. In *Proceedings of the International Workshop on Advanced Information Systems (IGIS 1994)*, volume 884 of *LNCS*, pages 15–24. Springer, 1994.

[CGR96] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill, 2001.

[CZ98] S. Chaudhuri and C. Zaroliagis. Shortest paths in digraphs of small treewidth, Part II: Optimal parallel algorithms. *Theoretical Computer Science*, 203(2):205–223, 1998.

[CZ00] S. Chaudhuri and C. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. *Algorithmica*, 27(3):212–226, 2000.

[DEL] DELFI, Durchgängige elektronische Fahrplaninformation. http://www.delfi.de/.

[Dia69] Robert Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12:632–633, 1969.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dji82] Hristo Nicolov Djidjev. On the problem of partitioning planar graphs. *SIAM Journal on Algebraic and Discrete Methods*, 3(2):229–240, 1982.

[DV97] Hristo Nicolov Djidjev and Shankar M. Venkatesan. Reduced constants for simple cycle graph separation. *Acta Informatica*, 34:231–243, 1997.

[EFA] EFA, a timetable information system by Mentz Datenverarbeitung GmbH, München, Germany. http://www.mentzdv.de/.

[Esr] Environmental Systems Research Institute. http://www.esri.com/.

[EUS] EU-Spirit, European travel information system. http://www.eu-spirit.com/.

[FET98] Lisa Fleischer and Éva Tardos. Efficient continuous-time dynamic network flow algorithms. *Operations Research Letters*, 23:71–80, 1998.

[FF58] L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6:419–433, 1958.

[FF62] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[Fli04] Ingrid C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. PhD thesis, Technische Universiteit Eindhoven, 2004.

[Fre91] G. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38:162–204, 1991.

[Fre95] G. Frederickson. Using cellular graph: Embeddings in solving all pairs shortest path problems. *Journal of Algorithms*, 19:45–85, 1995.

[Fre99] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46:473–501, 1999.

[FS02] Lisa Fleischer and Martin Skutella. The quickest multicommodity flow problem. In *Proceedings of the 9th Conference on Integer Programming and Combinatorial Optimization (IPCO'02)*, volume 2337 of *LNCS*, pages 36–53. Springer, 2002.

[FS03] Lisa Fleischer and Martin Skutella. Minimum cost flows over time without intermediate storage. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 66–75. SIAM, 2003.

[FSST86] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[Gol01a] A. V. Goldberg. Shortest path algorithms: Engineering aspects. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC 2001)*, volume 2223 of *LNCS*, pages 502–513. Springer, 2001.

[Gol01b] Andrew V. Goldberg. A simple shortest path algorithm with linear average time. In *Proceedigns of the 9th European Symposium on Algorithms (ESA 2001)*, volume 2161 of *LNCS*, pages 230–241. Springer, 2001.

[GSVGM98] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity search in databases. In *Proceedings of 24th International Conference on Very Large Data Bases (VLDB 1998)*, pages 26–37. Morgan Kaufmann, 1998.

[HAF] HAFAS, a timetable information system by HaCon Ingenieurgesellschaft mbH, Hannover, Germany.
http://www.hacon.de/hafas/.

[HD88] M. S. Hung and J. Divoky. A computational study of efficient shortest path algorithms. *Computers and Operations Research*, 15:567–576, 1988.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths in graphs. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[Hol03] Martin Holzer. Hierarchical speed-up techniques for shortest-path algorithms. Master's thesis, Fachbereich Informatik und Informationswissenschaft der Universität Konstanz, 2003.
http://www.ub.uni-konstanz.de/kops/volltexte/2003/1038/.

[HPS+04] Martin Holzer, Grigorios Prasinos, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Engineering planar separator algorithms. Technical report, DELIS Project, Universität Paderborn, Germany, 2004.
http://delis.uni-paderborn.de/docs/subproject1/papers2004/.

[HSW04] Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. In *Proceedings Third International Workshop on Experimental and Efficient Algorithms (WEA 2004)*, volume 3059 of *LNCS*, pages 269–284. Springer, 2004.

[IOAI91] K. Ishikawa, M. Ogawa, S. Azume, and T. Ito. Map navigation software of the electro multivision of the '91 Toyota soarer. In *Proceedings of the IEEE International Conference on Vehicle Navigation Information Systems*, pages 463–473, 1991.

[JMN99] R. Jakob, M. Marathe, and K. Nagel. A computational study of routing algorithms for realistic transportation networks. *The ACM Journal of Experimental Algorithmics*, 4, 1999.

[JP96] Sungwon Jung and Sakti Pramanik. HiTi graph model of topographical road maps in navigation systems. In *Proceedings of the 12th International Conference on Data Engineering*, pages 76–84. IEEE, 1996.

[JP02] Sungwon Jung and Sakti Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5), 2002. A previous version appeared as [JP96].

[Kar] George Karypis. MeTiS. http://www-users.cs.umn.edu/~karypis/metis.

[KLS02] Ekkehard Köhler, Katharina Langkau, and Martin Skutella. Time-expanded graphs for flow-dependent transit times. In *Algorithms - ESA '02*, volume 2461 of *LNCS*, pages 599–611. Springer, 2002.

[Koz92] D. Kozen. *The Design and Analysis of Algorithms*. Springer, 1992.

[KPSZ96] D. Kavvadias, G. Pantziou, P. Spirakis, and C. Zaroliagis. Hammock-on-ears decomposition: A technique for the efficient parallel solution of shortest paths and other problems. *Theoretical Computer Science*, 168(1):121–154, 1996.

[Len90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley, 1990.

[Lia92] A. M. Liao. Three priority queue applications revisited. *Algorithmica*, 7:415–427, 1992.

[LR99] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1, 2, and 3*. Springer, 1984.

[Mey01] Ulrich Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 797–806. SIAM, 2001.

[MHS] Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria Pareto search. In *Proceedings of the 4th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2004)*, LNCS. Springer. To appear.

[MHSW02] Matthias Müller-Hannemann, Mathias Schnee, and Karsten Weihe. Getting train timetables into the main storage. In *Proceedings of the 2nd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2002)*, volume 66 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[MHW01] Matthias Müller-Hannemann and Karsten Weihe. Pareto shortest paths is often feasible in practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *LNCS*, pages 185–198. Springer, 2001.

[Möh99] Rolf Möhring. Verteilte Verbindungssuche im öffentlichen Personenverkehr. In Patrick Horster, editor, *Angewandte Mathematik - insbesondere Informatik*, pages 192–220. Vieweg, 1999.

[Nac95] Karl Nachtigal. Time depending shortest-path problems with applications to railway networks. *European Journal of Operations Research*, 83:154–166, 1995.

[NM99] Stefan Näher and Kurt Mehlhorn. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. http://www.algorithmic-solutions.com.

[OR90] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3), 1990.

[OR91] A. Orda and R. Rom. Minimum weight paths in time-dependent networks. *Networks*, 21, 1991.

[Par06] Vilfredo Pareto. *Manuale di economia politica*. 1906. Translation by A. Schwier, Manual of political economy (A.M. Kelley, New York, 1969).

[PS98] S. Pallottino and M. Grazia Scutellà. Shortest path algorithms in transportation models: Classical and innovative aspects. In *Equilibrium and Advanced Transportation Modelling*, chapter 11. Kluwer Academic Publishers, 1998.

[PSWZ04a] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Experimental comparison of shortest path approaches for timetable information. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX 2004)*, pages 88–99. SIAM, 2004.

[PSWZ04b] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Towards realistic modeling of time-table information through the time-dependent approach. In *Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2003)*, volume 92 of *Electronic Notes in Theoretical Computer Science*, pages 85–103. Elsevier, 2004.

[Sch] Schwäbische Eisenbahn. http://de.wikipedia.org/wiki/Schw%C3%A4bische_Eisenbahn.

[Sch00] Frank Schulz. Efficient algorithms for a timetable information system. Technical Report 110, Preprints in Mathematics and Computer Science at University of Konstanz, January 2000. (Diploma thesis.) http://www.inf.uni-konstanz.de/Preprints/.

[SFG97] S. Shekhar, A. Fetterer, and G. Goyal. Materialization trade–offs in hierarchical shortest path algorithms. In *Proceedings of the International Symposium on Large Spatial Databases*, volume 1262 of *LNCS*, pages 94–111. Springer, 1997.

[SKC93] S. Shekhar, A. Kohli, and M. Coyle. Path computation algorithms for advanced traveler information system (ATIS). In *Proceedings of the 9th International Conference on Data Engineering*, pages 31–39. IEEE, 1993.

[ST91] L. Siklóssy and E. Tulp. The space reduction method: A method to reduce the size of search spaces. *Information Processing Letters*, 38(4):187–192, 1991.

[SV86] Robert Sedgewick and Jeffrey Scott Vitter. Shortest paths in Euclidean graphs. *Algorithmica*, 1:31–48, 1986.

[SV87] J. T. Stasko and J. S. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30:234–249, 1987.

[SWN92] J. Shapiro, J. Waxman, and D. Nir. Level graphs and approximate shortest path algorithms. *Network*, 22:691–717, 1992.

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE 1999)*, volume 1668 of *LNCS*, pages 110–123. Springer, 1999.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *Journal of Experimental Algorithmics*, 5(12), 2000.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX 2002)*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

[The95] D. Theune. *Robuste und effiziente Methoden zur Lösung von Wegproblemen*. Teubner, 1995.

[Tho99] Mikkel Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[TS88] Eduard Tulp and Laurent Siklóssy. TRAINS, an active time-table searcher. In *Eighth European Conf. on AI*, pages 170–175, 1988.

[UY91] J. D. Ullman and M. Yannakakis. High probability parallel transitive closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[War87] A. Warburton. Approximation of pareto optima in multiple-opjective, shortest-path problems. *Operations Research*, 34(1):70–79, 1987.

[Wax88] Bernard M. Waxman. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9), 1988.

[WW] Thomas Willhalm and Dorothea Wagner. Shortest path speedup techniques. In *Algorithmic Methods for Railway Optimization*, LNCS. Springer. To appear.

[WW03] Dorothea Wagner and Thomas Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proceedings of the 11th European Symposium on Algorithms (ESA 2003)*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.

[Zie01] M. Ziegelmann. *Constrained Shortest Paths and Related Problems*. PhD thesis, Naturwissenschaftlich-Technischen Fakultät der Universität des Saarlandes, 2001.

[Zwi01] Uri Zwick. Exact and approximate distances in graphs - a survey. In *Proceedings of the 9th Annual European Symposium on Algorithms table of contents (ESA 2001)*, LNCS, pages 33–48. Springer, 2001.

Zusammenfassung

Die elektronische Fahrplanauskunft ist im heutigen öffentlichen Personenverkehr zur Normalität geworden. Was vor einigen Jahren noch mittels Kursbüchern mühsam von Hand gesucht werden musste wird heute im Internet oder am Bahnschalter von Fahrplanauskunftssystemen, wie z.B. das von vielen Bahnunternehmen eingesetzte System Hafas [HAF], erledigt. Typischerweise werden zentrale Server eingesetzt, wobei pro Sekunde hunderte von Anfragen beantwortet werden müssen. Die Datenbasis ist enorm, die Auskunft der Deutschen Bahn etwa deckt den Großteil des europäischen Fernverkehrs sowie weite Teile des deutschen Nahverkehrs ab. Das System muss also fähig sein im Bruchteil einer Sekunde eine optimale Zugverbindung zu ermitteln. In der Praxis werden heutzutage meist heuristische Verfahren eingesetzt, die zwar schnell eine Zugverbindung berechnen, deren Optimalität aber nicht garantiert werden kann.

Mittelpunkt und Motivation dieser Arbeit ist der algorithmische Kern eines Fahrplanauskunftsystems, insbesondere spielen die Modellierung der betrachteten Probleme über kürzeste Wege in geeigneten Graphen, der Entwurf von Algorithmen sowie vor allem deren experimentelle Analyse eine wichtige Rolle. Somit ist die Arbeit hauptsächlich dem Gebiet des Algorithmen-Engineering zuzuordnen. Neben den kommerziellen Fahrplanauskunftsystemen sind nur wenig wissenschaftliche Arbeiten zu diesem Thema vorhanden (siehe etwa [Möh99, Nac95]), insbesondere gibt es kaum experimentelle Untersuchungen.

Erster wichtiger Punkt ist die Festlegung des Fahrplanauskunftproblems, d.h. die formale Definition des Begriffes *optimale Zugverbindung*. Für diverse Varianten und Optimierungskriterien dazu werden Graphmodelle und Algorithmen untersucht, die gemäß der jeweiligen Problemspezifikation garantiert optimale Verbindungen liefern. Es werden zwei Modellierungsansätze verfolgt, die beide das Fahrplanauskunftsproblem in ein kürzeste Wege Problem transformieren. Im ersten Fall wird ein Graph konstruiert, der für jede Abfahrt und jede Ankunft jedes Zuges im Fahrplan einen Knoten enthält. Es wird also gewissermassen im Graphen für jeden Bahnhof die Zeit expandiert, indem für jeden Zeitpunkt, an dem ein Zug abfährt oder ankommt, ein neuer Knoten eingeführt wird. Die Lösung wird dann mittels eines Algorithmus für kürzeste Wege bestimmt, wobei Varianten des Algorithmus von Dijkstra besonders geeignet sind. Im zweiten Modell entspricht der Graph im Prinzip dem zugrundeliegenden Streckennetz (zusammen mit weiteren Kanten, die Direktverbindungen ohne Zwischenhalte modellieren). In diesem Fall wird pro Kante eine ganze Liste von möglichen Abfahrts- und Ankunftszeiten gespeichert, und während des Algorithmus die jeweilige Ankunftszeit berechnet. Dies ist eine spezielle Form eines

zeitabhängigen kürzeste Wege Problems.

Zum Vergleich der beiden grundlegenden Modelle werden zunächst Umstiegszeiten vernachlässigt und das Kernproblem der Fahrplanauskunft betrachtet: Gegeben eine früheste Abfahrtszeit, einen Abfahrts- und einen Zielbahnhof, sucht man eine Verbindung, die baldmöglichst am Ziel ankommt. Anschließend wird untersucht, wie diese Modelle erweitert werden können um alle in der Realität vorkommenden Eigenheiten, wie Umstiegszeiten an Bahnhöfen oder Fußwege, zu integrieren. Des Weiteren werden neben der Fahrzeit andere Optimierungskriterien, wie z.B. die Anzahl der Umstiege, betrachtet. Schließlich sollen mehrere Kriterien gleichzeitig optimiert werden, einerseits durch Festlegen einer Reihenfolge und Bestimmung der lexikographisch ersten Lösung, oder im Allgemeinen durch Bestimmung aller Pareto-optimalen Lösungen.

Wie oben schon erläutert ist, neben der korrekten Modellierung, die mittlere Laufzeit pro Anfrage entscheidend für den Kern eines zentralen Fahrplanauskunftssystems. Die beschriebenen Modelle wurden implementiert, und auf echten Fahrplan- und Anfragedaten der Deutschen Bahn experimentell analysiert. Der zeitabhängige Ansatz ist bei der vereinfachten Problemstellung deutlich schneller. Dieser Vorteil wird aber relativiert, wenn man zu realistischen Problemstellungen übergeht. Die experimentellen Ergebnisse zeigen auch, dass die Basisalgorithmen (Dijkstra's Algorithmus bzw. eine zeitabhängige Variante davon) noch zu langsam für ein reales Auskunftssystem sind. Deswegen werden im Weiteren Techniken untersucht, wie Algorithmen für kürzeste Wege beschleunigt werden können ohne die Optimalität der Lösung zu verletzen.

Der Multi-Level Ansatz zur Beschleunigung von Algorithmen für kürzeste Wege besteht aus zwei Phasen. Zuerst wird in einem (langsamen) Vorverarbeitungsschritt der gegebene Graph hierarchisch durch Wegnahme wichtiger Knoten zerlegt und zwischen diesen mehreren Leveln wichtiger Knoten neue Kanten, die Information über kürzeste Wege tragen, berechnet. Später, wenn ein kürzester Weg berechnet werden soll, wird dieser nicht im Originalgraph gesucht, sondern nur in einigen Komponenten des hierarchisch zerlegten Graphen. Dadurch wird der Suchraum eingeschränkt, der gefundene kürzeste Weg hat jedoch garantiert die gleiche Länge wie ein kürzester Weg im Originalgraph. Das Problem, geeignete hierarchische Separatoren für diese Technik zu bestimmen, wird sowohl theoretisch als auch experimentell ausführlich untersucht.

Für eine Klasse von Graphen, für die „gutartige“ hierarchische Zerlegungen angegeben werden können, kann durch den Multi-Level Ansatz eine Beschleunigung der kürzeste Wege Suche garantiert werden. Wir stellen weiter ein Zufallsgraphmodell vor, mit dem Graphen generiert werden können, die zur oben erwähnte Graphklasse gehören, und sind somit in der Lage, die garantierte Beschleunigung experimentell zu validieren. Danach wird diskutiert, wie sowohl der Multi-Level Ansatz als auch andere Beschleunigungstechniken, die jeweils die Optimalität der Lösung erhalten, für die beschriebenen Modelle zur Fahrplanauskunft eingesetzt werden können, um für die Praxis akzeptable Antwortzeiten zu erhalten. Dabei kommt eine etwas abgewandelte Variante des Multi-Level Ansatz mit heuristisch bestimmten Separatoren zum Einsatz. Einen entscheidenden Punkt hierbei spielt wieder die experimentelle

Analyse der Techniken auf realen Daten. Abschließend soll die Frage der Kombinationsmöglichkeit von Beschleunigungstechniken, angewandt nicht nur auf Graphen aus dem öffentlichen Personenverkehr, sonder auch auf Graphen aus der Routenplanung und auf Zufallsgraphen, geklärt werden. Alle Kombinationen aus dem Multi-Level Ansatz, einer weiteren Vorverarbeitungs-basierten Technik, und den Standardtechniken zielgerichtete und bigerichtete Suche werden betrachtet.

Motiviert durch die im Multi-Level Ansatz benötigten Knotenseparatoren werden wir uns im letzten Kapitel Algorithmen zum Berechnen von Knotenseparatoren widmen. Dabei entfernen wir uns etwas von der Anwendung der Fahrplanauskunft, und beschäftigen uns mit planaren Graphen. (Die Graphen für die Fahrplanauskunft sind in der Regel nicht planar.) Für planare Graphen gibt es Linearzeitalgorithmen zur Berechnung von garantiert kleinen Knotenseparatoren, die gleichzeitig gewährleisten, dass die Komponenten, die durch Wegnahme der Separatorknoten entstehen, ungefähr gleich groß sind. Die Algorithmen basieren auf dem „Planar Separator Theorem“ von Lipton und Tarjan [LT79], wobei eine obere Schranke für die Größe eines Separators garantiert wird. Viele planare Graphen, die auch in der Realität vorkommen (z.B. Graphen zu Straßenkarten), haben jedoch Separatoren, die wesentlich kleiner sind als diese obere Schranke. Uns beschäftigt nun hauptsächlich die Frage, inwiefern die Algorithmen modifiziert werden können, um möglichst optimale Lösungen zu berechnen.

Für eine Vielzahl sowohl generierter planare Graphen also auch für planare Straßenkarten, mit unterschiedlichen Eigenschaften wie Durchmesser und minimale Größe eines möglichen Separators, wurden mit den klassischen und den modifizierten Algorithmen Separatoren bestimmt. Diese Experimente liefern Erkenntnisse wie sich die Algorithmen auf unterschiedlichen Graphen verhalten, und welche Phasen der Algorithmen überhaupt eintreten. Eine Modifikation des klassischen Algorithmus lieferte dabei durchaus erstaunliche Ergebnisse: Beim modifizierten Algorithmus ist die garantierte obere Schranke schlechter als beim klassischen Algorithmus, aber bei (fast) allen Graphen sind die vom modifizierten Algorithmus berechneten Separatoren besser, zum Teil sogar wesentlich besser, als die des klassischen Algorithmus.