Karlsruhe Institute of Technology

Algorithms for Contraction Hierarchies on Public Transit Networks

Diploma Thesis of

Alexander Wirth

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. rer. nat. Peter Sanders
Advisors:	Dipl.-Inform. Julian Dibbelt
	Moritz Baum, M.Sc.

Time Period: December 1st, 2014 – May 31st, 2015

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 31st May 2015

Abstract

Modern applications for route planning in public transit networks require algorithms that can answer earliest arrival queries within milliseconds. This thesis takes a look at the way Contraction Hierarchies can be used as a preprocessing technique for time-dependent networks of various sizes in order to achieve faster query times. We implement and evaluate the algorithms proposed by Dr. Robert Geisberger that make Contraction Hierarchies viable on public transit networks. We then propose alternative algorithms for earliest arrival queries and profile queries that work on the contracted networks and are simpler to implement while also being as fast or faster than previous algorithms. Finally we provide an alternative algorithm for finding witnesses during the construction of the hierarchy and show how it can be extended to allow a combination of time-dependent and time-independent edges. Our results show that our query algorithms perform no worse than previous algorithms and that our contraction techniques are viable on appropriate input networks.

Deutsche Zusammenfassung

Moderne Anwendungen für Routenplanung in öffentlichen Verkehrsnetzen benötigen Algorithmen, die früheste Ankunftszeiten in Millisekunden berechnen können. Diese Diplomarbeit betrachtet wie Contraction Hierarchies in zeitabhängigen Graphen zur Beschleunigung der Berechnung von frühesten Ankunftszeiten verwendet werden können. Wir implementieren die Algorithmen, die von Dr. Robert Geisberger vorgeschlagen wurden, um Contraction Hierarchies auf zeitabhängigen Verkehrsnetzen praktikabel zu machen, und vergleichen sie mit unseren alternativen Algorithmen für früheste Ankunftszeiten und Profilsuchen. Unsere Algorithmen sind einfacher zu implementieren und in der Praxis genau so schnell oder schneller wie bisherige Methoden. Zuletzt stellen wir eine alternative Methode für die Zeugensuche vor, die bei der Konstruktion von Contraction Hierarchies benötigt wird, und zeigen wie sie erweitert werden kann, um eine Mischung aus zeitabhängigen und zeitunabhängigen Kanten im Routengraphen zu unterstützen. Unsere Experimente zeigen, dass unsere Algorithmen das Potential für schnellere Laufzeit haben und unsere Kontraktionsmethoden auf geeigneten Eingabegraphen gute Ergebnisse liefern.

Contents

1. Introduction

In a modern world that grows ever closer together, traveling fast and efficiently plays an important role. So it is no wonder that route planning has been a major topic in algorithm engineering for a long time. The increasing demand for navigation systems for cars made it more important than ever that route planning can be done in real-time. Apart from car travel there is also a demand for route planning in public transit networks, especially since in comparison to road networks it is much more difficult for a person to plan a trip without computer assistance because train schedules can become quite complex and fastest routes can change throughout the day. Smartphones nowadays allow users to plan public transit routes either by using web-based services but also by downloading train schedules ahead of time and then running route planning algorithms locally on their phones. Therefore fast algorithms for public transit networks are also in the focus of route planning research.

1.1 Related Work

Although route planning is an old problem, in recent years it has become the subject of major algorithmic developments. The most basic problem in a road network is the shortest path problem which asks for the shortest possible path from a node X to a node Y. When the road network is viewed as a graph with travel times being used as edge weights, then the shortest path can be computed with Dijkstra's Algorithm [Dij59].

However, Dijkstra's Algorithm quickly becomes infeasible for larger networks and real-time query requirements where queries of more than a second are usually unacceptable and query times of not more than a few milliseconds are desired. Dijkstra's Algorithm still remains at the core of most approaches though, and many techniques have been developed to improve its speed on certain input graphs [Bau06]. These techniques usually augment the graph with some auxiliary data that allows more goal-directed searches and focus on skipping parts of the network that are not required for the current query without sacrificing correctness. Some examples for road networks are bidirectional search, highway-node routing [SS07], transit-node routing [BFSS07, BFM09], arc flags [MSS+06] and combinations of those techniques [BDS+08].

Apart from road networks there are also public transit networks to consider. The main difference between road networks and transportation networks is the complexity of the edges. Whereas road networks have simple constant travel time in most models, public transit networks are inherently time-dependent since trains, buses, etc. all depart at fixed times according to complex schedules and the resulting travel-time functions between two stations are very complex. There are two basic models for public transit routing, the time-expanded model [MW01, PS98] and the time-dependent model [BJ04, Nac95, PSWZ08]. The first one models each departure of a train at a specific station as its own event and node in the graph, which simplifies the graph into one that resembles road networks but at the cost of increasing its size significantly. The time-dependent model on the other hand keeps the complexity of public transit schedules in the travel-time function of the edges. In order to allow the introduction of minimum transfer times in the time-dependent model, Pyrga et al. add additional nodes that enforce transfer times [PSWZ08]. We will call this model the *route node model* throughout this thesis.

In public transit networks we are interested in two types of queries. The first type are earliest arrival queries, also called time queries, which ask for the earliest arrival at a station Y that can be achieved when departing from another station X at a specific time t. The second type are profile queries which ask for earliest arrivals not only for a specific departure time but for a whole range of departure times. Profile queries make sense in public transit networks because earliest arrivals can change throughout the day or when connections have just been missed. Profiles with a range of around an hour are a common use-case for public transit route planning for the end user but in algorithmic research we are typically interested in profiles for a 24 hour period.

Some of the techniques devised for road networks can be applied to public-transit networks as well, others introduce entirely new concepts to handle public transit networks like Round-Based Public Transit Routing [DPW12]. This thesis will look at one specific speed-up technique called Contraction Hierarchies [GSSD08] which were originally introduced for road networks and later also applied to public transit networks [BGS08, BDSV09, Gei09].

Lately multimodal transportation networks have also become a focus of algorithmic research [BBM06, DDP$^+$12]. A multimodal network combines several modes of transportation like car travel, public transit, walking, cycling and using a taxi. In these networks shortest path queries are not very interesting since they are usually dominated by the fastest mode of transportation. Instead there are usually multiple criteria to optimize, for example number of transfers, walking time, taxi fares and so on. The queries then become multi-criteria queries that result in pareto-sets of solutions at the target stop. Usually some filtering is then done to remove solutions that are impractical, unwanted or nonsensical in the real world [BBS13, DDP$^+$13].

Road networks and public transit networks in the real world are already closely related. Journeys do not typically start and end at public transit station so there is also the question of which station to walk to at the beginning of the journey and from which station(s) the goal can be reached. These do not necessarily have to be the closest stations to the start and end locations because stations that are farther away could provide faster trains or a more direct route to the target. It is therefore of interest to combine public transit at least with walking networks.

1.2 Contributions

In this thesis we implement the algorithms for Contraction Hierarchies on public-transit networks with realistic transfers proposed by Geisberger [Gei10] and evaluate them on a larger number of different public transit networks. We then combine them with ideas that we adapted from the Self-Pruning Connection-Setting algorithm by Delling et al. [DKP10] with the goal of improving preprocessing times and runtime of profile queries. We also introduce new ideas for earliest arrival queries that simplify previous algorithms and attempt to trade memory overhead for speed. Lastly we take a look at multimodal networks and attempt to remove some of the restrictions placed on footpaths and extend our algorithms to create Contraction Hierarchies on public-transit networks that are augmented by larger footpath networks. Our focus is primarily on earliest arrivals so our multimodal experiments do not use fully multimodal transportation networks and only combine walking and public transit for now.

1.3 Outline

Chapter 2 describes the model used for public transit network graphs with realistic transfers and shows what data is contained in a timetable, how this data is organized and some of the properties of the timetable and its connections.
Chapter 3 describes algorithms that can be used on the station model to answer earliest arrival queries without applying any preprocessing.
Chapter 4 shows how Contraction Hierarchies work, how they can be used on time-dependent networks and how existing algorithms can be modified and improved upon. The focus for the improvements in this area is the witness search for the contraction of nodes. This chapter also describes how the algorithms from the previous chapter can be adapted for earliest arrival queries and profile queries on the contracted graph as some changes are necessary to maintain correctness.
Chapter 5 will discuss extensions that remove restrictions on footpaths and allow hybrid public-transit/walking networks with earliest arrival queries.
Chapter 6 evaluates all algorithms on a variety of real-world data sets that range from local city traffic to long-distance train networks.
Finally, Chapter 7 completes this thesis by giving another overview of our results and discussing potential areas where future work can be done.

2. Preliminaries

In this chapter we first explain the structure of a public transit network, then we introduce the station model that we use to represent these networks and which is the basis for our algorithms. We then discuss in detail the terms and concepts used in public transit route planning as far as they are required for our algorithms.

A public transit network consists of a number of *stations* which are points where passengers may enter or exit public transit vehicles like trains, buses, trams, subways, ferries and so on. The only requirement for a public transit vehicle is that it operates according to a fixed schedule, the *timetable*, that defines exactly at which time a vehicle departs from a station and at which time it arrives at the next station. The timetable information consists of a number of *trips* where each trip represents a single vehicle as it moves from its starting station visiting a series of interim stations and finally ends at its last station with specific departure and arrival times at each of these stations. The public transit network can also include footpaths that can be used to travel by foot from one station to another station. They can be used at any point in time and have a specific length which is the time that is required to travel between those stations along the footpath.

The journey of a passenger in a public transit network begins at a certain time t at some station X and then consists of waiting at the current station, entering and exiting vehicles as they depart and arrive, or following footpaths to eventually reach some target station Y. In order to make these journeys more realistic there is an additional restriction on changing vehicles at a station: the *transfer time.* Each station has its own transfer time which is the time that a passenger must wait after exiting a vehicle at this station before he may enter any other vehicle departing at this station. The transfer times are meant to account for the time that it takes in the real world to move from one platform to another within a larger train station as well as the uncertainty that is typically introduced by real-world conditions where vehicles may arrive earlier or later than scheduled. The transfer time is therefore meant to act as a buffer that ensures that changing vehicles at a station is possible under typical real-world conditions.

2.1 Station Model with Realistic Transfers

The station model represents a public transit network as a graph $G = (S, E)$ in which every station has its own node $v \in S$. The timetable is modeled by a number of elementary connections where each elementary connection c represents the smallest segment of a trip in the timetable. An elementary connection c operates between two stations $S_{from}(c)$ and $S_{to}(c)$, departing at station $S_{from}(c)$ at time $t_{dep}(c)$ and arriving at station $S_{to}(c)$ at time $t_{arr}(c)$.

An edge $e := (S, T) \in E$ exists when there is at least one elementary connection c with $S_{from}(c) = S$ and $S_{to}(c) = T$. All elementary connections in the timetable belong to their appropriate directed edge $(S_{from}(c), S_{to}(c)) \in E$. There are no duplicate edges, so an edge can be uniquely identified by an ordered pair (S, T) of stations and stores all the connections that belong to it.

Definition 2.1. *A **trip** in the station model is represented as an ordered sequence $(c_1, c_2, ..., c_k)$ of elementary connections so that $\forall 0 \leq i < k : S_{to}(c_i) = S_{from}(c_{i+1})$ and $t_{arr}(c_i) \leq t_{dep}(c_{i+1})$.*

Definition 2.2. *A number of trips belong to the same **route** if they serve the exact same sequence of stations and no two trips pass each other, i.e. there is no pair of elementary connections c_1, c_2 from different trips of the same route with $S_{from}(c_1) = S_{from}(c_2), S_{to}(c_1) = S_{to}(c_2)$ and $t_{dep}(c_1) \leq t_{dep}(c_2) \leq t_{arr}(c_2) < t_{arr}(c_1)$.*

Every elementary connection must belong to exactly one trip, even if that trip consists only of this single connection. In the same way a trip must always belong to exactly one route, even if that route only consists of this single trip. In the real world there are usually many trips on the same route that follow a regular schedule throughout the day. However, especially early in the morning and late at night trips sometimes serve slightly different stations requiring separate routes for these trips to be created.

For every elementary connection c we store the information listed below. Note that $S_{from}(c)$ and $S_{to}(c)$ do not have to be stored explicitly, because they can be implicitly determined from the edge to which the connection belongs. The previous arrival and next departure time can also be determined by looking up the previous and next connections of c's trip, but for the sake of efficiency we require constant-time lookup of this information and thus store it directly with each connection, even if this introduces some duplicate data in our model. An elementary connection therefore stores:

- The departure time $t_{dep}(c)$ at station $S_{from}(c)$.
- The arrival time $t_{arr}(c)$ at station $S_{to}(c)$.
- The trip $Z(c)$ that this elementary connection belongs to.
- The previous arrival $t_{prev}(c)$ of the trip $Z(c)$ at $S_{from}(c)$ which may be $-\infty$ if the trip starts at $S_{from}(c)$.
- The next departure $t_{next}(c)$ of the trip $Z(c)$ at $S_{to}(c)$ which may be ∞ if the trip ends at $S_{to}(c)$.

For each trip we also store separately its route and for a connection c we can therefore look up the route $R(c)$ that it belongs to.

Implementation detail: Note that in the implementation we do not mark ∞ and $-\infty$ with some special flag, but instead choose an integer value η for infinity with the properties that $t_{max} + \text{transfer}_{max} < \eta$ and $\eta + t_{max} + \text{transfer}_{max} < \text{INT\_MAX}$, where transfer_{max} is the maximum transfer time at any station and t_{max} is the maximum time that is allowed to occur during contraction and queries. This allows us to perform arithmetic on these values and simplifies the domination for connections (see 2.4) and other operations by eliminating special cases. With 32-bit integers and all times stored as seconds the limits on t_{max} are still easily several years and thus so large that they will never occur during earliest arrival queries on real-world data.

Definition 2.3. *Every station S also has an associated* ***minimum transfer time*** *τ_S which is the minimum required waiting time when arriving at S with a certain trip before using a connection of a different trip is allowed. That means if we arrive at a station S with a connection c, then we cannot depart from S earlier than $t_{arr}(c) + \tau_S$ unless we continue with the next connection of the trip $Z(c)$ (i.e. stay in the vehicle).*

This is an important difference to earlier approaches like the route node model by Pyrga et al. [PSWZ08] which enforces the transfer time implicitly through the structure of the graph by introducing additional “route nodes” for each station. The route node model is discussed in more detail in Section 3.1.1. In the station model on the other hand we must make sure that our algorithms respect the transfer times at each station but in return we have a simpler graph that is vital for the performance of contraction algorithms.

2.2 Types of Connections

Definition 2.4. *Two (elementary) connections can be* ***linked*** *together to form a new connection, called a* ***shortcut connection****. In order to link connections c_1 and c_2 (in that order) it must be possible to use c_2 immediately after arriving with c_1 without any intermediate steps other than waiting. This means that $S_{to}(c_1) = S_{from}(c_2) =: S$ and either $t_{arr}(c_1) + \tau_S \leq t_{dep}(c_2)$ (meaning we can transfer at S from $Z(c_1)$ to $Z(c_2)$) or $Z(c_1) = Z(c_2)$ (meaning c_1 and c_2 use the same vehicle and we can stay in this vehicle as it passes through station S).*

The shortcut connection c created in this way represents the sequence of elementary connections (c_1, c_2) and it is necessary to store more detailed information about the connection:

- The departure time $t_{dep}(c) = t_{dep}(c_1)$ at station $S_{from}(c) = S_{from}(c_1)$.
- The arrival time $t_{arr}(c) = t_{arr}(c_2)$ at station $S_{to}(c) = S_{to}(c_2)$.
- The trip $Z_{first}(c)$ which is used to depart from $S_{from}(c)$.
- The trip $Z_{last}(c)$ which is used to arrive at $S_{to}(c)$.
- The previous arrival $t_{prev}(c)$ of the trip $Z_{first}(c)$ at $S_{from}(c)$ which may be $-\infty$ if the trip starts at $S_{from}(c)$.
- The next departure $t_{next}(c)$ of the trip $Z_{last}(c)$ at $S_{to}(c)$ which may be ∞ if the trip ends at $S_{to}(c)$.
- The station via(c) $= S_{to}(c_1) = S_{from}(c_2)$ at which c_1 and c_2 meet.

Two shortcut connections or an elementary and a shortcut connection can in turn be linked together again, resulting in shortcut connections that represent longer sequences $(c_1, c_2, ..., c_k)$ of elementary connections. Shortcut connections only exist in the contracted timetable. However, since we do not want to distinguish between elementary connections and shortcut connections, all connections in the timetable keep track of the information listed above. For elementary connections $Z_{first}(c)$ and $Z_{last}(c)$ are the same and via(c) is $\perp$. We show in Section 4.3.2 that for any shortcut connection c knowing via(c) in addition to the rest of the information is sufficient to reconstruct the complete sequence of elementary connections which c represents.

Definition 2.5. *We call a sequence $(c_1, c_2, ..., c_k)$ of elementary connections* ***consistent*** *if they can be linked together, respecting stations, trips and transfer times as described in Definition 2.4.*

Such a sequence can be a shortcut connection, but when we talk about them in other contexts such as a sequence of elementary connections that leads from a station X to a station Y as the answer to an earliest arrival query, we also refer to them as **paths** or simply **connections** and we talk about **consistent paths** and **consistent connections**.

For earliest arrival queries we do not always have to know the details about the departure of a connection. In this case we use **arrival connections**. An arrival connection can be thought of as just the "end" of a regular connection. It only stores information about the arrival (arrival time, last used trip, next departure of this trip) and none about the departure. Arrival connections are used as the labels of time queries, since we are only interested in dominant arrivals at each station. Domination for both arrival connections and shortcut connections are explained in Section 2.4.

2.3 Footpaths

Definition 2.6. *A **footpath** f is a time-independent connection that allows traveling from a station S to a station T with a duration of $t(f)$.*

In our station model each foot path is also associated with an edge $(S, T) \in E$. These footpaths connect stations that are close together, so that walking between them is possible, and since their departure is not time-dependent they can be used at any time.

Our standard contraction algorithm requires that these footpaths must be transitively closed, so that for every pair of footpaths (f_1, f_2) where f_1 leads from station S to station T with a travel time of $t(f_1)$ seconds and f_2 leads from station T to station U with a travel time of $t(f_2)$ seconds, there is also a footpath f_3 in the timetable that leads from station S to station U with a travel time of $t(f_3) \leq t(f_1) + t(f_2)$ seconds. This is required because linking two footpaths together creates a time-independent shortcut which the standard contraction algorithms' witness search cannot handle. In Chapter 5 we lift this restriction and propose an algorithm to find witnesses for footpath shortcuts.

Note that linking a footpath to a normal (time-dependent) connection is possible and results in a regular time-dependent shortcut connection. These shortcut connections are not ambiguous since it is always optimal to leave as early as possible when a footpath is linked *after* a connection and it is always optimal to leave as late as possible when a footpath is linked *before* a connection. In these cases we use a special trip ID to mark the first or last trip of the resulting shortcut connection as a footpath.

Regarding transfer times at each station there are several approaches for footpaths that can be taken. We could allow transfer between public transit and footpaths without waiting for the respective stations transfer time but intuitively it makes more sense to apply the transfer time at least either when entering a footpath or when changing from a footpath back into a public transit vehicle. Otherwise there might be stations where it is possible to circumvent the transfer time by using a footpath to another station and then immediately coming back. We can also require that the transfer time is respected on *both* ends of a footpath which also makes sense in a real life scenario as time is required to walk around the stations at both ends.

Since requiring transfer times at both ends has some nice properties that we can later exploit in our algorithms, we chose this approach. That means for a footpath between stations S and T, we must wait τ_S after arriving by public transit at S before the footpath may be taken and we must wait τ_T at T before we can enter another public transit vehicle. Later, when we remove the requirement for footpaths to be transitively closed, it is possible to use several footpaths in succession in which case no transfer time between two footpaths is required. This means the footpath ID can be used like a regular trip ID allowing two connections to link if the arrival and departure use the same trip regardless of whether this is a public transit trip or a footpath which simplifies the implementation.

2.4 Domination

In our algorithms we sometimes need to decide whether one connection *dominates* another. Domination applies to both paths/shortcut connections as well as arrival connections and we will look at them separately, although they basically work the same way.

2.4.1 Arrival Connections

Definition 2.7. *An arrival connection A at a station S dominates an arrival connection B at the same station if any consistent path that can be linked to B can also be linked to A.*

This means that two conditions must be fulfilled: (i) A arrives not later than B and (ii) it is possible to transfer from A's trip to B's trip at S or B's trip ends at S. More formally A dominates B iff both:

- (i) $t_{arr}(A) \leq t_{arr}(B)$
- (ii) $t_{arr}(A) + \tau_S \leq t_{next}(B)$ **or** $Z(A) = Z(B)$

In condition (ii) if the trips are the same, then a transfer from A's trip to B's trip is not necessary. Also note that (ii) holds if $Z(B)$ ends at S, because we have defined $t_{next}(B) := \infty$ in this case. In [Gei10] condition (ii) is also fulfilled when B does not have a *critical arrival*, with a critical arrival defined as $t_{arr}(c) + \tau_S > t_{next}(c)$. However, in this case we can see that

$$t_{arr}(A) + \tau_S \overset{(i)}{\leq} t_{arr}(B) + \tau_S \overset{\text{B non-critical}}{\leq} t_{next}(B) \tag{2.1}$$

and thus (ii) holds anyway, which allows us to drop that condition altogether.

2.4.2 Shortcut Connections

Definition 2.8. *A (shortcut) connection A dominates a (shortcut) connection B if for any path P that contains B we can substitute B with A to create a consistent path P' that does not depart earlier or arrive later than P.*

That means of course that A and B start at the same station S and end at the same station T although they need not share the same stations in between. It also means that it must be possible to transfer from B's trip to A's trip at $S_{from}(B)$ since we might arrive with $Z(B)$ if B is not at the beginning of P and likewise it must be possible to transfer from A's trip to B's trip at $S_{to}(B)$. We can formalize this into four conditions that must be fulfilled. A dominates B iff:

- (i) $t_{arr}(A) \leq t_{arr}(B)$
- (ii) $t_{arr}(A) + \tau_T \leq t_{next}(B)$ **or** $Z_{last}(A) = Z_{last}(B)$
- (iii) $t_{dep}(B) \leq t_{dep}(A)$
- (iv) $t_{prev}(B) + \tau_S \leq t_{dep}(A)$ **or** $Z_{first}(A) = Z_{first}(B)$

If conditions (i) and (ii) are fulfilled, we call A *dominant at arrival* with regard to B and if (iii) and (iv) are fulfilled, we call A *dominant at departure* with regard to B. Note that the first two conditions are the same as those for arrival connections, since arrival connections behave exactly like shortcut connections but without any specific departure.

Lemma 2.9. *Domination on shortcut connections and arrival connections is transitive, i.e.* $(A \text{ dominates } B) \wedge (B \text{ dominates } C) \Rightarrow (A \text{ dominates } C)$.

Proof. We check all four conditions to see if A also dominates C:

- (i) $t_{arr}(A) \leq t_{arr}(B) \leq t_{arr}(C)$ ✓
- (ii) If $Z_{last}(B) = Z_{last}(C)$ then also $t_{next}(B) = t_{next}(C)$ and therefore $t_{arr}(A) + \tau_S \leq t_{next}(B) = t_{next}(C)$ or $Z_{last}(A) = Z_{last}(B) = Z_{last}(C)$. Otherwise $t_{arr}(A) + \tau_S \overset{(i)}{\leq} t_{arr}(B) + \tau_S \leq t_{next}(C)$ ✓
- (iii) $t_{dep}(C) \leq t_{dep}(B) \leq t_{dep}(A)$ ✓
- (iv) If $Z_{first}(B) = Z_{first}(C)$ then also $t_{prev}(B) = t_{prev}(C)$ and therefore $t_{prev}(C) + \tau_S = t_{prev}(B) + \tau_S \leq t_{dep}(A)$ or $Z_{first}(A) = Z_{first}(B) = Z_{first}(C)$. Otherwise $t_{prev}(C) + \tau_S \leq t_{dep}(B) \overset{(iii)}{\leq} t_{dep}(A)$. And lastly if $Z_{first}(A) = \text{Footpath}$ this condition is trivially fulfilled. ✓

For arrival connections the first two conditions are sufficient. □

2.5 Periodic Timetables

The timetables available to us often only cover a single day of public transit schedules. Therefore it was necessary to make the timetable periodic, i.e. a connection with departure time $t_{dep}(c)$ can also be taken at any point in time $t_{dep}(c) + k * \pi$ with $k \in \mathbb{N}_0$, where π is the *period* of our timetable. The period is typically one day, but could also be chosen as a week or month depending on the train schedule information that is available. However, in order to model trains which only operate on certain days of the week or special holidays – or trains that do *not* operate on those days – it is a good idea to store information about the *days of operation* [MS09] (also called *traffic days*) for each trip, using bitflags to restrict for which k in the above formula a connection may be used. This way it is possible to model the complex details of real world schedules while restricting the actual departure times in our timetable to $0 \leq t_{dep}(c) < \pi$.

In all of our experiments we use a periodic timetable with $\pi = 1$ day $= 86400$ seconds with 1 second being the smallest unit of time in the timetable and all departure times being stored as seconds after midnight. Departure times are restricted to $[0, 86399]$ but arrival, next departure and previous arrival times remain relative to the departure and can fall outside of this range. We do not use days of operation because this information was not available to us. Adding days of operation to the earliest arrival algorithms is easily possible with only minor adjustments to linking and domination of connections. Combining Contraction Hierarchies with days of operation can be more complicated as witnesses for any possible combination and will lower the quality of the resulting contracted networks.

2.6 Problem Descriptions

There are two typical problems for which we provide algorithms in this thesis. Both are related only to earliest arrival times and the appropriate consistent paths.

Definition 2.10. *The **Earliest Arrival Problem** (EAP) gives a source station X, a target station Y and a departure time t and asks for the earliest possible arrival time at Y when departing from X at time t.*

The earliest arrival problem is the most fundamental problem in all of route planning. Typically it is also required that the consistent path from X to Y that leads to the earliest arrival time is part of the answer or can be constructed from the data that was collected while solving the EAP.

Definition 2.11. *The **Profile Problem** gives a source station X, a target station Y and a time interval $[t_0, t_1]$ and asks for the earliest arrival times for* all *departure times $t \in [t_0, t_1]$ at station X.*

The interval can be anything from a single point in time to the whole period of the timetable ($[0, \pi)$). Since in a timetable network the number of possible departure events from X is finite, the answer for the Profile Problem can be given as a number of (departure, arrival) pairs, so that the earliest arrival for any $t \in [t_0, t_1]$ can be found by looking at the pair with the smallest departure $\geq t$. For the Profile Problem – same as for the EAP – it is also often desirable that the consistent path for every (departure, arrival) pair can be determined.

3. Algorithms for Earliest Arrival Queries

In this chapter we will discuss algorithms that solve the earliest arrival problem for regular public transit networks without any preprocessing techniques. First we give a quick overview of Dijkstra's algorithm on the route node model and the connection scan algorithm by Dibbelt et al. [DPSW13] which we use as a baseline comparison for our own algorithms. Then we introduce the algorithm used by Geisberger [Gei10] which operates on the station model and which can also be used with a few modifications when Contraction Hierarchies for the network have been constructed to improve query times. Lastly we propose a new version of Geisberger's algorithm that is similar in complexity but simpler to implement and has the potential for faster query times.

3.1 Baseline Algorithms

We implement two algorithms for the earliest arrival problem that we can compare our results to and that can be used to verify the correctness of solutions produced by our own algorithms. The first one is a regular Dijkstra on the time-dependent model as defined by Pyrga et al. [PSWZ08] which we will call route node model. The second one is a slightly modified version of the Connection Scan Algorithm by Dibbelt et al. [DPSW13] that does not use a graph-based representation of the timetable.

3.1.1 Dijkstra's Algorithm on the Route Node Model

Pyrga et al. introduced a time-dependent model for realistic transfers [PSWZ08] that uses more than one node per station. In order to model the minimum transfer time at a station each station is split into one station node and several route nodes – one for every route that passes through this station. All trips depart and arrive at the appropriate route node to which their trip belongs instead of the station node itself. In addition there are footpaths from each route node to the station node with length 0 and footpaths from the station node to each of its route nodes with length τ_S. Figure 3.1 shows an example of the structure of the route node model compared to the station model.

Dijkstra's algorithm [Dij59] on this graph answers earliest arrival queries between two station nodes. Note that for a query departing at time t_d the actual departure time for Dijkstra's algorithm must be set to $t_d - \tau_S$ to allow the initial transfer from the starting station node to the first route node without losing any time. Minimum transfer times per station are enforced because in order to change from route R_1 to another route R_2 at a

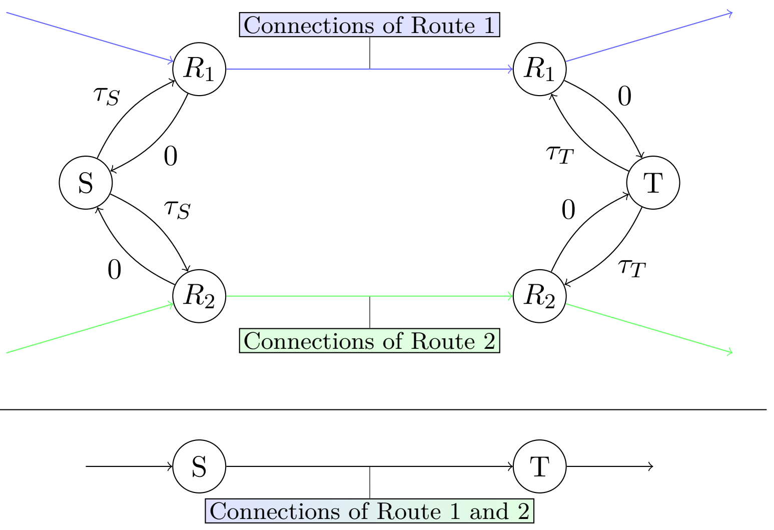


Figure 3.1: A comparison between the route node model (top) that uses additional route nodes per station to enforce minimum transfer times, and the station model that only has one node per station with only one (directed) edge from S to T. Transfer times in the station model are enforced by the algorithm instead.

station S one must first use a 0-length edge from the route node of R_1 to the station node of S and then an edge of length τ_S from the station node to the route node of R_2.

The difference to our algorithms is that in the route node model the transfer time at each station is enforced through the structure of the underlying graph itself instead of the algorithm as it is done in the station model. However, having more nodes leads to higher memory requirements and more priority queue accesses per query, negatively impacting performance. In addition the higher number of nodes makes Contraction Hierarchies infeasible on this model; see Chapter 4 for more details.

Dijkstra's algorithm is easy to implement but is also a lot slower than all other solutions due to the large amount of priority queue operations on the route node graph. There are ways to improve this model to use fewer route nodes [DKP12] but since we only use it for baseline comparison we only tested an unoptimized implementation.

3.1.2 Connection Scan Algorithm

The second comparison algorithm we implement is a version of the Connection Scan Algorithm (CSA) [DPSW13] that works on periodic timetables. CSA is simple to implement and quite fast for smaller timetables and networks where the average path between two nodes tends to be short. It is also easy to extend to problems other than earliest arrival queries.

The main idea behind CSA is not to represent the timetable as a graph at all. Instead it keeps just one large list $\mathbb{C}$ of all connections in the timetable sorted by their departure time $t_{dep}(c)$. Then for a query departing at time t_0 from source station X to target station Y an array of earliest arrival times is initialized as ∞ for all stations except X where the earliest arrival will be set to $t_0 - \tau_X$. A binary search finds the first connection $c_0 \in \mathbb{C}$ that departs not later than t_0. CSA then linearly scans through $\mathbb{C}$starting from c_0 and checks for every connection whether it can be used. A connection can be used when either $t_{min}(S_{from}(c)) + \tau_{S_{from}(c)} \leq t_{dep}(c)$ (where $t_{min}(S)$ is the tentative earliest arrival at S), meaning that the minimum transfer time is respected and we can transfer from whatever trip was used to reach $S_{from}(c)$ to c, or when the connection has been marked as being usable because it is possible to arrive with $Z_{first}(c)$ at $S_{from}(c)$.

CSA as described in [DPSW13] uses one flag per trip which is set to true whenever any connection belonging to the trip is used. All other connections of this trip may then be used without transfer. This is allowed because the connections are scanned in increasing order of departure which means that any subsequently encountered connection of a marked trip must be a later of the trip than the connection which triggered marking the trip and thus can be used without transfer time. However, since we use periodic timetables this is no longer correct. For long trips, like they can appear on the European level, we might encounter an earlier connection of the trip after the periodic wrap. To correct this we keep one flag per connection instead and every connection sets the flag of the next connection of the trip when it is used. This requires slightly more memory but restores correctness on periodic timetables.

3.2 Time Queries

In this section we explain two algorithms for earliest arrival queries – which we also call *time queries* – on the station model. The first one was proposed by Geisberger [Gei10] and uses multiple labels per station, one for every dominant arrival connection. The second one is our attempt to provide an alternative version of the algorithm that makes a memory-for-speed trade-off by using one label per route+station combination. Since Geisberger calls this algorithm simply "Time Query" we will introduce a new naming convention to make it easier to distinguish between the two algorithms. We will call Geisberger's algorithm Arrival Label Time Query (ALTQ) and our algorithm Event Label Time Query (ELTQ). With a few modifications both queries can also be used on the contracted timetable (see 4.3). Section 3.3 discusses implementation details and techniques that improve the average runtime.

3.2.1 Arrival Label Time Query / ALTQ

Instead of keeping only an earliest arrival time per station the idea behind the Arrival Label Time Query is to use a bag of arrival connections per station. A bag contains a label for every arrival connection at that station that might still be relevant for the earliest arrival at other stations. This uses the domination property (see 2.4) on arrival connections because an arrival connection B that is dominated by another arrival connection A at the same station does not contribute anything to an earliest arrival at some other station since any connection linked after B could also be linked after A. This means that the bags need to keep only non-dominated arrival connections.

Algorithm 3.1 shows the algorithm in pseudocode. A priority queue keeps stations that have already been reached with their earliest arrival as key. Every round the station with the smallest arrival time in the priority queue is expanded. When a station S is expanded, all edges (S, T) are considered and any connection that can be linked after an arrival connection in S's bag will produce a new arrival connection label that is merged with the labels in the bag of T. Afterwards in a domination step all dominated labels are removed and T is inserted into the priority queue as long as any new label was inserted into T's bag and not immediately dominated. When the target stop is expanded the earliest arrival in its bag is the answer for the time query. Note that for the source station a special label $\perp$ is introduced to which any connection departing later than t_d can link without regarding minimum transfer time.

The LINK and MERGE operations are most important for the performance of the algorithm. A naive implementation for either of them requires $O(n^2)$ operations. This can be improved to almost linear time on real-world data [Gei10] and on top of that LINK and MERGE can be combined into a single operation. We discuss an efficient implementation of link and merge in Sections 3.3.4 and 3.3.5.

Algorithm 3.1: Arrival Label Time Query

```
Input: Timetable $G = (V, E)$, Station X (Source) & Y (Target), Departure time $t_d$
Data: Priority queue Q, Array ac of bags containing dominant arrival connections
Output: Earliest arrival $t_a$ at station Y for a path departing from station X at $t_d$

// Initialization
1  forall $S \in V$ do  ac($S$) ← ∅
// Special label for the source station
2  ac(X).INSERT(⊥)
3  Q.INSERT(X, $t_d$)
4  while Q is not empty do
5      ($S, t$) ← Q.DELETEMIN()
6      if $S$ = Y then  return $t$
7      forall $e := (S, T) \in E$ do
           // Link possible connections and remove dominated ones
8          $N$ := MERGE(ac($S$).LINK($e.connections$), ac($T$))
           // If bag at T improved, insert or update T in Q
9          if $N \neq$ ac($T$) then
10             ac($T$) ← $N$
11             $t_{min} \leftarrow \min_{p \in N} t_{arr}(p)$
12             Q.INSERTORDECREASEKEY($T, t_{min}$)
13 return ∞
```

3.2.2 Event Label Time Query / ELTQ

ELTQ is a different approach to earliest arrival queries that draws inspiration from the way CSA handles transfer times. The idea is to use a separate array to mark which connections may be used without transfer in order to eliminate the need to store multiple arrival connections per station. The important observation is this:

Lemma 3.1. *When a connection c that arrives at station $S_{to}(c)$ at arrival time $t_{arr}(c)$ with a trip belonging to route $R_{last}(c)$ was used during the query, then any subsequently encountered connection c' with $S_{to}(c) = S_{from}(c'), R_{last}(c) = R_{first}(c')$ and $t_{arr}(c) \leq t_{dep}(c')$ can be used to depart from $S_{to}(c)$ even if the tentative earliest arrival $t_{min}(S_{to}(c))$ does not allow transfer to c'.*

Proof. The path from the source station to $S_{to}(c)$ must be consistent and can be represented as a series $(e_1, e_2, ..., e_k)$ of elementary connections and $R(e_k) = R_{last}(c) =: r$. That means either

$$\exists j := \max_{1<i\leq k} i : R(e_{i-1}) \neq r, R(e_i) = r$$

or

$$\forall 1 \leq i \leq k : R(e_i) = r.$$

In the former case since c is consistent the transfer time at $S_{from}(e_j)$ was respected, that is $t_{arr}(e_{j-1}) + \tau_{S_{from}(e_j)} \leq t_{dep}(e_j)$, and since trips of a route never pass each other we can board any trip of the same route departing later than $Z(e_j)$ at $S_{from}(e_j)$, especially the trip $Z_{first}(c')$ which must depart later than $Z(e_j)$ because – once again – trips of a route never pass each other. Therefore we can construct a consistent sequence $(e_1, ..., e_{j-1}, e'_j, ..., e'_k)$ which arrives at $S_{from}(c')$ with trip $Z_{first}(c')$ and can link c' to this without regard to transfer time. In the latter case we can achieve the same by simply waiting for the correct trip at station $S_{from}(e_1)$. □

Algorithm 3.2: Event Label Time Query

```
Input: Timetable G = (V, E), Stations X (Source) & Y (Target), Departure time $t_d$
Data: Priority queue Q, Earliest arrival $t_{min}$ per station, Earliest arr. $t_{ev}$ per event
Output: Earliest arrival $t_a$ at station Y for a path departing from station X at $t_d$

   // Initialization
1  forall $S \in V$ do $t_{min}(S) \leftarrow \infty$
2  forall $i \in Events$ do $t_{ev}(i) \leftarrow \infty$
   // Mark departure of the query
3  $t_{min}(X) \leftarrow t_d$
4  Q.insert(X, $t_d$)
5  while Q is not empty do
6      $(S, t) \leftarrow$ Q.deleteMin()
7      if $S = Y$ then return $t$
       // Earliest departure after transfer; special for X
8      edt $\leftarrow t_{min}(S) + \tau_S$ if $S \neq X$ else $t_{min}(S)$
9      forall Connections $c \in e := (S, T) \in E$ do
           // Expand the connection if using it is possible
10         if $edt \leq t_{dep}(c)$ or $t_{ev}(ev_{first}(c)) \leq t_{dep}(c)$ then
11             $t_{min}(T) \leftarrow t_{arr}(c)$
12             $t_{ev}(ev_{last}(c)) \leftarrow t_{arr}(c)$
           // If any value improved, T needs to be updated
13         if $t_{min}(T)$ or any $t_{ev}(i)$ improved then
14             Q.insertOrDecreaseKey($T, t_{min}(T)$)
15 return $\infty$
```

Definition 3.2. *We call a route passing through a station an* ***event****. More specifically when a route R services the sequence of stations $(S_1, S_2, ..., S_k)$ then we introduce events identified as $(R, i)\ \forall\, 1 < i < k$ and associate each event (R, i) with station S_i.*

Note that there can be multiple events per route at the same station if the route loops around and passes a station more than once. Connections no longer store their first and last trips but instead the first and last applicable *event*. During the search we remember the tentative earliest arrival $t_{min}(S)$ per station and for every event the tentative earliest arrival of any connection arriving with this event. For any connection we have to check only two things: (a) can we use the connection because the transfer time is respected $(t_{min}(S) + \tau_S \leq t_{dep}(c))$ or (b) can we use the connection because the departure event is marked with a smaller arrival time $(t(ev_{first}(c)) \leq t_{dep}(c))$ according to Lemma 3.1.

This leads to Algorithm 3.2 which is simpler to implement than ALTQ because no complex link and merge operations are required but the simplicity comes at the expense of higher memory requirements for the event label array. This can be a cause of slowdowns due to cache misses. Essentially ELTQ simulates the route nodes of the route node model by Pyrga et al. [PSWZ08] but without actually introducing new nodes. This reduces priority queue operations and size of the graph. A comparison between ALTQ and ELTQ performance can be found in Chapter 6.

3.3 Implementation Details

In this section we will discuss some of the most important techniques to improve the speed of ALTQ and ELTQ. Frequent profiling revealed several areas where large speed gains where possible. It also helped avoid unnecessary optimization in areas that do not contribute much to the total runtime.

3.3.1 Edge expansion

The most obvious start to improve speed is the expansion of an edge $e := (S, T)$. There can be several hundred connections attached to an edge but usually only a small amount of connections produce labels that are not immediately dominated. To take advantage of this we order the connections of e by departure first and arrival second. When expanding e we know the earliest time any connection may leave from S, which is $t_{min}(S)$, so a binary search will find the first connection with $t_{dep}(c) \geq t_{min}(S)$ or rather $t_{dep}(c) \geq (t_{min}(S) \bmod \pi)$ since the timetable is periodic. From this starting point we scan linearly through the connections. We can immediately ignore connections where $t_{arr}(c) \geq t_{min}(T)$ and $t_{next}(c) \geq t_{min}(T) + \tau_T$ since they will be dominated anyway. To gain the most advantage from this, newly linked connections should immediately update the tentative $t_{min}(T)$.

There is also only a small window in which we actually have to check if the connection can be linked, which is $t_{dep}(c) \in [t_{min}(S), t_{min}(S) + \tau_S)$. For these connection we have to check all arrival connection labels in the bag of S for one that arrived with the correct trip to allow linking without transfer. Every connection departing later than that can be linked by transferring from the earliest arrival connection so only the earliest arrival time at S is relevant.

Lastly we can stop linking connections when $t_{dep}(c) \geq t_{min}(T) + \tau_T$ since this connection and later connections cannot improve the solution. Since for edges with long connections this could still be a lot, we can further improve this by keeping track of the shortest length among the connections of each edge. That way there can be no more improvement as soon as $t_{dep}(c) + shortest(S, T) \geq t_{min}(w) + \tau_w$. A technique that [DPSW13] implements is to pre-process the *domination range* for every connection on e, that is the maximum number of connections that must be checked when starting from a certain connection until no more improvement is possible. Our implementation does not include this because the lower bound on connection length already does a pretty good job and since this preprocessing is not possible during the contraction where edges change frequently.

3.3.2 Interpolation Search

Profiling revealed that the binary search to find the first connection that must be checked makes up a large portion of the algorithm's runtime. This happens because accessing elements spread over a large portion of the array of connections will inevitably incur frequent cache misses. However, we can take advantage of two pieces of knowledge about our timetable: (1) The departure times are limited to $[0, \pi)$ since our timetable is periodic and (2) The departure times for real-world timetables are usually more or less uniformly distributed during the day and thin out during the night.

This allows us to use a variation on Interpolation Search [PIA78] to avoid jumping around the array in most cases. If the departure times for an edge with n connections were perfectly uniformly distributed then the first connection departing not earlier than t would be found at index $i := n * t/\pi$. Regular interpolation search would start at this index and then recursively perform interpolation search on the remaining interval. However, since our data is not actually uniformly distributed and night time distribution can vary between edges we found the biggest speed gain was achieved by checking index i as defined above first, then checking an index that is $(0.1 * n)$ larger/smaller than i (depending on the departure time found at i) and then performing regular binary search on the remaining interval. In a lot of cases the connection that we are looking for will fall between the first two indices and thus we reduce the number of cache misses significantly.

In our experiments Interpolation Search improved the overall runtime on our development hardware by up to 10% and was up to 25% faster than a regular binary search on the whole list of connections. This is only true for hardware with smaller caches though. On our primary hardware used for testing the caches were large enough that Interpolation Search brought no measurable benefit. However, the algorithm could also be implemented for devices like smartphones where cache-efficiency can make more of a difference.

3.3.3 Lazy Resetting

Another area which made up a large portion of the query time, not only for ALTQ and ELTQ but also our CSA implementation, is the preparation before each new query. The data structures used need to be reset to their default values. This includes the priority queue and bags for ALTQ, the priority queue and the event flags for ELTQ and the connection flags for CSA. However, resetting all of this for each query is wasteful since short queries will only access a fraction of this information.

We achieved speed improvements of up to 20% by performing lazy resets. That means we generate a unique run ID for each query and use it to keep track of whether a particular entry in our data structures has been reset during this run. This check is performed before each access and if the entry has not been reset then it is reset before its first access and then marked with the run ID. We cannot do this for the priority queue but it works well for the remaining data structures.

3.3.4 Linking

Linking normally involves two sets of connections: Given the arrival connections at a station S and all connections leaving S along an edge (S, T) we want to link all possible combinations of the two sets to form new arrival connections for station T. When implementing the improvements from the section on edge expansion (Section 3.3.1) however, there is not much complexity left in the linking operation. By pruning as many connections from (S, T) as possible by using the tentative earliest arrival at T, scanning linearly through the connections and checking trips only in the small window where it is necessary, we are already as efficient as we can be.

Linking will get more complicated when it is used in the witness search for our contraction algorithm. The details can be found in Section 4.2.1.

3.3.5 Merging

ALTQ requires frequent merging of two sets of arrival connections. When expanding an edge $e := (S, T)$ the new arrival connections created by linking connections from e to arrival connections at S need to be merged with arrival connections that already existed at T. The resulting set should be sorted, contain only dominant arrival connections and we need to report if anything in T's bag of arrival connections changed, because otherwise T does not need to be inserted into the priority queue for another update.

We call the bag holding the newly created labels *edgeBag* and the bag into which we merge *bag*(T). Before a merge operation begins we ensure that the labels in the *edgeBag* are sorted, contain no duplicates and there are no labels that are dominated by another label from this bag. The sorting criteria for arrival connections is the arrival time followed by the next departure and finally the trip ID. The labels are sorted using a standard sorting algorithm which will automatically place duplicate labels next to each other. There are only two ways in which an arrival connection in a bag can be dominated. Either there is a duplicate of the arrival connection – in which case they dominate each other and we want to keep only one of them – or it is dominated solely by arrival time – in which case we only need to check (*i*) and (*ii*) from Section 2.4 against the arrival time of the first label in the sorted bag. That means once the bag is sorted in $O(nlogn)$ we can perform one scan over all labels and remove in $O(n)$ all dominated labels and all duplicates.

The labels in $bag(T)$ will already be sorted, contain no duplicates and no dominated labels since this is the state in which all bags are left after the expansion of a node. The merging procedure then works as follows:

1. Check the trivial cases: If $bag(T)$ is empty we just insert all labels from $edgeBag$ and report that $bag(T)$ changed. If $edgeBag$ is empty we do not need to do anything.
2. Use the arrival time of the first label in $bag(T)$ to remove all labels that are dominated by it from $edgeBag$ in a linear scan.
3. Remove duplicates of labels in $bag(T)$ from $edgeBag$. Since both are sorted this can be done in linear time by scanning simultaneously over both sets, always incrementing the pointer of the smaller arrival connection.
4. If after (3) $edgeBag$ is now empty then stop here and report back that $bag(T)$ did not change.
5. If the earliest arrival of the first label in $edgeBag$ is earlier than that of the first label from $bag(T)$ then in another linear scan remove all labels in $bag(T)$ that are dominated by the first label of $edgeBag$.
6. Merge the labels from $edgeBag$ and $bag(T)$ into a sorted order, which can be done in linear time since both are already sorted, and report that $bag(T)$ changed.

Runtime analysis: All steps can be performed in linear time and therefore the whole merge is possible in $O(n)$ operations in the number of labels.

3.3.6 Reducing Necessary Events

ELTQ performance is largely affected by cache misses due to the large event label array and can be improved by keeping the number of events low. Not all events are actually required. An event for which all applicable trips have $t_{arr}(c) + \tau_{S_{to}(c)} \leq t_{next}(c)$ is not needed, since the next connection of these trips can never be used without automatically respecting the transfer time anyway. We also introduce two "catch-all" events, one of which is always ∞ and will only be read and the other will only be written. These are used in place of events that were removed through the criterium above and for route starts and ends to avoid handling special cases.

Using the conflict graph and coloring approach from [DKP12] the necessary events can be further reduced by assigning the same event to multiple routes when it can be guaranteed that they will never conflict. This requires that for any connection c_1 with arrival time $t_{arr}(c_1)$ any connection c_2 of a different trip sharing the event must have $t_{dep}(c_2) >= t_{arr}(c_1) + \tau_S$.

3.4 Complexity Comparison

ALTQ and ELTQ both operate on the station model are are very similar in the way they work. Both keep expanding the station node with the smallest arrival time among all nodes until the target node is expanded. That means they perform the same amount of delete minimum operations on the priority queue as well as the same amount of inserts and updates since they both use the same domination and pruning rules. Smarter edge expansion, lazy resetting and Interpolation Search for the first connection are all applicable to both algorithms.

The main advantage of ELTQ over ALTQ is that no sorting and merging of bags is necessary. The main disadvantage of ELTQ is that a larger amount of space is required. On hardware with large caches and timetables with a low number of events we therefore expect ELTQ to perform slightly better than ALTQ. We also expect ELTQ to perform better when the transfer times at most stations is high because with higher transfer times more non-dominated arrival connections remain for each stop which causes the sorting and merging to take longer.

A formal runtime analysis in terms of Landau Notation is of little value for these algorithms since the number of operations depends heavily on the structure of the underlying timetable and worst case examples can easily be constructed to slow down queries. Consider for example the case of two stations S and T with the same transfer time τ that have both been reached with the same earliest arrival time t_{min}. If there are a large number of trips from S to T and vice versa that both depart and arrive within $[t_{min}, t_{min} + \tau]$ then these stops will be expanded over and over again as each of these trips creates new non-dominated labels. This causes a quadratic number of linking checks and $O(|\mathbb{C}|)$ priority queue operations where we normally expect them to be almost linear in $|V|$. These worst case examples are far removed from reality however, and the actual performance should be measured on different real-world timetables as we do in Chapter 6.

4. Contraction Hierarchies

Contraction Hierarchies are a preprocessing technique that was originally developed for road networks [Gei08]. They work by defining a total order on the nodes of the network and then inserting additional edges so that the correct answer for an earliest arrival query can be found by searching forward from the source node using only edges to nodes of higher order and backward from the target node using only edges that come from nodes of higher order. There might be multiple points where the forward and backward search meet and can be joined together; the shortest path is guaranteed to be among those.

Applying Contraction Hierarchies to time-dependent networks introduces some challenges and makes the method more complex than for time-independent road networks. In Section 4.1 we formally describe how the contraction of a graph is defined and describe an algorithm to contract the station model graph. In Section 4.2 we then take a closer look at the witness search which is where we introduce ideas from [DKP12] to improve upon the algorithm by Geisberger [Gei10]. Finally in Sections 4.3 and 4.4 we discuss time queries and profile queries on the contracted network.

4.1 Contraction Algorithm

A graph $G = (V, E)$ is contracted by repeatedly contracting a single node $v \in V$ by removing it from G and adding new edges called shortcut edges which preserve the correctness of earliest arrival queries in $G' := (V \setminus \{v\}, E')$. When the nodes are contracted in the order $v_1, v_2, ..., v_{|V|}$ we use the notation $G_0 := G = (V, E) =: (V_0, E_0)$ for the original, uncontracted graph and $G_1, G_2, ..., G_{|V|}$ where $G_i = (V_i, E_i)$ with $V_i := V_{i-1} \setminus \{v_i\}$ and $E_i := \mathit{Shortcuts}(i) \cup E_{i-1} \setminus \{(u, v_i), (v_i, u) \in E_{i-1}\}$ for the (partially) contracted graphs. Here $\mathit{Shortcuts}(i)$ are the shortcut edges that need to be inserted when contracting v_i.

Definition 4.1. *An edge $(u, v) \in E$ is called an **upward edge** iff u was contracted before v or v is still uncontracted, and it is called a **downward edge** otherwise.*

We can apply this principle to time-dependent networks: When contracting a station v_i in the timetable network, we need to preserve earliest arrival queries in G_i by introducing shortcut edges. These shortcut edges are replacements for the edges that are removed from E_{i-1} which are all edges adjacent to v_i. In road networks a shortcut edge (u, w) simply represents the combined travel time of two edges $(u, v_i), (v_i, w)$. In a time-dependent network however, we need to consider all pairs of edges $(u, v_i), (v_i, w) \in E$ and link

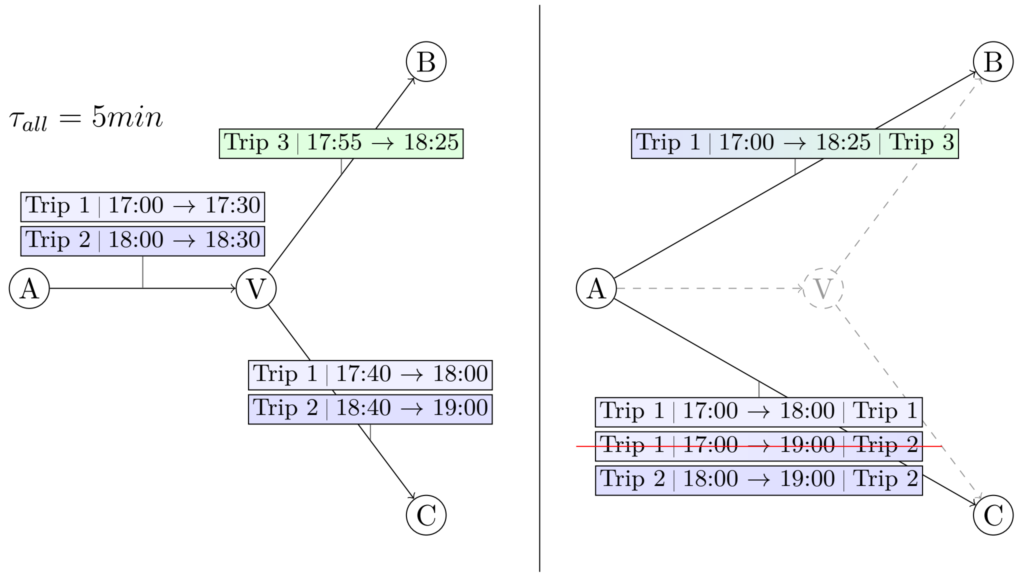


Figure 4.1: Example for the contraction of a node V. Connections on all pairs of incoming/outgoing edges are linked and added to new shortcut edges in order to preserve earliest arrivals in the contracted graph. Note that shortcut connections can start and end with different trips and one of the new shortcut connections from B to C is not required because it is dominated by the other connections on this edge.

connections in (u, v_i) to connections in (v_i, w) to form new shortcut connections that are inserted into the (possibly newly created) edge (u, w). Figure 4.1 shows an example of contracting a node in a time-dependent network.

Not all of these shortcut connections are actually required though. A connection can be omitted either if it is dominated by another connection or if we can find a *witness* for it.

Definition 4.2. *When contracting a node v_i, a* ***witness*** *for a connection c passing through v_i is a connection in G_i that dominates c.*

If a witness for c exists in G_i then the result of any earliest arrival query in G_{i-1} that uses c as part of its path will stay the same in G_i since c can be replaced by its witness in the path because of our definition of domination (see Section 2.4). Since adding additional shortcut edges and connections increases the size of the graph it is desirable to find witnesses for as many shortcut connections as possible. It is also a goal to keep the maximum and average hierarchy depth [Gei10, Vet09] of the contracted graph small.

Definition 4.3. *The* ***hierarchy depth*** *$depth(v)$ of a node is defined initially as $\forall v \in V : depth(v) = 0$ and is set to $\max(depth(v), depth(u) + 1)$ whenever a neighboring node u is contracted.*

Hierarchy depth can be used as a measure of the contraction quality. It is a limit on the maximum number of hops that can be performed in the contracted graph when relaxing only upward edges. When it is small then naturally queries on G will be faster. For both the number of inserted shortcut edges and connections as well as the average hierarchy depth the order in which the nodes of the graph are contracted is vitally important. We therefore require a method to sort the nodes into a sensible order. A common way is to use a function $p : V \to \mathbb{R}$ that assigns a priority to each node and then to contract them in increasing order of priority.

Since we are trying to minimize the number of shortcut edges, the number of shortcut connections and the hierarchy depth it is sensible to define $p(v)$ as a linear combination of these numbers, which can be computed through a simulated contraction of the node, i.e. contracting it to calculate the priority without actually constructing G_i. Since we also want to prevent E_i from becoming too big, we also consider the number edges that are removed from E by contracting v which are simply all edges adjacent to v. Our function p thus becomes

$$p(v) = (\alpha * \frac{\text{\# of new shortcut edges}}{\text{\# of edges removed}}) + (\beta * \text{\# of new shortcut conn.}) + (\gamma * \mathit{depth}(v))$$

with positive constants α, β and γ that we will select through experimentation as we show in Chapter 6, specifically Section 6.4.1.

The contraction algorithm now looks like this: (1) Initialization phase: Simulate contraction of every node to calculate the initial priorities. (2) Contraction phase: Repeatedly contract the node with the smallest priority until all nodes have been contracted. In order to contract nodes more quickly during the contraction phase we store the necessary shortcuts that were found during the simulated contraction. We then just have to insert them into the timetable when the node is actually contracted. However, when a node is contracted then the priorities of adjacent nodes can change which requires us to perform another simulated contraction in order to update the priority of those neighboring nodes. If at all possible we want to avoid this costly operation.

We check how the priority of a node u may change when a neighboring node v is contracted:

- The priority may **increase** when new shortcut connections on (u, v) or (v, u) were created during the contraction of v, because this may induce additional necessary shortcut connections when u is contracted.
- The priority may **increase** when the hierarchy depth of u increases after v is contracted.
- The priority may **increase** when the number of removed edges decreases because the neighboring edges (u, v) and/or (v, u) are removed during the contraction of v if they exist.
- The priority may **decrease** when previously necessary shortcut connections were using connections on the edges (u, v) or (v, u). These shortcut connections are then no longer necessary because the edges (u, v) and (v, u) are no longer part of G_i.

As we can see there is only one case in which the priority decreases. Since we already store the necessary shortcuts for every node we scan over the connections of all neighboring nodes and remove those that are no longer necessary, which can be done in linear time. If this is the only thing that we update after a node was contracted and then calculate a new (temporary) priority based on this information then this priority will never be *lower* than the actual priority that a full update of the node would produce. We can mark a node that was updated in this fashion as "dirty" and only perform another simulated contraction on it when its temporary priority is the lowest among all remaining nodes. If the updated priority is still the lowest, then we can insert the necessary shortcuts and have successfully contracted the node. Otherwise we remove the "dirty" flag and continue with the node that now has the lowest priority.

Theorem 4.4. *When a node is actually contracted in the scheme described above then its priority was the lowest among all nodes in the graph.*

Proof. A node v either has its real priority $p(v)$ assigned to it or a temporary priority. However, since we calculate the temporary priority after removing no longer necessary shortcut connections it must be a lower bound for $p(v)$ since anything else else can only increase the node priority. That means when a node v – after calculating its real priority – has the lowest priority among all real and temporary priorities and is thus selected for contraction, it also has the lowest priority among all real priorities. □

Corollary 4.5. *When keeping a temporary priority for a node v and delaying the update of its real priority until it has the lowest priority in the remaining graph we never perform more simulated contractions than if we update the real priority immediately, but we might perform less.*

Proof. We perform one wasted simulated contraction whenever v's temporary priority is the lowest among all remaining nodes but the real priority is not, which can only happen when a neighboring node u of v was contracted and this increased v's priority, which would also require a simulated contraction to update the real priority immediately.

However, if we always update the priority immediately and several neighboring nodes of v are contracted before v ever has the lowest priority in the remaining graph then we have done additional unnecessary simulated contractions. □

With this reasoning we perform lazy updates of priorities and end up with Algorithm 4.1 as our standard contraction algorithm. The functions CALCULATEPRIORITY and ADDCONNECTIONS should be straightforward whereas FINDNECESSARYSHORTCUTS is the heart of the algorithm and is discussed in the next section.

4.2 Witness Search

The core part of the contraction is the witness search. This is were 99% of the algorithm runtime is spent and therefore its efficiency is critical for the contraction algorithm. Remember that the purpose of the witness search is to find a path ω – the witness – in G_i for each shortcut connection c that is created when contracting v_i so that ω dominates c.

We can approach this problem by considering edges $e_{in} := (u, v_i) \in E_{i-1}$ one at a time. All connections on any edge $e_{out} := (v_i, x) \in E_{i-1}$ will be linked to the connections on e_{in} to form a set $\mathbb{S}$ of shortcut connections. In order to find witnesses for this particular set [Gei10] does a one-to-all profile query from u in G_i (where v_i is removed). Then for any shortcut c in $\mathbb{S}$ we can check for a witness ω among the paths that arrive at $S_{to}(c)$. Since true one-to-all profile queries are unnecessary as we are only interested in paths leading to the neighbors of v_i, a hop limit restricts the search space of the profile query to speed it up. This may lead to some witnesses not being discovered but this is a rare occurrence if the hop limit is chosen carefully and by tuning the hop limit we can make a speed-to-quality adjustment for the contraction. Using this approach a one-to-many profile query for every neighbor u of v_i with an edge (u, v_i) is required.

In this thesis we propose a different method of finding witnesses. Instead of computing complete profile queries we do a single time query for every possible shortcut $c \in \mathbb{S}$. This may sound slow at first but can be sped up by employing some techniques that are inspired by [DKP12]. The idea is to first look at some of the properties that a witness for c must have.

Theorem 4.6. *A witness ω in G_i for a shortcut connection c created when contracting v_i must have departure $t_{dep}(\omega) \geq \max\{t_{dep}(c), t_{prev}(c) + \tau_{S_{from}(c)}\}$ and arrival $t_{arr}(\omega) \leq \min\{t_{arr}(c), t_{next}(c) - \tau_{S_{to}(c)}\}$ unless c starts/ends with a footpath in which case ω may also depart/arrive with a footpath at $t_{dep}(\omega) \geq t_{dep}(c)$ and $t_{arr}(\omega) \leq t_{arr}(c)$ respectively.*

Algorithm 4.1: Public Transit Timetable Contraction

```
Input: Timetable $\mathsf{G} = (\mathsf{V}, \mathsf{E})$
Data: Priority queue $\mathsf{Q}$, Array of sets of shortcuts $\mathsf{S}$
// Initialization
1  forall $v \in \mathsf{V}$ do
2      $\mathsf{S}(v) \leftarrow$ FindNecessaryShortcuts($v$)
3      $p \leftarrow$ CalculatePriority($v$, $\mathsf{S}(v)$)
4      $\mathsf{Q}$.insert($v$, $p$)
// Main loop
5  while $\mathsf{Q}$ is not empty do
6      $v \leftarrow \mathsf{Q}$.deleteMin()
       // Lazy priority update
7      if dirty($v$) then
8          dirty($v$) $\leftarrow$ False
9          $\mathsf{S}(v) \leftarrow$ FindNecessaryShortcuts()
10         $p \leftarrow$ CalculatePriority($v$, $\mathsf{S}(v)$)
11         if $p >$ $\mathsf{Q}$.minPriority() then
12             $\mathsf{Q}$.insert($v, p$)
13             continue
       // Insert stored shortcuts
14     G.AddConnections($\mathsf{S}(v)$)
       // Quickly update adjacent nodes
15     forall $(u, v) \in \mathsf{E}$ or $(v, u) \in \mathsf{E}$ do
16         dirty($u$) $\leftarrow$ True
17         $\mathsf{S}(u) \leftarrow \{(x, u), (u, x) \in \mathsf{S}(u) : x \neq v\}$
18         $p \leftarrow$ CalculatePriority($u$, $\mathsf{S}(u)$)
19         $\mathsf{Q}$.insert($u$, $p$)
       // Remove node v
20     G.RemoveNode($v$)
```

Proof. To be a witness ω must dominate c and thus fulfill the four requirements in Section 2.4.2. However, except for footpaths, conditions (*ii*) and (*iv*) can never be fulfilled by $Z_{first}(c) = Z_{first}(\omega)$ or $Z_{last}(c) = Z_{last}(\omega)$ since c and ω by construction necessarily depart at $S_{from}(c)$ through different edges and arrive at $S_{to}(c)$ through different edges. Therefore (*i*) and (*ii*) directly imply the inequality for the arrival time and (*iii*) and (*iv*) imply the inequality for the departure time. The exception is if c and ω both begin with a footpath because then $Z_{first}(c) = Z_{last}(c)$ and only conditions (*i*)/(*iii*) are required. □

Definition 4.7. *We call* $t_{\omega d}(c) := \max\{t_{dep}(c), t_{prev}(c) + \tau_{S_{from}(c)}\}$ *the* ***witness departure*** *of* c *and* $t_{\omega a}(c) := \min\{t_{arr}(c), t_{next}(c) - \tau_{S_{to}(c)}\}$ *the* ***witness arrival*** *of* c.

We can then sort all $c \in \mathbb{S}$ by their witness departure in descending order. Now we perform one time query per connection c, departing at $t_{\omega d}(c)$, using the ALTQ algorithm. We enforce a hop limit for this query and also stop either when we expand $S_{to}(c)$ or when only nodes with an earliest arrival greater than $t_{\omega a}(c)$ remain in the queue. If the earliest arrival at $S_{to}(c)$ was smaller than $t_{\omega a}(c)$ then we have found a witness for c and do not need to add c to the list necessary shortcuts. Otherwise c is a necessary shortcut or we have missed a witness either because of the hop limit or because c starts with a footpath and we did not consider footpath departures in $[t_{dep}(c), t_{\omega d}(c))$.

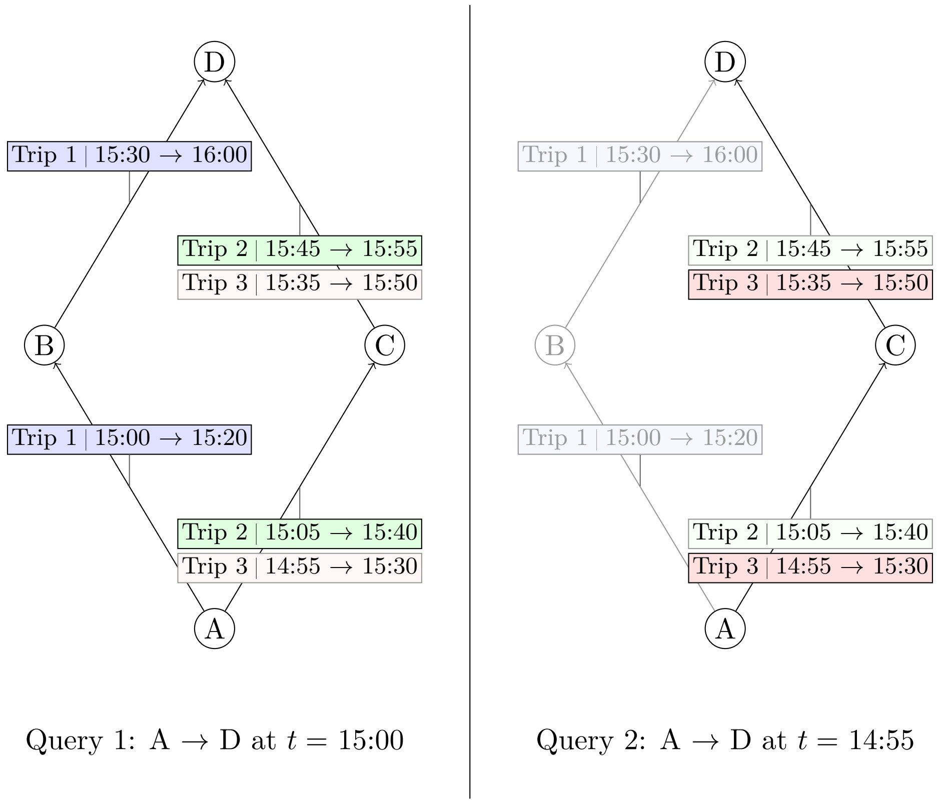


Figure 4.2: An example for reuse of data from previous queries to prune queries earlier. On both sides an earliest arrival search from A to D is performed. First with a departure time of 15:00 (left) and then with a departure of 14:55 (right). The first query expands the highlighted trips and the nodes in order A, B, C before finding the earliest arrival at D. Normally the second query would expand the same trips and nodes since they can all be reached with the earlier departure, but since the data from the previous query is not reset already used connections are not expanded again and nodes are only expanded again if they improve. Therefore the second query expands only nodes A and C and uses only the newly available trip 3 which improves the earliest arrival at C.

Our experiments show that only a very small portion of witnesses is missed because of a footpath departure in the mentioned interval. In return the advantage we gain from this method is that we do not have to reset the state of the ALTQ for successive time queries. Remember that all shortcut connections in $\mathbb{S}$ depart from the same station u as we are currently looking at $e_{in} := (u, v_i)$. For the next shortcut connection $c_2 \in \mathbb{S}$ that has $t_{\omega d}(c_2) \leq t_{\omega d}(c)$ we can first check if the earliest arrival at $S_{to}(c_2)$ is already good enough to dominate c_2 in which case we do not have to run a search for c_2. If not, we insert a label at u with departure time $t_{\omega d}(c_2)$, add the station to the priority queue and continue with the ALTQ in this state.

This will produce a correct result for the query departing at $t_{\omega d}(c_2)$ because this query either improves the bags on the way to its target node and thus re-inserts them into the priority queue and expands them again with the updated labels and earliest arrival time. Or the query does not improve the labels at any station along the way in which case best possible arrival must be the one computed by the previous query which is valid because when the queries are performed in descending order of witness departure we can always wait at the source station until $t_{\omega d}(c)$.
This way we can reuse work that has already been done for previous connections to make

Algorithm 4.2: FindNecessaryShortcuts

```
Input: Timetable $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, Node $v_c$ to be contracted, Hop Limit $\mathsf{h}$
Data: ALTQ instance TimeQuery, slightly modified for the witness search
Output: Array $\mathbb{S}$ of shortcut connections for which no witness was found
1  $\mathbb{S} \leftarrow \emptyset$
   // Temporarily take $v_c$ out of G for the witness search
2  $\mathsf{G}_x \leftarrow \mathsf{G} \setminus \{v_c\}$
3  forall $e := (u, v_c) \in \mathsf{E}$ do
4      $\mathbb{S}_e \leftarrow \emptyset$
5      forall $f := (v_c, w) \in \mathsf{E}$ do
6          $\mathbb{S}_e \leftarrow \mathbb{S}_e \cup \textsc{dominate}(e.\textsc{link}(f))$
7      $\mathbb{S}_e.\textsc{sortDescending}(key = t_{\omega d}(\cdot))$
       // Set all distances to $\infty$ and clear all data structures
8      TimeQuery.RESET()
9      forall $c \in \mathbb{S}_e$ do
           // Check whether $c$ is not already witnessed
10         if TimeQuery.DISTANCE($S_{to}(c)$) $\geq t_{\omega a}(c)$ then
               // Continue witness search with new starting label
11             TimeQuery.ADDLABEL($S_{from}(c)$, $t_{\omega d}(c)$)
12             TimeQuery.RUN($\mathsf{G}_x$, $u$, $S_{to}(c)$, $maxArrival = t_{\omega a}(c)$, $hopLimit = \mathsf{h}$)
               // Still not witnessed? $\Rightarrow$ Shortcut is necessary
13             if TimeQuery.DISTANCE($S_{to}(c)$) $\geq t_{\omega a}(c)$ then
14                 $\mathbb{S} \leftarrow \mathbb{S} \cup \{c\}$
```

the average ALTQ witness search very fast while at the same time we avoid doing costly profile searches for edges where only few shortcut connections exist. Algorithm 4.2 shows the witness search in pseudo code and Figure 4.2 gives an example for the reuse of previous query data.

4.2.1 Linking and Dominating

For the witness search an efficient linking operation is required that links connections from all pairs of edges $g := (u, v)$ and $h := (v, w)$ to create shortcut connections along these edges. We also need an efficient way to remove all dominated shortcut connections from the resulting sets. If $C(g)$ and $C(h)$ are the sets of connections that belong to g and h respectively then a naive implementation might simply link all combinations of connections from $C(g)$ and $C(h)$ resulting in $|C(g)| * |C(h)|$ shortcut connections. Geisberger [Gei10] uses more efficient link and domination algorithms in his profile queries which we adapt for our witness search.

For linking we sort the connections in ascending order of their departure time. In order to link as few connections as possible the important observation is this: When we link to a connection $c_1 \in C(g)$ and there is another connection $c_2 \in C(g)$ with $t_{dep}(c_2) \geq t_{dep}(c_1) + \tau_v$ then we only have to link connections from $C(h)$ to c_1 that depart in the interval $[t_{arr}(c_1), t_{arr}(c_2))$ since for any connection $d \in C(h)$ with $t_{dep}(d) \geq t_{arr}(c_2)$ the shortcut connection (c_1, d) is dominated by (c_2, d). This is true because c_2 dominates c_1 at departure due to enough buffer for the transfer time and because the linked connections have the same arrival because they both end with d thus dominating each other at arrival.

This means our link algorithm scans through the connections in $C(g)$ in descending order while remembering the minimum arrival time $t_{arr}(min)$ among connections that depart at

$t_{dep}(c) + \tau_v$ or later. Then for every connection we find the connections in $C(h)$ that depart in the interval $[t_{arr}(c), t_{arr}(min)]$ and check if those connections can be linked to c. In practice this interval is very small leading to an almost linear amount of connections being created instead of quadratically many as verified by Geisberger's experiments [Gei10].

In order to remove all dominated connections we first order the connections according to the following criteria, with later criteria being used as tiebreakers:

1. Their departure time $t_{dep}(c)$ ascending
2. Their arrival time $t_{arr}(c)$ (descending)
3. Non-critical arrivals (where $t_{arr}(c) + \tau_{S_{to}(c)} \leq t_{next}(c)$) before critical arrivals.
4. Non-critical departures (where $t_{prev}(c) + \tau_{S_{from}(c)} \leq t_{dep}(c)$) before critcial departures.

Theorem 4.8. *Given two connections A and B with $S_{from}(A) = S_{from}(B) =: S$, $S_{to}(A) = S_{to}(B) =: T$ and $A < B$ according to above sorting criteria, A cannot dominate B unless B also dominates A.*

Proof. Let A dominate B according to the criteria from Section 2.4. We want to show that B dominates A:

$$t_{dep}(B) \overset{(iii)}{\leq} t_{dep}(A) \overset{(1)}{\leq} t_{dep}(B) \Rightarrow t_{dep}(A) = t_{dep}(B) \tag{a}$$

$$t_{arr}(A) \overset{(i)}{\leq} t_{arr}(B) \overset{(2)}{\leq} t_{arr}(A) \Rightarrow t_{arr}(A) = t_{arr}(B) \tag{b}$$

Either $Z_{last}(A) = Z_{last}(B)$ (in which case both are also tied for sorting critierion 3) or $t_{arr}(A) + \tau_T \leq t_{next}(B) \overset{(a)}{\Rightarrow} t_{arr}(B) + \tau_T \leq t_{next}(B) \Rightarrow B$ has non-critical arrival $\overset{(3)}{\Rightarrow}$ A has non-critical arrival $\Rightarrow t_{arr}(A) + \tau_T \leq t_{next}(A) \overset{(a)}{\Rightarrow} t_{arr}(B) + \tau_T \leq t_{next}(A)$.
Either $Z_{first}(A) = Z_{first}(B)$ or $t_{prev}(B) + \tau_S \leq t_{dep}(A) \overset{(b)}{\Rightarrow} t_{prev}(B) + \tau_S \leq t_{dep}(B) \Rightarrow$ B has non-critical departure $\overset{(4)}{\Rightarrow}$ A has non-critical departure $\Rightarrow t_{prev}(A) + \tau_S \leq t_{dep}(A) \overset{(b)}{\Rightarrow}$ $t_{prev}(A) + \tau_S \leq t_{dep}(B)$.
$\Rightarrow$ All domination criteria are fulfilled so B also dominates A. □

For connections that dominate each other we only want to keep one of them and without loss of generality can keep the one that was sorted into the smaller position. That means for any connection we only have to check larger connections to find one that dominates it; or looking at it the other way around: for any connection we only have to check smaller connections to see if they are dominated by this connection. We therefore remove dominated connections by scanning over the connections in descending order, checking for each connection if it is dominated by a larger one.

To make this faster we keep a buffer of all relevant connections that might dominate the current connection c. The buffer is initially populated with connections from the "next day" where the departure times are shifted up by the timetables' period π. Connections are added to the buffer when they were not dominated and a connection d can be removed from the buffer when

$$t_{arr}(d) \geq \min_{i \in \text{Buffer: } t_{dep}(i) \geq t_{dep}(c) + \tau_S} t_{arr}(i) + \tau_T$$

because connection i dominates any connection smaller or equal than c that would have also been dominated by d as can easily be shown by checking the domination criteria in combination with the order by departure time that the connections are in.

Domination still has a theoretical worst case complexity of $O(n^2)$, but on real-world timetables it performs closer to $O(n)$.

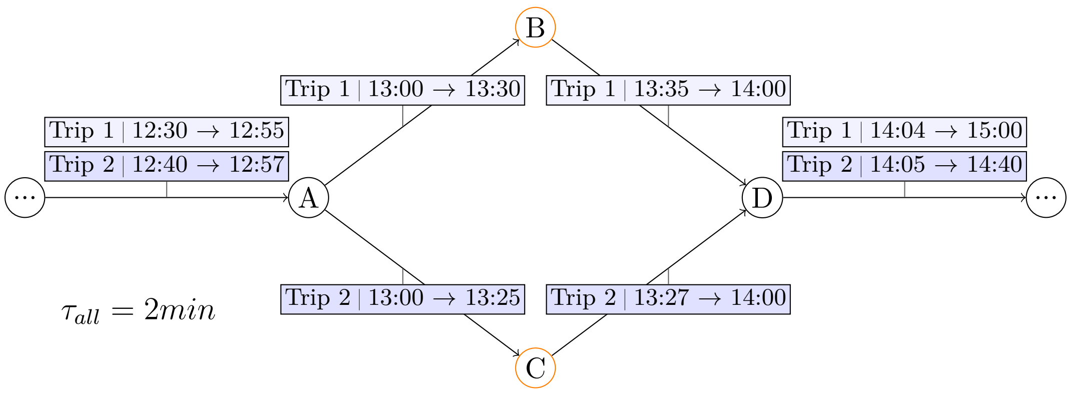


Figure 4.3: An example of two shortcut connections witnessing each other. The connections A-B-D with trip 1 and A-C-D with trip 2 have the exact same length and transfer from trip 1 to trip 2 and vice versa is possible at both A and D. Therefore neither of these connections is a necessary shortcut right now. If we do not update the necessary shortcuts of C after contracting B, then contracting C will leave the graph without any connection from A to D.

4.2.2 Problems and Edge Cases

There are several problems and edge cases in the current algorithm that need to be carefully considered to maintain correctness. The first problem occurs when there are two connections that witness each other, passing through nodes v_1 and v_2 that are not adjacent to each other. This can happen if both connections have the exact same length and and a low/high enough previous arrival and next departure respectively so that transfers at both ends are always possible. Normally this would not be a problem because once v_1 is contracted a new witness search for v_2 will no longer find the connection through v_1 as a witness. However, since we store the necessary shortcuts of a witness search and only update a stop when a neighbor has been contracted, v_2 might not get another update before it is contracted.

There are two solutions to this problem: Either we disallow witnesses with the exact same length so no two connections can witness each other or we perform an update for *every* node when it has the lowest priority and not just those that were marked “dirty”. The first solution lowers the quality of the witness search a tiny bit but its effect is usually unnoticeable. The second solution has a small negative impact on the contraction performance. Since the problem did not seem to occur at all in our real-world data sets we implemented the first solution as it had the smaller impact.

Another edge case that has to be considered are trips that pass through a station twice in quick succession so that it is sometimes preferable to stay in the vehicle rather than leave when the station is first encountered, because the transfer time is too high to re-enter the vehicle if that turns out to be the fastest connection. That means that our witness search has to support loops, i.e. shortcut edges (v, v).

Another thing that might cause problems are differing transfer times at stations so that it is sometimes best to change vehicles at a later station even if the trip that one transfers to visits a previous station immediately afterwards. An example of this is shown in Figure 4.4. Thankfully this problem can also be addressed simply by allowing loops.

4.3 Time Queries

Earliest arrival queries on contracted graphs usually consist of a forward search relaxing only upward edges and a backward search relaxing only downward edges. However, we

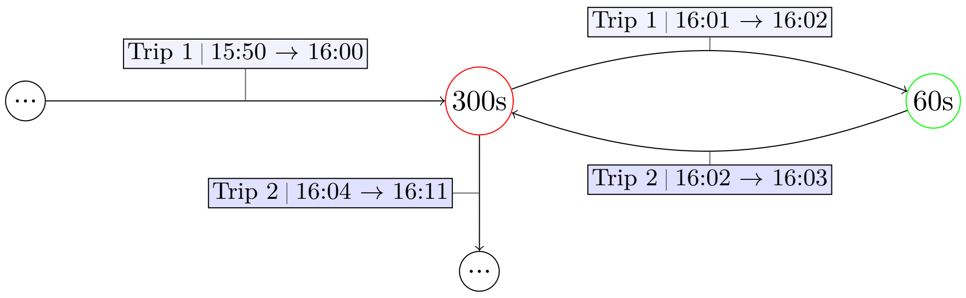


Figure 4.4: An example where differing transfer times may cause loops in the contracted graph. Transferring from trip 1 to trip 2 at the red stop is not possible, but it is possible at the green stop. So when the green stop is contracted, a shortcut edge red → red is required that stores the shortcut connection that changes trips via the green stop.

cannot perform a backward search in a time-dependent network if we do not know the arrival time in advance. Therefore a slightly different approach is necessary.

Theorem 4.9. *An earliest arrival query in the contracted, time-dependent network first performs a backward search from the target station on the* edges *only (disregarding specific connections) relaxing only downward edges and marking each downward edge that is encountered. Then a regular forward time query solves the EAP if it relaxes only upward edges and marked downward edges.*

Proof. By construction of the Contraction Hierarchy a shortest path can be found by following a series of upward edges from the source node followed by a series of downward edges to the target node, so any downward edge that must be expanded can be reached by the backwards search. Suppose the shortest path would at any point visit a downward edge (u, v) that has not been marked by the backwards search. Then it must be followed by an upward edge (v, w) and by definition v was contracted before u and w. Therefore there must either exist some path $u \to w$ that was a witness for the connection through v or there exists a shortcut connection that was inserted into the edge (u, w). Either can be used instead of the connections on (u, v) and (v, w). By recursively applying this reasoning we can conclude that the shortest path is indeed found by following a series of upward edges and then a series of marked downward edges and the algorithm finds the correct answer. □

We can use both ALTQ and ELTQ for the forward search with by modifying which edges should be relaxed. The backward search can be performed using a standard depth-first search or breadth-first-search.

4.3.1 Performance Improvements

One of the advantages of time queries on the contracted network is that far fewer edges have to be relaxed during a query. However, since the forward search cannot only relax upward edges we still have to check *all* adjacent edges because some of the downward edges could be marked. Even checking these edges creates unnecessary overhead that we would like to avoid.

Before we try to improve this we must explain briefly how the edges are stored. All edges are stored as one large array and each node keeps two lists, one containing pointers to all incoming edges and one containing pointers to all outgoing edges. During the contraction both lists are needed but after the contraction is complete we can make two improvements:

(1) For time queries the incoming edges are only needed for the backward search and since the backward search only follows downward edges we can delete all upward edges from the incoming edges list of each node. (2) We delete all downward edges from the outgoing edges list of each node. Then, during the query, the downward edges needed for the forward search are added back to the list dynamically during the backward search. This way we do not even need a marker for the downward edges that may be used since the useable downward edges are simply those that exist in the outgoing edges list of each station during the query. On every subsequent query the first time a node is encountered (which might happen either during the backward search or during the forward search) we remove the added downward edges from the previous run by simply resizing the list (assuming that the downward edges were added to the end of the list). This allows us to check far fewer edges and the overhead of adding and removing elements of the lists is lower than the overhead of checking additional edges.

4.3.2 Unpacking Shortcuts

Although our time queries produce the correct earliest arrival on contracted timetables we usually also want to know the path that needs to be taken in order to achieve this arrival time. This is straightforward in the uncontracted graph as for each arrival connection we just need to store which elementary connection was used last and can then work backwards through the graph. However in the contracted network we also need to *unpack* the shortcut connections that were used in order to find a proper sequence of elementary connections that represent the journey from source to target.

Lemma 4.10. *Given a shortcut connection c we can reconstruct the sequence of elementary connections* $(c_1, c_2, ..., c_k)$ *which the shortcut represents.*

As a reminder, each shortcut was created by linking two connections c_l and c_r one or both of which may themselves have been shortcut connections. Let $u := S_{from}(c_l), v := S_{to}(c_l) = S_{from}(c_r)$ and $w := S_{to}(c_r)$. We know u, v and w as (u, w) is the edge that c belongs to and v was stored as via(c) when the shortcut connection was created. Because we never remove connections from our timetable, c_l and c_r must still exist and be stored with the edges (u, v) and (v, w) respectively. We also know that $t_{dep}(c_l) = t_{dep}(h)$ and $t_{arr}(c_r) = t_{arr}(h)$ as well as the trips $R_{first}(c_l) = R_{first}(h)$ and $R_{last}(c_r) = R_{last}(h)$.

To find c_l we search the edge (u, v) for all connections that depart at $t_{dep}(c)$ and use $Z_{first}(c)$ as their first trip. Note that in the contracted timetable there might be several such connections because shortcut connections with the same departure (time and trip) but different arrival might exist that do not dominate each other. The resulting connections are candidates for c_l. We can speed up this search by using a binary search for the departure time. In the same way we find candidates for c_r by searching (v, w) for connections with the proper arrival time and trip.

Among the sets of candidates we now need to find any two that can be linked together to form a consistent connection. Even if these are not the two connections that were originally linked to form c they still provide a consistent sequence. If either c_l or c_r are shortcut connections themselves we recursively unpack them using the same method. In the end we will have reduced all shortcut connections to a sequence of elementary connections. Although there might be many candidates for c_l and c_r in most realistic cases there is rarely more than one and unpacking is very fast relative to query times.

4.4 Profile Queries

Apart from time queries profile queries are a type of query that is not applicable to road networks but is quite common in timetable networks. A profile query is similar to a time query in that it asks for shortest paths from station X to station Y but contrary to time queries we do not have a fixed departure time. Instead the departure time is an interval $[t_0, t_1]$ and we want shortest paths for *any* departure time in this interval. Profile queries can be posed as one-to-one queries with one source and one target station but can also be one-to-many or one-to-all queries.

We want to adapt our algorithms to also answer profile queries on the uncontracted graph as well as the contracted graph to benefit from the Contraction Hierarchy. A first naive approach is to perform one time query for every second in the interval $[t_0, t_1]$ (since a second is the lowest resolution in our timetable). This is of course much too slow for any practical use. One important observation is that as long as footpaths are disregarded there can be at most one unique shortest path for every connection departing from X in the interval $[t_0, t_1]$ [DKP12] because for any other departure time the journey just starts by waiting for the next train that leaves from X. If we include footpaths then we also need to take into account any train departing from stations that can be reached by footpath from X as a starting point for a unique path so that for any connection c at a station X' reachable by footpath in t_{foot} seconds from X there could be a unique path starting at $t_{dep}(c) - t_{foot}$ from X with this footpath.

We call the set of points in time $\mathbb{T}$ that have to be checked the **start times** of a profile query. With this observation we can improve the naive solution by only performing one time query for each of those start times. This is better and already performs well on both contracted and uncontracted networks when we use ALTQ or ELTQ as the time query. However, we can make it even faster by applying some of the principles that the witness search also uses, most importantly that we do not have to reset our time query data structures between queries. If we order $\mathbb{T}$ in descending order then a new time query can only ever improve the previous earliest arrival at the target station(s). That means we can reuse the data of the previous query which makes each individual query that *does not improve* the solution much faster as it will stop earlier. We can also speed up the backwards search that marks the downward edges that may be used in the contracted graph by performing one combined backwards search for all target nodes and do so only once during the initialization of the profile query.

Our implementation uses ELTQ as the base algorithm. We do not actually store full paths as solution but only departure and arrival for each path in the profile, although it is possible to store full paths by extracting them after each query that improved the arrival time. Any individual query can be stopped as soon as all target nodes have been expanded since their arrival time then can no longer improve. In the case that some target nodes are completely unreachable this can be detected in advance by a simple breadth first search on the edges since in a periodic timetable it is always possible to travel along an edge if it has at least one connection since one can wait for the next day when a connection has been missed. However, our actual implementation assumes that unreachable target nodes will never be used for the queries. Algorithm 4.3 shows the algorithm in pseudo-code.

Algorithm 4.3: Event Label Profile Query

```
Input: Timetable $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, Stations $\mathsf{X}$ (Source) and set $\mathbb{Y}$ (Targets), Departure
       interval $[t_0, t_1]$
Data: ELTQ instance TimeQuery
Output: For each target station $\mathsf{Y} \in \mathbb{Y}$ a list of (departure, arrival) pairs $\mathsf{P(Y)}$
for all unique shortest X- Y paths that depart in $[t_0, t_1]$

// Initialization
1  TimeQuery.reset()
2  forall $\mathsf{Y} \in \mathbb{Y}$ do  $\mathsf{P(Y)} \leftarrow \emptyset$
   // Find all start times and sort in descending order
3  $\mathbb{T} \leftarrow \emptyset$
4  forall Connection $c \in (X, \cdot) \in \mathsf{E}$ do
5      if $t_{dep}(c) \in [t_0, t_1]$ then $\mathbb{T} \leftarrow \mathbb{T} \cup \{t_{dep}(c)\}$
6  forall Footpath $(X, F) \in \mathsf{E}$ do
7      forall Connection $c \in (F, \cdot) \in \mathsf{E}$ do
8          if $t_{dep}(c) \in [t_0, t_1]$ then $\mathbb{T} \leftarrow \mathbb{T} \cup \{t_{dep}(c)\}$
9  $\mathbb{T} \leftarrow$ sortDescending($\mathbb{T}$)
   // Main Loop
10 forall t $\in \mathbb{T}$ do
11     TimeQuery.run(X,$\mathbb{Y}$, t)
12     forall $\mathsf{Y} \in \mathbb{Y}$ do
13         if TimeQuery.earliestArrival(Y) $< \min_{arrival}$ P(Y) then
14             P(Y) $\leftarrow$ P(Y) $\cup$ (t, TimeQuery.earliestArrival(Y))
   // Return the profiles of all target stations
15 return P($\mathbb{Y}$)
```

5. Contraction of Multimodal Networks

Multimodal route planning combines networks of different kinds of transportation and allows queries on the combined network. Typical networks are road, walking, public transit, bicycle, ferries and taxis. In this thesis we are not using fully multimodal networks that incorporate all those modes of transport. We are still interested in earliest arrivals and using Contraction Hierarchies to improve their query times; if we include data for road networks then the expectation is that car travel will always dominate public transit for earliest arrival since cars allow traveling directly to your destination whereas public transit has more overhead in getting to stations and waiting for trains, changing trains and so on. Instead we will try to combine walking and public transit only.

Right now our footpaths are extremely limited and restrictive. We require them to be transitive, so any component of n nodes that is connected by footpaths will require $n * (n-1) \in O(n^2)$ footpaths which clutters the timetable and slows down both queries and contraction. Ideally we not only want to lift the requirement for footpaths to be transitive but also connect larger groups of nodes to each other. In many cases it can make sense to walk the distance between two stations where direct connections do not exist or when the next train will not arrive for a long time, for example at night. Ideally we would want to include the walking distance among many such stations that are reasonably close to each other as footpaths in the network.

The problem in our current state however, is that queries in the contracted graph will not remain correct if footpaths are not transitive, because we would sometimes need to link two footpaths together to create a new footpath. This is a problem, because we cannot simply add them to the network or the number of footpaths would also increase very quickly to the order of $O(n^2)$. We cannot perform a standard witness search for them either because they have no specific departure time. Therefore we must introduce a new approach to make those footpaths viable.

5.1 Witness Search for Footpaths

Definition 5.1. *During the contraction of a node v for every pair of footpaths (u, v), (v, w) we get a candidate for a* ***footpath shortcut*** *by combining the two footpaths into a footpath (u, w) with the combined travel duration.*

For such a candidate we need to determine whether it is necessary to add it to the timetable. Since a footpath f has no fixed departure time it can be used at any time of the day and therefore we must also find one or multiple witnesses that dominate f for any possible departure in $[0, \pi)$. There are two ways to dominate a footpath shortcut f:

1. We find another sequence of footpaths that do not travel via v and have a shorter or equal combined travel duration. Because this sequence consist only of footpaths it can also be used at any time during the day and can therefore always replace f in any path.
2. We find a number of connections so that for any departure time $t \in [0, \pi)$ there is a connection c with $t + \tau_u \leq t_{dep}(c)$ and $t_{arr}(c) + \tau_w \leq t + length(f)$ that does not travel via v. Such a connection dominates c for departure time t.

So before doing anything else we perform one search in the footpath network only to find witnesses for all footpath shortcut candidates. The more candidates we can eliminate that way the less work has to be done in the remaining part. For any candidate remaining we must consider the second condition to dominate it. In the second case if there is any time of the day where no witness connection can be found then the footpath is necessary and must be inserted into the contracted timetable.

We make use of the fact that a witness for a regular shortcut connection can also be a witness for a footpath shortcut. During the regular witness search we are looping through the possible shortcut connections in descending order of their witness departure and find witnesses for them. Now say a witness c is found during this search, then c also dominates a footpath shortcut candidate f for all departures in $[t_{arr}(c) + \tau_w - length(f), t_{dep}(c) - \tau_u]$ since for any departure in this interval $t + \tau_u \leq t_{dep}(c) - \tau_u + \tau_u = t_{dep}(c)$ and $t + length(f) \geq t_{arr}(c) + \tau_w - length(f) + length(f) = t_{arr}(c) + \tau_w$ and the footpath is dominated.

So for every footpath shortcut candidate we remember the earliest time for which the footpath has not been dominated yet which is set to $(\pi - 1)$ initially. Let us call this time the **next witness time** of f and write is as $t_{wit}(f)$. Then during the regular witness search whenever we find a witness c for any of the regular shortcut connections we also reduce the earliest time for all footpaths to $t_{arr}(c) + \tau_w - length(f) - 1$. However, if the upper bound of the domination interval $t_{dep}(c) - \tau_u$ is smaller than the previous $t_{wit}(f)$ then there is a gap for which no witness exists yet. In this case we perform an extra witness search with departure $t_{wit}(f) + \tau_u$ and update $t_{wit}(f)$ (and all the other footpath shortcut candidates) with the resulting witness until either the gap is closed or no witness is found. As soon as we reach a departure time where no witness is found we can immediately add f to the set of necessary shortcuts and never have to consider it again. On the other hand if $t_{wit}(f)$ drops to zero the footpath shortcut is not necessary and can be deleted.

When there is a time in the day where no witness is found for a footpath shortcut candidate then it is very likely that this happens late at night. Insofar we have the additional advantage that we start our search at midnight as necessary footpath shortcuts will usually be identified with the very first witness search (which finds no witness). In order to efficiently update $t_{wit}(f)$ and check whether we need additional witness search queries we use an additional priority queue that holds all the footpath shortcut candidates with $t_{wit}(f) + \tau_u$ as their key. Whenever the maximum key in this priority queue is larger than the witness departure of the current regular shortcut connection we perform one additional witness search then update the key of this footpath.

Algorithm 5.1 shows the algorithm in pseudo-code.

5.2 Time-Limited Footpath Shortcuts

With the improvement to our witness search in the previous section we now no longer have to construct cliques around connected sets of nodes in the footpath part of a timetable.

However when we start connecting larger node sets with footpaths there is another problem we can encounter: For most real world train schedules around 3am at night there is the least amount of activity. During this time it is likely that footpath shortcuts will not be dominated. When we are working with a city-sized network like London it can be best to walk for a long time during the night which results in similar problems as we had before when we are trying to contract the network: The number of necessary footpath shortcuts becomes very large again.

One solution we want to attempt is to divide the timetable into a daytime and a nighttime network. When adding new footpaths we consider a footpath dominated when it is dominated at any time during the **day time interval** $[d_0, d_1]$. Our queries will now be incorrect during the night though. During a query when the current arrival time at a station falls outside of the day time interval then we must also consider *all* the footpaths in the network and not only those on upward edges. This is a trade-off that makes the contraction for these networks feasible, but lets the queries benefit less from the contraction. The actual values for the day time interval must be carefully chosen. We want the interval to be as large as possible without reaching the point where suddenly a lot of footpath shortcuts are no longer dominated.

5.2.1 Preliminary Experiments

We did some preliminary experiments with the day time interval on a number of public transit networks that were augmented with additional footpaths. The day time interval did reduce the number of necessary footpath shortcuts by roughly 20% in all cases but unfortunately had little impact on the quality of the resulting contraction. The number of shortcut connections that were created remained mostly unchanged, contrary to our expectation that fewer footpath shortcuts would also lower the number of shortcut connections because fewer combinations of footpaths and connections would have to be linked. The runtime in the contracted network showed no improvements. One thing we gained was a small reduction in the size of the timetable due to the lower number of footpath shortcuts.

Algorithm 5.1: FindNecessaryShortcutsMultimodal

```
Input: Timetable $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, Node $v_c$ to be contracted, Hop Limit $\mathsf{h}$
Data: ALTQ instance TimeQuery, Priority queue Q for footpath candidates
Output: Array $\mathbb{S}$ of shortcut conn. / footpaths for which no witness was found
1  $\mathbb{S} \leftarrow \emptyset$
   // Temporarily take $v_c$ out of G for the witness search
2  $\mathsf{G}_x \leftarrow \mathsf{G} \setminus \{v_c\}$
3  forall $e := (u, v_c) \in \mathsf{E}$ do
       // Collect all shortcut candidates starting on edge e
4      $\mathbb{S}_e \leftarrow \emptyset$
5      forall $f := (v_c, w) \in \mathsf{E}$ do
6          $\mathbb{S}_e \leftarrow \mathbb{S}_e \cup$ DOMINATE($e$.LINK($f$))
7          if e and f both have footpaths then
               // Footpath candidate unless witnessed by other footpath
8              $p \leftarrow$ LINKFOOTPATHS($e$, $f$)
9              if TimeQuery.FOOTPATHDISTANCE($G_x, u, w$) $>$ $p$.LENGTH() then
10                 $t_{wit}(p) \leftarrow \pi - 1$
11                 Q.INSERT($p$)

       // Set all distances to $\infty$ and clear all data structures
12     TimeQuery.RESET()
       // Find witnesses for candidates in descending order
13     while not Q.EMPTY() or not $\mathbb{S}_e = \emptyset$ do
           // Find candidate with maximum witness departure
14         $c \leftarrow \max_{\text{witness departure}} \{$Q.MAXIMUM() $\cup\, \mathbb{S}_e\}$
15         remove $c$ from Q or $\mathbb{S}_e$
           // Continue witness search with new starting label
16         TimeQuery.ADDLABEL($S_{from}(c)$, $t_{\omega d}(c)$)
17         TimeQuery.RUN($\mathsf{G}_x$, $u$, $S_{to}(c)$, maxArrival $= t_{\omega a}(c)$, hopLimit $= \mathsf{h}$)
           // Not witnessed? $\Rightarrow$ Shortcut is necessary
18         if TimeQuery.DISTANCE($S_{to}(c)$) $\geq t_{\omega a}(c)$ then
19             $\mathbb{S} \leftarrow \mathbb{S} \cup \{c\}$
20         else if c is footpath then
               // Update next witness time of footpath and re-insert
21             $t_{wit}(p) \leftarrow$ TimeQuery.DISTANCE($S_{to}(c)$) $+ \tau_{S_{to}(c)} - p$.LENGTH()
22             if $t_{wit}(p) \geq 0$ then Q.INSERT($p$)
23
```

6. Experiments

In this chapter we measure the performance of all algorithms on a number of different timetables ranging from local city traffic to long-distance trains covering parts of Europe. We compare the runtime and preprocessing times for Contraction Hierarchies on those timetables and we try different contraction parameters. We test time queries, profile queries and multimodal queries and show a few ways to optimize the performance of Contraction Hierarchies on public transit networks.

6.1 Platform Specification

Experiments were primarily conducted on a single core of a 4-core Intel Xeon E5-1630v3 clocked at 3.7 GHz, 128 GiB of DDR4-2133 RAM, 10 MiB of L3 and 256 KiB of L2 cache. All development was done on an Intel Core 2 Duo clocked at 2.4 GHz, 4GiB of DDR3-1067 RAM and 3MB L2-Cache. Some of the experiments refer to performance on the development hardware to highlight the difference that smaller caches make in relation to the primary hardware. Any experiment using the development hardware are marked as such.

Programs were compiled with the GNU C++ compiler (GCC) version 4.8.3 using optimization level 3 (-O3). All code was running sequentially on a single core.

6.2 Data Sets

We tested all algorithms on a number of country-sized and city-sized public transit networks. Table 6.1 shows the different networks and some statistics about them including number of stations, number of edges, number of connections, number of trips, number of routes and number of footpaths that are in the network. The network for Europe is the same as the one used by Geisberger [Gei10] and our data of Berlin/Brandenburg covers roughly the same area but uses updated schedules. These networks allow us to make some comparison to the results of Geisberger although we are also using faster hardware.

The timetable for Germany contains all trains operated by Deutsche Bahn and is using the winter schedule of 2001/02. Europe is based on the winter schedule of 1996/97 and consist mostly of long-distance trains. Sweden and Switzerland contain both long-distance and local transit and were extracted from `trafiklab.se` and `gtfs.geops.ch` respectively. The data set named VBB is from fall 2014 and contains trains, subways and buses for

Table 6.1: Overview of the data sets that were used for our experiments.

Network	#Stops	#Edges	#Connections	#Trips	#Routes	#Footpaths
Europe	30,517	87,818	1,639,869	166,176	44,824	0
Germany	6,822	18,440	500,757	49,172	9,365	0
Sweden	51,081	117,339	4,388,621	194,622	22,887	1,144
Switzerland	26,288	76,865	4,749,585	562,318	39,669	12,659
VBB	13,054	35,975	1,096,510	53,134	10,548	786
Chicago	11,991	24,624	1,194,512	20,303	710	12,082
London	20,755	69,612	4,941,261	129,263	2,184	45,326
Madrid	4,651	7,293	3,006,388	110,181	1,615	1,284
New York	16,136	60,361	1,824,717	45,297	1,400	44,528
Vitoria-Gasteiz	331	756	34,305	1,770	42	335

Berlin as well as the state of Brandenburg including rural areas with few connections per day. It is publicly available from the Berlin Open Data project and is made available by the Verkehrsverbund Berlin-Brandenburg (VBB). London was extracted from the London Data Store and contains data from a Tuesday during the periodic summer schedule of 2011. It includes the subway, buses, trams, the Dockland Light Rail and ferries. New York contains subway and bus schedules from the MTA New York for Wednesday, August 10, 2011. Data for Madrid was extracted from `emtmadrid.es` and contains only bus schedules and its footpaths were generated using a known heuristic. Footpaths in all other networks were part of the input. Vitoria-Gasteiz is a very small test timteable of very little practical consideration and is listed only for completeness.

Note that in our model every footpath gets its own edge unless an edge between the two stations that it connects already exists which usually is not the case. Therefore the number of edges also includes footpaths. None of our data sets included any transfer times for stations, so for most experiments we have chosen a global transfer time that is the same for all stations in the network. Unless mentioned otherwise the transfer time was 300 seconds on Germany, Europe, Sweden and Switzerland and 180 seconds on the city-sized networks.

6.3 ALTQ and ELTQ

Before we compare the runtime of ALTQ and ELTQ we first check the performance of the first baseline algorithm, which is a standard unidirectional Dijkstra on the time-dependent network with route nodes (see 3.1). In addition to per-query runtime we also report the number of extract/delete minimum operations (DeleteMin) of the priority queue which is equal to the number of stops that were expanded, since the next stop to be expanded is always taken from the priority queue. The results for all data sets can be found in Figure 6.1.

The results of all time query experiments in this chapter are averages of 10,000 random queries that were generated by choosing a source and target station and a departure time in $[0, \pi)$, our 24 hour period, uniformly at random. One query set was generated for each timetable and different algorithms on the same timetable use the same query set.

We can see that the runtime is directly proportional to the number of DeleteMins. We tried the algorithm with different transfer times and as expected the runtime remained constant, because the transfer time in this model is handled by the route nodes and does not produce any additional work. Query times are faster on city-sized networks even if they have more nodes than some of the national networks which can be explained by the fact that on national networks it is more likely that two random stations require a journey that takes several hours and thus more connections throughout the day have to be checked.

Network	Time [ms]	#DeleteMins
Vitoria-G.	0.08	600
Chicago	8.7	25,000
Madrid	14.9	26,200
New York	12.3	32,800
London	25.9	55,000
Germany	21.8	65,000
VBB	32.4	111,000
Europe	77.2	257,900
Switzerland	142.5	320,400
Sweden	108.4	341,000

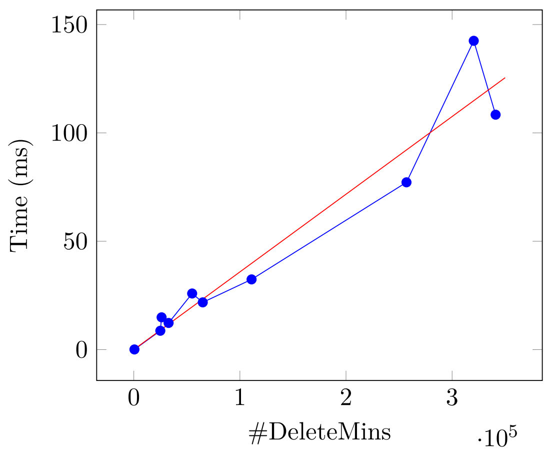


Figure 6.1: Per query runtime and number of delete minimum operations on the time-dependent network with route nodes. The red line shows a linear fit between runtime and DeleteMins.

Next we tried the Arrival Label Time Query (ALTQ) by Geisberger [Gei10], the Event Label Time Query (ELTQ) that we proposed, and the Connection Scan Algorithm (CSA) by Dibbelt et al. [DPSW13] on all networks. The descriptions of the algorithms can be found in Chapter 3, Sections 3.2.1 (ALTQ), 3.2.2 (ELTQ) and 3.1.2 (CSA).

Table 6.2 shows a comparison of all runtimes. The reported delete minimum operations are applicable to both ALTQ and ELTQ because those expand the exact same sequence of nodes. The number of DeleteMins is obviously much smaller than in the model by Pyrga et al. since we only have station nodes and no route nodes. CSA performance is very good especially for city-sized networks where it beats the priority queue based approaches due to short journeys that do not require many connections to be scanned. On national networks on the other hand it can be slower but remains competitive. The last column lists the speed up of ELTQ in comparison to Dijkstra's algorithm on the route node model; they range from a factor of 5 up to a factor of 25.

The ELTQ consistently performs around 20% better than ALTQ in all instances which is also a result of the large caches of our primary hardware. On our development hardware the difference was smaller, especially for small transfer times where the ALTQ overhead

Table 6.2: Comparison between Dijkstra's Algorithm, CSA, ALTQ and ELTQ on uncontracted public transit networks. The last column shows the speed up of our approach over Dijkstra's Algorithm on the route node model.

Network	Dijkstra [ms]	CSA [ms]	ALTQ [ms]	ELTQ [ms]	#DeleteMins (ALTQ/ELTQ)	Speed-up of ELTQ over Dijkstra
Germany	21.8	0.73	1.15	0.86	3,353	25
Europe	77.2	8.15	6.75	5.23	14,565	15
Sweden	108.4	10.68	15.30	11.82	26,425	9
Switzerland	142.5	7.84	8.93	6.67	13,417	21
VBB	32.4	1.17	2.89	2.23	6,579	14.5
Chicago	8.7	0.47	1.66	1.36	6,056	6.5
London	25.9	1.85	4.65	3.78	10,640	7
Madrid	14.9	0.87	0.97	0.79	2,367	19
New York	12.3	0.99	2.74	2.18	8,183	5.5

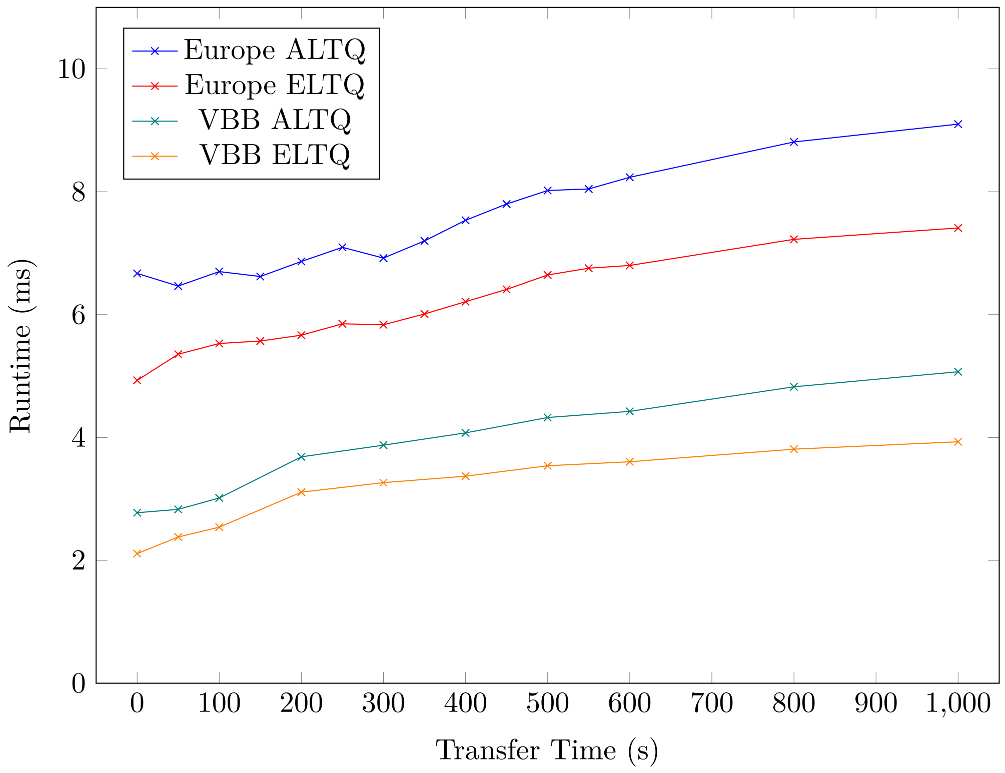


Figure 6.2: The effect of different transfer times on the runtime of ALTQ and ELTQ in the European and Berlin/Brandenburg network.

of linking and merging bags has less impact. A comparison between the two algorithms for different transfer times can be found in Figure 6.2. As expected higher transfer times also result in higher query times as more work per station has to be done. On our primary hardware, for both Europe and Berlin/Brandenburg the ratio between ALTQ and ELTQ runtime remained similar for transfer times between 0 and 1000 seconds.

6.4 Contraction

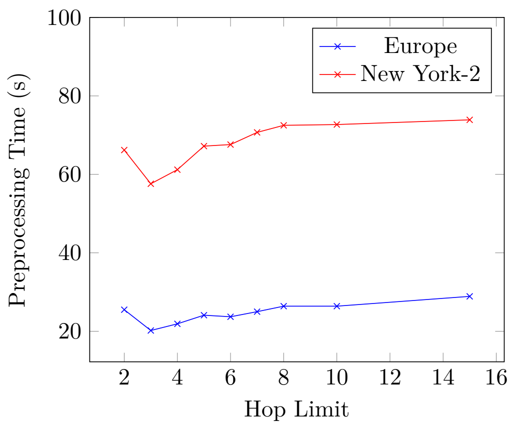
In the next experiment we contract all networks to achieve faster query times and we also look at ways to tune both the contraction parameters as well as the timetables themselves to achieve better results.

6.4.1 Parameter Tuning

Measuring contraction performance is not straightforward. Ultimately the goal is to reach the smallest possible query times, but we also want reasonable preprocessing times and not too much increase in the size of the final timetable. The most important adjustment we can make to improve the quality of the contraction – and therefore the one we focus on first – are the parameters of our priority calculation that determines the order in which the nodes are contracted. As discussed in Chapter 4 the formula we chose for the priority calculations is

$$p(v) = (\alpha * \frac{\text{\# of new shortcut edges}}{\text{\# of edges removed}}) + (\beta * \text{\# of new shortcut conn.}) + (\gamma * \mathit{depth}(v))$$

with positive parameters α, β and γ which we have to choose carefully. We call a triple (α, β, γ) a **parameter set** and our measure of performance for a parameter set is the



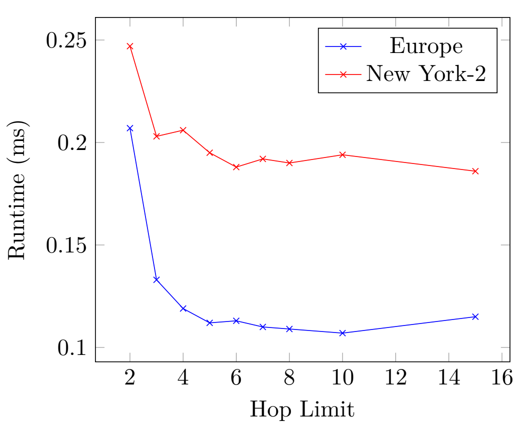


Figure 6.3: The effect of different hop limits on the preprocessing time (left) and on query runtimes (right). New York-2 is an updated version of the New York timetable and is explained in Section 6.4.2

average query runtime on a network contracted with these parameters. The parameter set suggested by Geisberger is $(10, 0, 1)$ but our tests showed it to perform poorly on all networks. The reason for this is the difference in how contraction is done in Geisberger's paper [Gei10]: They are doing multiple rounds of contraction where every round all nodes with a minimum priority in their 2-neighborhood are contracted. This way the order is naturally spread out over the whole timetable which keeps the hierarchy depth lower. It also has the benefit of making it easier to parallelize the algorithm because the contractions done in every round do not affect each other. We experimented with this approach but ultimately found that a higher factor γ for the hierarchy depth of a node had the same positive effect on the contraction quality.

Since contracting and then measuring performance requires a lot of time, we could not try too many possible parameter sets and do something like a gradient descent. Instead we chose a kind of genetic approach by generating a number of random parameter sets and measuring their quality, then keeping only the ten best sets that are not "too close" to each other and generate new parameter sets by walking in a random direction in the search space for each of them. We do this for several rounds, reducing the amount by which existing parameter sets are adjusted each round.

Using this method we searched for optimal parameters in the networks of Germany, Europe and VBB. The resulting parameter sets were quite different, showing that different networks have different requirements, but they also allowed us to find a good starting point for all networks by averaging the results. Our standard parameter set for all networks is $(5, 1, 200)$ placing the most importance on the hierarchy depth, then on the ratio of new edges to removed edges and lastly on the number of new shortcut connections created.

In order to improve preprocessing times we can adjust the hop limit of the witness search. The hop limit exists because we expect that witnesses for shortcut connections usually only follow a small number of edges and we want to avoid searching too much of the timetable for a witness that does not exist. However, since we perform one witness search for every shortcut connection in contrast to the one-to-many profile search for each incoming edge of the contracted stop done by Geisberger [Gei10] we expect the hop limit to have less of an effect on our witness search. In theory a lower hop limit causes some witnesses to be missed, resulting in more shortcuts and slower queries, but it also speeds up the witness search and thus reduces preprocessing time. Figure 6.3 shows the effect of different hop limits that we experimented with. For most networks a hop limit of 9 gives good results

Table 6.3: Contraction results, including increase in edges/connections and hierarchy depth of the resulting networks. Hop limit of the witness search was 9. Contraction time limit was 900 seconds and was reached by the four networks marked in red.

Network	Time [s]	Contraction	#Edges	#Conn.	Avg Depth	Max Depth
Germany	4.4	100%	+73.2%	+84.8%	1.43	36
Europe	28.6	100%	+70.1%	+85.5%	1.48	58
VBB	59.3	100%	+135.7%	+130.6%	2.48	96
New York	772	100%	+153.3%	+553.3%	6.9	262
Sweden	843	100%	+105.8%	+102.7%	1.71	99
Chicago	900	99.1%	+299.3%	+550.6%	5.76	167
London	900	90.9%	+122.5%	+310.2%	3.85	36
Madrid	909	98.5%	+239.5%	+231.4%	5.51	102
Switzerland	1,069	97.3%	+117%	+94.2%	2.59	37

and as expected we noticed little difference when using higher hop limits since our witness search already naturally terminates early in many cases.

The last thing we can tune is the amount of nodes that we contract. Usually the contraction of nodes that require a large amount of shortcuts and introduce many new edges is delayed until very late during the contraction process. On networks that are not well suited for contraction these remaining nodes sometimes form tight clusters that result in an enormous increase of shortcut edges and connections when they are finally contracted. This not only makes the contraction of the last nodes very slow it can also have a negative impact on query times as the timetable becomes too large and inefficient.

Since our algorithms also work without modification on partially contracted networks we can simply stop before all nodes are contracted. The query performance on these partially contracted networks is often faster than on the fully contracted network and the contraction itself takes much less time since most of the heavy work is done during the contraction of the last few hundred nodes. We can use three limits to stop our contraction before it has contracted 100% of the nodes: (1) Stop when a certain time limit is reached, (2) stop when a certain number of new edges have been created, expressed as a percentage of the original edges in the graph, and (3) stop when a certain number of new connections has been created, also expressed as a percentage of the original. We could also directly limit the number of nodes that may be contracted in case the other limits do not provide good results.

In general there are no optimal parameters that give the best result for all networks. Every contraction needs to be carefully tuned to provide the best results. However, once good parameters for a network have been found it is unlikely that they will change much in the future – even when train schedules change – since major changes in the structure of a public transit network require a lot of time, effort and money and are usually a very slow process in the real world.

Table 6.3 shows our contraction results for all timetables. We limited total preprocessing time to 900 seconds which resulted in four of the networks getting only partially contracted. The limit was checked after the contraction of every node and thus actual values can be slightly above the limits. In the example of Switzerland the contraction of the last node before we stopped took more than 170 seconds by itself and contracting 100% of the nodes would have been infeasible. The witness search uses our standard parameter set $(5, 1, 200)$ and a hop limit of 9.

The contraction works very well on Germany, Europe and Berlin/Brandenburg and takes less than a minute on all three. The amount of created shortcut edges and shortcut

Table 6.4: ALTQ and ELTQ performance per query on the contracted networks

Network	ALTQ [ms]	Spd-up	ELTQ [ms]	Spd-up	#DeleteMins	Factor
Germany	0.060	19	0.036	24	57	59
Europe	0.171	39	0.112	47	129	113
VBB	0.338	8.5	0.241	9.3	214	31
New York	1.34	2.0	1.02	2.1	339	24
Sweden	0.426	36	0.267	44	227	116
Chicago	0.94	1.8	0.70	1.9	250	24
London	2.23	2.1	1.66	2.3	1,408	7.5
Madrid	0.351	2.8	0.238	3.3	132	18
Switzerland	1.67	4.7	1.10	6.1	734	18

connections as well as the average and maximum hierarchy depth can give a sense of how good the performance of time queries on the contracted timetable will be. Again Germany, Europe and Berlin/Brandenburg as well as Sweden are promising as their size does not grow too large and their hierarchies are flat. On the other hand the city-sized networks perform notably worse. For most of them the contraction reached the 900 second time limit and was stopped before the network was fully contracted. Their large increase in edges and connections and their higher average depth promise only small gains from the contraction. In the next section we first look at actual time query performance before addressing the problems with those timetables.

6.4.2 Time Queries

Table 6.4 shows the runtime of ALTQ and ELTQ on the contracted timetables along with the speed-up in comparison to the respective uncontracted runtime. Once again the number of delete minimum operations are the same for both algorithms so they are listed only once together with the factor by which they were reduced. We can see that the performance is very good on the timetables of Germany, Europe and Sweden, decent on Berlin/Brandenburg and not very good for the rest of the networks as we expected from the contraction results. Interestingly enough Switzerland also performs notably worse despite being a country-sized network where we expect better results than on city-sized networks.

A closer look reveals that there is a correlation between the number of footpaths and the quality of the contraction. All timetables that perform badly also have a lot of footpaths whereas the two networks that perform best, Germany and Europe, have no footpaths at all. There are two problems related to the high number of footpaths. The first is that they need to be transitively closed as explained in Section 2.3, so that when footpaths $u \to v$ and $v \to w$ exist then $u \to w$ must also exist. This means that large connected components require many additional footpaths. In fact, our data sets come from real world data that contained fewer footpaths and additional footpaths had to be inserted to create the transitive closure. The second problem is that some of the timetables are too detailed and are modeling a large train station as a number of nodes for every platform with footpaths in between. In London almost all stations are modeled as two nodes with a footpath between them. This is something that the station model wants to specifically avoid and having only one node for a larger train station with a station-specific transfer time is preferred.

We can address this by combining nodes that are connected by footpaths into one station node. We start by finding cliques that are completely connected by footpaths with less than half our global transfer time (so 150 seconds for countries and 90 seconds for cities).

Table 6.5: Size of the updated networks after combining station nodes with short footpath distance.

Network	#Stops	#Edges	#Footpaths
Chicago-2	7,493 (-38%)	12,465 (-49%)	456 (-96%)
London-2	11,649 (-44%)	24,660 (-65%)	18,170 (-60%)
New York-2	9,577 (-40%)	16,469 (-72%)	15,010 (-66%)
Switzerland-2	24,087 (-8%)	56,580 (-26%)	1,707 (-87%)

Table 6.6: Contraction results on the updated networks with combined station nodes.

Network	Time [s]	Contraction	#Edges	#Conn.	AvgDepth	MaxDepth
New York-2	70.6	100%	+157.3%	+137.5%	2.53	118
Chicago-2	71.3	100%	+186.5%	+160.7%	2.59	124
London-2	385	100%	+175%	+152.3%	3.93	170
Switzerland-2	983	99.1%	+102%	+81.7%	2.00	47

Table 6.7: Performance of ELTQ on the regular and contracted networks with combined station nodes. Numbers in brackets show how the speedup improved in comparison to the original unchanged timetables.

	Regular		Contracted			
Network	ELTQ		ELTQ			
	[ms]	#DeleteMin	[ms]	Speedup	#DeleteMin	Factor
New York-2	1.44	4,932	0.19	7.6 (+5.5)	133	37 (+13)
Chicago-2	1.03	3,813	0.18	5.7 (+3.8)	118	32 (+6)
London-2	2.54	6,111	0.53	4.8 (+2.5)	233	26 (+18.5)
Switzerland-2	6.13	12,383	0.42	14.6 (+8.5)	333	37 (+19)

There are efficient algorithms for finding *maximum cliques* in a graph [TT77, Rob86], but for our purposes a simple linear algorithm for finding *maximal* cliques proved good enough.

Combining stations was done for Switzerland, Chicago, London and New York. Table 6.5 shows how it affected the size of the timetable and Table 6.6 shows contraction results for those four networks. Finally Table 6.7 shows results for ELTQ on the uncontracted and contracted networks and the new speed-up that we achieve. The timetables shrink considerably and the quality of the contraction increases significantly underlining the importance of matching the available data appropriately to the station model. The only remaining timetable with poor Contraction Hierarchy performance is Madrid where we could not find any changes that gave a notable boost to contraction quality. The most likely cause for this is that the Madrid data consists entirely of bus lines which are not ideal for Contraction Hierarchies as they can follow the road network in a mostly direct way to their target whereas trains are bound to the rails and as such are more structured and require fewer shortcut connections.

6.5 Profile Queries

We test our profile query algorithm on both the uncontracted and the contracted networks. Our timings are the average runtime of 1,000 queries and we try one-to-one, one-to-ten and one-to-100 queries with randomized source and target stations and with a time interval of $[0, \pi)$ covering the whole period of the timetable which is 24 hours. The profile query implementation that was tested gives a list of (departure time,arrival time) pairs for each target station, so that for any departure time t the pair with the smallest departure time

Table 6.8: Average runtime in milliseconds per query for 1-to-1, 1-to-10 and 1-to-100 profile queries on the uncontracted networks (column *Reg.*) and the contracted networks (column *Con.*). The table is sorted by the speed-up that was achieved by contracting the timetable which is shown in the *Spd.* columns.

	1-to-1			1-to-10			1-to-100		
Network	Reg. [ms]	Con. [ms]	Spd.	Reg. [ms]	Con. [ms]	Spd.	Reg. [ms]	Con. [ms]	Spd.
Europe	36.89	0.70	47.2	59.61	1.77	33.7	62.65	4.24	14.8
Sweden	122.63	3.35	36.6	176.82	7.07	25.0	175.22	15.33	11.4
Germany	11.30	0.38	29.7	17.35	1.18	14.7	17.61	3.33	5.3
Switzerland-2	106.92	7.59	14.1	153.72	12.78	12.0	153.26	20.84	7.4
New York-2	88.19	9.64	9.1	150.50	22.39	6.7	167.08	41.71	4.0
VBB	35.02	4.61	7.6	48.59	7.82	6.2	46.57	13.54	3.4
London-2	245.35	39.21	6.3	425.35	75.11	5.7	452.86	135.49	3.3
Chicago-2	58.51	9.58	6.1	102.28	19.05	5.4	106.14	34.32	3.1
Madrid	93.45	25.05	3.7	149.87	49.85	3.0	157.51	90.41	1.7

$\geq t$ at the desired target station provides the earliest arrival for a time query from the source to this target at time t. It does not generate the consistent paths that must be taken to achieve these arrival times, which would add some amount of overhead.

Table 6.8 shows the results of one-to-one and one-to-many profile queries. Speedups are especially good for one-to-one queries; for one-to-many queries the overhead of storing the results becomes larger with more targets and reduces the benefits of the contraction.

6.6 Comparison to the Original Approach

We want to compare our results to the previous work on Contraction Hierarchies. We will be using the numbers reported by Geisberger that can be found in [Gei11] which are the most recent and were done on slightly faster hardware than the numbers reported in [Gei10]. His experiments were done on "one core of an Intel Xeon X5550 processor (Quad-Core) clocked at 2.67 GHz with 48 GiB of RAM and 8 MiB of Cache running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3." [Gei11]

There are two timetables used by both us and Geisberger: (1) The timetable for Europe which is exactly identical to ours except that we removed some unnecessary connections when making the timetable periodic. (2) The timetable for Berlin/Brandenburg where we cover roughly the same area since both data sets come from the same source but we use data from the winter period of 2014 whereas the original data set is from 2000/01. The difference is a slightly higher number of stations and a significantly larger number of connections in our data set. Table 6.9 shows the differences in more detail.

Table 6.9: Comparison between the data sets used by Geisberger [Gei11] marked with a *G* and our data sets that are named the same as in previous sections.

Network	#Stations	#Edges	#Connections
G-Europe	30,517	88,091	1,669,666
Europe	30,517	87,818	1,639,869
G-VBB	12,069	33,473	680,176
VBB	13,054	35,975	1,096,510

Table 6.10: Comparison between Geisberger's results and our results with and without preprocessing. Slower runtimes in Geisberger's case are a result of slower hardware and runtimes are not directly comparable. Time speed-ups refer to ALTQ over Dijkstra for Geisberger's data and to ELTQ over Dijkstra for our data. The two rows of preprocessing for Geisberger's data sets were achieved with a hop limit of 7 for the upper row and a hop limit of 15 for the lower row.

		Preprocessing		Time Queries				Profile Queries	
		Time		ALTQ	ELTQ	Time	#Del.	Time	Time
Data	Impl.	[s]	#Edges	[ms]	[ms]	Spd.up	Mins	[ms]	Spd.up
Europe	[Gei11]	—	—	9.4	—	—	14,504	242.0	—
		210	+88%	0.251	—	37.5	192	3.7	65.1
		619	+86%	0.216	—	43.5	183	3.4	71.4
Europe	ours	—	—	8.15	6.75	—	14,565	36.9	—
		29	+70%	0.171	0.112	47	129	0.70	47.2
VBB	[Gei11]	—	—	4.0	—	—	5,969	215.0	—
		216	+135%	0.544	—	7.3	207	27.0	8.0
		685	+128%	0.434	—	9.2	186	24.2	8.9
VBB	ours	—	—	2.89	2.29	—	6,579	35.0	—
		59	+136%	0.338	0.241	9.3	214	4.61	7.6

Table 6.10 shows a comparison for time and profile queries on both contracted and uncontracted timetables. The uncontracted timetables are those without preprocessing information. The time speed up for time queries refers to ALTQ on Geisberger's data sets and to ELTQ on our data sets. Geisberger used two different preprocessing settings, one with a hop limit of 7 and one with a hop limit of 15. In his case the hop limit had a strong effect on the preprocessing time whereas for us it makes little difference. The quality of the contraction did slightly improve for larger hop limits.

We can directly compare the number of delete minimum operations of the priority queue and to some degree the speed-up values because they depend more on the algorithm than on the hardware. For time queries our algorithms and contraction parameters seem to perform slightly better on Europe where we get 30% fewer DeleteMins and a higher speed-up, whereas on Berlin/Brandenburg the values are quite similar to those of Geisberger. Although our absolute number of delete minimums for contracted VBB is higher we also had a higher number in the uncontracted case due to our slightly larger data set and the ratios are quite close ($214/6579 = 0.0325 \doteqdot 0.0311 = 186/5969$). Profile queries gain much faster speed-ups for Geisberger but when looking at the actual runtime we are faster by a factor of ~5.

When we try to account for hardware differences it is best to compare the ALTQ values for both uncontracted and contracted Berlin/Brandenburg where the time speed-up and the factor by which delete minimums were reduced were most similar. We divide our result for uncontracted and contracted time queries on VBB (2.89ms and 0.338ms) by the best results by Geisberger (4.0ms and 0.338ms). After adding a little leeway this suggests that Geisberger's runtimes can be multiplied by 0.65–0.75 to gain a rough estimation by which we can compare the rest of the results. Looking at preprocessing times, Geisberger achieved better results with a hop limit of 15 but the preprocessing time became greater than 10 minutes which, when adjusted by a factor of 0.65–0.75, is still more than 5 times slower on Berlin/Brandenburg and 10 times slower on Europe compared to our approach. Using the faster result with hop limit 7 still results in a bit over a minute for Geisberger whereas our algorithm takes less than a minute in both cases. Our profile queries also seem to be a lot

Table 6.11: Using a heuristic to add footpaths to the timetables, stations were connected with 70% probability to other stations within 300m radius assuming a random walking speed between 3-6km/h. The table shows the amount of added footpaths and redundant footpaths that were removed afterwards.

Network	Created footpaths	Redundant footpaths	Network	Created footpaths	Redundant footpaths
Germany	100	1,631	VBB	4,627	2,254
Europe	585	10,327	London-2	40,348	21,522
Sweden	21,089	5,426	Chicago-2	27,488	4,838
Switzerland-2	18,672	3,811	New York-2	36,197	8,737

faster even when accounting for hardware differences. At the very least all results seem to be no worse than Geisberger's results and most promise to be faster.

6.7 Multimodal Time Queries

We first try a simple scenario to verify the performance and correctness of our multimodal witness search. For every footpath f in the timetable we check if there is a series of footpaths $(f_1, f_2, ..., f_k)$ so that $\sum_{i=1}^{k} length(f_i) = length(f)$ and if this is the case then we remove f from the timetable since it is unnecessary and was probably added to create the transitive closure of the footpaths. We extend our priority weights by another parameter δ that weighs the importance of necessary footpath shortcuts so our formula becomes:

$$p(v) = (\alpha * \frac{\#\text{ new edges}}{\#\text{ edges removed}}) + (\beta * \#\text{ new conn.}) + (\gamma * depth(v)) + (\delta * \#\text{ new footpaths})$$

We then run a contraction with the parameter set $(\alpha, \beta, \gamma, \delta) = (5, 1, 200, 8)$ which is our standard parameter set extended with $\delta = 8$ which was chosen through experimentation with a couple of different values. As expected our contraction worked as before with almost exactly the same results and no impact on preprocessing times. The difference to the previous results was less than 5 percent for all timetables which is why we don't list these results in a separate table. That means that our multimodal witness search is good enough to be used in scenarios where our input data contains footpaths that are not transitively closed and possibly connect larger groups of nodes.

For our second experiment we use a heuristic to generate additional foot paths between stations that are close together and we also assign a walking duration to all edges that already exist in the timetable. Normally this would be done by overlaying a road/walking network over the station graph, connecting each station to its closest node in the road network by geographical distance and then searching walking distances in the road network. We are using a simplified approach however, and generate foot paths simply by assigning a walking duration that is based on air-line geographical distance (our data contains geo-coordinates for every station) and a random walking speed between 3km/h and 6km/h for each individual edge. We connect all stations within a distance of 300m to each other with a probability of 70% to avoid creating fully connected components. Table 6.11 shows how this affects the size of the timetable. Except for Germany, Europe and VBB the number of inserted footpaths is quite large. The number of redundant footpaths depends on randomness as it is caused by the differing walking speeds making some of the newly created footpaths immediately obsolete because they are dominated. Since we also assign walking distance to existing edges there can even be more footpaths removed than were added.

Table 6.12: Contraction results on the timetables that were combined with random footpaths. The time queries section lists runtime for both the uncontracted and the contracted network and the speed-up that was achieved. The hop limit was set to 9 and the time limit to 1200 seconds.

	Preprocessing					Time Queries [ms]		
Network	Time [s]	#Nodes contr.	#Edges	#Conn.	#Foot-paths	ELTQ regular	ELTQ contr.	Spd. up
Germany	10.6	100%	+74%	+89%	+55%	0.84	0.036	23
Europe	48.8	100%	+71%	+93%	+50%	5.2	0.119	44
Sverige	539	100%	+120%	+190%	+86%	11.0	0.69	16
Switzerland-2	1695	97%	+122%	+127%	+79%	5.8	0.80	7.2
VBB	154.2	100%	+160%	+215%	+92%	1.8	0.40	4.5
New York-2	1200	92.3%	+190%	+704%	+93%	1.53	0.89	1.7
Chicago-2	1200	94.2%	+212%	+814%	+111%	1.13	0.64	1.8
London-2	1200	93.7%	+170%	292%	+95%	2.55	0.89	2.9

We then contract these timetables with a time limit that was increased to 1200 seconds to accommodate their larger size. The results can be found in Table 6.12. Once again the contraction works well on Germany, Europe and Sweden but that is to be expected since footpaths should have very little impact on long-distance trains. More interesting are the results for the city-sized networks.

Switzerland and Berlin/Brandenburg still have somewhat decent speed-ups but the other instances perform worse. On New York and Chicago the number of connections increases by more than 700% even though less than 95% of the nodes were contracted which is much higher than previous experiments. There are still speed-ups gained from the contraction but they are only within a factor of two and much lower than previous speed-ups.

7. Conclusion

This thesis is based heavily on previous work on Contraction Hierarchies for public transit networks done by Geisberger [Gei10, Gei11]. We have implemented the existing ideas and algorithms for time queries on public transit networks and compared our results to the previous ones. We then introduced new algorithmic ideas and combined them with approaches adapted from research on profile queries in transportation networks [DKP12] in order to provide alternative algorithms for time queries, profile queries and the contraction of the graph itself.

Our experiments have shown that our approach has the potential to perform better than previous solutions on appropriate input data. While time query speed-ups were not significantly faster, both preprocessing time and profile queries showed promising results. Furthermore the Event Label Time Query was simpler to implement and never performed worse than the time query algorithm proposed by Geisberger [Gei10, Gei11] which we call Arrival Label Time Query in this work.

We have tested both existing and new algorithms on a larger variety of timetables than was previously done. Especially our analysis of the influence of footpaths is far more complete. We discussed the problems that arise when applying our approaches to city-sized networks and proposed solutions. Converting the available transportation data by combining stations into a more contraction-friendly graph showed that speed-ups of up to an order of magnitude can still be achieved on such networks when the parameters and structure of the network are carefully tuned.

The extension of our witness search to time-independent witnesses removed the requirement for footpaths to be transitively closed. This allows more complex footpath networks to be added to timetables and possibly makes it easier to use realistic real-world data for the algorithms. However, a fully multimodal combination of public transit and walking networks that connected all nodes in the graph through footpaths did not work well with Contraction Hierarchies. This was caused by the large increase in size of the graph and the relatively little structure that existed among the footpaths.

7.1 Future Work

We only did a few simple multimodal experiments and we did not use real road data for the walking network. This is an area where further research could be done to perform experiments with more realistic and more structured data. An open question is also the feasibility of Contraction Hierarchies when multi-criteria multimodal queries are considered.

Another idea that still needs to be explored is the effect of the day time interval introduced in Chapter 5 on the performance of Contraction Hierarchies in combined public transit and walking networks based on realistic data. In our preliminary experiments it unfortunately showed very little improvement but that might be caused by the already suboptimal structure of our generated footpath networks.

Lastly it could be possible to contract public transit networks only partially and combine Contraction Hierarchies with other speed-up techniques that are then applied to the core of the network.

Bibliography

[Bau06] Reinhard Bauer. Dynamic Speed-Up Techniques for Dijkstra's Algorithm. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2006.

[BBM06] Maurizio Bielli, Azedine Boulmakoul, and Hicham Mouncif. Object modeling and path computation for multimodal travel systems. *European Journal of Operational Research*, 175(3):1705–1730, 2006.

[BBS13] Hannah Bast, Mirko Brodesser, and Sabine Storandt. Result diversity for multimodal route planning. In *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems-2013*, volume 33, pages 123–136. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013.

[BDS$^+$08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, June 2008.

[BDSV09] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 175–192. American Mathematical Society, 2009.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[BGS08] Gernot Veit Batz, Robert Geisberger, and Peter Sanders. Time Dependent Contraction Hierarchies - Basic Algorithmic Ideas. Technical report, ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH), 2008.

[BJ04] Gerth 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'03)*, volume 92 of *Electronic Notes in Theoretical Computer Science*, pages 3–15, 2004.

[DDP$^+$12] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing and Evaluating Multimodal Journeys. Technical Report 2012-20, Faculty of Informatics, Karlsruhe Institute of Technology, 2012.

[DDP$^+$13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing Multimodal Journeys in Practice. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DKP10] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–12. IEEE Computer Society, 2010.

[DKP12] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. *ACM Journal of Experimental Algorithmics*, 17(1):4.4:4.1–4.4:4.26, July 2012.

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013.

[DPW12] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140. SIAM, 2012.

[Gei08] Robert Geisberger. Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf`.

[Gei09] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. Technical report, ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH), 2009.

[Gei10] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer, May 2010.

[Gei11] Robert Geisberger. *Advanced Route Planning in Transportation Networks.* PhD thesis, Karlsruhe Institute of Technology, February 2011.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[MS09] Matthias Müller–Hannemann and Mathias Schnee. Efficient Timetable Information in the Presence of Delays. In *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2009.

[MSS$^+$06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11(2.8):1–29, 2006.

[MW01] 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'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

[Nac95] Karl Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154–166, 1995.

[PIA78] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search – a log log n search. *Commun. ACM*, 21(7):550–553, July 1978.

[PS98] Stefano Pallottino and Maria Grazia Scutellà. Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects. In *Equilibrium and Advanced Transportation Modelling*, pages 245–281. Kluwer Academic Publishers Group, 1998.

[PSWZ08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

[Rob86] J.M Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425 – 440, 1986.

[SS07] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.

[TT77] Robert Endre Tarjan and Anthony E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[Vet09] Christian Vetter. Parallel Time-Dependent Contraction Hierarchies. Technical report, Karlsruhe Institute of Technology, 2009. `http://algo2.iti.kit.edu/download/vetter_sa.pdf`.