Masterarbeit

On Compact Representation and Robustness of Transfer Patterns in Public Transportation Routing

Jonas Sternisko

27. März 2013

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Work period
October 2012 – March 2013

Supervisor
Prof. Dr. Hannah Bast
Prof. Dr. Christian Schindelhauer

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Abstract

Transfer pattern routing is a state-of-the-art routing algorithm for public transportation. It allows to answer multi-criteria shortest path queries within a few milliseconds. The algorithm is based on an expensive precomputation of shortest paths between stations. This thesis examines two aspects of transfer pattern routing. The first objective is to find a compact representation of the transfer patterns. We detect redundancy in the transfer patterns and present novel ways to store the information within less memory. The second objective is to analyze the robustness of the transfer patterns towards delay and to answer the question whether the patterns still allow for optimal routing, if the underlying network changes. We address the second question by introducing a framework to evaluate the robustness in networks with different delay scenarios and we present experimental results indicating that transfer patterns are quite robust.

Zusammenfassung

Transfer Pattern Routing ist gegenwärtig einer der effizientesten Algorithmen zur Suche von kürzesten Wegen im öffentlichen Personenverkehr. Das Verfahren basiert auf einer aufwändigen Vorberechnung von kürzesten Wege in Verkehrsnetzen. Gegenstand dieser Arbeit sind zwei mit dieser Information verbundene Problemstellungen. Zum einen wird untersucht, wie die Größe der berechneten Information minimiert werden kann. Wir entwickeln neue Möglichkeiten die Information der Transfer Patterns kompakt darzustellen und in weniger Speicher abzubilden. Zum anderen wird die Robustheit der berechneten Transfer Patterns untersucht und eine Antwort auf die Frage gegeben, ob deren Information auch unter Echtzeit-Veränderungen des zugrunde liegenden Netzwerkes noch Gültigkeit besitzt. Wir stellen ein Framework vor, mit dem die Robustheit des Algorithmus untersucht werden kann und präsentieren Ergebnisse, welche die Robustheit von Transfer Patterns belegen.

Acknowledgments

I wish to thank, first and foremost, my Professor Hannah Bast for supervising this thesis. Her outstanding lecture on “Efficient Route Planning” awoke my interest in route planning algorithms in the first place and her helpful advice during the course of this project pushed ahead this work. No less I appreciate Professor Christian Schindelhauer’s agreeing to review this thesis as second supervisor.

Furthermore, I thank my proofreaders Leticia Pfeifer and Manuel Braun for all their help in fine-tuning this text. I would like to thank Mirko Brodesser for the helpful exchange of ideas we had in the last six months.

Last but not least, I wish to thank Carolin Baer for mentally supporting me and respecting the increased workload during the last weeks.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 2

2 Routing with Transfer Patterns **5**
- 2.1 Graphs, Shortest Paths and Dijkstra's Algorithm . . . 5
- 2.2 Modeling Timetable Data As Graph . . . 6
  - 2.2.1 General Transit Feed Specification GTFS . . . 6
  - 2.2.2 Time Expanded Graph . . . 7
  - 2.2.3 Walking between Stations . . . 8
  - 2.2.4 Location-to-Location Queries . . . 9
  - 2.2.5 Used Datasets . . . 9
- 2.3 Shortest Path Problem in Public Transportation . . . 9
- 2.4 Transfer Pattern Routing . . . 12
  - 2.4.1 Computing Transfer Patterns . . . 12
  - 2.4.2 Exploiting Optimal Transfer Patterns for Search . . . 16
  - 2.4.3 Heuristics . . . 17

3 Compact Representation of Transfer Patterns **19**
- 3.1 A Less Informed Approach: First-Transfer Routing . . . 19
  - 3.1.1 Idea . . . 19
  - 3.1.2 Implementation . . . 21
  - 3.1.3 Results . . . 22
- 3.2 Reducing Redundancy in Directed Acyclic Graphs . . . 23
  - 3.2.1 Isomorphic Reductions . . . 24
  - 3.2.2 Shared Entry-Points . . . 25
  - 3.2.3 Joint DAG . . . 26
- 3.3 Further Ideas . . . 29
- 3.4 Results . . . 29
- 3.5 Summary . . . 35

4 Robustness of Transfer Patterns **37**
- 4.1 Transfer Patterns and Real-Time Updates . . . 37
- 4.2 Measuring Robustness . . . 37
- 4.3 Modeling Delay . . . 38
- 4.4 Experiments . . . 39

5 Conclusion **47**
5.1 Compact Representation of Transfer Patterns 47
5.2 Robustness of Transfer Patterns 48

Bibliography **49**

1 Introduction

1.1 Motivation

In the recent past, mobility has become more and more important. People travel over long distances much more often than 20 years ago. This evolution went in parallel with the evolution of public transportation networks: The world-wide area served with public transportation grows and services become increasingly flexible and fine-grained. High-speed or local trains, subways, trams and buses of various sizes serve areas with capacity and frequencies adapting to the customers' needs. Journeys often lead the traveler to territory previously unknown to him. The growing spread of mobile devices in the last years increased the demand for route planning applications for public transportation. Such applications have to respond to thousands of queries per second. Thus, the answer to a single query has to be found within very short time.

From a computer scientist's point of view, finding shortest paths in public transportation networks differs from road networks, where efficient algorithms have been developed during the late 90's and early 00's. The reasons lay within the different structure of the underlying networks and the fact that the quality of a route is not only determined by its duration, but also by the total fare and the number of transfer. Hannah Bast et al. introduced transfer pattern routing [2], which allows to find multi-criteria shortest paths in real-time even for huge transportation networks.

A transfer pattern of a connection is the sequence of stations where the transportation vehicle is changed. For example, traveling from Freiburg to Munich by train is either possible by taking an ICE with one transfer at Karlsruhe or with a series of local trains with transfers in Titisee and Ulm. Furthermore, there is a direct bus between the two cities. The transfer patterns for this connection are *Freiburg–Karlsruhe–Munich*, *Freiburg–Titisee–Ulm–Munich* and *Freiburg–Munich*.

In short, the algorithm exploits the observation that optimal paths between two places, independent of the departure time, always follow a limited set of optimal transfer patterns. Once these patterns are determined, queries for shortest paths can be answered in a few milliseconds. The algorithm has two key components: The computation of the optimal transfer patterns and a data structure for efficient answer of direct connection queries. Transfer pattern routing works well for realistic public transportation routing. For example, it is employed by *Google Transit*. However, it has some important drawbacks.

One critical aspect is to represent the transfer patterns within a compact data structure. Unlike personal navigation devices for route planning in road networks, this information has a size such that it has to be provided by a server responding to queries from remote clients. The size of the patterns plays a role for the required hardware and cost of the route planning system. It is also important for access speed: Small data can be cached efficiently, which increases the overall performance of the search. Furthermore, current research tries to apply transfer pattern routing for more complicated, namely multi-modal shortest path problems. Here, a shortest path has to be found in a graph representing more than one mode of transportation and there is a higher order of variation among the shortest paths. Thus, there are more optimal paths between two places and so the amount of information increases. Finding a more *compact representation* for the transfer patterns is the first objective of this thesis.

An even more crucial problem of the algorithm is the time-consuming precomputation to discover the optimal transfer patterns in a transit network. Computing the transfer patterns every time the underlying timetable data changes is impracticable. But in reality, timetables are never met with perfection: There is always delay, because of traffic jams, maintenance of tracks or other reasons. Given a steady flow of real-time updates to the transit data, it is questionable if transfer pattern routing still allows for optimal responses. The second aspect this thesis addresses is about an important property of transfer patterns–*robustness*: Does transfer pattern routing, once the patterns are computed, correctly answer shortest path queries in the presence of real-time updates to the transit network?

This introduction is followed by an overview about research related to this thesis. In chapter 2 we describe transfer pattern routing and our implementation in detail. Thereafter, we present different approaches and results for a more concise representation of transfer patterns in chapter 3. Afterwards chapter 4 reports about our investigations on the robustness of transfer patterns. The text concludes by a summary of our results and an evaluation of their relevance and contribution.

This thesis has two different parts, because during research on the first topic we found out that in the first publication on transfer pattern routing [2] some undocumented improvement must have been applied in order to achieve the reported data sizes. Considering this, the problem of the data size is just not as critical as it seemed to when the work on this thesis started.

1.2 Related Work

Pyrga et al. [17] give an overview over common graph models for timetable information enabling Dijkstra-based routing in public transportation networks. The authors compare two important graph representations, namely the *time-expanded*

and the *time-dependent* graphs. Hannah Bast [1] describes the differences between route planning in road networks and for public transportation and why the latter is harder.

The notion of transfer patterns has been introduced by Hannah Bast et al. in 2010 [2]. The authors explain the basic principle of answering queries based on precomputed information. They describe a fast algorithm to answer direct connection queries using timetable information. The precomputation of transfer patterns is explained and various heuristics to make it feasible are presented, most notably the notion of hub stations. The authors show how to answer shortest path queries using the precomputed data and the direct connection query algorithm in a few milliseconds. An analysis of the computational effort of the precomputation, of the size of the computed data and of the query times is provided. Robert Geisberger, one of the authors, presents a more detailed description of the underlying techniques in his PhD thesis [10]. He analyzes how walking between stations can be modeled in order to allow for realistic routes. Furthermore, he elaborates the incorporation of hub stations into the precomputation and search.

Recently, some surveys have been made how transfer pattern routing can be applied for multi-modal route planning. In his master's thesis, Manuel Braun studies route-variations during the precomputation of transfer patterns [3]. He identifies some reasons and suggests models and heuristics to decrease the variations. In another (yet unpublished) master thesis, Mirko Brodesser approaches the problem with a different model [4]. He tries to reduce the computational effort of the precomputation by a combination of the road network with a flattened graph representing the transportation network.

Graph and tree data structures play an important role in computer science. The first part of this thesis is about graph compression. Efficient ways to represent graphs have been discussed by Turán [21], Kannan et al. [13] and Jacobson [12]. The set of transfer patterns can be seen as dictionary of words, which can be stored as a *trie*. Indeed, the data structure used to store the patterns is similar to a trie. The graph variant of this is known as Minimal Acyclic Finite State Automaton, for which Daciuk et al. [6] introduced an algorithm to compress equal suffixes at construction time, or Directed Acyclic Word Graph, for which Chrochemore and Vérin [5] proposed a compact form.

The second part of this thesis investigates how an existing route planning algorithm can handle stochastic delay. In his diploma thesis, Ben Strasser encounters the problem of delay in transit networks from a different point of view [18]. He introduces a stochastic algorithm which solves a minimum expected arrival time problem and suggests reliable alternative routes.

2 Routing with Transfer Patterns

This thesis contributes to the knowledge about transfer pattern routing in two ways. Before we introduce the actual advances, this chapter elaborates the foundations of transfer pattern routing and our implementation, which is required to comprehend the remainder of this document. In the following, we give fundamental definitions for route planning in general. As the chapter proceeds, topics become more specific to public transportation routing. Finally, we explain the components of transfer pattern routing and details of our implementation.

2.1 Graphs, Shortest Paths and Dijkstra's Algorithm

Graphs In this thesis graphs play an important role. We will later model timetable information as a graph and we will store information in graphs. A *graph* $G = (V, E)$ is composed of a set of *nodes* or *vertices* V and a set of *edges* E. Each edge $e \in E$ connects two vertices $e = (v_1, v_2)$, $v_1, v_2 \in V$. A graph is directed, if $(v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$ does not hold. A graph is cycle free, iff no sequence of arcs $(v_1, v_2), (v_2, v_3) \dots (v_{k-1}, v_k)$ with $v_1 = v_k$ exists. A graph with these two characteristics is called *directed acyclic graph* DAG. We refer to the edges of a directed graph as *arcs* and denote the nodes connected by such an arc e as startpoint $e[0]$ and endpoint $e[1]$. In a *weighted graph,* edges have a certain *weight* or *cost* $c(e)$ associated to them. This cost can be scalar or vector-valued.

Shortest Paths A *path* between two nodes v_1 and v_n in a Graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \dots, v_n$ successively connected by arcs $(v_i, v_{i+1}) \in E$. Likewise, a path can be seen as a sequence of arcs $e_1, e_2, \dots, e_n$ such that each arc starts from the node its predecessor pointed to: $e_i[0] = e_{i-1}[1]$ for $i = 1 \dots N$ and $e_1[0] = v_1$ and $e_n[1] = v_n$. The cost of a path is defined as the sum of the arc costs along the path. The shortest path between two vertices v_s and v_t in a graph is a path that starts at v_s and ends at v_t with costs less or equal to the costs of all other paths between these two nodes.

Dijkstra's Algorithm The problem of finding a shortest path in a graph with non-negative edge weights was first solved by Edsger W. Dijkstra [8]. Starting at a source node with distance 0 and distance ∞ for all other nodes, the algorithm

iteratively settles the node with the smallest tentative distance. The node's outgoing arcs are relaxed and the tentative costs of each connected node is updated with the cost of the node being settled plus the cost of the respective arc. Unsettled nodes with tentative costs other than ∞ are maintained in a queue. This is repeated until either the target node is settled or the queue is empty. In the latter case, there is no path from the source to the target node. The optimal path can be reconstructed by storing a pointer from each node to its parent. The parent is the node which caused the child node's last cost update. The optimal path is extracted by traversing the parent pointers from the destination.

In combination with an appropriate implementation of a priority queue, this algorithm solves shortest path problems within time $O(|E|+|V|\,log\,|V|)$ [9]. The term *Dijkstra's algorithm* commonly refers to that priority queue based implementation. It constitutes the base for many path exploration and search algorithms to be covered later on.

2.2 Modeling Timetable Data As Graph

After the basics, this section goes into more detail about the representation of timetable information. We start by explaining GTFS, a common format specification for timetable data. From this we create an intermediate representation, which we will need again for the experiments on robustness in chapter 4. Then we explain our graph model for public transportation networks. Finally, we extend this model to allow for realistic routes.

2.2.1 General Transit Feed Specification GTFS

Since its publication the *general transit feed specification* has become the most popular format to serve information on public transportation networks to routing service providers [11, 20]. A GTFS-feed consists of multiple files. These define the stations served by public transit and describe the routes between them by departure and arrival times, frequencies, trajectories, service days and fare information. See table 2.1 for an overview of the most important components of the specification.

In the remainder of the text a *trip* denotes the tour of one vehicle characterized by a sequence of *stops*, i.e. stations where passengers get on or off. A *line* is a set of trips which share the same sequence stops and do not overtake each other. For two stations which are subsequent stops of a trip we say an *elementary connection* exists between them.

From the GTFS data we create a meta representation. We set up a mapping between trip ids, service activities and frequencies. We collect information about the stations

Table 2.1: Important files of a GTFS-feed [11].

Filename	Defines
agency.txt	One or more transit agencies that provide the data in this feed.
stops.txt	Individual locations where vehicles pick up or drop off passengers.
routes.txt	Transit routes. A route is a group of trips that are displayed to riders as a single service.
trips.txt	Trips for each route. A trip is a sequence of two or more stops that occurs at specific time.
stop_times.txt	Times that a vehicle arrives at and departs from individual stops for each trip.
calendar.txt	Dates for service IDs using a weekly schedule. Specify when service starts and ends, as well as days of the week where service is available.
...	

Table 2.2: Example for intermediate timetable data.

trip_A	stops	s_0	s_1	s_2	s_3
	times	$(-, 12{:}05)$	$(12{:}30, 12{:}30)$	$(13{:}00, 13{:}05)$	$(13{:}15, -)$
trip_B	stops	s_4	s_1	s_7	
	times	$(-, 20{:}12)$	$(20{:}17, 20{:}17)$	$(20{:}35, -)$	

in the network and the subsequent arrival and departure times of each trip. Using the mapping and the raw trip schedule we create a list of its stops and a list of corresponding arrival and departure times for each trip. Table 2.2 gives an example for this intermediate data. Based on this data, we construct the graph as described below.

2.2.2 Time Expanded Graph

In order to use Dijkstra's algorithm for shortest path search, the timetable information contained in a GTFS-feed has to be modeled as a weighted graph. Pyrga et al. [17] compare two different graph representations. In the *time-dependent graph* each station is modeled by a node. There is an arc between two nodes if there is any elementary connection between these stations. The arcs do not have fixed costs but a special cost function which depends on the time for which the arc is traversed. In this thesis, we focus on another model.

In the *time-expanded graph* as described in [17, 16] each departure and arrival event along a trip is explicitly modeled as a node with an according time stamp. Thus,

there are *arrival nodes* and *departure nodes* for each stop of a vehicle at a station. The successive nodes are connected with arcs of costs corresponding to the time difference between the two events. Beside these two kinds of nodes, there are nodes modeling transfers and waiting at a station. There is a *transfer node* with the same time stamp for each departure node and connected to it with an arc of cost 0. All transfer nodes of a station are connected to the next transfer node in time forming a waiting-chain. Every transfer node corresponds to the decision-event "After waiting until now, do I board the train that goes right now, or do I keep on waiting?". To model transfers from one vehicle to another, an arc connects each arrival node to a transfer node shortly after. Usually, a traveler cannot instantly change from one train to another. We model this with the difference between the arrival and the connected transfer node not being shorter than a fixed transfer buffer. In our experiments, the transfer buffer is 120 seconds.

To incorporate the travel time *and* the number of transfers into the cost function each arc gets a tuple weight. The first component corresponds to the difference of the timestamps of the adjoint nodes. The second component is called *penalty*. Arcs from arrival to transfer nodes have a penalty of one. All other arcs have a penalty of zero. An excerpt of the basic time-expanded graph is depicted in figure 2.1.

In order to decrease the size of the graph and accelerate the transfer pattern computation, we remove departure nodes by redirecting incoming arcs to respective successors [2].

2.2.3 Walking between Stations

To allow for realistic routing, transfers with walking from one station to another must be possible. Imagine you get off a bus and the best connection to your destination goes from the bus stop right on the other side of the street. Therefore you cross the street and wait over there for the connection to arrive. In the basic time-expanded graph presented above, there are no means to model such a direct transfer between the two stations. The only way to arrive on the other side of the street would be an awkward detour, where each change of vehicle involves just one station. If walking is not taken into account, the routing application will suggest obscure routes to users, which would certainly drive them away from using it. In order to give evidence for realistic use-cases, experiments have to be conducted on a model which allows for walking between stations.

We therefore maintain an additional walking graph with arcs between neighboring stations. It is possible to walk from a station to each of its neighbors. Every arc has the duration of walking as costs and penalty one, because walking corresponds to a transfer. For the sake of a simple model, we take the straight line distance between stations and assume a fixed speed of 5 km/h to compute the walking time. The neighborhood of each station is limited to a fixed radius. Thereby, the out-degree of

our graph is limited which is important for the running time of Dijkstra's algorithm in the upcoming computations.

During graph traversal, walking is only allowed from arrival to transfer nodes. When expanding a label at an arrival node at station S and time t_{arr}, the walking graph is used to determine the first transfer node after $t_{arr} + \mathit{walktime}(S, T) + \mathit{transfer\ buffer}$ for every neighbor T. The time difference between the two nodes and penalty 1 are used as weight for the relaxed virtual arc. To speed up the search we compute and store the list of appropriate successor nodes for every arrival node. This does not increase the graph size too strong for a neighborhood radius up to 1000 meters. Hence, we prefer this instead of searching the successors during search. Figure 2.1 also shows such a shortcut walking arc.

This model implicitly forbids successive walking over multiple stations (so called *via-walking*), because after walking from an arrival to a transfer node another vehicle has to be boarded before a new walk becomes possible.

2.2.4 Location-to-Location Queries

In realistic routing applications shortest path queries are typically issued between locations instead of stations. Let X and Y denote two locations. To model a query $X@t \rightarrow Y$ with our graph, we extend it with two stops at X and Y. The former has one arrival node at time t, with outgoing arcs to the first transfer node of each neighbor N_X after time $t + \mathit{walk}(X, N_X)$. The latter has one transfer node with incoming arcs from all arrival nodes of every neighbor N_Y with cost $\mathit{walk}(N_Y, Y)$ respectively. Every of these arc has penalty 0. A feasible path for a location-to-location query starts at X and ends at Y.

2.2.5 Used Datasets

We conduct our experiments on four GTFS datasets of different size. *Hawaii* is a medium-sized data set and contains both urban and rural areas. *Detroit* is a medium-sized, mostly urban dataset with a good structure. *Toronto* and *NYC* are large metropolitan transit networks with the latter being composed of many datasets, both independent borough-wise and city-wide transit feeds. Table 2.3 gives an overview of the examined datasets.

2.3 Shortest Path Problem in Public Transportation

Shortest path problems in graphs with scalar edge weights have been addressed by many publications. There is a variety of fast algorithms finding shortest routes on road networks, where the edge weight solely is the time of travel. In public

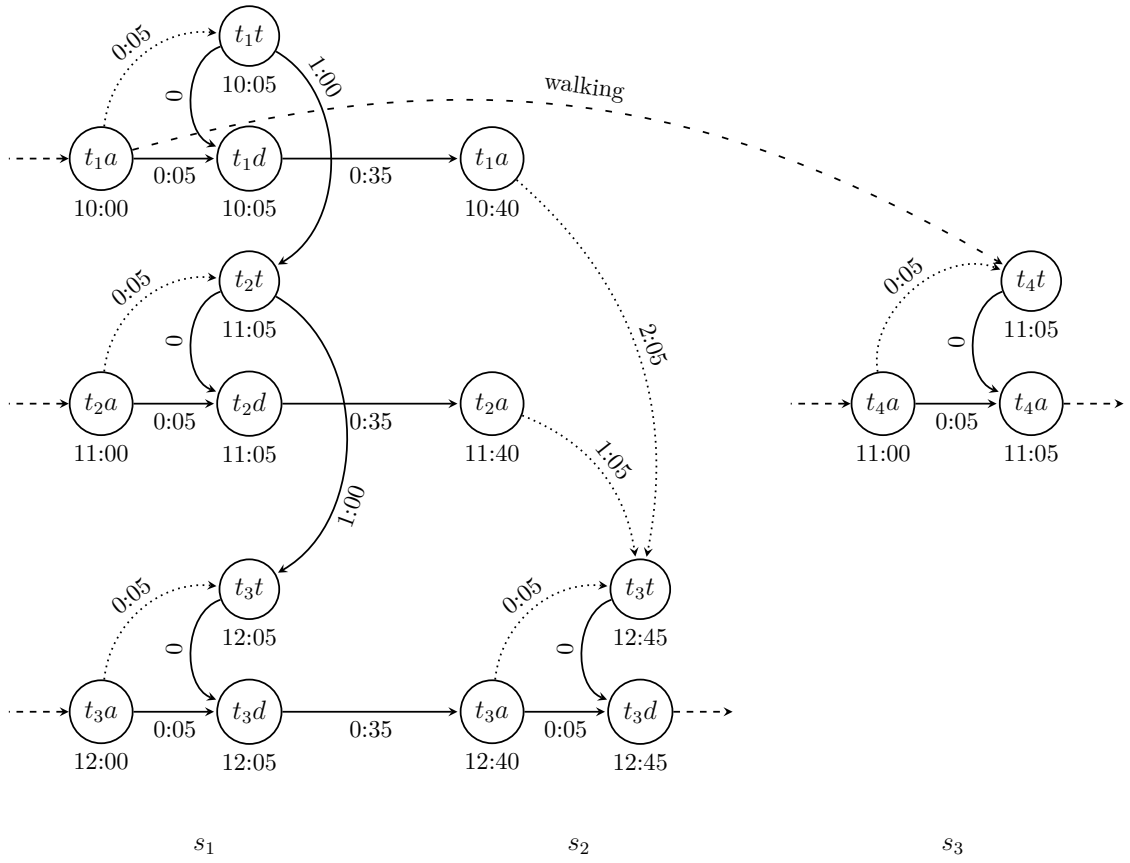


Figure 2.1: Excerpt of the time-expanded graph with three stations and four trips t_1–t_4. Dotted arcs denote transfers and have penalty 1. All other arcs have penalty 0. Walking is possible between s_1 and s_3 and takes seven minutes. Note that the loosely dashed arc is virtual, it is not part of the graph. Courtesy of Manuel Braun.

Table 2.3: Size of the GTFS data sets (after contraction of departure-nodes).

Dataset	Hawaii	Detroit	Toronto	New York City
#stations	3924	5770	10883	16765
#nodes	410K	405K	3.0M	4.6M
#arcs	820K	799K	6.0M	9.2M

transportation the best connection is not necessarily the fastest route. Other criteria like the number of transfers, the minimum time buffer when changing means of transport or the total fare for a connection play an important role. In the following, we extend the notion of shortest paths and their determination defined in section 2.1 to adapt with the models presented in section 2.2.

Multi-Criteria Shortest Paths For the arcs with tuple costs in the time-expanded graph, we refine the notion of a shortest or optimal path. When comparing the costs of two paths in that graph we have to take more than one component into account. One solution would be to define a cost function that linearly combines all components to a scalar value. This approach assumes fixed preferences about the importance of the different components and would yield just one path optimally fulfilling these. But users have varying opinions about what is best. Because of this, we take another approach which allows multiple optimal solutions.

Consider n-tuples $a, b \in \mathbb{R}^n$ and let a_i denote the i-th component of a. We say a tuple a *dominates* a another tuple b *in the Pareto-sense*, $a <_\mathcal{P} b$, if at least one component is less and no component is greater than the respective component of the other:

$$\begin{aligned} a <_\mathcal{P} b &\iff \exists i : a_i < b_i \wedge \neg\exists i : a_i > b_i \qquad a, b \in \mathbb{R}^n,\ i = 1 \dots n \\ &\iff \exists i : a_i < b_i \wedge \forall i : a_i \leq b_i \end{aligned}$$

Furthermore, we say a tuple a is *Pareto-optimal* within a set C if there is not any other tuple a' in C which dominates a:

$$a \text{ Pareto-optimal in } C \iff \neg\exists a' \in C : a' <_\mathcal{P} a$$

Two tuples a and b are *incomparable* if both $a \not<_\mathcal{P} b$ and $b \not<_\mathcal{P} a$. Note that a set of tuples can have an arbitrary number of Pareto-optimal elements and that all these are pairwise incomparable. For example, consider the set $S = \{(20, 0), (30, 0), (24, 1), (10, 2)\}$. The tuples $(20, 0)$ and $(10, 2)$ are the Pareto-optimal elements as they dominate all other elements, but are incomparable to each other ($(20, 0) \not<_\mathcal{P} (10, 2)$ and $(10, 2) \not<_\mathcal{P} (20, 0)$). In the remainder we speak of *labels* instead of tuples.

Multi-Label Dijkstra To solve the shortest path problem with multivariate costs, a multi-criteria variant of Dijkstra's algorithm [14, 16, 19, 15] is used. Instead of a single tentative cost value, it maintains a set of labels for each node. From a single label $(0, 0)$ at the source it yields the set of Pareto-optimal costs to a target, representing the optimal paths. For bi-criteria costs this is initially $\{(\infty, \infty)\}$. During its main loop the algorithm settles labels instead of nodes and maintains Pareto-optimal

labels in the priority queue. The order, in which labels are settled, is determined by a linear combination of the components. With each relaxation of an arc a label with cost being the settled label's cost plus the cost of the arc is inserted into the set at the end point of the arc. All dominated labels are then removed from this set. This variant of the algorithm terminates when all labels at the destination are settled or the queue is empty. The optimal paths can be reconstructed by maintaining a pointer from each label to its parent.

2.4 Transfer Pattern Routing

Now we have all the prerequisites for transfer pattern routing. Remember, a *transfer pattern* denotes the sequence of stations along a journey where one boards, changes or alights vehicles. Independent of the departure time, the optimal paths in public transportation networks follow a limited set of optimal transfer patterns. The key idea of the algorithm is that the optimal transfer patterns between stations describe a restricted search space which is orders of magnitude smaller than the original search space in the time-expanded graph. Given these optimal transfer patterns are known and the cost for direct connections between stations can be looked up efficiently, a fast search becomes possible.

This section gives a detailed explanation of the main components of transfer pattern routing: In the beginning we explain how transfer patterns in a transit network are computed. We show how these procedures have to be adapted for initial and final walking in location-based queries and point out a critical pitfall. We then show how fast direct connection queries work and how they are used during search. In the end of the section we discuss important heuristics needed to make the computation of transfer patterns practicable. Please note that all algorithms and data structures mentioned in this section refer to the original publication on transfer patterns by Hannah Bast et al. [2], if not stated otherwise.

2.4.1 Computing Transfer Patterns

Profile Queries To discover all optimal transfer patterns in a network, we use an important property of Dijkstra's algorithm. When settling an optimal label, every label settled before is optimal too. So running Dijkstra from a single source station until all Pareto-optimal labels are settled yields all optimal paths starting from that station. We consider time of travel and number of transfers as cost criteria and run Dijkstra's algorithm from initial labels $(0, 0)$, one at each transfer node of the source station. In the following text we call such a transfer pattern exploring Dijkstra a *profile query*.

Algorithm 2.1 Compute transfer patterns for a departure station S.

TRANSFER_PATTERNS(S)
1. Run an unlimited profile query from departure nodes of station S.
2. For each destination D reached from S, run *arrival_loop*(D).
3. Backtrack optimal labels selected by 2. to yield transfer patterns.
4. Create a DAG for these patterns considering prefixes starting at S.

end

Arrival Chain Algorithm After the multi-criteria Dijkstra terminates, the settled labels at arrival nodes represent optimal paths to the respective station. But because every arc in the time-expanded graph is on at least one optimal path [1], there are suboptimal connections among these. Recall the example Freiburg to Munich. There are also trains from Vienna to Munich. For each train arriving at Munich main station there is an arrival node. As the profile query from Freiburg main station explores the full network, all these nodes have at least one optimal label. But the connection Freiburg–Vienna–Munich is always suboptimal. We perform a post-processing to get rid of suboptimal paths. The *arrival chain algorithm* selects an optimal subset among the arrival labels of a station S: For all distinct arrival times $t_1 < t_2 < ... < t_n$ at S it selects a dominant subset in the set of labels consisting of (1) the labels settled at time t_i and (2) the labels settled at time t_{i-1} with duration increased by the waiting time $t_i - t_{i-1}$, ties are broken in preference of (2).

Retrieval and Storage of Transfer Patterns From the selected subset the paths to the departure station are backtracked using the parent pointers of the labels. Reducing the paths to stations with boarding or alighting yields the transfer patterns. In terms of nodes, these are the stations of the very first and last nodes, of every transfer node where the predecessor is an arrival node and also of the arrival node, if it is at a different station than the transfer node. All transfer patterns starting at station A are stored in a DAG for A. This graph has three different types of nodes: A single *departure node* (*root node*) representing the source station, for each destination B reachable from A a unique *destination node* named B and for each prefix $AC_1 \dots C_i$ in any transfer pattern $AC_1 \dots C_k B$ a *prefix node* labeled C_i. The set of transfer patterns from station A to destination B is represented by all paths from the destination node for B to the root A.

Figure 2.2 shows an example for this data structure. In addition to each DAG, there is a mapping which indicates the appropriate entry-point for every destination reachable from the departure station. In our C++ implementation, this mapping is a `std::vector<std::pair<int,int>>`, with pairs of station id and node index. The vector is sorted by ascending station ids and allows for lookup in $O(log\, N)$ using `std::lower_bound`. The DAG is implemented as a vector of station ids with an adjacency list for every node. Listing 2.1 summarizes the full algorithm to compute the transfer patterns from a source station.

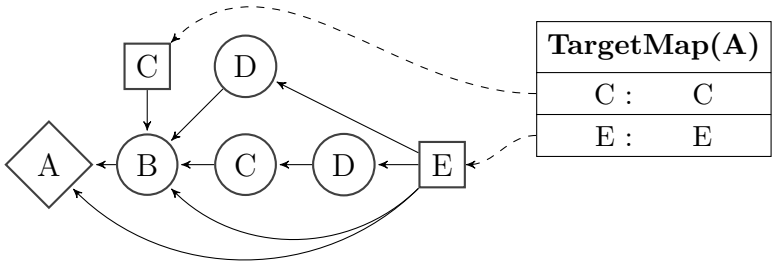


Figure 2.2: The DAG representing the patterns 'ABC', 'ABE', 'ABDE' and 'ABCDE'. Note that beside destination nodes, every other node has at most one outgoing arc. A map indicates the entry-point into the DAG for every reachable destination.

Location-to-Location versus Station-to-Station The procedure described so far computes transfer patterns between stations. Realistic routing applications are queried for shortest paths between locations. Typical users accept routes which have a walking portion in the beginning and end. In the following, we compute location-to-location transfer patterns. One might suppose that this causes a significant increase in the number of patterns. But actually computing such transfer patterns requires the same number of profile queries and the number of distinct patterns decreases.

The reason for the reduced number of patterns is this: When starting a journey at a station it often makes sense to walk to another station in the beginning and get a much faster connection there. If the patterns are computed such that each path starts aboard a vehicle, awkward detours resulting in useless patterns are the consequence. Avoiding this is an integral part of location to location queries: When searching from a location as in section 2.2.4 the resulting paths begin with walking to nearby stops where the best connection departs aboard a vehicle. We make use of this and allow the profile queries to walk in the beginning and end, while all stored transfer patterns start and end with riding a vehicle (with an exception explained in section 2.4.3).

The pattern exploration search starting from $(0, 0)$-labels at all transfer nodes at departure station S is extended: For each neighbor N add initial labels $(walk(S, N), 0)$ at its respective transfer nodes. To solve the disadvantage of paths with final walking to destinations as mentioned in section 2.2.3 the arrival chain algorithm of section 2.4.1 is extended to consider all arrival events in the neighborhood of the destination T. The cost of all paths arriving at neighbor N are increased with time $walk(N, T)$ before considering them in the arrival chain.

Backtracked paths may now yield transfer patterns not only between the departure and destination station, but between every possible combination of neighbors of the two. For example, a profile query from 'Freiburg, Faculty of Engineering' to

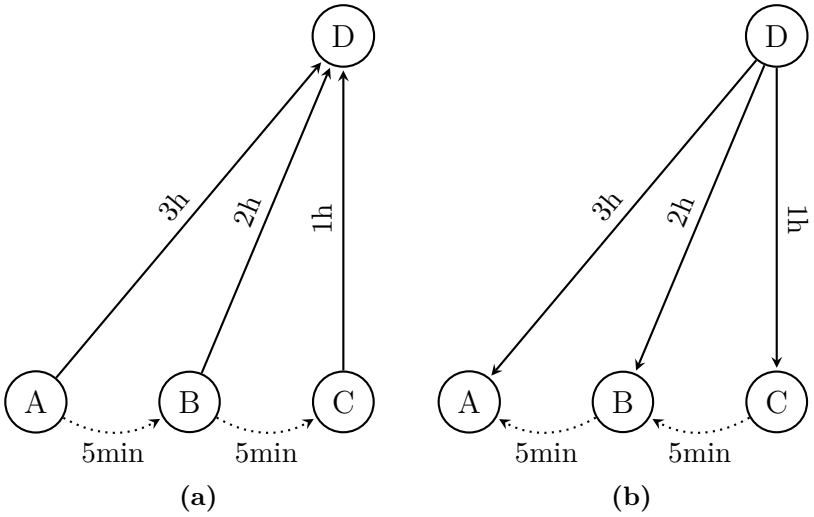


Figure 2.3: A setup where the shadowed initial (a) and final (b) walking problem arises. Computation of transfer patterns without initial and final walking fails, if not every dominating pattern is stored. Straight arcs denote direct connections. Dotted arcs denote possible walking.

Freiburg main station may yield a pattern from 'Freiburg, Bärenweg' to 'Freiburg ZOB', which are each within 1 km of the original departure and destination. Because it is optimal for at least one other profile query, it is important to store that pattern in the respective DAG, even though it may not be optimal for a profile query from 'Freiburg, Bärenweg'. Otherwise in the context of limited neighborhoods the *shadowed walking problem* arises. Figure 2.3 depicts the setting in which this problem is present. Assume a profile query from station s would only yield patterns starting from s aboard a vehicle. Consider the connection $A \to D$, for which the direct connection is dominated by a connection starting with walking to a neighbor station B. The profile query from station A relies on a pattern starting at its neighbor B being found by the profile query from that station, as the patterns of its neighbors will be used to construct the query graph at search time. During the profile query from B, that pattern is dominated by another pattern starting from station C, which is outside the neighborhood of A. Consequentially neither the direct connection from A to D nor the optimal connection with initial walking to B could be found. To avoid this problem in general, a profile query from s has to return transfer patterns starting at its neighbors, and these have to be stored in the DAG of the respective neighbor. A similar problem exists for walking at the end of a journey.

These location-based computation of transfer patterns avoids artificial transfer patterns. Thereby, the patterns become somewhat more homogeneous, as the artificial are specific for every stop. With increasing radius for initial and final walking this reduces the number of transfer patterns. Also, the model becomes more realistic. However, not limiting the neighborhood radius renders the computation infeasible:

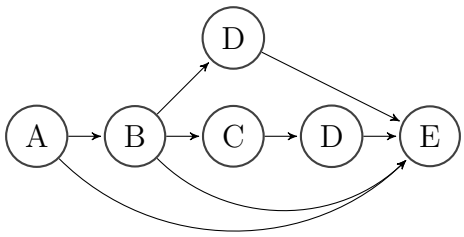


Figure 2.4: Query Graph for the query $A \to E$ constructed from the DAG in figure 2.2.

The out-degree of arrival nodes becomes too large. So one could argue for different radii for initial, intermediate and final walking. But as the arrival chain takes the whole neighborhood of every destination into account, the asymptotic runtime of the precomputation would become cubic in the number of stations.

2.4.2 Exploiting Optimal Transfer Patterns for Search

Query Graph To answer a query from station S to station T a *query graph* is constructed from the transfer patterns in the following way. The destination node for T is searched in the transfer patterns DAG for S. If there is no such node, no path between S and T exists. Otherwise let the current node in the DAG be u with label U have k successors with labels $V_1 \dots V_k$. Add arcs (V_i, U) for all i to the query graph. Recursively repeat this for all successors of u. Figure 2.4 shows an example for such a query graph.

Robert Geisberger extended this procedure for location to location queries [10]. For a query $X \to Y$ let N_X and N_Y be the neighborhood stations of the two locations. The query graph is initialized with two nodes x and y with arcs (x, X') to all $X' \in N_X$ and arcs (Y', y) from all $Y' \in N_Y$. Now the query graph can be constructed from the patterns between station-pairs in $N_X \times N_Y$. The construction of a query graph with one departure and multiple destinations can be done simultaneously when starting from the destination nodes for N_Y for each X'. So the construction time just increases by linear factor $|N_X|$ compared to the station to station version.

Direct Connection Queries The edges in the query graph correspond to direct connections between stations. The only exception are the initial and final walking arcs for location-to-location queries, for which the walking time can be stored in a separate table. To search for shortest paths on such a graph quickly, an efficient lookup data structure is required. The trips extracted from GTFS (see section 2.2.1, table 2.2) are grouped to distinct lines and stored in a table.

line_A	s_4	s_{12}	s_7	...
$\text{trip}_{A,1}$	$(-, 6{:}00)$	$(6{:}10, 6{:}12)$	$(6{:}20, 6{:}20)$	...
$\text{trip}_{A,2}$	$(-, 6{:}30)$	$(6{:}40, 6{:}42)$	$(6{:}50, 6{:}50)$	...
$\text{trip}_{A,3}$	$(-, 7{:}00)$	$(7{:}10, 7{:}12)$	$(7{:}20, 7{:}20)$	...
...	...	...	...	...

For each station a list of incident lines and its position along the line is stored.

station	incident lines with position			
s_1	$(\text{line}_A, 0)$	$(\text{line}_D, 7)$	$(\text{line}_F, 3)$	...
s_7	$(\text{line}_A, 2)$	$(\text{line}_B, 4)$	$(\text{line}_E, 2)$	...
...	...			

The query for a direct connection $s_A \longrightarrow s_B@t$ is answered by intersecting the incidence lists of s_A and s_B, determining lines which serve both stations and s_A before s_B. For all such lines which stop at s_A before s_B, the first departure time after t is looked up in the line's table in column s_A. If there is a trip departing after that time, the difference to the arrival of the trip at s_B is returned. In the example above a query $s_1 \longrightarrow s_7@6{:}25$ would find the incident line_A with positions 0 and 2 and return a duration of 35min for the direct connection.

With this data structure a direct connection query can be answered within microseconds. For the search within the query graph, where each relaxation of an arc corresponds to a direct connection query, this results in query times of only a few milliseconds.

2.4.3 Heuristics

Knowing the optimal transfer patterns enables this routing algorithm to answer shortest path queries very fast. Its main drawback is the time-consuming precomputation of the patterns. The asymptotic runtime has at least quadratic dependency on the size of the underlying network. For instance, computing the complete transfer patterns for the New York City data set with 1.000 meters walking radius takes about two weeks using 20 cores. The computation of the patterns for transit networks of the size of Northern America with the algorithm as described above is infeasible. To shorten the computation time some heuristics have been successfully tested [2, 10].

Important Stations During long range journeys typically at least one transfer takes place at an important station, e.g. a large bus hub or the central train station of a city. For the computation of transfer patterns this can be exploited such that only profile queries from hub stations explore the full transit network and profile queries

from regular stations are limited until every path did a transfer at an important station.

We change the algorithm 2.1 in two ways. For hub stations the procedure is extended such that it explores also paths which begin with a transfer at the hub. On this account, these global profile queries start from $(0, 0)$-labels at the arrival nodes in addition to the labels at departure nodes. These new labels allow paths which initially walk to neighbors of the hub. For profile queries from non-important stations a flag for every label indicates whether the path to this label has transfered at a hub. The search terminates as soon as all open paths did such a transfer. As an exception to location-to-location routing forbidding patterns with final walking, local profile queries may yield patterns terminated by walking to a hub.

Other implementations extend this termination criterion with paths that just traveled through the important station. This forced transfer is resolved during query graph construction by checking for a direct connection between the stops before and after the hub [10, 2]. However, we did not pursue this last extension, as the decreasing computation time was not the focus of this work.

The important stations are selected by running a limited number of Dijkstras from random stations on a compressed variant of the graph. In this graph every station has just one arrival and transfer node. Both nodes are connected to the arrival node of every station which is connected by an elementary connection. Walking and transfers are modeled by arcs from arrival to transfer nodes. The arc cost between stations correspond to the fastest connection during the course of the day. After each Dijkstra the number of transferring paths is counted for each station. The one percent with the most transfers are selected as hub stations.

Limiting the Precomputation Further heuristics to shorten the computation time of the transfer patterns limit the search space for profile queries. This is done by setting maximum cost and penalty for paths. In the patterns we computed, paths from non-hubs have penalty ≤ 3 (up to three transfers) and penalty ≤ 5 from hubs. Although this reportedly improves the computation time a lot, we did not limit the duration of paths within these limits as it decreases optimality.

A Note on Optimality With the heuristics introduced beforehand, the computation of transfer patterns becomes feasible, however optimality is relinquished. Criticism on transfer pattern routing often focuses on its partial suboptimality [7, 18]. No results examining the correspondence between the heuristics and loss of optimality have been published yet. The experiments in chapter 4 give some evidence to this question.

3 Compact Representation of Transfer Patterns

This chapter covers our results to the first objective of this thesis: A compact representation of transfer patterns. In the first section we propose an algorithm similar to transfer patterns, but less informed. In the remainder of the chapter we analyze the structure of the DAGs representing transfer patterns as described in the primary article [2] and introduce multiple methods to decrease the size of the information. The chapter is concluded by comprehensive experimental results proving the improvement of these techniques.

3.1 A Less Informed Approach: First-Transfer Routing

3.1.1 Idea

One of the mechanisms that render transfer pattern computation feasible is the concept of important stations. Remember section 2.4.3, where we computed patterns up to the first hub. At query time the search space is reconstructed from the patterns between the departure station, its hubs and the target station. Looking at the exploration of the patterns from a source station A we observe that the optimal set of hubs, which minimizes the computation time *and* the size of the DAG, is such that every first transfer involves a hub. For example in the network sketched in figure 3.1, all paths from A to *any* other station have a first transfer at B or C**.** If B and C were hubs and A was not, the pattern exploration would stop as soon as every expanded path transfered for the first time.

If the number of important stations was unlimited, the local profile query for every station could be minimized. Thereby the information size would become minimal, too. From a hub station a global profile query must be computed. Because of this, considering all stations as important is impractical. However, we observe that the result of global profile queries always overlap: In the example, the patterns from B to D are a superset of the patterns from A to D which start with a transfer at B (despite the first station A).

Let $FT(s,t)$ denote the stations where optimal paths from station s to t have their first transfer and let $TP(s,t)$ denote the optimal transfer patterns between s and t.

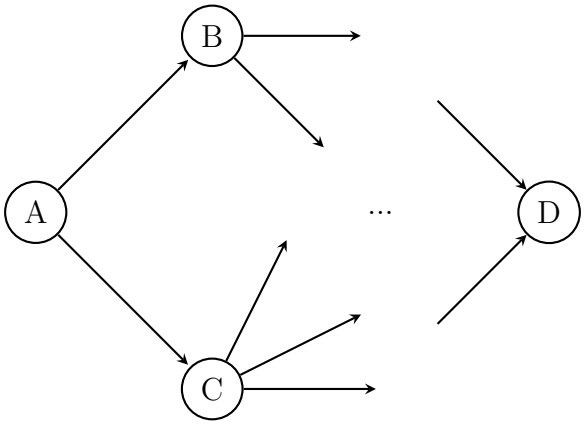


Figure 3.1: Motivation for first-transfer information. The nodes denote stations, the arcs possible connections from and to each station. The set of hubs minimizing the transfer patterns starting at A is $\{B, C\}$.

Then it holds

$$TP(s,t) \subseteq \bigcup_{ft \in FT(s,t)} \{s\} \times TP(ft,t)$$

In other words, when prepending s to all optimal transfer patterns between every first-transfer station ft to t, then the optimal transfer patterns from s to t are among these. Using this recursive formula, a superset of the search space induced by the optimal transfer patterns can be constructed from the first-transfer information.

To illustrate the idea in more detail, consider the initial example from Freiburg to Munich. The optimal transfer patterns were *Freiburg–Karlsruhe–Munich*, *Freiburg–Titisee–Ulm–Munich* and *Freiburg–Munich.* The set *Karlsruhe, Titisee, Munich* contains all stations where optimal connections from Freiburg to Munich have the first transfer. During query graph construction we add arcs from *Freiburg* to these stations. Then, we look up the first transfers from Karlsruhe to Munich, which yields the set *Mannheim, Stuttgart*, *Munich*. These stations are added to the query graph as well and the procedure continues recursively with *Titisee*. At some point, all paths in the query graph end in *Munich*.

The idea of *first-transfer routing* is similar to the principle of transfer pattern routing. If the first transfers of all optimal paths in a network independent of the departure time are known, a fast search on a restricted search space is possible. In contrast to the original algorithm we do not store full transfer patterns but only the first transfers along the optimal paths, thereby decreasing the size of the information. Because of this, first-transfer routing is less informed than transfer pattern routing, suggesting the search time is increased. In the following we explain how we explore and store the first transfers. After that, we report about the results of first experiments with this technique, indicating the different information size and search performance of first-transfer and transfer pattern routing.

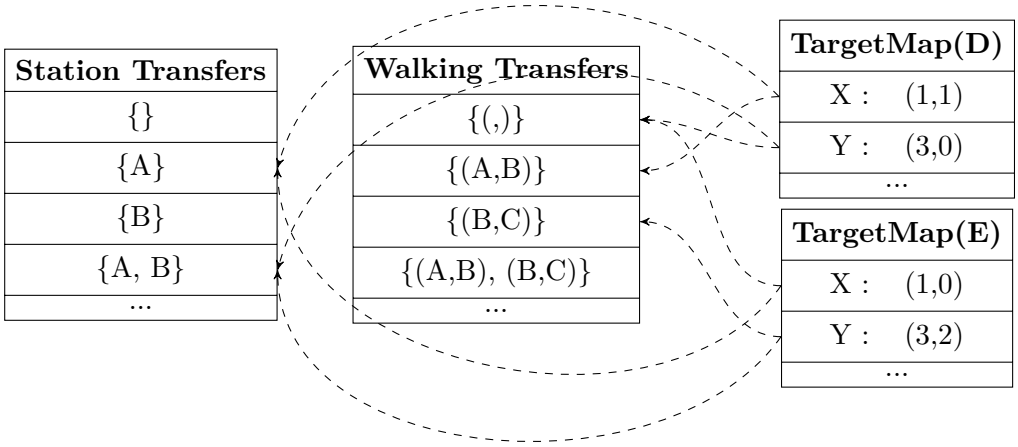


Figure 3.2: The data structure storing first-transfer information. Unique sets of transfers are separated into transfers at one station and those which include walking between stations. For each departure station, a map indicates the two respective fields which represent the first transfers to a destination.

3.1.2 Implementation

We adapt the algorithm computing optimal transfer patterns from chapter 2 to yield the first transfers only. The specializations concerning location to location queries with initial and final walking are directly embedded: The search is started from labels with respective walking costs from a station and its neighbors and the arrival loop algorithms takes additional labels at neighbors of each destination into account. From the generated optimal paths we extract the set of first transfers to every destination.

In a first experiment the first-transfer information just comprised a single stop. We recognized that this algorithm was too poorly informed as to compete with full transfer patterns in terms of query times. Transfers including walking between two stations and those which transfer on-site have to be distinguished. Distinct sets of first transfers are stored in an array for each type. Every departure-destination pair maps to one entry in each of these. Figure 3.2 illustrates this data structure. Here, to give an example, optimal paths from E to Y have their first transfer at station A, station B or by walking from B to C.

We now explain how to construct the query graph between two stations from this information. The construction for location to location queries is a straightforward extension in analogy to section 2.4.2. Starting with a queue containing only the departure station, all first transfers to the destination are retrieved. The query graph is extended with the respective arcs and the successors are added to the queue. This procedure continues until every stop in the queue has been

visited.

Note that the query graph may also be constructed at search time. Each settling of a node in the graph adds the adjacent arcs using the first-transfer information and updates the costs of the successors. The order of settlement follows the tentative costs of the nodes. However, we did not pursue this further, as the query graph construction takes only a fraction of the response time and so this would improve the search time just slightly.

3.1.3 Results

This section contains experimental results on the behavior of first transfer routing. We compare the output size and the query performance between routing with transfer patterns and first transfers only.

Setup To get an overview of the utility of this approach, we compute the information on the two smallest of the four networks described in section 2.2.5. We compare the data size and the performance of the search between routing with full transfer patterns TP and first transfers FT. The time expanded graph was generated for 26 and a half hours from 3:00 am to 5:30 am the next day, with a maximum walking distance of 1000 meters. Like in the initial article [2] we select one percent of the stations as hubs for the transfer patterns. Unlike the authors, we did not force paths to transfer at hubs and had no limit on the time of travel. The paths were limited to at most 3 transfers (4 legs) for local and 5 transfers for global profile queries. This implies that paths with up to 8 transfers can be found by transfer pattern routing. We adapted the computation limits for first transfers accordingly, to allow the new variant of the algorithm to find paths of the same maximum length. Note that the computation time is larger than for transfer pattern, as we have to run a global profile query for every departure station.

Output Size At first we compare the size of the computed information. Table 3.1 summarizes the created data for transfer patterns using the basic DAG for every station (see figure 2.2) and the first transfer information stored as explained before. Indeed, the output sizes are smaller for first transfers, because on the one hand the first transfers comprise less information and on the other hand, the data can be stored efficiently with the employed data structure (figure 3.2). For both data sets, the memory used for FT is about half as large. However, we think that without introducing a heuristic similar to the important stations in section 2.4.3, this advantage will vanish for large networks as the share of the entry-point maps becomes larger. Because of the global profile queries the first transfers are always known to almost every destination, while the transfer patterns are restricted to those reachable within 4 legs.

Table 3.1: Memory size of the computed information for transfer patterns (TP) and first transfers (FT) in Byte.

Dataset	TP	FT
Hawaii@1000m	265.8M	123.2M
Detroit@1000m	588.3M	266.4M

Query Performance To get a first glimpse of the different performance of the search with the two information types, we executed 10 000 random queries on the different networks. While the increased time for the recursive construction of the query graph did not have a large impact, it turned out that in general the query graphs for FT are about twice as large as for TP. With the asymptotic complexity of the search algorithm being quadratic in the size of the underlying network, it is not surprising that the search is slower than for TP. So routing with FT cannot compete with transfer pattern routing in terms if speed, nonetheless it is still orders of magnitude faster than computing the Pareto-optimal paths using multi-label Dijkstra.

Interpretation The first results demonstrated that routing with the information of first-transfers instead of full transfer patterns is possible. The information can be represented by more compact structures than the transfer patterns, at least for small datasets. As the underlying network grows, first transfer routing suffers its main disadvantage: It does not (yet) exploit any of the networks structure by such means as important stations. Because of this the computation is more expensive than for transfer patterns, were we distinguish local and global profile queries. As a matter of fact, it is not very convenient to perform an expensive computation and then discard most of the retrieved information. Due to larger query times and with the advantage in information size being only marginal compared to techniques we present hereafter, routing with first transfers seemed not promising enough to be pursued further. However, the query graph constructed recursively from the first-transfer information might provide for alternative routes when a direct connection along the optimal transfer patterns becomes delayed. This may become of interest in chapter 4.

3.2 Reducing Redundancy in Directed Acyclic Graphs

The previous section discussed an approach that aims to reduce the information size by using less information. As we could observe, the search speed suffers from the reduced knowledge while the size of the data is not improved so much. Opposed to that, we now present three techniques that reduce the size of the representation of the transfer patterns while the information is maintained.

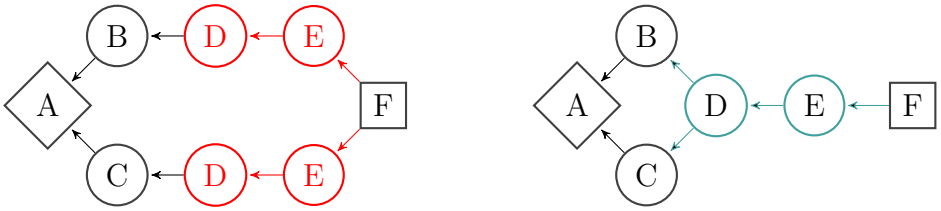


Figure 3.3: An example for subgraph equality. The DAG for the patterns 'ABDEF' and 'ACDEF' and the reduced DAG.

3.2.1 Isomorphic Reductions

Idea The DAG representing the transfer patterns is prefix-free, which means that all equal prefixes of the contained patterns are represented by a path from the same internal prefix-node to the departure-node. The DAG is thereby related to a prefix-tree or trie, used to store a set of words by paths originating at the root-node.

Consider the DAG for the patterns 'ABDEF' and 'ACDEF' in figure 3.3. Obviously the two equal suffixes 'DEF' are expressed by two separate branches. If we merge the branches and let the node for 'D' point to both 'B' and 'C', two nodes and two arcs can be saved. The set of patterns represented by this more compact DAG is the same.

Consider the set of transfer patterns in a network as dictionary. Dictionaries can be represented as *Directed Acyclic Word Graphs* DAWG also known as *Minimal Acyclic Finite State Automata* MA-FSA. For these data structures the same problem of equal suffixes is given. There are different approaches to construct a compact DAWG (Chrochemore and Vérin [5]) or compact MA-FSA (Daciuk et al. [6]). Inspired by these works we implemented a top-down algorithm to find and remove subgraph equality in the transfer pattern DAG. As the number of distinct transfer patterns is not too large, we can decrease the size of the data structure after construction.

Top-Down Algorithm Starting from the original DAG we construct the prefix-tree representing all paths in reversed direction (i.e. starting at the departure-node). The leafs of this tree correspond to the destinations. Figure 3.4 (b) shows this tree for the DAG from the initial example.

Starting from the root we traverse the tree in preorder depth-first-search. Every traversed node is the root of a subtree. Once we find a subtree equal to a subtree visited before, we redirect the connection from its parent to the root of the formerly visited subtree. Thereby, the current subtree is decoupled (c). The traversal continues with the next unvisited node until the full tree has been processed. Then the compact DAG is re-assembled from the pruned tree (d).

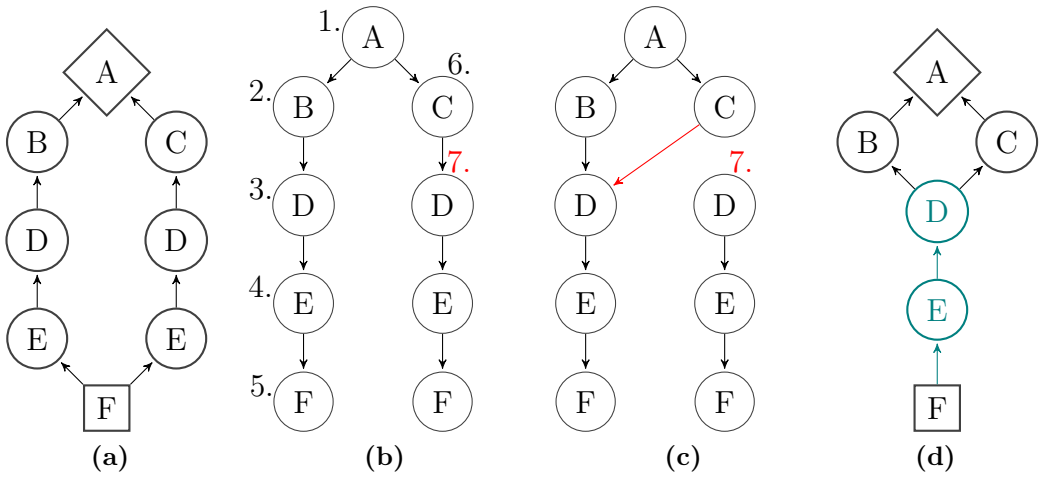


Figure 3.4: From the DAG (a) the inverse tree is constructed and traversed via preorder DFS (b). The small ordinary numbers denote the order of visited nodes. In the 7th step a subtree equal to step 3 is found and decoupled (c). In the end, only subtrees connected to the root form the improved DAG (d).

To compare subtrees efficiently, we compute a hash value for every node and store it in a table. The hash value of a node is a combination of its station id and the hash values of all its children. Due to this recursive formulation, the hash values can be computed in linear time bottom-up for all nodes of the tree. Only when two subtrees have the same hash value they are compared node by node in order to resolve hash collisions.

This algorithm efficiently removes duplicates of equal suffixes among the transfer patterns. For instance, compressing the 380 million transfer patterns for New York City takes about 5 minutes. The resulting DAG is free of redundant prefixes *and* suffixes. Note that unlike in the original data structure, an internal node may have more than one outgoing arc in this graph. The query graph construction from chapter 2.4.2 has to be extended such that every node of the transfer pattern DAG is visited just once, otherwise the query graph may have duplicate arcs. This can be achieved by maintaining a mark for each node.

This algorithm removes redundant substructures from a DAG and shrinks the data without loss of information. Before we present results for this compression, we will introduce some further mechanisms.

3.2.2 Shared Entry-Points

Observation Remember that there is a mapping along with the DAG from section 2.4.1 which tells us where the appropriate destination node is. The information comprised in a destination-node is its station id and its set of successors. The

successors determine the set of transfer patterns to this destination by all paths from the destination-node to the departure-node of the DAG. The station id of the node is *also* determined by the mapping. Thus, the station id label of the node is redundant. With the only substantial information being the set of successors, all destination nodes with equal successors can be merged.

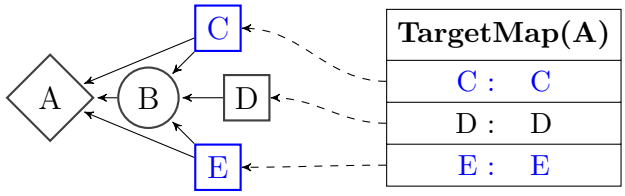
For example, in the DAG representing the patterns 'AC', 'ABC', 'ABD', 'AE' and 'ABE' depicted in figure 3.5, the patterns from A to C and to E are equal except for the last stop. They can be represented by a common destination node. Its label is irrelevant as it can be determined from the mapping. The destination node for D, however, represents a different set of patterns, so it cannot be combined with the other one.

Improvement In the original DAG (see [2] and figure 2.2), the majority of nodes are destination nodes. The redundancy among them can be resolved by replacing all nodes with the same set of successor nodes by a single incognito node, i.e. a node without a label. The query graph construction must be adapted such that it assigns a label to the incognito node when traversing it. This is a simple, yet very effective change to the data structure. We combine this improvement with the previous one and refer to the combination as *compressing* the transfer pattern DAG. Results for this compression can be found in section 3.4.

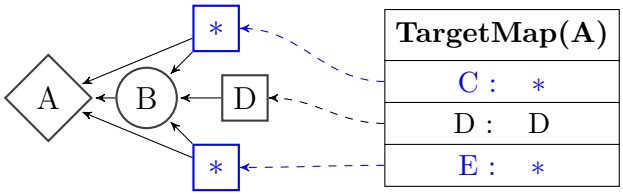
3.2.3 Joint DAG

Observation When inspecting the structure of the transfer patterns we made another observation: For neighboring stations the transfer patterns are somewhat similar. Especially for remote destinations there is only little variance. To exploit this redundancy and decrease the size of the data structures further we pursue an idea that was incepted by the foregoing section. We could introduce incognito nodes, because the label (the station id) of a node can be assigned from the mapping. In fact, we always use the transfer patterns DAG within a context: We either add patterns or we construct a query graph from it. In either case, we know the destination *and* the departure station from context.

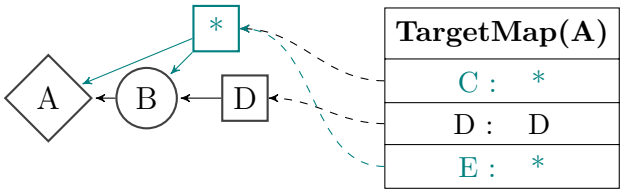
Joint Pattern DAG We use this observation and change the preceding data structures and algorithms in the following way. The root node of the DAG becomes an incognito node. Instead of one DAG per departure station, we maintain one joint DAG (jDAG) with a single root for all departure stations. For each departure, the mapping indicating the entry point for each reachable destination is kept as before. But this mapping now points into the jDAG. This data structure automatically resolves the redundancy observed above. Note that also all arcs pointing to the root-node are now obsolete. Despite this, we keep them in the graph, as their number



(a) Example DAG for the patterns 'AC', 'ABC', 'ABD', 'AE' and 'ABE' with the original data structure.



(b) The map determines the station id of the destination-nodes, so their labels are arbitrary.



(c) The destination nodes for C and E are now equal and can be merged.

Figure 3.5: Merging redundant destination nodes.Label of destination-node D is unchanged for better distinction.

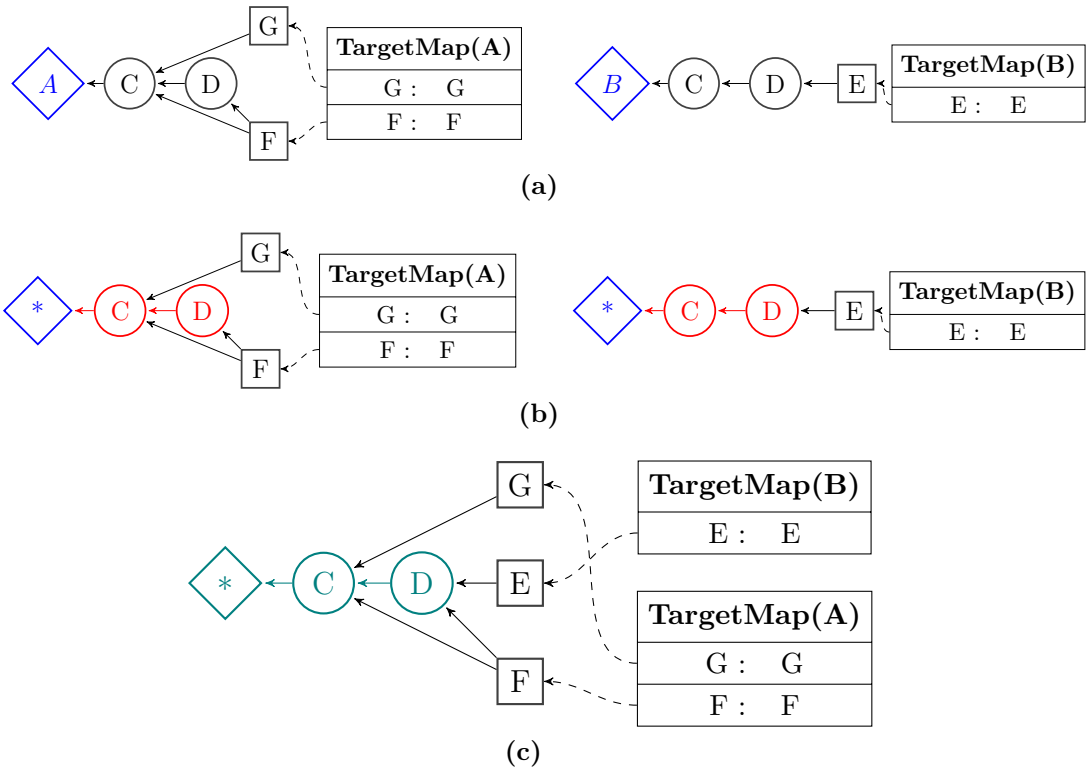


Figure 3.6: Example for a joint pattern DAG. The two DAGs in (a) have an equal internal structure given the label of their departure-node is known from context (b). The joint DAG resolves the redundancy (c).

is limited by the number of distinct stations and the algorithms for DAG and Query Graph construction need to be changed otherwise.

Figure 3.6 gives an example for the patterns 'ACG', 'ACF', 'ACDF' and 'BCDE'. The patterns represented by the graphs for departures A and B can be represented by a joint graph. This saves two nodes and arcs.

Beside the reduced size, the jDAG has even more advantages: It easily adapts to the essential construction and retrieval processes of chapter 2. When extending the DAG with a new pattern, we ignore the id of the pattern's first stop. During query graph construction, the root node's label is assigned from the context. Moreover, the compression techniques introduced in sections 3.2.1 and 3.2.2 can be used on the joint graph, such that all redundancies discussed so far are removed. We show results for this joint representation alone and compressed by the previous techniques in the end of this chapter.

Furthermore, the location-based query graph construction can be extended such that it not only creates the graph from one departure to multiple destinations in parallel [10], but also from multiple departures to multiple destinations: Start at the destination nodes for all departures and all destinations. For each destination node keep an indicator for the departure station(s) from which it can be reached

and start backtracking the paths in the joint DAG. When reaching the incognito root node, connect the path to the departure nodes in the query graph denoted by its indicator.

The concept of the joint DAG is quite flexible. Instead of one joint DAG for all stations, one could consider a partitioning of the network, where each group of stations shares a joint DAG.

3.3 Further Ideas

In the compressed joint DAG there is no more obvious redundancy. The mapping from each departure-destination pair to its corresponding entry-point (destination node) in the DAG has become the largest part of the data. A more space-efficient structure for this mapping would further improve the size of the data. In the DAG, the majority of the nodes are destination nodes. Here, destination nodes could be separated from the remaining graph and their successors may be stored as space-efficient lists of integers, with variable width or gap encoding. However, such an implementation goes beyond the scope of this thesis.

Another idea we investigated partially is to formulate the destination maps recursively. Let $\mathsf{m}_s = \{(s,t) \to i\}$ denote a map with entries from departure s and destination t to index i. We did first experiments looking for successive stations s_1 and s_2 along lines and counting maps m_{s_1} for which $\mathsf{m}_{s_1} \supseteq \mathsf{m}_{s_2}$ holds, i.e. m_{s_1} has the same entries as m_{s_2} and maybe some in addition. This suggests that given a suitable recursive formulation m_{s_1} could be represented by only the elements $\mathsf{m}_{s_1} \setminus \mathsf{m}_{s_2}$ and a pointer to m_{s_2}. We conducted a first experiment counting the entries which become obsolete this way. The results suggest that between 10 and 20 percent of the map entries could be saved by such a recursive formulation. As announced above, we did not pursue this approach further.

3.4 Results

Experimental Setup In the remainder, we present results outlining the contribution of the first part of this thesis. To evaluate the improvement of the techniques introduced before, we computed the transfer patterns on the four datasets from 2.2.5 for the same period as in 3.1.3. To understand if and how the advancement depends on the structure of the network, we repeated the computation on time expanded graphs generated for different walking transfer radii (100m, 500m and 1000m). The same maximum walking distance applies for walking at the beginning (initial labels at neighbors) and in the end (arrival loop candidates) of a path. Like in previous experiments (section 3.1.3) we select one percent of the stations as hubs. Local and global profile queries are limited to 4 and 6 legs respectively. There was no limit on the cost of paths.

The generated transfer patterns are represented using four different data structures. The original DAG (see figure 2.2) with the destination mappings forms the baseline (TP). We compare it to the compressed version with removed suffix redundancy and merged destination nodes (TP^c), the joint DAG ($jDAG$) and the compressed joint DAG ($jDAG^c$). This subdivision permits a detailed understanding which technique decreases the size of by what magnitude. For all data structures we measured the number of internal nodes, destination nodes and arcs as well as the consumed memory. We also calculate the average number of Bytes required to represent one pattern.

The tables 3.2–3.4 show the output sizes of all computations. For each part of these tables the header specifies the dataset and the walking distance. The two following numbers report the total number of optimal transfer patterns and the average number of such patterns between two stations.

A Note on Data Sizes When comparing the results at hand to the data sizes documented in the original article on transfer pattern routing [2], the different parameter settings for the transfer pattern computation must be taken into account. Most notably, the authors limited local searches to 3 legs, while we used 4 legs. This increases the output size of the local transfer patterns for our approach. The restriction of the maximum time of travel in their paper accounts for the time-expanded graph wrapped for multiple days which they employed. It should not be responsible for a large difference in comparison to our results.

As mentioned in the introduction, we found out that, in order to achieve the data sizes reported in the original paper [2] (table 3), some undocumented improvement or non-trivial implementation detail must have been used. According to personal communication with the current project maintainers, a technique similar to merging destination nodes with equal successors (section 3.2.2) was used.

Evaluation In general the tables reflect the sizes of the underlying GTFS data sets. In larger networks there are more transfer patterns. The average number of optimal transfer patterns between two stations is an indicator for the variance among the optimal paths during the course of the day, and for the number of routes with a different trade-off between travel time and number of transfers.

In the three tables the influence of the walking distance becomes visible: Generally speaking a larger walking radius leads to a smaller output size. When routes can walk farther in the beginning, often a faster connection can be reached, which dominates other routes. A longer distance for final walking causes more labels being dominated in the arrival chain algorithm. This makes the optimal paths more uniform. As the patterns are stored without marginal walking, the number of patterns decreases. However, no limit to the walking distance would render the computation time impracticable, so this is not a good idea. For instance, the computation of the

transfer patterns for New York City with 1 000 meter walking radius did not finish in time to be included in this thesis.

We now inspect the difference between the representations. Starting with the first two columns, we observe that the number of internal nodes is slightly decreased for TP^c. This is the effect of the compact representation of equal suffixes achieved with the algorithm of section 3.2.1. The reduction of the number of internal nodes ranges from one eighth to one fourth between the different data sets. A more significant improvement is the reduced number of destination nodes with the representation TP^c. This is on the account of the merging of these nodes with the technique from section 3.2.2. It removes between 50% and 80% of the destination nodes depending on the data set. Because the destination-nodes usually have more than one successor (other than internal nodes), this technique mainly accounts for the drop in the number of arcs. With the destination-nodes representing the majority of nodes in the original DAGs of TP, the reduced memory size of this representation is mostly due to the merging of these nodes.

The third column ($jDAG$) shows the size of the structures when using a joint DAG from section 3.2.3 instead of one DAG for each station. This significantly decreases the number of internal nodes. As every internal node in the original DAGs has at most one outgoing arc, the reduction of the number of arcs is almost equal. Compared to the sum of internal nodes of the separate DAGs, the joint DAG has between 78% and 88% less nodes. Because the number of saved arcs is not as large as for TP^c, a little less memory is saved with this representation.

Finally the fourth column ($jDAG^c$) shows the results for the patterns represented by a joint DAG, which is compressed with the two other methods. Compared to the plain $jDAG$, the number of internal nodes decreases further, even if the effect of the compact representation of suffixes is not as large as between the columns one and two. Merging equal destination nodes works even better than for TP^c. When located in separate graphs, the redundancy of the destination-nodes between the graphs cannot be resolved. Because all the nodes are now part of the same graph, this becomes possible.

Clearly, $jDAG^c$ is the most compact of these representations. Compared to the original data structure, the internal parts of the graphs shrinks by 80%. The number of destination nodes decreases by 65%–95%. The improvements are relatively independent from the different walking radii in the computation of the transfer patterns, but the savings by merging equal destination nodes seems to be a bit lower for large walking distances. Most probably this is because for long walkways there are less patterns anyway, and so there is less redundancy.

The increasing compactness of the graph structures does not express proportionally in the decrease of memory consumption. This is due to the tables which indicate the entry-point for all destinations reachable from a station. These mappings stay the same for all four representations and form the largest part of the information in $jDAG^c$.

Table 3.2: Size of the transfer patterns in different representations for Hawaii, Detroit, Toronto and New York City with walking radius 100m. The numbers in the dataset-headers denote the total number of transfer patterns represented and the average number of transfer patterns between two stations. The columns show different representations: TP is the original DAG structure. TP^c applies the suffix and destination node compression from sections 3.2.1 and 3.2.2. $jDAG$ uses the joint DAG from section 3.2.3. $jDAG^c$ applies the compression methods on top of the joint DAG.

	TP	TP^c	$jDAG$	$jDAG^c$
Hawaii@100m: 51.0M, 5.8				
# internal nodes	7.2M	5.4M	822.4K	726.5K
# destination nodes	8.9M	2.5M	8.9M	1.3M
# arcs	58.3M	26.2M	51.9M	15.0M
Memory size (Byte)	497.4M	270.4M	394.7M	155.6M
Byte/pattern	9.7	5.3	7.7	3.0
Detroit@100m: 61.8M, 2.9				
# internal nodes	8.1M	7.1M	755.8K	716.3K
# destination nodes	21.0M	3.9M	21.0M	1.4M
# arcs	69.9M	24.4M	62.6M	9.5M
Memory size (Byte)	797.0M	397.0M	680.2M	232.1M
Byte/pattern	12.9	6.4	11.0	3.8
Toronto@100m: 215.8M, 4.9				
# internal nodes	36.8M	27.8M	4.5M	3.9M
# destination nodes	44.5M	13.3M	44.5M	5.8M
# arcs	252.7M	116.7M	220.3M	56.7M
Memory size (Byte)	2.3G	1.3G	1.8G	700.0M
Byte/pattern	10.9	6.1	8.5	3.2
New York City@100m: 611.6M, 5.2				
# internal nodes	114.8M	91.2M	19.5M	16.9M
# destination nodes	117.5M	46.6M	117.5M	26.8M
# arcs	726.4M	400.5M	631.1M	251.7M
Memory size (Byte)	6.6G	4.2G	5.1G	2.5G
Byte/pattern	10.8	6.9	8.4	4.0
	TP	TP^c	$jDAG$	$jDAG^c$

Table 3.3: Size of the transfer patterns in different representations for Hawaii, Detroit, Toronto and New York City with walking radius 500m. The numbers in the dataset-headers denote the total number of transfer patterns represented and the average number of transfer patterns between two stations. The columns show different representations: TP is the original DAG structure. TP^c applies the suffix and destination node compression from sections 3.2.1 and 3.2.2. $jDAG$ uses the joint DAG from section 3.2.3. $jDAG^c$ applies the compression methods on top of the joint DAG.

	TP	TP^c	$jDAG$	$jDAG^c$
Hawaii@500m: 34.4M, 5.2				
# internal nodes	4.6M	3.4M	648.9K	578.5K
# destination nodes	6.6M	1.8M	6.6M	1.1M
# arcs	39.0M	16.7M	35.0M	11.2M
Memory size (Byte)	343.4M	183.0M	279.6M	118.1M
Byte/pattern	10.0	5.3	8.1	3.4
Detroit@500m: 64.5M, 3.6				
# internal nodes	8.5M	6.9M	932.3K	880.8K
# destination nodes	17.7M	3.8M	17.7M	1.9M
# arcs	73.0M	26.8M	65.4M	14.0M
Memory size (Byte)	748.2M	377.4M	626.7M	230.5M
Byte/pattern	11.6	5.9	9.7	3.6
Toronto@500m: 238.3M, 6.8				
# internal nodes	36.6M	28.6M	4.9M	4.4M
# destination nodes	35.1M	13.8M	35.1M	8.4M
# arcs	274.9M	150.3M	243.1M	101.0M
Memory size (Byte)	2.2G	1.4G	1.7G	838.8M
Byte/pattern	9.4	5.8	7.3	3.5
New York City@500m: 382.9M, 5.3				
# internal nodes	72.6M	61.7M	16.1M	15.1M
# destination nodes	72.4M	35.7M	72.4M	25.0M
# arcs	455.5M	298.1M	399.0M	229.1M
Memory size (Byte)	4.1G	2.9G	3.2G	2.0G
Byte/pattern	10.8	7.7	8.5	5.2
	TP	TP^c	$jDAG$	$jDAG^c$

Table 3.4: Size of the transfer patterns in different representations for Hawaii, Detroit and Toronto with walking radius 1000m. The numbers in the dataset-headers denote the total number of transfer patterns represented and the average number of transfer patterns between two stations. The columns show different representations: TP is the original DAG structure. TP^c applies the suffix and destination node compression from sections 3.2.1 and 3.2.2. $jDAG$ uses the joint DAG from section 3.2.3. $jDAG^c$ applies the compression methods on top of the joint DAG. NYC is missing here because the computation did not finish in time.

	TP	TP^c	$jDAG$	$jDAG^c$
Hawaii@1000m: 25.1M, 4.5				
# internal nodes	3.4M	2.7M	542.5K	504.5K
# destination nodes	5.5M	1.7M	5.5M	1.0M
# arcs	28.5M	13.1M	25.7M	8.8M
Memory size (Byte)	265.8M	149.8M	219.7M	97.9M
Byte/pattern	10.6	6.0	8.7	3.9
Detroit@1000m: 48.8M, 3.4				
# internal nodes	6.7M	5.8M	821.1K	795.6K
# destination nodes	14.3M	3.5M	14.3M	1.7M
# arcs	55.5M	22.3M	49.7M	12.3M
Memory size (Byte)	588.3M	314.9M	494.1M	193.9M
Byte/pattern	12.0	6.4	10.1	4.0
Toronto@1000m: 162.0M, 5.8				
# internal nodes	24.2M	20.6M	3.2M	3.0M
# destination nodes	27.9M	11.7M	27.9M	7.2M
# arcs	186.2M	110.0M	165.2M	78.3M
Memory size (Byte)	1.6G	1.1G	1.3G	659.8M
Byte/pattern	9.8	6.5	7.8	4.1
	TP	TP^c	$jDAG$	$jDAG^c$

3.5 Summary

Addressing the first objective of this thesis, we proposed a variant of transfer pattern routing which is less informed and we developed a set of techniques representing the transfer patterns more compact. The first approach relies on a recursive construction of the search space, which allows to work with a fraction of the transfer pattern information, namely the first transfer. While this algorithm requires less memory to store the information, it has some major disadvantages compared to transfer pattern routing.

In the second part of this chapter, experiments emphasize the utility of the techniques explained in sections 3.2.1–3.2.3. It turns out that compressing equal suffixes in the transfer patterns is less advantageous than we expected. When storing longer sequences, this technique may become more beneficial. Merging equal destination-nodes proved to be a very important approach to reduce the size of the information. It transpires that another large amount of redundancy can be resolved by the joint representation of all patterns in one DAG. By proposing a more compact representation for the transfer patterns, we have accomplished the first objective.

4 Robustness of Transfer Patterns

4.1 Transfer Patterns and Real-Time Updates

The substantial drawback of transfer pattern routing is the expensive computation of the patterns. For the New York City dataset with 1000 meters walking distance for example, it takes about two weeks on a machine with 20 cores with the settings from section 3.4. When the search answers queries using the transfer patterns, it assumes that the timetable data on which the patterns were computed is still valid. But due to delay, this is not true in general. This chapter investigates whether the information of the transfer patterns is still valid with realistic real-time updates to the underlying network. In other words, we search an answer to the question if transfer pattern routing finds optimal routes when connections are delayed.

In the beginning of this chapter we explain our methodology in addressing this question and how we model real-time updates in the network. After that, we present and discuss experimental results.

4.2 Measuring Robustness

Remember that transfer pattern routing has two main components: There are the transfer patterns representing optimal connections between stations independent of the departure time, which are used to construct a relatively small query graph (section 2.4.1). And there is the efficient data structure for direct connection queries, which allows for fast search on these graphs (section 2.4.2). If a connection in the underlying transportation network changes, it is hard to say which optimal connections the change affects. A new computation of the transfer patterns is impractical, especially if the timetable data is updated frequently. In contrast, the direct connection data can be updated within a few minutes ([2], table 2).

We say the transfer pattern information is robust to real-time updates, if a search using updated direct connection data and the original patterns finds optimal paths. To study the robustness, we generate the time-expanded graph for one traffic day and compute the transfer patterns. Then different delay scenarios are applied to the network and to the direct connection data. In one experiment per scenario, a large number of location-to-location queries is answered by transfer pattern routing and the resulting paths are compared to the optimal paths found by multi-label Dijkstra

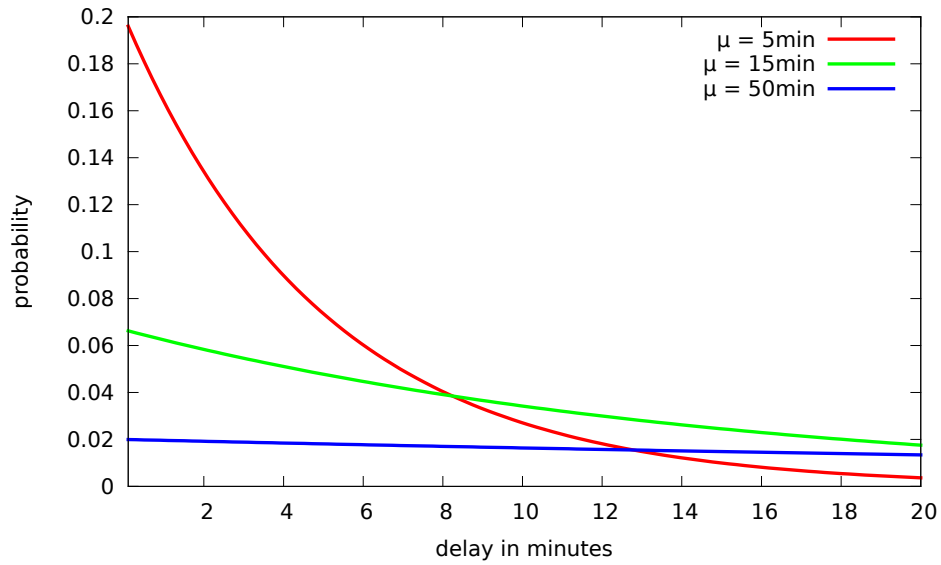


Figure 4.1: Probability density functions for exponential distributions $p(x) = a \cdot exp(-ax)$, $a = 1/\mu$ of different mean μ.

on the updated time-expanded graph. The next section will go into more detail about how the delay is modeled.

4.3 Modeling Delay

Each scenario partitions the set of trips into groups of common average delay. A random subset of the trips is selected. All the trips in that set are delayed with a random number of seconds drawn from the same exponential distribution. For every trip the random delay is added to the stop times, starting at a random position along its sequence of stops.

In the experiments we compare six different delay scenarios. There are three generic scenarios where one quarter of the connections are delayed with 5 (Low), 15 (Medium) and 50 minutes (High) delay in average. Figure 4.1 shows the probability density functions for these average delays. In addition, we generate three scenarios with different groups of average delay, which we call Switzerland, Germany and India. They have an increasing amount of delay. In the last scenario, every trip is delayed. The scenarios are specified in table 4.1.

With the introduction of hub stations and a transfer limit for local profile queries, transfer pattern routing becomes suboptimal. It cannot find optimal paths from non-important stations to destinations with more than three transfers, which do not transfer at a hub. Therefore, we perform the same queries on the unchanged network too, forming the baseline Null for the delayed scenarios.

Table 4.1: Delay scenarios applied to the timetable data.

Scenario	Share of trips and average delay
LOW	25% : 5 min
MEDIUM	25% : 15 min
HIGH	25% : 50 min
SWITZERLAND	10% : 5 min, 3% : 15 min, 1% : 50 min
GERMANY	20% : 5 min, 10% : 15 min, 5% : 50 min
INDIA	40% : 5 min, 40% : 15 min, 20% : 50 min

4.4 Experiments

Setup In the experiments we evaluate the robustness of the transfer patterns computed in chapter 3. For each data set we focus on the patterns computed with the largest walking distance, as this gives the most realistic results.

We issue random location-to-location queries with departure times from an interval of 24 hours starting at 4am. This is repeated until 50,000 queries with at least one connection have been answered. To account for varying traffic density during the day, the departure time is selected in the following way: With probability 0.5 it is drawn uniformly from the rush hours 6am–10am and 16pm–20pm. Otherwise it is drawn from the full time range 4am–4am. Thus, the probability that a query is issued during the rush hours is four times as high as on other daytimes. Although both routing algorithms could answer queries from arbitrary locations correctly, comparing the resulting paths would be hard. For the sake of simplicity we choose random stations and take their geographic coordinates as departure or destination location. The selection follows the density of traffic in different areas. A station is selected with a probability that respects the number of incident connections at this station. Let n_s denote the number of departure events at station s. Then the probability that s is selected is

$$p_s = \frac{\sqrt{n_s}}{\sum_{s' \in S} \sqrt{n_{s'}}}$$

Furthermore, we disallow queries where the distance between the departure and destination is less than $2r$, with r denoting the walking distance in the precomputation. Because via-walking is not possible in our model, such queries yield obscure paths a human person would never accept.

Each query $x \to y@t$ is answered by transfer pattern routing using the patterns computed on the original network and the direct connection data from the current scenario. The paths it finds are compared to the results of a multi-label Dijkstra on the scenario's time-expanded graph. To compute the latter, the graph is temporarily extended with two nodes according to section 2.2.4. Dijkstra's algorithm is started

from with a label $(0, 0)$ at the node for x and terminated as soon as all open labels are dominated by the labels at the destination node y.

The shortest paths found by the two algorithms are compared and the paths of transfer pattern routing are classified as follows: If a path of equal costs is among the paths found by Dijkstra's algorithm, the response is OPTIMAL. For every Dijkstra-generated path which has no equal path found by transfer pattern routing, the most similar path is selected. If there is no such path, the result is BAD. If there is a path with the same penalty, the duration difference is inspected. If the path found by transfer pattern routing is less than 5% of the total travel time *and* less than 5 minutes slower than the optimal path, it is ALMOST OPTIMAL A . If it is less than 10% *and* less than 10 minutes late, it is ALMOST OPTIMAL B. Otherwise the path is classified as BAD.

Class	Difference d to optimal path costs c^*
OPTIMAL	$d = 0$
ALMOST OPTIMAL A	$d \leq 5\text{min} \wedge \frac{d}{c^*} \leq 5\%$
ALMOST OPTIMAL B	$d \leq 10\text{min} \wedge \frac{d}{c^*} \leq 10\%$
BAD	otherwise

Results The tables and figures on pages 42, 43, 44 and 45 show the results of our experiments conducted for Hawaii, Detroit, Toronto and New York City. The classification results listed in the tables express the influence of the scenarios: With increasing average delay, the share of optimal responses decreases. The tables also show that for the baseline NULL the paths found by transfer pattern routing are mostly optimal. For each data set there are just a couple of suboptimal responses. Regarding the robustness, even for INDIA the share of suboptimal paths is never above 3.5%. Beside this, the classification results show that most of the suboptimal paths are quite close to the optimum: The major part is classified as ALMOST OPTIMAL A and the share of BAD paths is never larger than 0.7%. When examining the suboptimal responses in detail we observed a large number of paths which reach the destination with the same time of travel, but one more transfer.

For suboptimal paths the figures next to the tables show the distribution of the relative differences to the corresponding optimal path by a box-plot for each scenario. In these diagrams the red bar marks the median, the box contains the interval between the upper and lower quartile of the data. The whiskers have a length of 1.5 times the distance between the quartiles.

The influence of the scenarios' amount of delay is not reflected as clearly in the distributions as for the classification results above. For example, the distribution for the scenario GERMANY is less wide-spread as for SWITZERLAND in some data sets. However, we can observe that the median of the distribution is below 0.2 for all data sets. There is a noteworthy difference between the plots for Hawaii and Detroit

and the ones for Toronto and New York City. The relative cost differences of the former are generally larger. For the latter two data sets, the relative difference of the upper quartile is often within 0.1, meaning that 75% of the suboptimal responses are less than 10% off the optimal path cost. This dissimilarity between the data sets is probably a consequence of the different average path length for the networks: Hawaii and Detroit are small compared to the other two, so random paths have a smaller average time of travel. Moreover, the average number of patterns between two stations is larger for Toronto and New York City (see table 3.3, for example), so there are more alternatives available if the optimal route becomes suboptimal due to delay.

In the distribution of responses, there are plenty of outliers, some of which are far off from the optimal path. Manual inspection of these critical outliers showed that they typically stem from queries between remote places with bad connectivity.

These results indicate that transfer patterns are quite robust to delay. Even in the worst scenario the share of suboptimal responses is very small, and most of these paths are almost optimal.

Table 4.2: Classification of paths under different delay scenarios for **Hawaii**@1000m.

	OPTIMAL	ALMOST A	ALMOST B	BAD
NULL	99.99%	0.00%	0.00%	0.00%
LOW	99.80%	0.11%	0.01%	0.05%
MEDIUM	99.56%	0.31%	0.03%	0.08%
HIGH	99.34%	0.45%	0.04%	0.15%
SWITZERLAND	99.84%	0.11%	0.01%	0.03%
GERMANY	99.56%	0.27%	0.04%	0.12%
INDIA	98.50%	1.07%	0.11%	0.31%

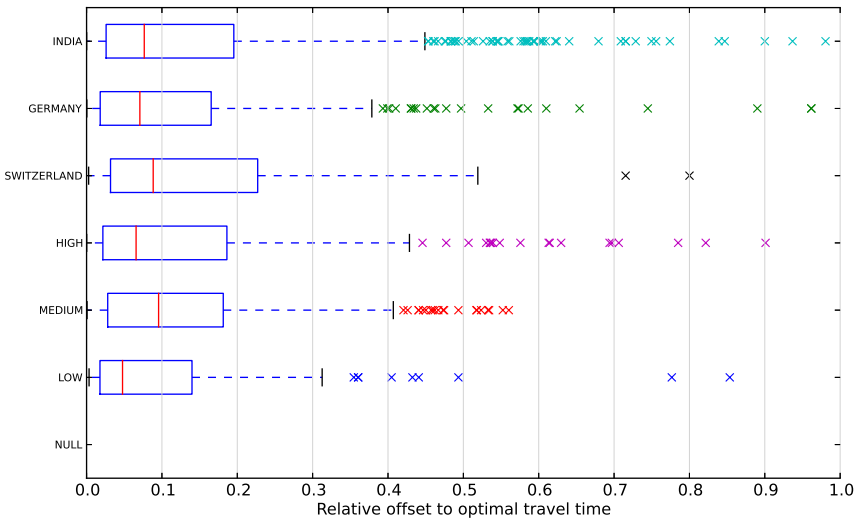


Figure 4.2: Suboptimal paths: Time of travel difference to the optimal path with equal number of transfers. Results for **Hawaii**@1000m. Outliers above 1.0 are not shown.

Table 4.3: Classification of paths under different delay scenarios for **Detroit**@1000m.

	OPTIMAL	ALMOST A	ALMOST B	BAD
NULL	99.99%	0.00%	0.00%	0.00%
LOW	99.19%	0.59%	0.07%	0.13%
MEDIUM	99.04%	0.67%	0.09%	0.18%
HIGH	98.77%	0.85%	0.11%	0.25%
SWITZERLAND	99.57%	0.31%	0.03%	0.07%
GERMANY	98.83%	0.85%	0.10%	0.20%
INDIA	96.56%	2.50%	0.35%	0.57%

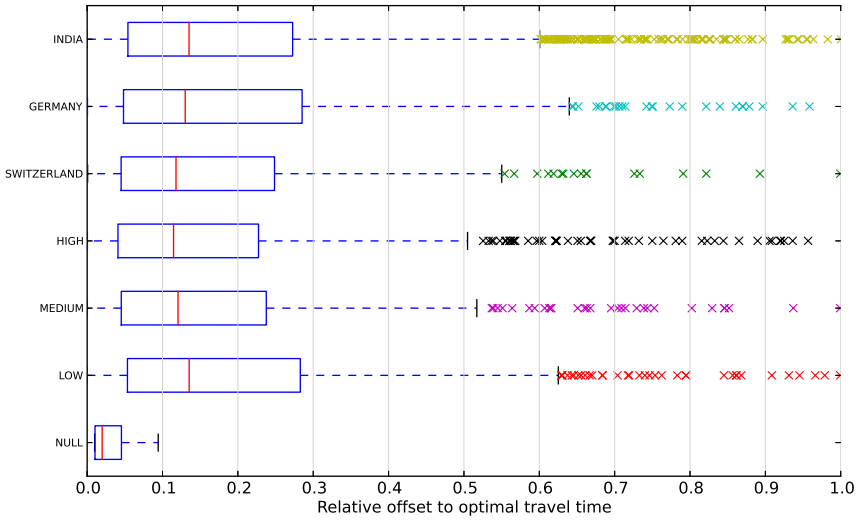


Figure 4.3: Suboptimal path: Time of travel difference to the optimal path with equal number of transfers. Results for **Detroit**@1000m. Outliers above 1.0 are not shown.

Table 4.4: Classification of paths under different delay scenarios for **Toronto**@1000m.

	OPTIMAL	ALMOST A	ALMOST B	BAD
NULL	99.98%	0.00%	0.00%	0.00%
LOW	99.73%	0.12%	0.04%	0.08%
MEDIUM	99.59%	0.20%	0.06%	0.13%
HIGH	99.49%	0.27%	0.07%	0.16%
SWITZERLAND	99.82%	0.09%	0.02%	0.05%
GERMANY	99.54%	0.22%	0.06%	0.16%
INDIA	97.85%	1.12%	0.31%	0.70%

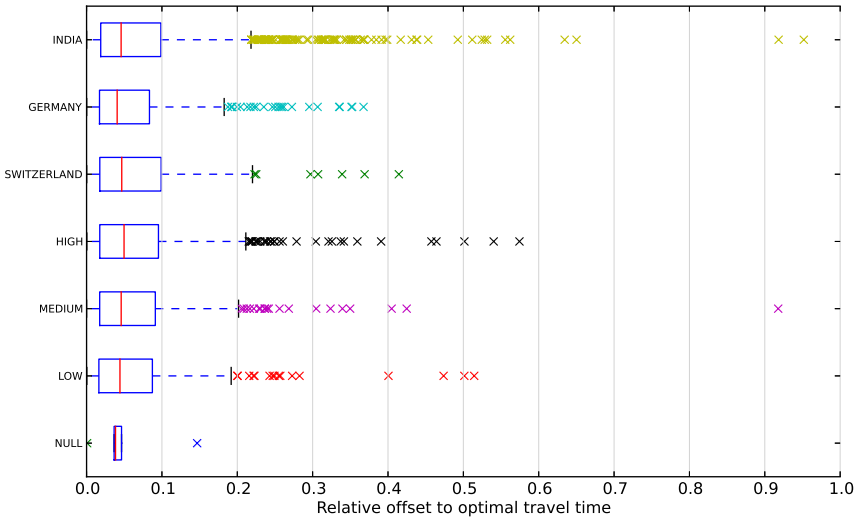


Figure 4.4: Suboptimal paths: Time of travel difference to the optimal path with equal number of transfers. Results for **Toronto**@1000m. Outliers above 1.0 are not shown.

Table 4.5: Classification of paths under different delay scenarios for **New York City**@500m.

	OPTIMAL	ALMOST A	ALMOST B	BAD
NULL	99.97%	0.02%	0.00%	0.00%
LOW	99.77%	0.12%	0.02%	0.07%
MEDIUM	99.72%	0.16%	0.03%	0.07%
HIGH	99.57%	0.27%	0.04%	0.09%
SWITZERLAND	99.84%	0.10%	0.01%	0.03%
GERMANY	99.66%	0.20%	0.03%	0.09%
INDIA	98.76%	0.74%	0.15%	0.33%

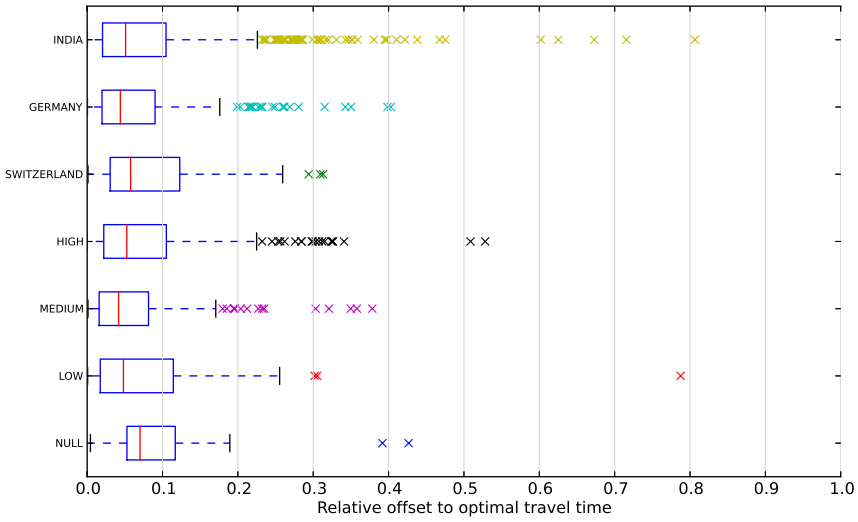


Figure 4.5: Suboptimal paths: Time of travel difference to the optimal path with equal number of transfers. Results for **New York City**@500m. Outliers above 1.0 are not shown.

5 Conclusion

In this chapter we discuss the importance of the results of this thesis. We emphasize its key contributions and suggest consequential topics for future research.

5.1 Compact Representation of Transfer Patterns

The main outcome of the first part is the identification, documentation and resolution of redundancy in the data structure proposed by the original article [2]. We apply existing techniques for isomorphic graph reduction to the transfer patterns and we propose new approaches to the representation, most noteworthy the joint DAG. The thesis successfully examines the combination of these methods and corroborates their utility by a set of experiments on real-world data.

In summary, the most effective of these advances allow to reduce the number of nodes in the internal graph by a factor between five and eight, in the total graph by a factor between five and ten. The total size of the data is reduced by 50%–75%.

Furthermore, this thesis documents the influence of walking on the size of the transfer patterns: While an increasing walking radius slows down the computation, it shrinks the size of the data. Also, we point out important pitfalls for the implementation of transfer patterns routing.

As a consequence of the removed redundancy in the graphs representing the transfer patterns, the main part of the data size is due to the mapping of destinations to entry-points, for which we suggest possible improvements. Future research should focus on a compact representation for these mappings.

Our experimental results indicate that the approaches benefit from heuristic improvements as the concept of important stations. However, we did not study this in detail. Also our computational capacity was too limited as to examine networks larger than New York City. So to some extend, the scalability of the presented approaches remains an open question.

Another topic arising from this thesis is a space-efficient implementation of the graph structures. We did not put much effort into that, nonetheless we think a refined implementation would use only half as much memory.

5.2 Robustness of Transfer Patterns

The results of the second part of this thesis indicate the robustness of transfer patterns. Even under extreme delay scenarios, the quality of the results is very high. The share of suboptimal responses is never above 3.5%. More than three quarter of the suboptimal paths are almost optimal. Less than one percent of all found paths is actually bad.

Beside the evidence for robustness, this thesis gives a more detailed statement about the quality of transfer pattern routing in the context of heuristic improvements. We document the share of suboptimal responses when computing transfer patterns using important stations and limited local profile queries in more detail than any of the publications on transfer patterns so far.

The inherent disadvantage of the random scenarios is that they model delay independently. In realistic public transportation, there are strong dependencies between connections. On the one hand, delay is often systemic. For example, a traffic jam will delay a series of trips, a bridge under maintenance will cause route deviations. On the other hand, there are mechanisms to compensate delay: A bus can drive faster to catch up with its schedule. Another example would be that in the EU, traffic agencies are bound by law to reimburse passengers for delay. Because of this the agencies have their own complicated calculations which sometimes make connections wait for a delayed train. The acquisition of realistic delay data and repetition of the experiments on top of that is one topic for future research.

Another problem is the dependency of the robustness towards parameters of the transfer pattern computation: How does the robustness change with different hub selection strategies or increasing walking distance?

Although the quality of the paths found is generally quite good, there are a few exceptions. Future work should focus on increasing the robustness of transfer pattern routing, which means reducing the share of suboptimal responses and the magnitude of outliers.

With the introduction of a new representation for transfer patterns, which has proven to be a lot more compact than previous approaches, and detailed experiments establishing the remarkable robustness of the routing algorithm, this thesis contributes to the knowledge about transfer pattern routing to a great extent.

Bibliography

[1] Hannah Bast. Car or Public Transport - Two Worlds. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 355–367. Springer, 2009.

[2] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns. In Mark de Berg and Ulrich Meyer, editors, *ESA (1)*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[3] Manuel Braun. Multi-Model Route Planning with Transfer Patterns. Master's thesis, Albert-Ludwigs-Universität Freiburg, December 2012.

[4] Mirko Brodesser. Multi-Modal Route Planning. Ongoing master's thesis, Albert-Ludwigs-Universität Freiburg, April 2013.

[5] Maxime Crochemore and Renaud Vérin. On Compact Directed Acyclic Word Graphs. In Jan Mycielski, Grzegorz Rozenberg, and Arto Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer, 1997.

[6] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard Watson. Incremental Construction of Minimal Acyclic Finite State Automata. *Computational Linguistics*, 26(1):3–16, 2000.

[7] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In David A. Bader and Petra Mutzel, editors, *ALENEX*, pages 130–140. SIAM / Omnipress, 2012.

[8] Edsger Wybe Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. In *FOCS*, pages 338–346. IEEE Computer Society, 1984.

[10] Robert Geisberger. *Advanced Route Planning in Transportation Networks*. PhD thesis, Karlsruhe Institute of Technologie, 2011.

[11] General Transit Feed Specification Reference. `https://developers.google.com/transit/gtfs/reference`.

[12] Guy Jacobson. Space-efficient Static Trees and Graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989.

[13] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit Representation of Graphs. In Janos Simon, editor, *STOC*, pages 334–343. ACM, 1988.

[14] Ronald Prescott Loui. Optimal Paths in Graphs with Stochastic or Multidimensional Weights. *Commun. ACM*, 26(9):670–676, 1983.

[15] Rolf H. Möhring. *Verteilte Verbindungssuche im öffentlichen Personennahverkehr: Graphentheoretische Modelle und Algorithmen*, pages 192–220. Vieweg, 1999.

[16] Matthias Müller-Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-criteria Pareto Search. In Frank Geraets, Leo G. Kroon, Anita Schöbel, Dorothea Wagner, and Christos D. Zaroliagis, editors, *ATMOS*, volume 4359 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2004.

[17] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12, 2007.

[18] Ben Strasser. Delay-Robust Stochastic Routing in Timetable Networks. Diploma thesis, Karlsruhe Institute of Technologie, July 2012.

[19] D. Theune. *Robuste und effiziente Methoden zur Lösung von Wegproblemen*. Teubner-Texte zur Informatik. Teubner, 1995.

[20] Trillium Solutions Inc. Opportunities to leverage GTFS. `https://docs.google.com/document/d/17CoxTgGQPoUenz1HCW3kdCnX0sciPLvTa_wGRPZcqPc`.

[21] György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.