Albert-Ludwigs-Universität Freiburg
Fakultät für Angewandte Wissenschaften
Institut für Informatik

**Masterarbeit**

# Multi-Modal Route Planning with Transfer Patterns

Manuel Braun

Dezember 2012

**Supervisor**
Prof. Dr. Hannah Bast

**Primary Reviewer**
Prof. Dr. Hannah Bast

**Secondary Reviewer**
Prof. Dr. Christian Schindelhauer

**Date**
2012-12-14

# Acknowledgments

First, I want to thank Prof. Dr. Hannah Bast for awakening my interest in route planning, supervising and reviewing this thesis, and the time she spent for all the afternoons of helpful and guiding discussions. I also appreciate her letting me take part in some interesting meetings about route planning. Furthermore, I want to specially thank Sabine Storandt for helpful discussions and the ideas and feedback she provided. I want to thank Prof. Dr. Christian Schindelhauer for reviewing this thesis as the secondary supervisor.

I also want to thank Jonas Sternisko and Mirko Brodesser for proofreading this thesis. Further thanks go to my friends, family, Guido van Rossum, Bjarne Stroustrup, and all others, whose work or support, knowingly or not, made this thesis possible.

# Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Inhalte als solche kenntlich gemacht habe. Desweiteren erkläre ich, dass diese Abschlussarbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens war oder ist.

Freiburg, den 14. Dezember 2012 Manuel Braun

## Abstract

We examine a way to extend routing using transfer patterns – a state of the art route planning approach for fast routing in public transportation networks – to a multi-modal scenario that allows using public transport, cars, and walking. We specifically examine how to precompute transfer patterns for multi-criteria, multi-modal route planning. When done naively with a Pareto cost model, this leads to a large number of undesirable route variations that render precomputation computationally infeasible. We identify reasons why these variations appear and present ways to reduce the number of variations with appropriate graph models and by applying limits during search. Our results indicate that precomputing multi-modal transfer patterns on large datasets is feasible with such a restricted model.

## Zusammenfassung

Wir untersuchen eine Möglichkeit, Routenplanung mit Transfer-Pattern – ein aktueller Ansatz für schnelle Routenplanung auf öffentlichen Verkehrsnetzen – an ein multi-modales Szenario anzupassen, das die Benutzung von öffentlichen Verkehrsmitteln, Autos und Laufen erlaubt. Namentlich untersuchen wir, wie sich Transfer-Pattern für multi-modale Routenplanung bezüglich multiplen Kriterien vorberechnen lassen. Macht man dies auf naive Art mit einem Pareto-Kostenmodell, führt dies zu einer großen Vielzahl von unsinnigen Variationen von Routen, was die Vorberechnung vom Berechnungsaufwand her unmöglich macht. Wir identifizieren Gründe weshalb solche Variationen entstehen und präsentieren Möglichkeiten, die Anzahl von Variationen durch entsprechende Graph-Modelle und durch Anwendung von Limits während der Suche zu reduzieren. Unsere Ergebnisse deuten darauf hin, dass die Vorberechnung von multi-modalen Transfer-Patterns auf großen Datensätzen mit einem solchen, eingeschränkten Modell vom Berechnungsaufwand her machbar ist.

# Contents

# 1 Introduction

Probably most of us are facing the question *how do I get from here to some target location* on a somewhat regular basis. When it comes to using public transportation, we've got used to being instantly provided with answers to that question by our mobile devices. Some of us would probably not even know any more how to figure out a non-trivial public transportation route without using a route planning system. Yet these systems still have limitations. One evident limitation is, that they are not designed for handling multiple modes of transportation. So even if there exists a preferable route that includes using a taxi or car sharing service, they will just not find it.

In this thesis, we investigate adapting Routing using Transfer Patterns [5] – a state of the art algorithm for routing on public transportation networks – to a multi-modal setting. We develop a better understanding for challenges that arise from such a setting and show ways to address them.

## 1.1 Motivation

Great progress has been made in developing very fast algorithms for routing on road networks, as well as on public transportation networks. These algorithms allow query times of only a few milliseconds or even microseconds even on huge networks and have proven their practical usefulness on real-world systems like Google Maps that answer millions of queries daily.

Yet there is still room for improvement. One issue is the limitation to one mode of transportation. When looking up a route for example on Google Maps, you are offered multiple options for the mode of transportation. You can choose if you want to walk, go by bike, use a car or use public transportation. This choice might be an over-simplification of the problem you are actually trying to solve. For example you might want to take your bike with you in public transportation. While you can obviously still do this with the routes offered when choosing public transportation, the route planner does not consider that you have a bike with you. There might exist a much faster route that includes a transfer between two somewhat distant train stations that only make sense to take when having a bike available. The route planner will just not find such

connections. As there are often bikesharing services available in big cities, it might not even be necessary to take your own bike for such a route to be advantageous. If you riding a bike appears like a completely unrealistic scenario to you, let's consider the following example. It can make perfect sense to take a train to the airport, take a flight from there and finally take taxi to your destination. This includes three modes of transportation (not counting walking), is a completely realistic, quite common scenario and nevertheless no existing route planning algorithm fully covers it.

One important characteristic of routing in public transportation networks is the lack of a single meaningful optimization criteria in real world scenarios. While for routing on road networks, travel time alone already makes up a quite good objective, for public transportation things turn out not to be so easy. Let's assume you want to go from Freiburg to Berlin by train. There is a direct train that takes six and a half hours and goes directly from Freiburg to Berlin. But there is also a connection that requires to transfer in Mannheim but takes only six hours. Is half an hour saving in travel-time worth one more transfer? This cannot be answered in general but depends on the users individual preference. For example when travelling with a lot of luggage, the zero-transfer connection will probably be preferable, while when being under time-pressure, the faster connection will be it. Considering multiple modes of transportation adds even more options to go to Berlin. For example one could also take a taxi to Basel and take a flight to Berlin from there. While this the fastest option, it requires at least two transfers (at the airports) and using an airplane. It is easy to see, that for multi-modal route planning, a single optimization criteria makes even less sense than for public transportation alone.

In this thesis, we investigate multi-criteria, multi-modal route planning. Specifically we stick with walking, car and public transportation as modes of transportation. While other modes of transportation like planes or bicycles are inarguable of practical importance, we believe that this setting is a good staring point that already covers many challenging aspects of multi-modal route planning.

## 1.2 Background

### 1.2.1 Route Planning in Road Networks

A basic approach to route planning in road networks models it as graph search problem. The road network is represented as a weighted graph, nodes correspond to geographical locations and edges to roads that connect them. Costs are assigned to edges, for example travel-time or the distance between two adjacent nodes on the road. The problem of finding a route then compiles down to finding a path with minimal costs (sum of edge weights) between a designated source and target node. It is also called the *shortest path*

*problem.*

One of the simplest algorithms that solves the shortest path problem is **Dijkstra's algorithm**. Dijkstra claims to have invented it in about 20 minutes in his head, without pencil and paper, while sitting tiredly in a café in Amsterdam with his fiancée [16]. It was published three years later [13] and there was little interest in the shortest path problem at that time, as Dijkstra stated in an interview:

> Now, at the time, an algorithm for the shortest path was hardly considered mathematics: there was a finite number of ways of going from A to B and obviously there is a shortest one, so what's all the fuss about? [16].

Dijkstra's algorithm computes the shortest path from a single source node to all other nodes in the graph. It is commonly implemented by maintaining tentative distances to all nodes and a priority queue, that is initialized with zero for the source node and infinity for all other nodes. Then, iteratively, the node with minimal tentative distance is taken out of the priority queue (settling a node) and its incident edge weights are used to update tentative distances of adjacent nodes (relaxing edges). Whenever a tentative distance improves, it is added to the priority queue together with its respective node.

Route planning in road networks has received much research attention since to speed up query times. Most of the algorithms rely on some sort of preprocessing step, that is computed offline and helps to speed up actual shortest path queries. The performance of such algorithms is usually stated in terms of speed-up achieved in comparison with Dijkstra's algorithm at query time. An trivial example for such a precomputation scheme would be to compute a complete distance table between all pairs of nodes. Queries can then be answered by simple table lookups. Obviously, this is intractable for most real-world graphs due to the quadratic time and space requirement.

**Contraction Hierarchies** [15] is a simple and yet effective state of the art speed-up technique. It works by incrementally contracting nodes in order of importance, while assuring that shortest paths between all yet uncontracted nodes are preserved by adding shortcut edges between adjacent nodes, if necessary. The order of contraction induces an ordering of the nodes. It is guaranteed that in the resulting graph shortest paths exist, consisting of only two types of node sequences. One with strictly increasing node order and one with strictly decreasing node order. These shortest paths can be found by a bidirectional Dijkstra search in the so called upward and downward graph (ignoring all edges with non-increasing or non-decreasing node order). This limits the search space dramatically. The authors achieved query times of only a few hundred microseconds for a road network of Western Europe with over 18 million nodes and 42 million edges. The order of node contraction is an important ingredient for this algorithm. During the contraction phase, a priority queue is used to determine the node to be contracted next. A linear combination of several terms is used as priority, for example Edge Difference

(number of shortcuts introduced when contracting a node minus number of incident edges) or the number of already contracted neighbours. We refer to the original work for a more detailed discussion.

**Transit Node Routing** [7, 6, 8] is based on the intuitive observation that when driving somewhere far away, the current location is left through only a few important traffic junctions. A complete distance table is computed between such junctions called *transit nodes*, which allows to answer queries between far-away nodes in only a few microseconds by a few table lookups. The set of transit nodes that are first encountered when starting from a specific node are called its *access nodes*. The number of access nodes is small on road networks (about 10), so they can be explicitly stored. In order to compute the shortest-path distance between a source and target node, their respective access nodes are looked up to together with their pairwise distances. The shortest path distance is just the minimum of these distances plus the distance to the access nodes. Local queries can be answered using any existing technique.

A more detailed overview of routing algorithms can be found in the PhD thesis of Geisberger [14].

### 1.2.2 Route Planning in Public Transportation Networks

**Modelling** We have seen that modelling a road network as a graph is quite easy. For public transportation networks, this is a bit more complicated. Public transportation networks are inherently time-dependent and a route typically does involve transfers between vehicles. Transfers from one vehicle to another can not happen instantly but require a minimum amount of time to get off the first vehicle, walk to the other vehicle and board it. This time required to switch vehicles is called the *minimum transfer duration*.

We assume that a public transportation network is given as timetable data that consists of a set of *stations* $\mathcal{S}$ (train stations, bus stops etc.) and a set of trips. A trip is given as the sequence of stations served by a single vehicle $((s_1, t_{a1}), (s_1, t_{d1}), (s_2, t_{a2}), (s_2, t_{d2}), \dots)$ where $t_{ai}$ is the *arrival time* and $t_{di}$ is the *departure time* of the vehicle at station $s_i$. We call $((s_1, t_{d1}), (s_2, t_{a2}))$ an *elementary connection* from $s_1$ to $s_2$ (a train connects both stations without intermediate stops).

One way to model a public transportation network as a graph, is the **time-expanded model**. For each arrival and departure event (i. e. the event of a train arriving or departing from a station) an *arrival node* or *departure node* is added to the graph. Edges between these nodes either correspond to travelling on a train between two stations or waiting at a station. At a station, it is possible to transfer between trains, as long as the minimum transfer duration is respected. One way to achieve this is to add a *transfer*

*node* next to each departure node at the same time and connect it to the departure node with a zero-cost edge. Transfer nodes at a station are connected in increasing order of time using the associated waiting time as costs. Finally arrival nodes are connected with the first transfer node that respects the minimum transfer duration. Please note that this yields a static graph, that in principle all graph search methods like Dijkstra's can be used on without any modification. An example is shown in Figure 1.1.

An alternative model is the **time-dependent model** which has a node for each station and edges between two stations if there is an elementary connection between them. This yields a much smaller graph than the time expanded model but requires non-trivial treatment to handle minimum transfer durations. An example is shown in Figure 1.2.

The **line model** also called *train-route-model* consists of a node for each station and an additional node for each line-station-pair, for stations served by the respective line. A *route* or *line* is a set of trips that share the same sequence of stops. This yields a small graph and handling transfers correctly is easy. An example is shown in Figure 1.3.

Please refer to the work of Pyrga et. al. [18] for a more detailed discussion on modelling.

**Algorithms** It has been observed that methods which yield great speed-ups on road networks, generally do not perform well on public transportation networks [4]. This is especially the case when considering large municipal areas with poor structure. One reason for this is that there is hardly any hierarchy in public transportation networks, at least not within a city area. Methods like contraction hierarchies only work well if there actually is hierarchy they can exploit. Transit node routing does not work well due to the lack of efficient local search algorithms. A detailed discussion on the difference between routing on road and public transport networks can be found in an article by Hannah Bast [4].

A **multi-label** variant of Dijkstra's algorithm can be used to compute a set of multi-criteria shortest paths on any of the models described earlier. It maintains a set of labels for each node in the graph. A label carries a cost-tuple with multiple components instead of just a scalar cost. For example the tuple $(t, p)$ can be used as cost where $t$ is the total travel-time (including time for transfers and waiting) and $p$ is the number of transfers. We say cost *a dominates* cost *b in the Pareto sense* if it is strictly better in at least one component $\exists i : a_i < b_i$ and not worse in all other components $a_i \leq b_i \forall i$. Note that this definition is not restricted to two components. Two labels that do not dominate each other are called *incomparable*. Initially all but the starting label sets are empty. Just like in the scalar version of Dijkstra's algorithm, yet unprocessed labels are added to a priority queue. In every step a minimal label is taken from the priority queue and the incident edges of the associated node are relaxed. An edge is relaxed by calculating

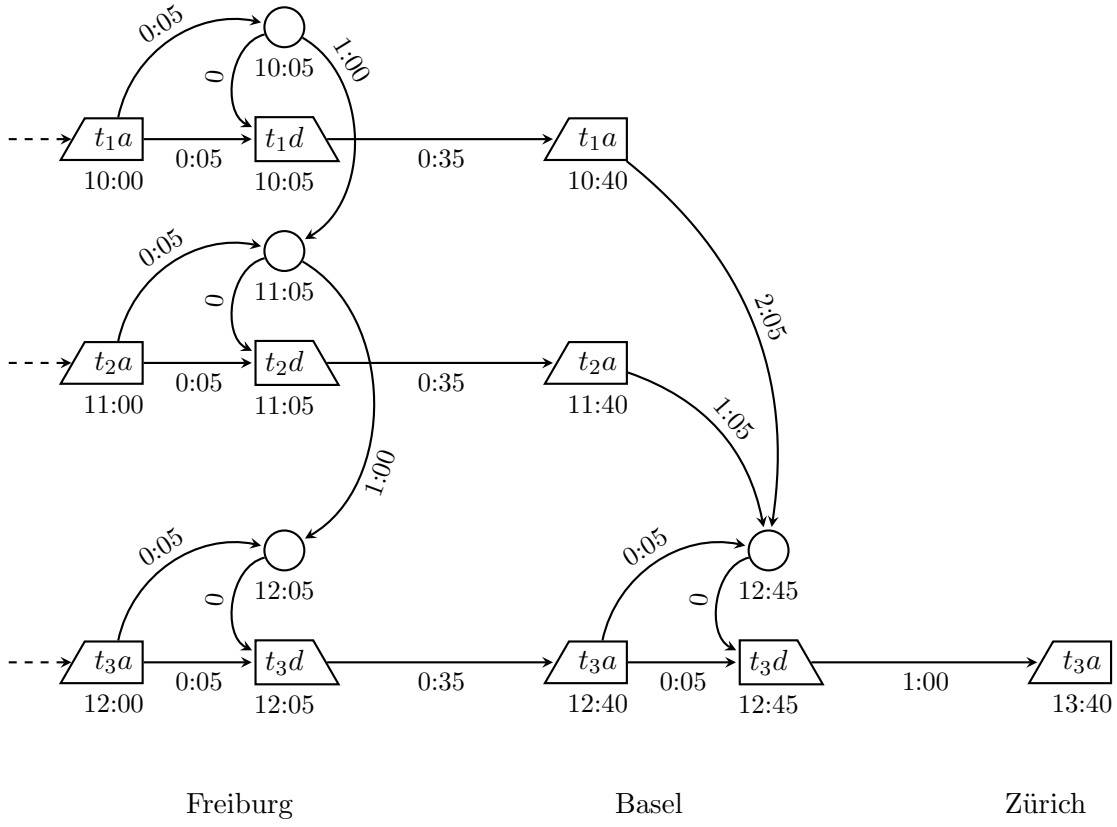


**Figure 1.1:** Time-expanded model showing three trips. The first two trips $t_1, t_2$ go from Freiburg to Basel. The last trip $t_3$ goes from Freiburg via Basel to Zürich. Circle shaped nodes are transfer nodes. Nodes denoted with $a$ and $d$ are arrival and departure nodes. Labels on edges denote the travel-time along an edge.

$$c_t(e_1, t) = \begin{cases} 10{:}00 - t + 0{:}35 & t \leq 10{:}00 \\ 11{:}00 - t + 0{:}35 & 10{:}00 < t \leq 11{:}00 \\ 12{:}00 - t + 0{:}35 & 11{:}00 < t \leq 12{:}00 \end{cases}$$

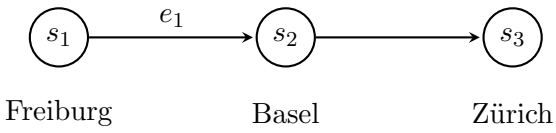


**Figure 1.2:** Time dependent model showing a connection from Freiburg via Basel to Zürich. This can either be a direct connection or one with a transfer in Basel. The example cost function for $e_1$ shows the travel-time component from Freiburg to Basel, for trains leaving at 10:00, 11:00 and 12:00 at Freiburg and take 0:35 to Basel.

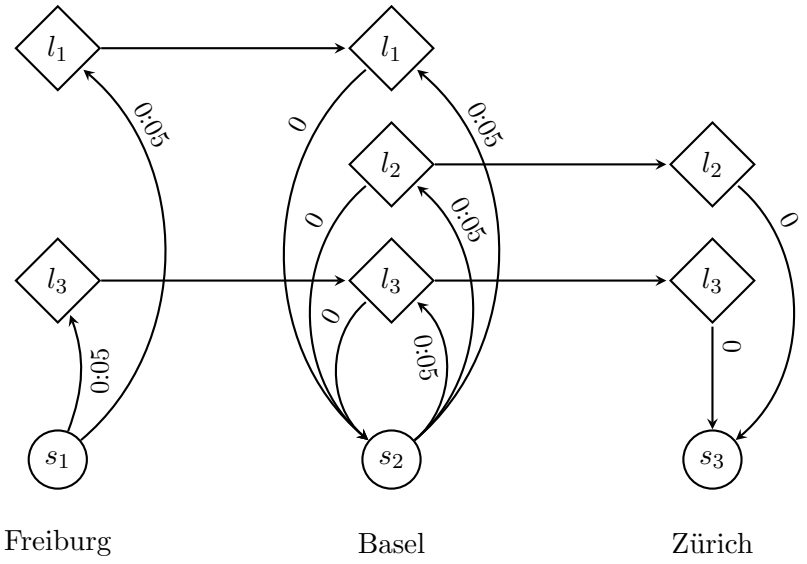


**Figure 1.3:** Line model with three lines and a minimum transfer duration of 0:05. Line $l_1$ goes from Freiburg to Basel, $l_2$ from Basel to Zürich and $l_3$ from Freiburg via Basel to Zürich.

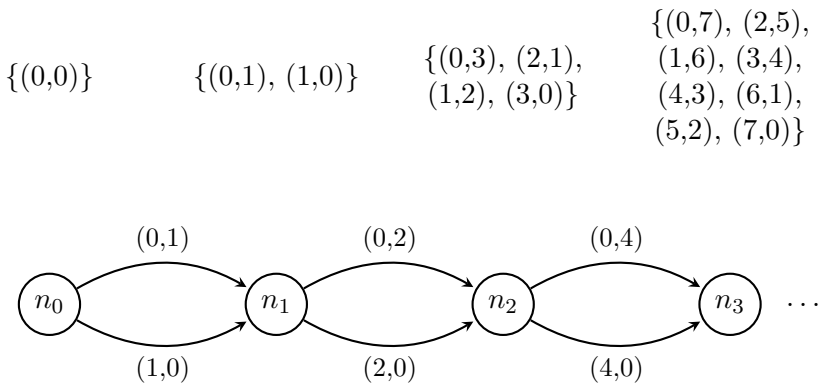


**Figure 1.4:** Example graph and associated label sets for paths starting at $n_0$ with cost $(0, 0)$. The number of Pareto-optimal labels grows exponentially as each path yields a disjunct label. At node $n_i$ the number of labels is $2^i$. The usage of double edges is not of importance for this example. The same effect can be achived with single edges by adding an intermediate node between two nodes instead.

the sum of the cost of the label that is currently being processed and the cost on the edge. If the resulting cost $c$ is dominated by any cost in label set of the head, no update is required. Otherwise $c$ is added to the set of labels and to the priority queue. If $c$ dominates any of the existing labels, they are removed from the label set. This ensures that all remaining labels are pairwise incomparable.

It is important to note that in theory, the number of Pareto-optimal labels is not bound polynomially by the size of the input graph. This already holds for costs with two components and can easily be seen by considering the example given in figure 1.4. In this example, the number of labels grows exponentially with the size of the graph. As the result of the output can already grow exponentially, obviously no algorithm with polynomial runtime exists. So with a Pareto cost model, the size of the output (and runtime of any algorithm) essentially depends on the actual graph and cost structure. For public transportation networks, travel-time and the number of transfers $(t, p)$ have been observed to behave well. Intuitively, this is because the number of transfers is discrete and bound by a small constant for the fastest routes. For example if the fastest route from A to B requires 3 transfers, then the number of labels for routes from A can be at most 4 (assuming a single departure-time).

**Routing using Transfer Patterns** [5] has been proposed as a method that achieves fast query times of only a few milliseconds even on very large transportation networks. This is achieved by precomputing optimal *transfer patterns*, i. e. sequences of transfer

stations on a shortest path. For a given source and target station, the number of optimal transfer patterns is small. Given the set of optimal transfer patterns from a source to a target station, a query can be quickly answered with few direct connections lookups.

Consider the following example from the original paper that we reproduce here for the sake of completeness: Assume we want to find shortest paths from Freiburg to Zürich, departing at 10:00. Further assume that we are given the information that any optimal path from Freiburg to Zürich, no matter on which time or day, either goes directly from Freiburg to Zürich, or has exactly one transfer at Basel. Further assume that we can quickly look up direct connections between two stations. With this information, answering the query is easy. Find direct trains from Freiburg to Zürich, that leave no earlier than 10:00. Let's assume there is a train that leaves Freiburg at 12:55 and arrives in Zürich at 14:52. Also find the next direct connections from Freiburg to Basel that departs no earlier than 10:00. Say it leaves at 10:02 and arrives in Basel at 10:47. Then find the next direct connection from Basel to Zürich at 10:47. Say it leaves at 11:07 and arrives in Zurück at 12:00. In the cost model used in the paper, these two connections are incomparable (one is faster, one has less transfers) and thus are reported both. It is guaranteed that the found connections are the only optimal ones, as no other optimal transfer pattern exists. The set of optimal transfer patterns is very little information and yet it decreases the search space dramatically.

Transfer patterns for a given station $A$ are computed by running a multi-label Dijkstra profile query from all transfer nodes of station $A$ in the time-expanded model. Then for each target station $B$, a set of dominant labels is computed for all arrival times at $B$. This is done by reading off the labels at the individual arrival nodes, while taking into account that it can be better to arrive at an earlier node and just wait at the target station. Then all resulting labels are tracked back, yielding the set of optimal transfer patterns from $A$ to $B$. Transfer patterns are stored space-efficiently in a DAG, exploiting the fact many shortest paths share similar transfer patterns.

In order to evaluate a query from station $A$ to $B$, the corresponding transfer patterns are used to build a *query graph*. The query graph is a DAG that consists of nodes for stations $A$ and $B$ and all stations that occur in any transfer pattern from $A$ to $B$. Edges represent direct connections as given by the transfer patterns. So for example the two transfer patterns (Freiburg, Basel, Zürich) and (Freiburg, Zürich) would result in a query graph with edges from Freiburg to Basel, from Basel to Zürich and from Freiburg to Zürich. A query is then evaluated by running a multi-criteria Dijkstra on the query graph, asking a special direct connection structure for the appropriate edge costs.

The *direct connection data structure* consists of:

- A set of lines. A line is list of trips that share the same sequence of stops, sorted by departure time, with the additional requirement that they do not overtake it

other. The latter ensures the FIFO property of a line, meaning it is never better to depart with a later trip. A line is stored in a 2D array like this:

| Line ICE 123 | Basel | Freiburg | | Karlsruhe | | Mannheim |
|---|---|---|---|---|---|---|
| trip 1 | 14:22 | 14:55 | 14:57 | 15:58 | 16:00 | 16:22 |
| trip 2 | 15:22 | 15:55 | 14:57 | 16:58 | 17:00 | 17:23 |
| ... | ... | ... | ... | ... | ... | ... |

- A sorted list of incident lines for each station with the associated stop position(s) of the line on the station. So for example consider a line ICE 123 with stop sequence Basel, Freiburg, Karlsruhe, Mannheim, Frankfurt and a line ICE 321 with stop sequence Stuttgart, Mannheim, Karlsruhe, Freiburg, Basel. Then the list of incident lines for Freiburg are: (ICE 123, 2), (ICE 321, 4) as ICE 123 has its second stop in Freiburg and ICE 321 has its 4th stop in Freiburg.

A direct-connection query from station A to station B at departure time $t$ is answered by:

1. Intersecting the incident lists of $A$ and $B$. Please note that this can be done efficiently as the incident lists are sorted. It yields all lines that have stops at $A$ as well as on $B$ with associated stop positions. It is only necessary to consider lines that have a stop with earlier stop position on $A$ (otherwise they only go in the wrong direction).
2. For each such line, reading of the costs for the earliest feasible trip (departing no earlier than $t$ at $A$) and returning the minimum one.

Please refer to the original work [5] for more details.

**RAPTOR** [11] is an approach to multi-criteria public transport routing. It does not rely on preprocessing, which allows to use it in dynamic scenarios. Opposed to most other routing methods, it is not based on Dijkstra but instead operates in rounds. Starting at a source station at a given time, it computes the earliest arrival time any other stop can be reached by using at most $k$ trips. This is repeated for increasing numbers of $k$ until no improvement is achieved anymore. In each round, each route is scanned at most once. The authors have achieved speedups of about one order of magnitude compared to a multi-label Dijkstra implementation on large metropolitan datasets like London or New York.

### 1.2.3 Multi-Modal Route Planning

Multi-modal route planning combines multiple modes of transportation into a single problem. Delling, Pajor et. al. [10, 17] proposed **Access-Node Routing** that adapts

the idea of transit-node routing to a solve a multi-modal label constrained shortest path problem, with road usage restricted to the beginning and end of a path. This separates routing in the road and public transportation networks into two problems. In their experiments the authors use a simple single-criteria Dijkstra implementation to compute routes in the public transportation network, but they point out that a speed-up technique could be used instead. The approach is limited to long-range queries, for local queries a separate algorithm has to be used.

Dibbelt et. al. [12] describe an approach to speed up single-criteria label constrained shortest path queries based on contraction hierarchies. It does not fix the constraints during precomputation but sees them as input that is individually given by the user at query time. Graphs for the individual modes are contracted independently while ensuring that transit nodes (where mode change occurs) are not contracted. This way, a shortcut never includes a modal change. Opposed to Access-Node Routing, it also supports local queries.

Besides label constrained approaches, it has been proposed to use linear combinations of multiple criteria to compute multi-modal shortest paths using a single-criteria routing algorithm [19, 3]. This requires to choose a set of weights upfront, which we think is undesirable. While one could argue that one could run such an approach with multiple different weights to generate a diverse set of route suggestions, this still implies trade-offs, like *30 minutes of travel time equals one transfer* or *taking a taxi is like taking two buses* in the actual routing, which seems a bit arbitrary. It is unclear if not doubtable if one can guarantee not to miss interesting routes which such an approach.

In a very recent work, Delling et. al. [9] describe an approach to multi-modal, multi-criteria route planning. The authors describe an algorithm based on RAPTOR [11] that computes a full Pareto set of optimal costs of multiple criteria such as travel time, number of transfers, walking duration and costs for using a taxi. As this set can get very large, they introduce a generalized definition of domination using fuzzy logic, which is then used in a post processing step to filter it. They also introduce some heuristics, for example already using a fuzzy definition of domination during search or discretization of costs. Using these heuristics results in giving up optimality. The authors have observed, that including taxis (i. e. costs of using them) in the multi-modal scenario, renders computation infeasible without heuristics. They have achieved query times of about one second on big metropolitan datasets (London) in the heuristic version with taxis enabled.

## 1.3 Outline

In chapter 2, we describe the initial approach of this thesis, to adapt the idea of routing with transfer patterns to road networks. There has been much work on adapting methods that work well on road networks to public transportation networks with little success. This chapter follows the idea of going the other way round. Chapter 3 then describes our work in trying to compute multi-modal transfer patterns that could be used for routing like in the transfer patterns approach described earlier. After an initial introduction motivating the general idea behind this, we describe various approaches for the computation and try to understand their problems, hopefully resulting in a deeper insight into characteristics of multi-criteria, multi-modal route planning. We finally show some experiments on real-world datasets in chapter 4 and conclude with chapter 5.

# 2 Routing in Road Networks with Transfer Patterns

## 2.1 Motivation

It has been reported that algorithms that where engineered for road networks, and perform very well there, do not perform well on realistically modelled public transportation networks at all [4]. Up to our knowledge the other way round, running an algorithm that performs well on a public transportation network on a road network, has not been reported yet. However, if this was possible with good performance, we would have an algorithm that works well on both, road and public transportation networks – probably a good foundation on the way to multi-modal route planning. In this chapter, we describe our approach to adapt routing with transfer patterns to road networks.

The concept of transfers is central to routing with transfer patterns, as the name implies. In road networks however, there is no such thing as a transfer. No matter how a route looks like, there is never a need to change vehicles. But consider the actual query algorithm for routing with transfer patterns. It basically relies on asking a fast data structure for costs between predetermined pairs of locations (direct connections). In principle, this isn't restricted to direct train connections. The fast data structure could be anything that can be implemented sufficiently efficient. Also, patterns do not have to correspond to transfers, they could be anything that can compactly describe a path as a sequence of subproblems. For road networks, a pattern could be a sequence identifying independent subgraphs of the whole graph. Asking the fast data structure would then correspond to solving a shortest path problem in such a subgraph.

## 2.2 Implementation

When looking at route directions for cars within Germany at Google Maps between two arbitrary locations, we observed that they are usually constituted by a relatively low number of items (around 20). One item (sometimes a few) correspond to the name of a street that is used. So the number of street names involved in an optimal car-route seems

to be low. It's typically a few local roads to leave the city of the source location, using a few highways to get near the target location, and finally using a few local roads again. We won't make use of this hierarchy here, but just stick with the fact that the number of street names is low. Please note that when considering lower speeds like walking, the number of involved street names is much higher, so we restrict ourselves to cars.

As a first approach, we've implemented a translation of a road network to a public transportation network. We've worked with data from OpenStreetMap (OSM). A OSM dataset contains nodes (geographical locations) and ways (edges between multiple nodes) that yield a graph. Ways (edges) carry various additional tags, among them the name of the street. In the following we refer to a *street* as the largest connected subgraph with identical name tags on all edges. Typically a street is just a linear graph, but it can also contain forks or even loops. The translation we've implemented takes an OSM dataset as input and creates a line-model graph and a direct connection data structure as output.

The actual **translation** from OSM to a public transportation network works in the following way:

1. Create a station for each OSM node.
2. For each street, create a time-independent line. If a street contains forks, split it into multiple lines. Store these lines in a direct connection data structure as described in section 1.2.2 with the only exception that a line is stored in a 1D array with only one entry for departure and arrival time (they are equal). As times, use the relative travel-time of a car on the street starting from the first station. Travel-time is calculated by travelled distance along the street divided by an average speed depending on the road type. The road types are given as tags in the OSM input data.
3. From the lines and stations, build a line-graph (line model) as described in section 1.2.2. Note, that costs of edges between line nodes are time-independent and can be can be stored directly on the edges.

After this translation, we can now run the transfer patterns algorithm on road networks.

## 2.3 Observations

In our translation, we create a station for every OSM node. In public transportation networks, the number of stations is quite low. The number of OSM-nodes of a road network is higher by orders of magnitudes. For example in Texas, the public transportation network has less than 12000 stations while the largest connected component of the

OSM-graph covering the same region already contains over 1 million nodes. Transfer patterns are precomputed by running a Dijkstra search from every station. So for a large number of stations, precomputation is very expensive.

The authors of the original work [5] on transfer patterns have introduced the concept of *hub stations* to reduce precomputation costs. They compute global searches into the whole network only from a preselected set of important hub stations. For all other stations, only a local search is performed that is terminated as soon as all open paths are covered by hub stations. Transfer patterns for non-hub stations only need to be stored up to hub stations. They select hub stations with a sampling-based approach on a time-independent graph. The usage of hub stations reduces precomputation costs and the size of the output far enough for transit networks. However precomputation still remains quite expensive, mainly due to the local searches.

For our naive translation of OSM-nodes to stops, this is a severe issue as stations are large in numbers. In our implementation we pick hubs with the sampling-based approach proposed in the work on transfer patterns and did not apply any speed-up techniques for local searches. This makes local searches too expensive for any realistically sized network. One could likely engineer more efficient means to perform local searches. The computation of access nodes for transit node routing [7, 6, 8] seems quite comparable to the problem of computing local searches up to hub stations in our scenario. There it is essential to do this very efficiently to keep precomputation costs within reasonable bounds. For example when transit nodes are computed using contraction hierarchies, the hierarchy of the network can be exploited as transit nodes are chosen at an higher order in the hierarchy. So a limited search in the upward graph suffices. This could be adapted to our scenario by choosing transit nodes as hubs.

However, storing transfer-patterns for all hubs up to all stations would still require too much space. Recall that for transit node routing, a complete distance table is only computed and stored between transit nodes. Adapting this to our setting would correspond to a 2-layer concept of hub stations, where patterns for hubs are only stored up to other hub stations. So copying concepts from transit node routing seems to provide solutions to cope with the problems that arise from the large number of stations of our street-based translation approach. But in the end the question would be if anything has been gained compared to native transit node routing on road networks. For our simple translation, we do not see any advantage. This could be different for more abstract definitions of stations, transfers and lines.

We did not pursue this however, but instead shifted focus to a different approach described in the next chapter that seemed more promising.

# 3 Computing Multi-Modal Transfer Patterns

## 3.1 Central Idea

At query time, the transfer patterns algorithm [5] relies on querying a fast direct connection structure for connections between precomputed patterns of stations. We've already noted in the previous chapter that these direct connections do not necessarily have to be lines. One could imagine using anything instead that returns the cost of a direct connection fast enough. The authors have reported query times of about 10 µs for the direct connection data structure. So we can safely assume that anything that achieves similar query times could be used in substitution. Similar query times can be achieved on road networks with transit node routing [7, 6, 8]. The authors have reported query times of about 5 µs to 20 µs. So multi-modal routing with transfer patterns seems achievable if the following two assumptions hold:

1. Multi-modal transfer patterns can be computed efficiently enough.
2. The number of multi-modal patterns between two locations is not substantially higher than for a (uni-modal) public transportation network.

In this chapter, we describe our approaches to computing multi-modal transfer patterns. A *multi-modal transfer pattern* $\mathcal{P} = (p_1, p_2, \ldots, p_n)$ is a sequence of stations or geographical locations on a path from $p_1$ to $p_n$ where a change of vehicle or mode occurs (including source and target location). We denote the associated modes with $\mathcal{M}_\mathcal{P} = (m_{1,2}, \ldots, m_{n-1,n})$, where $m_{i,j} \in \Sigma$ is is a mode in $\Sigma := \{\Sigma_t, \Sigma_c, \Sigma_w\}$. As input we consider three graphs covering the same geographical region:

- A *transit graph* $G_t$ corresponding to the line model (see section 1.2.2) of a public transportation network.
- A *car graph* $G_c$ that models a road network. Edge costs correspond the average travel time along an edge using a car.
- A *walk graph* $G_w$ that also models a road network. But here edge costs correspond to the time it takes to walk along an edge.

We are interested in computing optimal, multi-modal transfer patterns of paths that jointly use these graphs in a way meaningful for a human traveler. At this point we do not have a formal definition of for *meaningful* in this context so we just stick with the intuition of it.

In the rest of this chapter we describe our approaches to compute transfer patterns in the chronological order as we have examined them.

## 3.2 Naively Joint Road and Transit Graphs

The simplest and most naive idea for computing multi-modal transfer patterns is probably to just join all input graphs and run a Dijkstra search on the joint graph. This is exactly what we describe in this section.

### 3.2.1 Graph Model

In our experiments we build input graphs from publicly available data. For the transit graph we process GTFS feeds [1] (General Transit Feed Specification) which are available for many cities, mainly in the US. To build the car and walk graphs we use an extract from OpenStreetMap covering the same geographical region. For the walk graph, we assign edge costs that correspond to an average walking speed of $4\,\frac{\text{km}}{\text{h}}$. For the car graph we assign edge costs that corresponds to an average travel speed depending on the road type, ranging from $5\,\frac{\text{km}}{\text{h}}$ to $110\,\frac{\text{km}}{\text{h}}$. For ease of implementation we assume that all roads can be travelled in both directions and do not handle any turn restrictions. So our car and walk graphs share exactly same structure except for their edge costs.

We further assume that a change of mode can only occur at stations and only from the transit graph to exactly one of the road graphs or vice versa. This means that transitions between the walk and car graph are not allowed. Please note that in the real world it can make sense to walk a few meters to a road that is better accessible by car and take a car from there. But as in our model the car and walk graphs share identical structure and edge costs of the car graph are strictly lower, one could never save travel time by allowing such transitions. So given the simplicity of our model for the road graphs we argue that this is a reasonable assumption to make.

In order to model this restriction we use a slight variation of the line model introduced in section 1.2.2 for the transit graph. Namely, instead of having just a single station node and line node at each station, we add a *station arrival node* and a *station departure node* replacing a single station node and a *line arrival node* and a *line departure node* replacing a single line node. For every pair of those nodes, we add a zero-cost edge

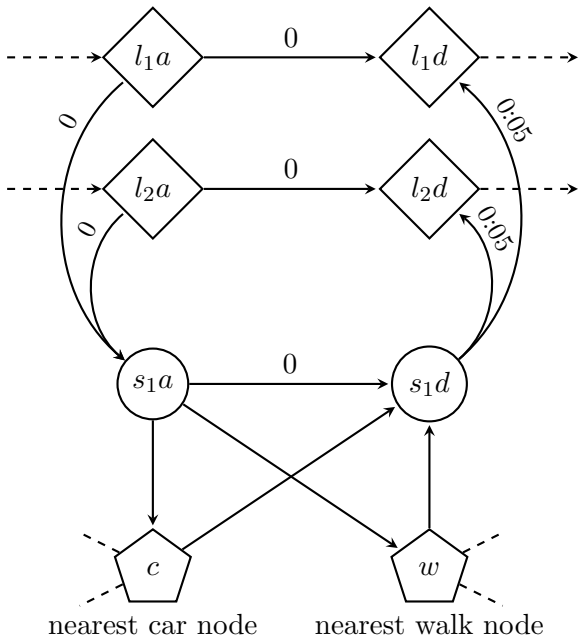


**Figure 3.1:** Joint graph at a station with two lines. Diamond shaped node are line nodes, circular nodes are station nodes and pentagon nodes are the nearest nodes of the car and walk graphs. Edges from the station departure node to the line departure node are labelled with the travel time component of the edge costs.

from the arrival to the departure node. We also add an edge from each line arrival node to the corresponding station arrival node and from the station departure node to all corresponding line departure nodes. Note that with this structure there are no paths from the station departure to the station arrival nodes. We now build a joint graph $G = (V, E)$ by just merging all graphs $V = V_t \cup V_c \cup V_w$, $E = E_t \cup E_c \cup E_w$. For every station, we add an edge from the nearest car node $v_c$ and walk node $v_w$ to the station departure node and an edge from the station arrival node to these nodes. See figure 3.1 for an illustration. Note that transitions between the walking and car graphs via the transit graph are not possible without using public transportation in-between.

### 3.2.2 Computing Profile Queries

Assume that we want to compute transfer patterns between a given set of locations, i. e. nodes of the joint graph. Recall that in the original transfer patterns paper [5] this is achieved by running a multi-label Dijkstra profile query from all transfer nodes

of a given station. We do not use the time-expanded model, so we have to do that in a slightly different way that we'll describe here.

Recall that an order relation on labels defines a set of dominant labels used during the Dijkstra search. More formally, let $l \in \mathcal{L}$ be a set of labels and $<_\mathcal{L}$ be an order relation. Then the set of dominant labels of $\mathcal{L}$ is defined by $\{l \in \mathcal{L} \mid \neg\exists l_o \in \mathcal{L} : l_o <_\mathcal{L} l\}$. We define a label as a tuple $l = (t_d, c)$, consisting of departure time $t_d \in \Gamma$ and cost $c \in \mathcal{C}$. We then define an order $<_\mathcal{L}$ on the set of labels based on a given order relation on the costs as:

$$(t_1, c_1) <_\mathcal{L} (t_2, c_2) \Leftrightarrow (t_1 > t_2 \wedge c_1 \leq_\mathcal{C} c_2) \vee (t_1 \geq t_2 \wedge c_1 <_\mathcal{C} c_2)$$

This means a label dominates another label if it has a later departure time and no higher costs, or if it has lower costs and does not depart earlier. Please note that for the time-expanded model, it suffices to use $<_\mathcal{C}$ directly as order relation for the labels. This is possible because the arrival time is encoded in the nodes and the structure of the graph ensures that a label with earlier departure time is never compared to one with earlier arrival time. The relation $<_\mathcal{C}$ of course also suffices if all initial labels share the same departure time (which is not true for our profile queries).

The departure time $t_a$ is either a number encoding the time or a special value indicating that the departure time is arbitrary $\Gamma := \mathbb{N}_0 \cup \{\Gamma_{\text{any}}\}$. The order relation on the departure times is given by their natural order with the exception $\Gamma_{\text{any}}$ is treated like infinity. This means that a free to choose departure time is better than any fixed departure time regarding $<_\mathcal{L}$.

All of our actual cost models contain a travel-time component. We denote the *travel time component* of cost $c$ as $c_t$. The *arrival time* of a label $l = (t, c)$ (at the respective node) is given by $t + c_t$ if $t \neq \Gamma_{\text{any}}$. During a Dijkstra search, the arrival time is used to determine the first feasible trip of a line. The travel time of this trip plus the waiting time until the trip departs, constitutes the *travel time* component $\omega(e, l)_t$ of the edge cost $\omega(e, l)$ of an edge $e$ between a line departure and line arrival node. Labels are propagated along edges by adding the edge cost to the cost component of the label, whereas the departure time is copied from the preceding label. For the special case when a label with departure time $\Gamma_{\text{any}}$ hits a line departure node, a successor label is generated for each trip of that line. The departure time of these labels is fixed to the departure time of the trip minus the travel time component of the label cost. The travel time component of edges from station departure to line departure nodes is set to the transfer buffer, as illustrated in figure 3.1.

With these definitions we can now run profile queries on our joint graph, suitable for computing transfer patterns. For example, running a profile query from all transfer nodes in the time-expanded model is equivalent to running a profile query with an

initial label $l = (\Gamma_{\text{any}}, 0)$ at the station departure node. Allowing initial walking or car usage corresponds to putting the same initial label at the station arrival node.

### 3.2.3 Contracting Road Graphs

One of the first things one will notice when running a profile query on the joint graph, is that a lot of resources are consumed to propagate labels within the road networks. Please recall that we generate labels for every departure time of a line. For a non-trivial cost model many of these labels will eventually end up at a station arrival node and then be propagated through the road graphs. Now consider how a multi-modal path between to arbitrary locations $A$ and $B$ (station, car, or walk nodes) actually looks like. It can be split into three potentially empty subpaths:

1. A subpath from $A$ to some station node that solely uses either the car or walk graph.
2. A subpath that uses the transit graph and the road graphs, but the latter only between pairs of station nodes.
3. A subpath from some station node to $B$.

As we are not interested in computing actual paths but just transfer patterns, the path on the road network between two station nodes is irrelevant. So for exploring 2), there is no need to know the complete road input graphs. Instead, we can substitute the road graphs by any smaller graph as long as distances between stations are preserved. To compute a profile query we compute the distance from $A$ to all station nodes on the original road network and use these to initialize our actual profile search. After completion, for any target location $B$ we analogously collect labels at all station arrival nodes and add the costs to go to $B$ on the road networks. This gives us the same set of dominant labels as we would get without substituting the road graphs.

We use node contraction as known from Contraction Hierarchies [15] to compute a *core graph* that preserves distances between all station nodes. As priority term we use the sum of Edge Difference plus the number of already contracted neighbours. We fix the priority to infinity for all nodes that will be connected with station nodes. This artificially holds those nodes up in the hierarchy. To avoid ending up with a too dense graph, we stop node contraction as soon as the edge difference raises above a threshold (10 in our experiments). Recall that node contraction preserves path costs between all not yet contracted nodes. So after completion we can just remove all contracted nodes from our graph, yielding a much smaller core graph. For New York for example, this reduces the number of nodes in the car (or walk) graph roughly by a factor of 50 from about 2 million to about 36000 nodes in our experiments. We provide more detailed

numbers in chapter 4.

### 3.2.4 Observations and Problems

We have described a way to compute multi-modal transfer patterns but we did not define yet how the actual costs look like. In fact, choosing an appropriate cost model turned out to be a non trivial task – something we did not expect in the beginning. In the following we give some examples for cost models one might intuitively choose and describe why they do not work.

**Pareto: Time and Number of Transfers** One of the most obvious cost models is to just use what works well in a pure public transportation network. Bast et. al. [5] use a tuple $(t, p)$ where $t$ is the travel time and $p$ a penalty that basically corresponds to the number of transfers. The order relation is defined in the Pareto sense:

$$(t_1, p_1) <_{\mathcal{C}} (t_2, p_2) \Leftrightarrow (t_1 < t_2 \wedge p_1 \leq t_2) \vee (t_1 \leq t_2 \wedge p_1 < p_2)$$

This cost model makes a lot of sense for public transportation networks. So let's assume we use it in our multi-modal setting and analogously count one transfer not only for transfers within the public transportation network but also for boarding a car. It turns out that with this cost model we will get virtually no routes that use the public transportation network at all. It is easy to see why this happens: there always exists a zero-transfer route that just uses a car from the source to the target location. The only way a transit route can survive, is by being faster than the car-only route. Within cities this is hardly ever the case. Just consider a city that provides public transportation services only by buses. Here taking a car will alway be faster in our model. So using this cost model clearly does not give desirable results.

**Fixed Penalty for Boarding a Car** Now let's assume we use the same cost model as described above, but use a fixed, larger penalty for boarding a car. Larger than the one used for a transfer (i. e. boarding a line). Intuitively this corresponds to paying a fixed amount for using a taxi. With such a cost model, once a taxi is used, there is hardly any advantage in ever using public transportation again. A direct car-only route is nearly always cheaper than any combination of car and public transport. Intuitively the taxi is already completely paid once it is used, so there is no point in not using it for the whole route (again except for situations where public transport is faster of course, which is rarely the case within cities with our model). So this model also does not give useful results.

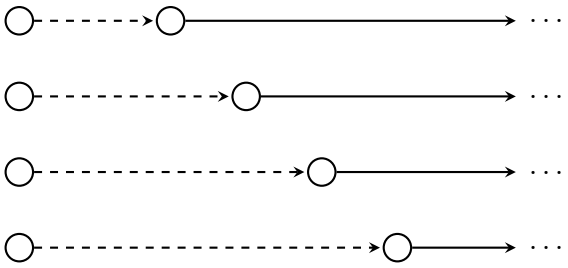

**Figure 3.2:** Illustation of follow-a-bus variations. Dashed lines are travelled by car, lined ones using public transportation. Circles denote boarding a vehicle.

**Penalizing Car Travel Time** An other quite obvious idea for a cost model is to penalize the time travelled with a car. This can be achieved by using Pareto costs on a triple $(t, p, t_c)$ analogously, where $t_c$ stands for the time spend in the car graph plus an additional fixed time for boarding a taxi. Intuitively this can be seen as paying for taxi based on the time using it.[1] This cost model sounds quite reasonable at first glance, but produces mostly unreasonable variations of paths. The most obvious unreasonable kind of variations are what we call *follow-a-bus* routes: a set of routes where a car is used up to a bus stop and the rest of the route is travelled using public transport. Variations of such routes exist for many bus stops on the route. See figure 3.2 for an illustration.

To understand why these variations appear, recall that in the Pareto sense, a cost does not dominate another cost if it is not less or equal in all components. In our case this especially means that the travel time of a dominating cost cannot be greater, not even the slightest bit. Now consider a route where the first part is travelled by car up to some bus stop and the route continues with using a bus. Numerous Pareto-optimal variations exist of such routes. Instead of using a car only up to the closest bus stop of a line, there is a variation that uses a car up to the second closest bus stop of the same line and is identical for the rest of the route. This variation is better in terms of travel time but worse in terms of car usage. There exists also a variation that uses a car up to the third closest bus stop and so on. None of these variations dominates the others. The variations also appear analogously for the last part of a route.

So with this cost model, we get a great number of optimal routes. It would not make any sense to show all these variations to a user. Apart form that, the number of labels generated for a profile query with this cost model just makes it intractable for any dataset of somewhat realistic size. One approach to cope the number of variations would be to loosen the definition of dominance. For example take two routes: one that uses a taxi for

[1] Adding the time spend on the road network to $p$ instead of adding a third cost component does not change the described effects but has the additional disadvantage of implying a comparability of car-time and number of transfers.

10 minutes up to a train station and then a train for 2 hours directly to the destination. And another one, that uses taxi for 9 minutes and a train for 5 hours directly to the destination. From a human perspective, the first route is clearly preferable. One could think of several way to address such cases. One approach would be to discretise car time, for example in 5 minute steps. In this example it would lead to the desired result. It would also make some of the follow-a-bus variations disappear. But one can easily construct examples where variations of identical structure still appear. In our example just consider 30 and 24 minutes of car time instead. Apart from this, applying discretization already during the Dijkstra search easily leads to non-optimal solutions. However, applying it on the results in a post-processing step does not help to decrease computational costs. Despite from discretization, one could think of other concepts for loosening the definition of dominance. For example one could somehow relate car time with total travel time in a non-linear way. But again, one will easily loose optimality when applying it already during search. Despite intensive thought, we could not come up with any cost model (a definition of $\mathcal{C}$ and $<_{\mathcal{C}}$) that makes unwanted variations disappear and nevertheless allows to give an optimality guarantee (or at least does not also make some obviously wanted variations disappear too). A refined approach for loosening the definition of dominance is described by Delling et. al. [9]. However again, results are heuristic when applied during search.

In the rest of this thesis, we follow a different idea. As we failed to come up with a convincing definition of $<_{\mathcal{C}}$, we instead experiment with ways to disallow undesirable variations explicitly in the graph model already. The intuition for this is that modelling gives more control to influence generated routes by reasoning about their structure, which is hard to achieve by looking at costs only. To come up with a reasonable model, a deeper understanding of the reason for variations is required. Especially for those which make computation intractable on larger datasets. We have understood follow-a-bus variations so far, so this is a starting point.

## 3.3 Reduced Boarding Variations

In previous section, we have described follow-a-bus variations of paths. In this section, we describe a graph model that disallows such variations. The idea is quite simple. Assume we are at any arbitrary location and want to use a car to access the public transportation network. In the graph model described in the previous section, it is possible to drive to any station departure node and take any line that departs from that station. Which means every line can be boarded at every served station. Being able to board a line at all possible stations is exactly what allows follow-a-bus variations. So we make the following assumption:

**Assumption 1.** *Given a location $A$ and a line $l$ that serves stations $\mathcal{S}_l$, taking a car from $A$ to $s \in \mathcal{S}_l$ in order to board $l$ is only reasonable for a small number of stations $s$.*

Intuitively this means, if you want use a car to reach a specific line, there is just one or two station you would drive to. Usually it is is just one station. If you are somewhere in-between two stations and the line goes in two directions, both stations might be reasonable.[2]

### 3.3.1 Graph Model

We assume without loss of generality, that the road network is only used between stations. Please refer to section 3.2.3 for an explanation why this is sufficient. For the public transportation part, we use the same graph as in the previous section. In order to add the road networks we assume to be given an oracle $O_c$, that given a location $p$ (for example a station) and a line $l$, returns a set of reasonable stations $O_c(p, l) \in \mathcal{P}(\mathcal{S})$ for taking $l$ using a car to get to the station. For walking, follow-a-bus variations do not appear as walking is usually slower than a bus. Nevertheless, for the sake of symmetrical beauty we assume to be given an oracle $O_w$ for walking analogously. For each line $l$ and station $s$, we add an edge from the station arrival node of $s$ to all line departure nodes of $l$ at stations given by $O_c(s, l)$. Edge costs are set to the shortest path costs in the car graph between the two involved stations. For the walk graph we add the same edges analogously. Please note that the number of edges added for each road input graph is $|\mathcal{L}| \cdot |\mathcal{S}| \cdot n$ where $n$ is the average number of stations returned by the respective oracle. So even for small $n$ this are quite some edges. See figure 3.3 for illustration of the graph structure. We call this graph model *stops-to-lines model*.

### 3.3.2 Observations and Problems

We do not have a clear understanding of how to implement a reasonable oracle at this point. For our experiments we used a very simple implementation that for $O(s, l)$, just returns the nearest station served by the line. Where *nearest* means with respect to the shortest path costs on the road graph. If $l$ happens to serve $s$, we just return the empty set. In the following we assume that this oracle is used if not otherwise stated. We are aware that we will miss some reasonable routes with this simple oracle. So what we observe is not necessarily generalizable. But it is likely that unwanted variations that we observe with this simple oracle will also appear using a more sophisticated oracle. So at least for finding more reasons for unwanted variations this simple oracle is sufficient.

[2] One could argue that if you are very late, you might indeed want to follow a bus to catch one specific trip. But this is a very specific, probably rare use case. If it's that urgent and it's faster by car, you would probably just take the car directly to your target location instead.

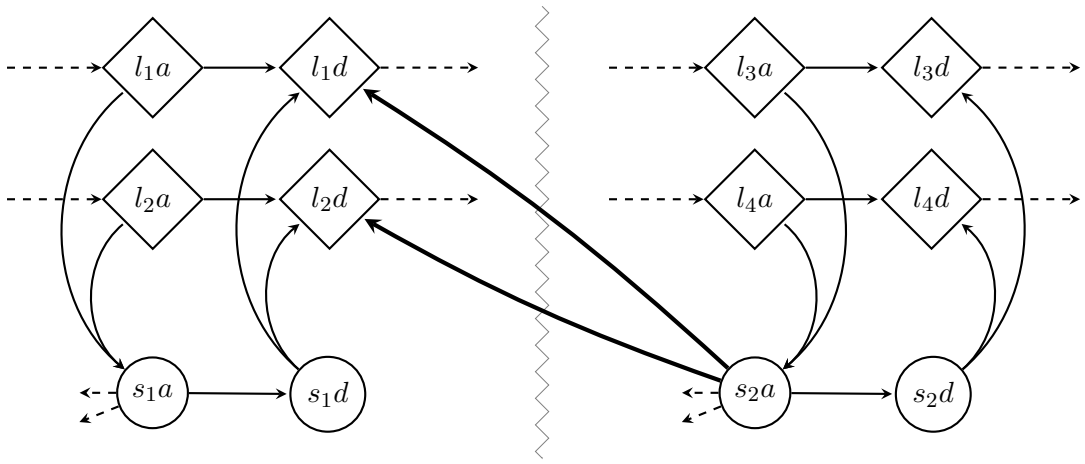


**Figure 3.3:** Stops-to-lines graph model. Thick lines indicate the type of the edges added for each road network as given by the oracle. In this example station $s_1$ is a reasonable station to board $l_1$ and $l_2$ from $s_2$ via the road network.

Using the stops-to-lines model, one still gets a lot of unwanted variations. The most obvious kind we recognized are of the following kind: On a taxi trip, use a bus for a few stations to save some car time. Please recall that we do increase the car time $t_c$ every time a taxi is boarded by a fixed amount (typically 5 minutes in our experiments). So using two taxis in a row and a bus in-between does only pay off if it saves at least this fixed amount of car time. Nevertheless such variations do appear. At the beginning or end of a car route, variations do not even need to save a minimum amount of car time to be non-dominated. A typical example is to use a bus for one station and use a taxi for the rest of the route. We call such variations *insane-skinflint* routes. See figure 3.4 for an illustration.

Please note that these variations are much less frequent than the follow-a-bus variations described in the previous section, which disappear with this model. Still they are the most obvious unreasonable variations we noticed. So in the next section we will describe a way to avoid them by imposing limits to the Dijkstra search.

## 3.4 Limited Mode Changes

In this section, we introduce a way to avoid insane-skinflint variations. The idea behind this compiles down to one simple assumption:

**Assumption 2.** *A path with transfer pattern* $\mathcal{P} = (\dots, p_i, p_j, p_k, \dots)$ *and modes* $m_{i,j} = \Sigma_c$ *and* $m_{j,k} = \Sigma_t$ *is unreasonable, if the shortest path cost* $c_{i,k}$ *of the car-only path*

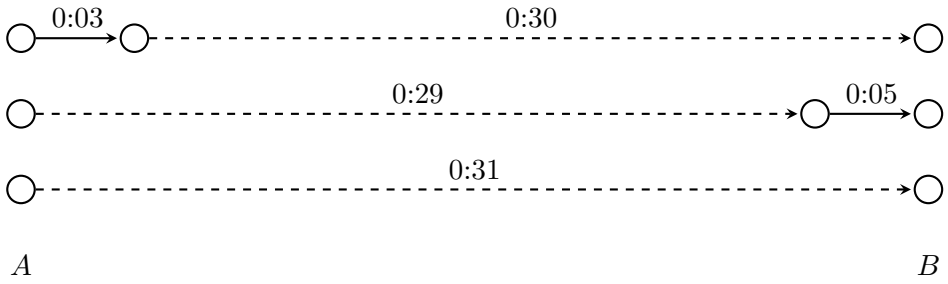


**Figure 3.4:** Illustation of two insane-skinflint variations. Dashed lines are travelled by car, lined ones using public transportation. Circles denote transfers or the source and target locations.

*from $p_i$ to $p_k$ is bound by $c_{i,j} + C_{save}$ for a reasonably chosen constant $C_{save}$. It is also unreasonable for the reverse case with modes $m_{i,j} = \Sigma_t, m_{j,k} = \Sigma_c$.*

For example in figure 3.4 the car time of the direct car connection is only little greater than the car times of the indirect connections between $A$ and $B$. So for a *car save limit* $C_{\text{save}} = 0{:}02$ both variations would be considered as unreasonable. Please note that this also applies to any subpaths. So in the example, $A$ and $B$ do not necessarily have to be the source and target location but could correspond to a pair of transfers within a more complex path.

### 3.4.1 Implementation

We modified our Dijkstra implementation to prune all paths early, as soon as it becomes clear that they are unreasonable with respect to assumption 2. This is done in the following way:

We compute a full distance table between all pairs of stations on the car graph. For this purpose, we run a regular scalar-valued Dijkstra search on the core graph. In the following we refer to the values in this table by $c_{sp}(s_1, s_2)$. During search we additionally maintain for each label:

- The station $s_{bl}$ where a line was boarded last and the car time $c_{bl} := t_c$ immediately before boarding.
- The station $s_{bc}$ where a car was boarded last and the car time $c_{bc} := t_c$ immediately before boarding.

These values are propagated during the search, i. e. copied from the predecessor if not updated. We prune search whenever:

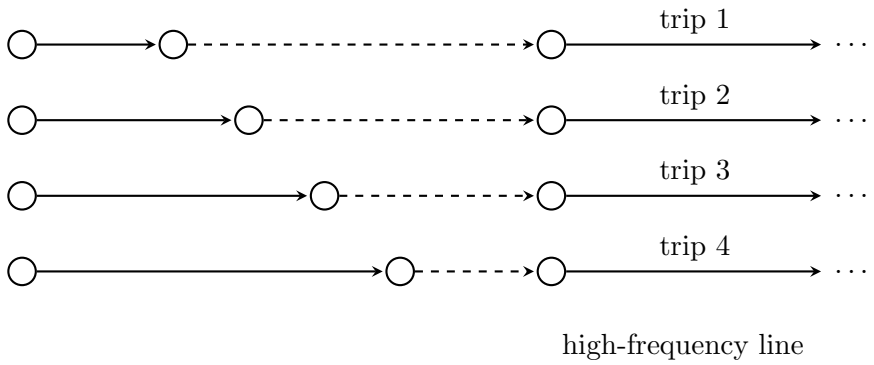


**Figure 3.5:** Illustation of high-frequency line-to-line variations. Dashed lines are travelled by car, lined ones using public transportation. Circles denote boarding a vehicle. Depending on where one gets off the first line, one can catch an earlier trip of the second line.

- Getting of a car at $s$, public transport was used before, and $c_{sp}(s_{bl}, s) < t_c - c_{bl} + C_{\text{save}}$ holds, i. e. using the car from $s_{bl}$ instead is preferable.
- Boarding a car at $s$, public transport was used before, and $c_{sp}(s_{bl}, s) < C_{\text{save}}$ holds. This implies the previous pruning rule, but applies earlier.
- Getting of a line at $s$, a car was used before and $c_{sp}(s_{bc}, s) < t_c - c_{bc} + C_{\text{save}}$ holds, i. e. using the car up to $s$ instead is preferable.

### 3.4.2 Observations and Problems

We have noticed one major source for remaining unreasonable variations when running experiments with the stops-to-lines model and a car save limit. We didn't notice it on small datasets but it became apparent on datasets of large cities. We call them *high-frequency line-to-line* variations, because they appear on transfers from one line to another line that is served with a high-frequency schedule. Imagine a bus line with final destination at a metro station, where a trip of a high-frequency line departs every two minutes. The most reasonable way to transfer from the bus line to the metro would be to just wait till the bus arrives at the metro station. But in our stops-to-lines model, it is also possible to leave at some earlier stop and take a taxi from there to the metro station. This way it is possible to catch an earlier trip of the metro line. Which trip can be caught, depends on the station where the transfer to the taxi is made. So it is not uncommon to get a variation for every single stop of the bus on its way to the metro station. See figure 3.5 for an illustration.

The variations are structurally comparable to follow-a-bus variations, they just appear in a different setting. If they appear, the number of these variations tends to grow large. But they appear much less often and only for some queries of large cities that have high-frequency lines. Please refer to chapter 4 for experimental details.

## 3.5 Explicitly Modelled Line to Line Transfers

To eliminate high-frequency line-to-line variations, we follow an idea very similar to the one we used to eliminate follow-a-bus variations. We model connections between lines over the road network explicitly. The initial boarding of a line and the final getting off is treated as in the stops-to-lines model. Our model is based on the following assumption:
**Assumption 3.** *Consider all reasonable paths that use two lines $l_1, l_2$ and a car in-between. Let $\mathcal{T}_{l_1,l_2}$ be a set with elements $(s_1, s_2) \in \mathcal{T}_{l_1,l_2}$ identifying the stations where a transfer between the lines occurs, i. e. $l_1$ is used up to station $s_1$ then a car is used to drive to station $s_2$ in order to board line $l_2$ there. The number of elements $|\mathcal{T}_{l_1,l_2}|$ is small on average.*

For the public transportation part, we use the same graph as in the previous sections. To add the road networks, we assume to be given an oracle $O_{lc}$, that given two lines $l_1, l_2$ returns a set of reasonable transfers station tuples $(s_1, s_2) \in O_{lc}(l_1, l_2)$ identifying a reasonable transfer from $l_1$ to $l_2$ with car usage in-between. Again, we assume to be given an oracle $O_{lw}$ for walking analogously. For each pair of lines $l_1, l_2$ and stations $(s_1, s_2) \in O_{lc}$, we add an edge from the line arrival node of $l_1$ at $s_1$ to the line departure node of $l_2$ at $s_2$. Edge's costs are set to the shortest path cost from $s_1$ to $s_2$ on the car graph. For walking, we proceed analogously. An illustration of the resulting graph is given in figure 3.6. We call this model *lines-to-lines model*.

### 3.5.1 Observations and Problems

In our experiments we again use a very simple implementation of an oracle that just returns a single tuple $O_{lc}(l_1, l_2) = \{(s_1, s_2)\}$ with minimal shortest path costs on the road graph from $l_1$ to $l_2$. Please note that potentially desirable routes will be missed with this simple oracle. But it is good enough for the purpose of finding out if other computationally intensive variations exist despite the ones we have spotted so far.

Running Dijkstra profile queries on the lines-to-lines model turned out that variations that made precomputation prohibitively expensive seem to have disappeared. We did not recognize any remaining obviously unreasonable kind of variations. What remains are variations that we call *obvious variations*:

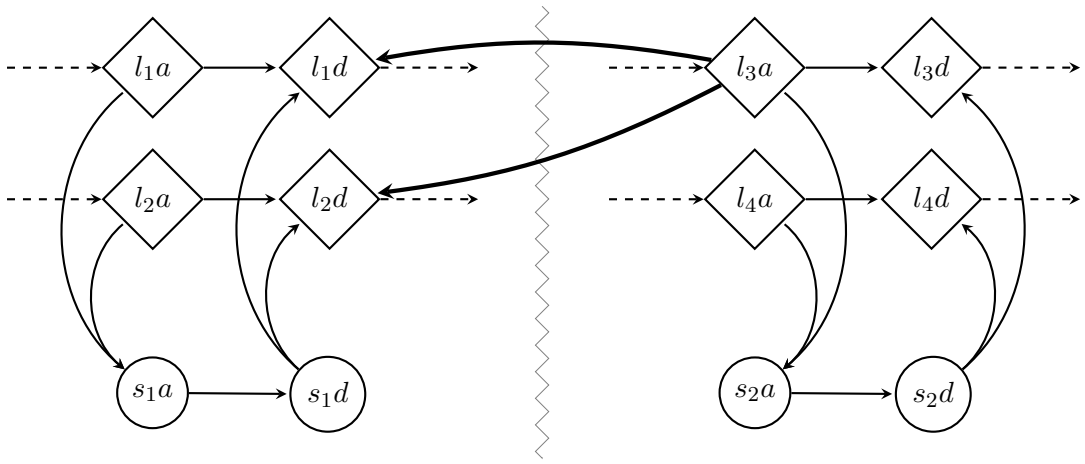


**Figure 3.6:** Lines-to-lines graph model. Thick lines indicate edges added for each road network as given by the oracle. In this example edges from $l_3$ to $l_1$ and $l_2$ for the transfer tuple $(s_2, s_1)$ are shown.

- **Walking to save a transfer.** For a route that uses at least two vehicles, walking instead of taking a line can save one transfer. This trades a transfer for more travel time, which is incomparable in our cost model. For the first and last part of a route, it is always possible to walk instead of taking a vehicle (with adapted departure and arrival times).
- **Car instead of walking.** It sometimes is an option to either walk to the first station or take a taxi to the first station. Same holds from the last station to the target location. Combinations which walk the first part but use a car for the final part, or the other way round, also occur.

These variations do not occur in such large numbers as the ones we've eliminated. So they could also be filtered in a post-processing step. But this is not the focus of this thesis.

# 4 Experiments

For our experiments we used publicly available GTFS [1] feeds for Léon (Spain), Austin, Dallas, Toronto and New York. For the road networks, we've extracted OSM [2] data, covering a region that contains all stations of a GTFS feed. The datasets are of increasing size in the stated order, whereas Léon is very small and New York is the largest publicly available GTFS feed. We only process data of a single Monday from the GTFS feeds. Details about the datasets are given in table 4.1.

Table 4.2 shows details about the size of the resulting graph models. Note that for the stops-to-lines and lines-to-lines models there are also walk and car nodes reported. That is because in our implementation we add a walk and a car node next to each line departure node to distinguish edge types. So this detail is slightly different from the description of the graph models in chapter 3, where we assumed that no additional information is necessary to determine the type of an edge, especially its Pareto cost tuple.

We ran profile queries from random samples of source locations. In our experiments the source location always corresponds to a station. This was done solely to ease implementation and is not a general limitation. It especially does not impose any restriction on the first vehicle used. It is still possible to walk or go to any other station before departing with public transport. During Dijkstra search, we discard labels with travel-times worse by factor $C_{\text{spread}}$ (10 in our experiments) than the best known travel-time at the respective node. This eliminates some edge-case routes and also naturally limits walking. For example one will never get something like *walk from Freiburg to Frankfurt Airport for 2 days* as there are means to get to Frankfurt at least 10 times faster. We assume a transfer buffer (minimal transfer duration) of 5 minutes for all transfers. We also add the transfer buffer to the travel time for the initial boarding of a vehicle. So for example boarding a taxi and using it for one minute has a total travel-time of 6 minutes.

All source code of our C++ implementation, including evaluation scripts used to generate the result tables, is available on Bitbucket[1]. It may be considered as released into the public domain.

[1] `https://bitbucket.org/lohre/transitrouter`

| | Leon | Austin | Dallas | Toronto | New York |
|---|---|---|---|---|---|
| **Transit** | | | | | |
| Stations | 247 | 2.72 K | 11.6 K | 10.9 K | 16.9 K |
| Lines | 85 | 235 | 563 | 1.12 K | 1.98 K |
| Trips | 87 | 6062 | 10.8 K | 40.7 K | 62.8 K |
| Departures | 1.95 K | 161 K | 473 K | 1.47 M | 2.24 M |
| **Road** | | | | | |
| Nodes | 234 K | 339 K | 1.37 M | 399 K | 1.95 M |
| Edges | 501 K | 723 K | 2.98 M | 880 K | 4.27 M |
| **Car core** | | | | | |
| Nodes | 369 | 3.56 K | 23.6 K | 11.6 K | 36.7 K |
| Edges | 3.57 K | 23.7 K | 153 K | 61.9 K | 220 K |
| Nodes (%) | 0.16 | 1.04 | 1.71 | 2.91 | 1.88 |
| **Walk core** | | | | | |
| Nodes | 429 | 5.06 K | 23.1 K | 12.5 K | 37.6 K |
| Edges | 4.74 K | 32.6 K | 179 K | 75.4 K | 270 K |
| Nodes (%) | 0.18 | 1.49 | 1.67 | 3.14 | 1.92 |

**Table 4.1:** Datasets used in our experiments. We parse transit data for a single Monday, so numbers of trips and departures are for one day only. The road graphs are the largest connected component of the OSM data. The number of nodes in the core graphs is significantly lower, speeding up further processing.

## 4.1 Computational Costs

Computational costs mainly depend on the size of the graph and the number of labels generated. Recall that a multi-label Dijkstra search maintains a set of dominant labels for each node. Whenever a new label is generated, it is inserted into this set eliminating all dominated labels, potentially the newly generated label itself. We use a naive implementation to maintain label sets that requires checking all existing labels of a node for dominance whenever a new label is inserted. So inserting $n$ pairwise non-dominating labels sequentially into an initially empty label set requires at least $\frac{1}{2}n(n-1) \in \Omega(n^2)$ comparisons. There might exist more efficient methods for maintaining labels sets, but implementing these was not the focus of this thesis. Just keep in mind the computation time of a query grows at least quadratic with the number of labels per node in our implementation (for a fixed graph structure).

We ran random profile queries for all graph models (naive, stops-to-lines and lines-to-lines) on all feasible datasets. We report the average number of labels separately for all node types that remain after a query has completed. We also report the average CPU time per query. A machine with two Intel Xenon E5649 CPUs with 12 cores in total and 96GB of RAM was used for our experiments. Our C++ implementation was compiled

| | Naive | s2l | l2l |
|---|---|---|---|
| #car nodes | 369 | 1.79 K | 2.13 K |
| #walk nodes | 429 | 1.82 K | 2.15 K |
| #station nodes | 494 | 494 | 494 |
| #line nodes | 3.98 K | 3.98 K | 3.98 K |
| #transit nodes | 4.48 K | 4.48 K | 4.48 K |
| #nodes | 5.27 K | 8.09 K | 8.75 K |

**(a)** Leon

| | Naive | s2l | l2l |
|---|---|---|---|
| #car nodes | 3.56 K | 6.55 K | 9.42 K |
| #walk nodes | 5.06 K | 6.72 K | 9.77 K |
| #station nodes | 5.44 K | 5.44 K | 5.44 K |
| #line nodes | 14.7 K | 14.7 K | 14.7 K |
| #transit nodes | 20.2 K | 20.2 K | 20.2 K |
| #nodes | 28.8 K | 33.5 K | 39.4 K |

**(b)** Austin

| | s2l | l2l |
|---|---|---|
| #car nodes | 22.1 K | 27.8 K |
| #walk nodes | 22.5 K | 28.9 K |
| #station nodes | 23.3 K | 23.3 K |
| #line nodes | 50.2 K | 50.2 K |
| #transit nodes | 73.5 K | 73.5 K |
| #nodes | 118 K | 130 K |

**(c)** Dallas

| | s2l | l2l |
|---|---|---|
| #car nodes | 37 K | 52.1 K |
| #walk nodes | 37.6 K | 53.7 K |
| #station nodes | 21.8 K | 21.8 K |
| #line nodes | 82.5 K | 82.5 K |
| #transit nodes | 104 K | 104 K |
| #nodes | 179 K | 210 K |

**(d)** Toronto

| | s2l | l2l |
|---|---|---|
| #car nodes | 57 K | 86.8 K |
| #walk nodes | 58.6 K | 92.2 K |
| #station nodes | 33.7 K | 33.7 K |
| #line nodes | 123 K | 123 K |
| #transit nodes | 156 K | 156 K |
| #nodes | 272 K | 335 K |

**(e)** New York

**Table 4.2:** Sizes of the naive, stops-to-lines (s2l) and lines-to-lines (l2l) graph models. Transit nodes refers to station or line nodes.

with gcc version 4.6.3. The results are reported in tables 4.3 to 4.7. The shortcuts s2l (stops-to-lines), l2l (lines-to-lines) indicate the model, the digit the car save limit in minutes. Values are averaged over multiple profile queries from random source stations. For comparison, we also report the number of labels without car usage. We therefore just count the number of labels with $t_c = 0$ of the same query, so no separate CPU time is reported. Please note that number of labels is not a good measure for the number of resulting transfer patterns. A single high-frequency connection easily produces several hundred labels that compress to a single transfer pattern, while a connection that is only valid for one specific time also produces a transfer pattern but has little impact on computational costs. So we report the number of transfer patterns separately in section 4.2.

The computational effort for the naive model quickly grows beyond any reasonable limit. For Austin about 1700 labels are generated per node and a single query requires more than 5 hours on average. We could not even compute a single query on our larger datasets within reasonable time limits. With the stops-to-lines model, computational effort decreases significantly. For Léon and Austin, the number of generated labels drops by a factor of 5 compared to the naive model. Knowing about the existence of follow-a-bus variations, this reduction is not too surprising.

A car save limit of 10 minutes reduces the number of generated labels by about one order of magnitude for the stops-to-lines model. For lines-to-lines model, it reduces them by a factor of about 5.

Comparing the number of generated labels between the stops-to-lines and the lines-to-lines model for a fixed car save limit shows, that the total number of labels for the stops-to-lines is larger by a factor of about 2 to 3. However the difference in terms of CPU-time is about 1 to 2 orders of magnitude. Neither can be explained completely by avoided high-frequency line-to-line variations. Looking at the distribution of labels among the different types of nodes, one can see that the difference in the number of labels at transit nodes is quite insignificant. The difference in the total number of generated labels mostly goes back to labels at car and walk nodes. So it turns out that inserting car and walk nodes to distinguish edge types is not a very good idea. One could probably save a lot by not inserting these nodes. This also holds for the lines-to-lines model, but here the saving would be much less as the number of labels at walk and car nodes is already much lower. Another part of the saving in CPU time of the lines-to-lines model compared to the stops-to-lines model can be accounted to the reduced degree of the graph. Recall that there is an edge from every station to every line for the former, whereas for the latter there is only an edge from every line to every line. The number of stations is typically larger by an order of magnitude (refer to table 4.1 for actual numbers).

| | Naive | s2l-0 | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 |
|---|---|---|---|---|---|---|---|
| Samples | 83 | 84 | 79 | 82 | 82 | 80 | 83 |
| CPU Time | 0:00:07 | < 1 s | < 1 s | < 1 s | < 1 s | < 1 s | < 1 s |
| #labels at car nodes | 124 K | 36.1 K | 3.84 K | 426 | 966 | 102 | 77 |
| #labels at walk nodes | 80.5 K | 26.7 K | 9.25 K | 8.56 K | 963 | 373 | 390 |
| #labels at station nodes | 109 K | 2.97 K | 1.03 K | 968 | 2.31 K | 943 | 878 |
| #labels at line nodes | 27.8 K | 5.09 K | 3.5 K | 3.31 K | 3.56 K | 2.86 K | 2.77 K |
| #labels at transit nodes | 137 K | 8.06 K | 4.53 K | 4.28 K | 5.87 K | 3.8 K | 3.65 K |
| #labels total | 342 K | 70.8 K | 17.6 K | 13.3 K | 7.8 K | 4.27 K | 4.12 K |
| #labels/car node | 337 | 20.2 | 2.14 | 0.238 | 0.454 | 0.048 | 0.0362 |
| #labels/walk node | 188 | 14.6 | 5.08 | 4.7 | 0.449 | 0.174 | 0.182 |
| #labels/station node | 222 | 6.01 | 2.09 | 1.96 | 4.69 | 1.91 | 1.78 |
| #labels/line node | 6.98 | 1.28 | 0.878 | 0.832 | 0.894 | 0.717 | 0.696 |
| #labels/transit node | 30.7 | 1.8 | 1.01 | 0.957 | 1.31 | 0.849 | 0.816 |
| #labels/node | 64.9 | 8.76 | 2.18 | 1.64 | 0.892 | 0.489 | 0.471 |
| **w/o car usage** | | | | | | | |
| #labels at walk nodes | | 8.95 K | 7.95 K | 8.48 K | 360 | 367 | 390 |
| #labels at station nodes | | 1 K | 882 | 964 | 792 | 810 | 874 |
| #labels at line nodes | | 1.96 K | 1.83 K | 1.93 K | 1.29 K | 1.31 K | 1.4 K |
| #labels at transit nodes | | 2.97 K | 2.71 K | 2.9 K | 2.08 K | 2.12 K | 2.28 K |
| #labels total | | 11.9 K | 10.7 K | 11.4 K | 2.44 K | 2.49 K | 2.67 K |
| #labels/walk node | | 4.91 | 4.36 | 4.66 | 0.168 | 0.171 | 0.182 |
| #labels/station node | | 2.03 | 1.79 | 1.95 | 1.6 | 1.64 | 1.77 |
| #labels/line node | | 0.493 | 0.459 | 0.486 | 0.323 | 0.329 | 0.352 |
| #labels/transit node | | 0.663 | 0.605 | 0.648 | 0.464 | 0.474 | 0.509 |
| #labels/node | | 1.89 | 1.69 | 1.81 | 0.368 | 0.376 | 0.403 |

**Table 4.3:** Average computational costs of profile queries on the **Léon** dataset for various graph models and car save limits.

| | Naive | s2l-0 | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 |
|---|---|---|---|---|---|---|---|
| Samples | 17 | 26 | 20 | 20 | 20 | 20 | 20 |
| CPU Time | 5:18:43 | 1:21:31 | 0:04:55 | 0:00:34 | 0:01:34 | 0:00:09 | 0:00:03 |
| #labels at car nodes | 14.4 M | 6.18 M | 1.34 M | 330 K | 968 K | 115 K | 20 K |
| #labels at walk nodes | 13.2 M | 3.15 M | 913 K | 432 K | 793 K | 239 K | 116 K |
| #labels at station nodes | 13.7 M | 1.25 M | 439 K | 201 K | 988 K | 376 K | 179 K |
| #labels at line nodes | 7.34 M | 1.63 M | 805 K | 454 K | 1.32 M | 668 K | 408 K |
| #labels at transit nodes | 21 M | 2.88 M | 1.24 M | 655 K | 2.3 M | 1.04 M | 587 K |
| #labels total | 48.6 M | 12.2 M | 3.49 M | 1.42 M | 4.06 M | 1.4 M | 723 K |
| #labels/car node | 4.05 K | 942 | 204 | 50.3 | 103 | 12.2 | 2.13 |
| #labels/walk node | 2.61 K | 469 | 136 | 64.3 | 81.2 | 24.5 | 11.8 |
| #labels/station node | 2.51 K | 229 | 80.7 | 37 | 182 | 69.1 | 32.9 |
| #labels/line node | 498 | 111 | 54.6 | 30.8 | 89.3 | 45.3 | 27.7 |
| #labels/transit node | 1.04 K | 143 | 61.6 | 32.5 | 114 | 51.7 | 29.1 |
| #labels/node | 1.69 K | 365 | 104 | 42.4 | 103 | 35.5 | 18.4 |
| **w/o car usage** | | | | | | | |
| #labels at walk nodes | | 380 K | 365 K | 327 K | 93 K | 96.9 K | 95.5 K |
| #labels at station nodes | | 165 K | 166 K | 146 K | 128 K | 134 K | 133 K |
| #labels at line nodes | | 248 K | 248 K | 223 K | 188 K | 196 K | 195 K |
| #labels at transit nodes | | 414 K | 414 K | 369 K | 316 K | 331 K | 328 K |
| #labels total | | 793 K | 779 K | 697 K | 409 K | 427 K | 423 K |
| #labels/walk node | | 56.5 | 54.2 | 48.7 | 9.52 | 9.92 | 9.78 |
| #labels/station node | | 30.4 | 30.5 | 26.9 | 23.5 | 24.7 | 24.4 |
| #labels/line node | | 16.8 | 16.8 | 15.1 | 12.8 | 13.3 | 13.2 |
| #labels/transit node | | 20.5 | 20.5 | 18.3 | 15.7 | 16.4 | 16.2 |
| #labels/node | | 29.5 | 29 | 25.9 | 13.7 | 14.3 | 14.1 |

**Table 4.4:** Average computational costs of profile queries on the **Austin** dataset for various graph models and car save limits.

| | s2l-0 | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 |
|---|---|---|---|---|---|---|
| Samples | 13 | 20 | 20 | 20 | 20 | 20 |
| CPU Time | 17:06:05 | 1:10:51 | 0:09:37 | 0:08:20 | 0:00:49 | 0:00:19 |
| #labels at car nodes | 24.1 M | 6.61 M | 1.77 M | 2.92 M | 563 K | 149 K |
| #labels at walk nodes | 10.1 M | 3.55 M | 1.98 M | 2.16 M | 761 K | 477 K |
| #labels at station nodes | 6.06 M | 2.35 M | 1.34 M | 5.32 M | 2.16 M | 1.26 M |
| #labels at line nodes | 7.27 M | 3.76 M | 2.31 M | 6.48 M | 3.32 M | 2.17 M |
| #labels at transit nodes | 13.3 M | 6.11 M | 3.65 M | 11.8 M | 5.49 M | 3.43 M |
| #labels total | 47.5 M | 16.3 M | 7.4 M | 16.9 M | 6.81 M | 4.05 M |
| #labels/car node | 1.09 K | 299 | 80.1 | 105 | 20.2 | 5.36 |
| #labels/walk node | 450 | 158 | 88.3 | 74.5 | 26.3 | 16.5 |
| #labels/station node | 260 | 101 | 57.7 | 229 | 93 | 54.3 |
| #labels/line node | 145 | 74.9 | 46 | 129 | 66.2 | 43.2 |
| #labels/transit node | 181 | 83.2 | 49.7 | 161 | 74.7 | 46.7 |
| #labels/node | 403 | 138 | 62.8 | 130 | 52.3 | 31.1 |
| **w/o car usage** | | | | | | |
| #labels at walk nodes | 1.19 M | 1.27 M | 1.3 M | 307 K | 261 K | 278 K |
| #labels at station nodes | 792 K | 835 K | 857 K | 764 K | 657 K | 694 K |
| #labels at line nodes | 1.02 M | 1.08 M | 1.1 M | 978 K | 848 K | 897 K |
| #labels at transit nodes | 1.81 M | 1.91 M | 1.96 M | 1.74 M | 1.51 M | 1.59 M |
| #labels total | 3.01 M | 3.18 M | 3.26 M | 2.05 M | 1.77 M | 1.87 M |
| #labels/walk node | 53 | 56.6 | 58 | 10.6 | 9 | 9.61 |
| #labels/station node | 34 | 35.9 | 36.8 | 32.8 | 28.2 | 29.8 |
| #labels/line node | 20.4 | 21.4 | 22 | 19.5 | 16.9 | 17.9 |
| #labels/transit node | 24.7 | 26 | 26.7 | 23.7 | 20.5 | 21.7 |
| #labels/node | 31.3 | 33.2 | 34 | 20 | 17.2 | 18.3 |

**Table 4.5:** Average computational costs of profile queries on the **Dallas** dataset for various graph models and car save limits.

| | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 |
|---|---|---|---|---|---|
| Samples | 2 | 24 | 20 | 20 | 20 |
| CPU Time | 1 day, 0:06:14 | 3:14:24 | 5:11:41 | 0:35:36 | 0:04:45 |
| #labels at car nodes | 29.9 M | 9.43 M | 23.1 M | 4.97 M | 969 K |
| #labels at walk nodes | 22.1 M | 10.7 M | 18.1 M | 7.17 M | 3.11 M |
| #labels at station nodes | 7.69 M | 3.78 M | 19.9 M | 7.82 M | 3.38 M |
| #labels at line nodes | 13.6 M | 7.31 M | 26.5 M | 13 M | 6.36 M |
| #labels at transit nodes | 21.3 M | 11.1 M | 46.5 M | 20.8 M | 9.73 M |
| #labels total | 73.4 M | 31.3 M | 87.7 M | 32.9 M | 13.8 M |
| #labels/car node | 810 | 255 | 444 | 95.4 | 18.6 |
| #labels/walk node | 588 | 285 | 337 | 133 | 57.8 |
| #labels/station node | 353 | 174 | 916 | 359 | 155 |
| #labels/line node | 165 | 88.6 | 321 | 157 | 77 |
| #labels/transit node | 204 | 106 | 445 | 199 | 93.3 |
| #labels/node | 410 | 175 | 417 | 157 | 65.7 |
| **w/o car usage** | | | | | |
| #labels at walk nodes | 5.54 M | 6.82 M | 2.38 M | 2.29 M | 2.16 M |
| #labels at station nodes | 1.97 M | 2.39 M | 2.42 M | 2.33 M | 2.22 M |
| #labels at line nodes | 2.84 M | 3.36 M | 3.4 M | 3.28 M | 3.16 M |
| #labels at transit nodes | 4.81 M | 5.74 M | 5.82 M | 5.61 M | 5.38 M |
| #labels total | 10.3 M | 12.6 M | 8.2 M | 7.9 M | 7.54 M |
| #labels/walk node | 147 | 181 | 44.4 | 42.7 | 40.1 |
| #labels/station node | 90.5 | 110 | 111 | 107 | 102 |
| #labels/line node | 34.4 | 40.7 | 41.1 | 39.7 | 38.3 |
| #labels/transit node | 46.1 | 55.1 | 55.7 | 53.8 | 51.6 |
| #labels/node | 72.9 | 88.5 | 51.9 | 50 | 47.7 |

**Table 4.6:** Average computational costs of profile queries on the **Toronto** dataset for various graph models and car save limits.

| | s2l-10 | l2l-5 | l2l-10 |
|---|---|---|---|
| Samples | 20 | 20 | 20 |
| CPU Time | 6:44:16 | 1:26:16 | 0:17:31 |
| #labels at car nodes | 13.6 M | 8.52 M | 1.76 M |
| #labels at walk nodes | 19.1 M | 17.8 M | 8.43 M |
| #labels at station nodes | 6.22 M | 13.4 M | 6.3 M |
| #labels at line nodes | 11.9 M | 23.2 M | 11.9 M |
| #labels at transit nodes | 18.1 M | 36.6 M | 18.2 M |
| #labels total | 50.8 M | 62.9 M | 28.4 M |
| #labels/car node | 239 | 98.1 | 20.3 |
| #labels/walk node | 326 | 193 | 91.4 |
| #labels/station node | 184 | 397 | 187 |
| #labels/line node | 96.9 | 189 | 97.2 |
| #labels/transit node | 116 | 234 | 117 |
| #labels/node | 187 | 188 | 84.7 |
| **w/o car usage** | | | |
| #labels at walk nodes | 11.8 M | 4.87 M | 4.81 M |
| #labels at station nodes | 3.83 M | 3.57 M | 3.41 M |
| #labels at line nodes | 5.47 M | 5.2 M | 4.99 M |
| #labels at transit nodes | 9.3 M | 8.76 M | 8.4 M |
| #labels total | 21.1 M | 13.6 M | 13.2 M |
| #labels/walk node | 202 | 52.8 | 52.1 |
| #labels/station node | 114 | 106 | 101 |
| #labels/line node | 44.6 | 42.4 | 40.7 |
| #labels/transit node | 59.5 | 56.1 | 53.7 |
| #labels/node | 98.3 | 54.9 | 53.1 |

**Table 4.7:** Average computational costs of profile queries on the **New York** dataset for various graph models and car save limits.

## 4.2 Generated Patterns

We extracted transfer patterns from the profile queries and report some statistics about them in tables 4.8 to 4.12. We generate transfer patterns for our sample queries to 1000 random target locations. A target locations always corresponds to the location of a station. This is a limitation of our current implementation, but it does not impose any restriction on the last vehicle used. It is still possible to arrive at the target location by car or walking. We also report the number of *mode patterns*, which are transfer patterns that are considered different if a different mode is used between two stations. For example the two routes *walk to the main station and take a direct train to Frankfurt* and *take a taxi to the main station and take a direct train to Frankfurt* are counted as a single transfer pattern but two mode patterns. For patterns that use public transportation there is a station where the public transportation network is entered the first time and a station where the public transportation network is exited the last time. We refer to these stations as *access stations*. The number of access stations that are accessed by walking (#walk stations) or by car (#car stations) or somehow (#stations) is reported in the tables. The latter also includes the source and target stations themselves. Some transfer patterns are only valid at a single time, i. e. backed by only one label. We report the same numbers with these patterns filtered out. So the filtered figures do not take these edge cases into account. All numbers are reported for each feasible model separately. We additionally include the naive model with car usage disabled (w/o car) for comparison. This roughly corresponds to the original uni-modal transfer patterns approach, with the exception that walking is not limited. So it does include variations, which use walking to save a transfer as described in section 3.5.1. It can also include routes with large amounts of walking. Please note that in our models with car usage enabled, walking is implicitly more restricted as the worst travel-time is bound by the best travel time multiplied by $C_{\text{spread}}$ and the direct connection by car implies a lower limit.

To illustrate the distribution of the number of patterns we also provide box plots in figures 4.1 to 4.5. They do not show the naive model and the maximum observation for some models, as they would require a too large scale.

The number of access stations is about equal for the stops-to-lines and lines-to-lines model for a fixed car save limit. This is what had to be expected, as the lines-to-lines model only restricts variations between pairs of lines. It ranges from about 8 for Dallas to 17 for New York with a car save limit of 10 minutes. The number of walk access stations is consistently lower than the number of car access stations. Note that the number of access stations and the number of patterns is higher for the naive model without car usage than for the lines-to-lines model with a car save limit of 10 minutes for some datasets. This is an indication that our simple oracles do indeed miss some interesting

connections. It is not too surprising as we just stuck with very simple implementations of oracles as a first attempt. They can easily miss connections. For example if a bus line goes in two directions, with different stops for each direction on the respective side of the street, our oracles just return connections to one of these stops, which can be the one that goes in the wrong direction for an actual route. So any evaluation of the quality of the resulting routes does not seem very reasonable at this stage. We have initial ideas for more sophisticated oracles, but neither a implementation nor experimental results so far.

| | Naive | s2l-0 | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 | w/o car |
|---|---|---|---|---|---|---|---|---|
| Samples | 83 | 84 | 79 | 82 | 82 | 80 | 83 | 75 |
| #patterns avg | 119 | 12.7 | 4 | 2.76 | 11 | 4.17 | 2.7 | 7.67 |
| #patterns stdev | 94.2 | 7.46 | 2.41 | 2.57 | 6.08 | 2.69 | 2.79 | 4.43 |
| #mode patterns avg | 137 | 15 | 4.28 | 2.92 | 13.1 | 4.47 | 2.86 | 7.84 |
| #mode patterns stdev | 108 | 9.35 | 2.61 | 2.66 | 7.8 | 2.91 | 2.89 | 4.46 |
| #walk stations avg | 39.4 | 5.67 | 3.1 | 1.63 | 5.34 | 3.29 | 1.68 | 7.15 |
| #walk station stddev | 26.7 | 3.15 | 2.35 | 2.34 | 2.96 | 2.53 | 2.61 | 4.39 |
| #car stations avg | 55.3 | 6.54 | 0.353 | 0.0221 | 5.93 | 0.419 | 0.0146 | 0 |
| #car station stddev | 37.7 | 3.9 | 0.917 | 0.192 | 3.49 | 1.01 | 0.148 | 0 |
| #stations avg | 65.4 | 11.7 | 4.23 | 2.49 | 10.7 | 4.42 | 2.4 | 8.18 |
| #station stddev | 41.5 | 5.54 | 2.6 | 2.6 | 4.95 | 2.77 | 2.9 | 4.51 |
| **Filtered** | | | | | | | | |
| #patterns avg | 76.6 | 6.88 | 2.32 | 1.83 | 6.42 | 2.44 | 1.92 | 4.01 |
| #patterns stdev | 52.3 | 3.69 | 1.41 | 1.38 | 3.31 | 1.53 | 1.6 | 2.05 |
| #mode patterns avg | 94.6 | 9.14 | 2.64 | 2.14 | 8.57 | 2.79 | 2.22 | 4.18 |
| #mode patterns stdev | 65.9 | 6.22 | 1.7 | 1.56 | 5.58 | 1.83 | 1.74 | 2.12 |
| #walk stations avg | 31.7 | 4.37 | 2.57 | 1.78 | 4.3 | 2.73 | 1.97 | 4.97 |
| #walk station stddev | 19.6 | 2.5 | 1.72 | 1.74 | 2.46 | 1.83 | 2 | 2.67 |
| #car stations avg | 45 | 4.66 | 0.301 | 0.0358 | 4.48 | 0.354 | 0.0239 | 0 |
| #car station stddev | 28.6 | 2.65 | 0.713 | 0.231 | 2.52 | 0.778 | 0.183 | 0 |
| #stations avg | 53.3 | 8.5 | 3.6 | 2.94 | 8.23 | 3.74 | 3.07 | 5.88 |
| #station stddev | 31.1 | 3.62 | 1.73 | 1.62 | 3.43 | 1.82 | 1.87 | 2.63 |

**Table 4.8:** Patterns for **Leon** for various graph models and car save limits.

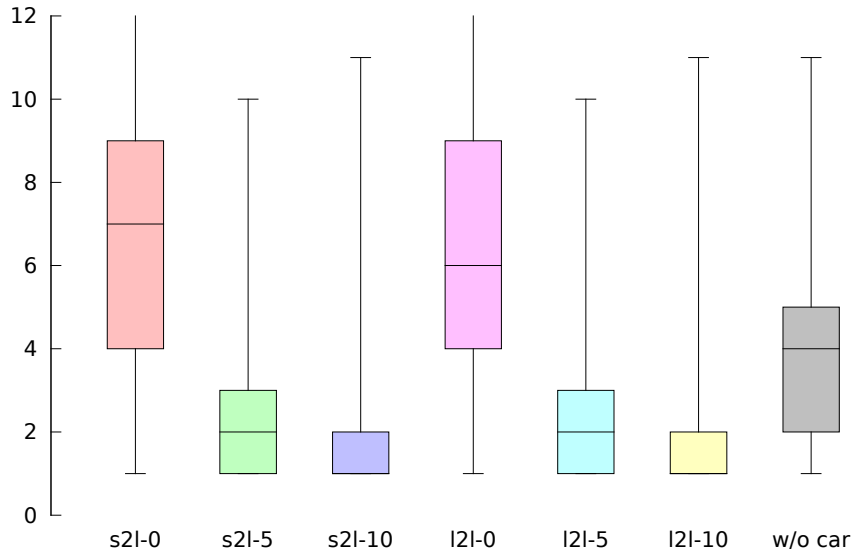


**Figure 4.1:** Number of filtered transfer patterns for **Léon**.

| | Naive | s2l-0 | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 | w/o car |
|---|---|---|---|---|---|---|---|---|
| Samples | 17 | 26 | 20 | 20 | 20 | 20 | 20 | 20 |
| #patterns avg | 686 | 135 | 36.6 | 16 | 91.5 | 28.1 | 13.4 | 26.7 |
| #patterns stdev | 500 | 102 | 33.3 | 13.2 | 67.8 | 22.9 | 11.1 | 16.5 |
| #mode patterns avg | 725 | 149 | 42 | 17.2 | 103 | 32.9 | 14.3 | 26.9 |
| #mode patterns stdev | 531 | 115 | 39.3 | 15 | 77.7 | 27.9 | 12.1 | 16.6 |
| #walk stations avg | 45.1 | 13.2 | 8.48 | 5.98 | 12 | 8.33 | 6.07 | 12.5 |
| #walk station stddev | 20.6 | 5.86 | 4.52 | 3.89 | 5.61 | 4.44 | 3.99 | 5.78 |
| #car stations avg | 194 | 38 | 9.22 | 1.69 | 36.1 | 8.75 | 1.66 | 0 |
| #car station stddev | 97.2 | 21.5 | 8.81 | 3.34 | 20.6 | 7.96 | 3.17 | 0 |
| #stations avg | 212 | 47.6 | 16.8 | 8.47 | 44.7 | 16.2 | 8.5 | 13.9 |
| #station stddev | 100 | 23 | 10.5 | 5.49 | 21.8 | 9.57 | 5.47 | 5.94 |
| **Filtered** | | | | | | | | |
| #patterns avg | 341 | 70.6 | 21.1 | 9.14 | 53.8 | 17.4 | 8.07 | 12.6 |
| #patterns stdev | 216 | 46.7 | 17.9 | 7.31 | 35.8 | 13.7 | 6.39 | 7.03 |
| #mode patterns avg | 381 | 84.8 | 26.6 | 10.5 | 65 | 22.3 | 9 | 12.8 |
| #mode patterns stdev | 247 | 59.9 | 24.4 | 9.55 | 46.2 | 19.1 | 7.64 | 7.12 |
| #walk stations avg | 31 | 11.6 | 7.29 | 5.13 | 10.7 | 7.31 | 5.29 | 8.95 |
| #walk station stddev | 13.5 | 5.16 | 3.83 | 3.04 | 4.95 | 3.74 | 3.1 | 4.32 |
| #car stations avg | 142 | 28.9 | 7.64 | 1.57 | 28.1 | 7.31 | 1.53 | 0 |
| #car station stddev | 69.1 | 15.7 | 7.14 | 2.92 | 15.5 | 6.42 | 2.77 | 0 |
| #stations avg | 156 | 37.6 | 14 | 7.4 | 36 | 13.8 | 7.51 | 10.1 |
| #station stddev | 71.3 | 17.3 | 8.64 | 4.38 | 16.8 | 7.75 | 4.35 | 4.44 |

**Table 4.9:** Patterns for **Austin** for various graph models and car save limits.

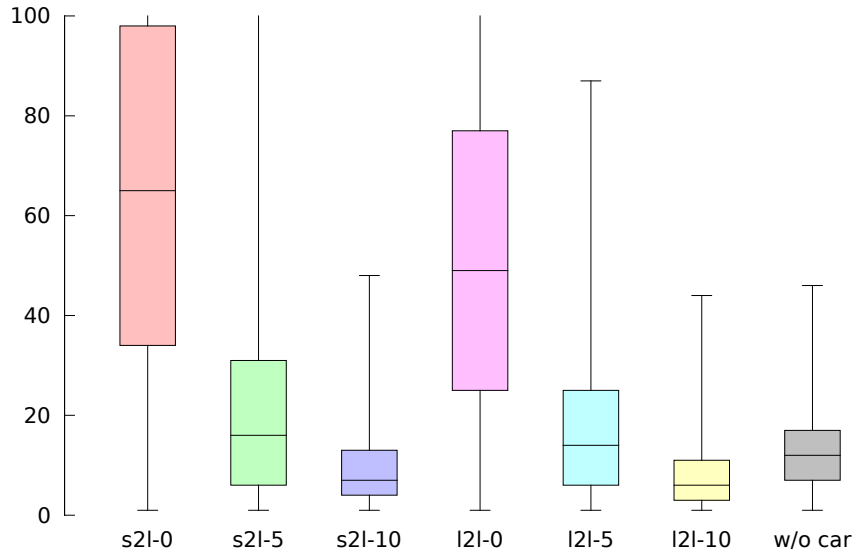


**Figure 4.2:** Number of filtered transfer patterns for **Austin**.

| | s2l-0 | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 | w/o car |
|---|---|---|---|---|---|---|---|
| Samples | 13 | 20 | 20 | 20 | 20 | 20 | 20 |
| #patterns avg | 204 | 54 | 25.1 | 150 | 46.7 | 21.7 | 39.5 |
| #patterns stdev | 137 | 39.4 | 18.1 | 103 | 32.5 | 15.7 | 21 |
| #mode patterns avg | 234 | 66.3 | 29.6 | 179 | 55.9 | 26.2 | 40.9 |
| #mode patterns stdev | 157 | 48.5 | 22 | 121 | 39.3 | 19.3 | 21.7 |
| #walk stations avg | 12.7 | 9.48 | 8.01 | 12.7 | 9.32 | 7.44 | 16.3 |
| #walk station stddev | 4.7 | 3.93 | 3.7 | 4.89 | 3.95 | 3.8 | 6.04 |
| #car stations avg | 56.9 | 14.8 | 3.71 | 55.6 | 15.1 | 4.38 | 0 |
| #car station stddev | 29 | 10.2 | 4 | 28.4 | 10 | 4.47 | 0 |
| #stations avg | 65.7 | 23 | 12.1 | 64.5 | 23.3 | 12 | 17.8 |
| #station stddev | 29.6 | 11.4 | 5.84 | 29.5 | 11.4 | 6.35 | 6.21 |
| **Filtered** | | | | | | | |
| #patterns avg | 110 | 32.9 | 15.2 | 90.2 | 30 | 14.5 | 15.9 |
| #patterns stdev | 68.5 | 22.6 | 10.6 | 57.1 | 20.1 | 9.99 | 7.03 |
| #mode patterns avg | 140 | 45.3 | 19.9 | 119 | 39.2 | 19.2 | 17.2 |
| #mode patterns stdev | 90.8 | 32.7 | 15.4 | 77.3 | 27.5 | 14.3 | 8.27 |
| #walk stations avg | 11.2 | 8.35 | 6.85 | 11.2 | 8.16 | 6.58 | 10.3 |
| #walk station stddev | 4.12 | 3.39 | 2.98 | 4.29 | 3.4 | 3.02 | 3.85 |
| #car stations avg | 42.8 | 12.2 | 3.25 | 42.9 | 12.5 | 3.98 | 0 |
| #car station stddev | 21.7 | 8.23 | 3.42 | 21.4 | 8.11 | 3.86 | 0 |
| #stations avg | 50.8 | 19.4 | 10.5 | 50.9 | 19.7 | 10.8 | 11.6 |
| #station stddev | 22.3 | 9.4 | 4.87 | 22.4 | 9.37 | 5.18 | 4.02 |

**Table 4.10:** Patterns for **Dallas** for various graph models and car save limits.

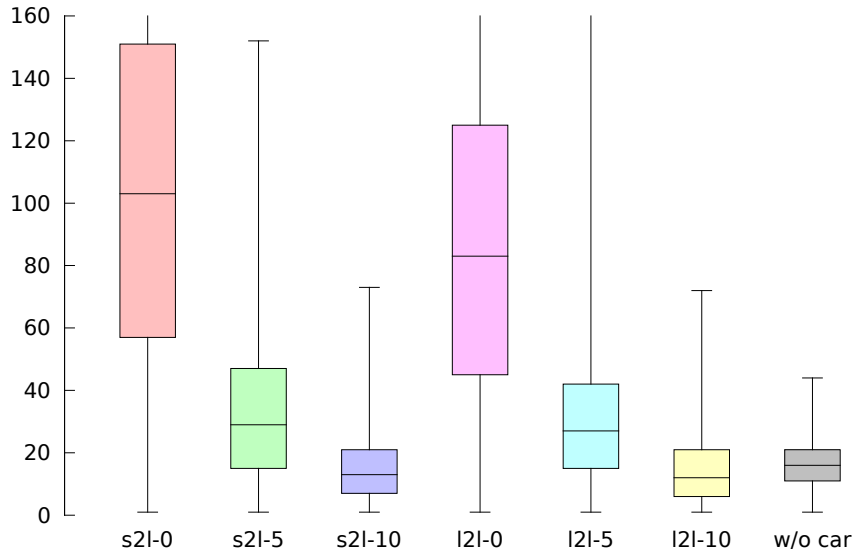


**Figure 4.3:** Number of filtered transfer patterns for **Dallas**.

| | s2l-5 | s2l-10 | l2l-0 | l2l-5 | l2l-10 | w/o car |
|---|---|---|---|---|---|---|
| Samples | 2 | 24 | 20 | 20 | 20 | 20 |
| #patterns avg | 91.2 | 32 | 190 | 69.3 | 30 | 40.8 |
| #patterns stdev | 94.1 | 27.7 | 145 | 62.1 | 26.1 | 26.5 |
| #mode patterns avg | 96.7 | 34.4 | 207 | 76 | 32 | 41 |
| #mode patterns stdev | 97.6 | 30.3 | 156 | 67 | 28.3 | 26.5 |
| #walk stations avg | 9.76 | 8.06 | 15.1 | 10.3 | 8.03 | 16.2 |
| #walk station stddev | 3.71 | 3.63 | 5.4 | 4.78 | 4.04 | 7.01 |
| #car stations avg | 22 | 5 | 61.6 | 21.4 | 4.75 | 0 |
| #car station stddev | 18.6 | 6.11 | 34.2 | 18.6 | 6.24 | 0 |
| #stations avg | 30.5 | 13.5 | 72.5 | 30.5 | 13.3 | 17.4 |
| #station stddev | 19.9 | 7.46 | 35.8 | 20.4 | 7.77 | 7.14 |
| **Filtered** | | | | | | |
| #patterns avg | 54.2 | 20.2 | 121 | 45.4 | 19.2 | 21.3 |
| #patterns stdev | 52.3 | 17 | 85.9 | 39.2 | 16.1 | 12.8 |
| #mode patterns avg | 59.7 | 22.6 | 139 | 52.1 | 21.2 | 21.4 |
| #mode patterns stdev | 56 | 19.8 | 98.1 | 44.5 | 18.4 | 12.8 |
| #walk stations avg | 8.26 | 6.81 | 13.2 | 8.74 | 6.72 | 11.1 |
| #walk station stddev | 3.16 | 3.1 | 4.75 | 3.93 | 3.24 | 4.72 |
| #car stations avg | 17.1 | 3.96 | 48.1 | 16.7 | 3.7 | 0 |
| #car station stddev | 14.1 | 4.95 | 25.9 | 14.2 | 4.91 | 0 |
| #stations avg | 24.3 | 11.2 | 57.7 | 24.4 | 10.9 | 12.2 |
| #station stddev | 15.2 | 6.1 | 27.4 | 15.8 | 6.08 | 4.81 |

**Table 4.11:** Patterns for **Toronto** for various graph models and car save limits.

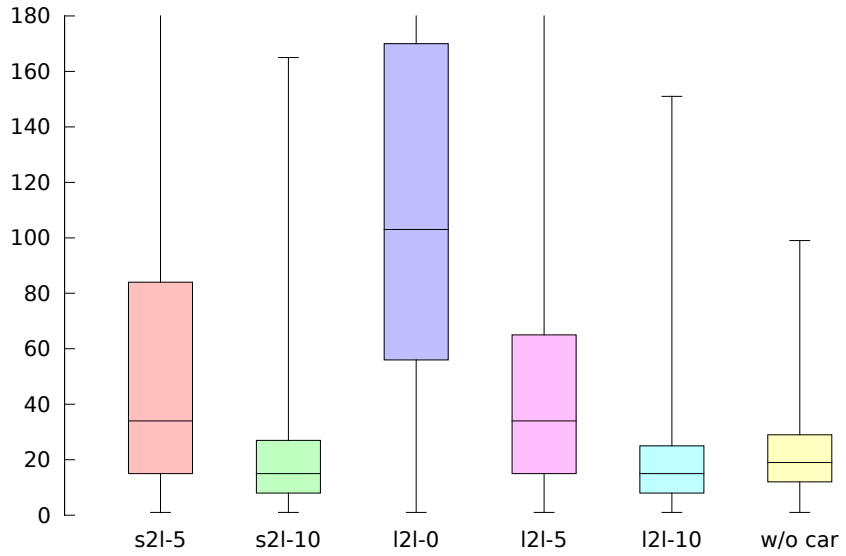


**Figure 4.4:** Number of filtered transfer patterns for **Toronto**.

| | s2l-10 | l2l-5 | l2l-10 | w/o car |
|---|---|---|---|---|
| Samples | 20 | 20 | 20 | 20 |
| #patterns avg | 55.4 | 129 | 60.7 | 53.2 |
| #patterns stdev | 68.2 | 133 | 68.9 | 40.2 |
| #mode patterns avg | 61.1 | 151 | 67 | 53.2 |
| #mode patterns stdev | 76.7 | 155 | 79 | 40.2 |
| #walk stations avg | 11.4 | 15.3 | 12.1 | 21.4 |
| #walk station stddev | 5.78 | 6.58 | 5.98 | 8.85 |
| #car stations avg | 6.79 | 28 | 8.71 | 0 |
| #car station stddev | 9.48 | 21.2 | 11.4 | 0 |
| #stations avg | 17.6 | 39.8 | 20.2 | 22.2 |
| #station stddev | 12.1 | 23.6 | 13.9 | 9.03 |
| **Filtered** | | | | |
| #patterns avg | 31.9 | 76.5 | 35.8 | 24.8 |
| #patterns stdev | 33.8 | 70.7 | 37.1 | 15.2 |
| #mode patterns avg | 37.7 | 98.4 | 42.3 | 24.8 |
| #mode patterns stdev | 43 | 93.6 | 47.7 | 15.2 |
| #walk stations avg | 9.51 | 12.9 | 10.2 | 14.5 |
| #walk station stddev | 4.53 | 5.53 | 4.66 | 6.12 |
| #car stations avg | 5.97 | 22.6 | 7.57 | 0 |
| #car station stddev | 8.03 | 16.8 | 9.55 | 0 |
| #stations avg | 14.8 | 32.3 | 17.1 | 15.1 |
| #station stddev | 9.84 | 18.6 | 11.3 | 6.26 |

**Table 4.12:** Patterns for **New York** for various graph models and car save limits.

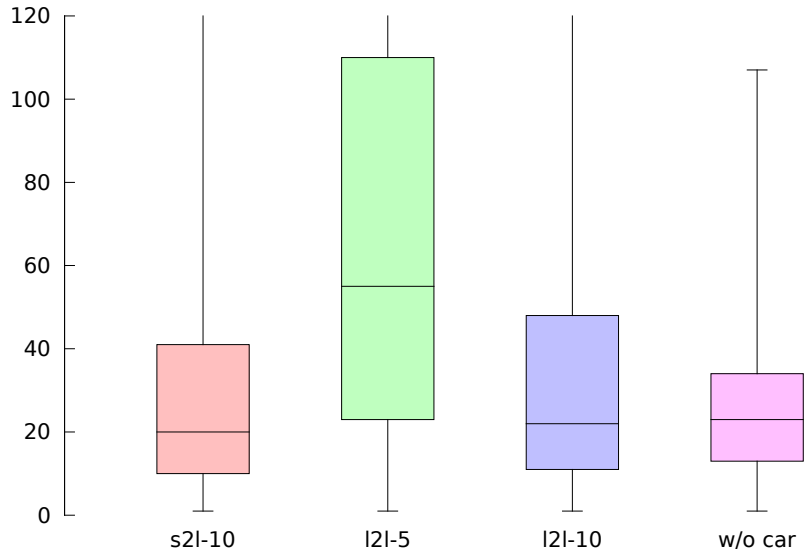


**Figure 4.5:** Number of filtered transfer patterns for **New York**.

# 5 Conclusions

We first implemented an approach for routing with transfer patterns on road networks. We therefore treated a change of street name like a transfer. This naive approach however had issues with the large number of stations, so we've switched focus to computing transfer patterns for multi-modal route planning. We therefore abstracted from routing on road networks and treated a route between two locations on the road network just as a direct connection, assuming that these connections could be looked up quickly at query time using any existing, efficient routing algorithm for road networks. We examined ways to compute multi-modal transfer patterns, starting from a naively joint multi-modal, line-based graph model with a Pareto cost model. This turned out to generate many unreasonable route variations that render precomputation infeasible. We have identified reasons why these variations appear and introduced restricted graph models and the car save limit, that together eliminate most of the variations and make precomputation of profile queries feasible, even on large datasets.

To construct our graph models, we assumed to be provided with reasonable connections by an oracle. For our experiments, we have used a very simple implementation of an oracle. This was good enough to allow us to confirm that the kind of variations we suspected where indeed the ones that make precomputation infeasible. However, it is unclear if precomputation still remains feasible with more advanced and realistic oracles. So unfortunately, we cannot make any claim about the practical utility of our models. This is the most obvious limitation of our work and probably the first thing one would want to examine in future work.

For practical multi-modal route planning based on precomputed transfer patterns, one would also like to be able to execute location-to-location queries between arbitrary locations. It would be infeasible to precompute and store transfer patterns between all pairs of OSM-Nodes the way we did it for a set of sample locations. So it is an open question how such queries can be executed or how transfer patterns for all locations can be computed and stored efficiently. We do not yet have a solution for this and it remains an interesting challenge for future work.

Taking these limitations into account, the most useful contribution of this work is probably a better understanding of the route variations that occur in multi-modal route planning with a Pareto cost model. We hope, that this insight is valuable for the future

development of multi-modal route planning algorithms, even if the models we examined should turn out not to be the way to go.

# Bibliography


[1] General transit feed specification. `https://developers.google.com/transit/gtfs/`.

[2] Openstreetmap. `http://www.openstreetmap.org/`.

[3] Leonid Antsfeld and Toby Walsh. Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm. In *Proceedings of the 19th ITS World Congress*, 2012.

[4] 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-Verlag, 2009.

[5] 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 *Proceedings of the 18th annual European Conference on Algorithms*, ESA'10, pages 290–301. Springer-Verlag, 2010.

[6] Holger Bast, Stefan Funke, and Domagoj Matijevic. Transit – ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge*, 2006.

[7] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*. SIAM, 2007.

[8] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[9] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato Werneck. Computing and evaluating multimodal journeys. Technical report, Karlsruher Institut für Technologie, 2012.

[10] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating multi-modal route planning by access-nodes. In *Proceedings of the 17th Annual European Conference on Algorithms*, ESA'09, pages 587–598. Springer, 2009.

[11] Daniel Delling, Thomas Pajor, and Renato Werneck. Round-based public transit routing. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments*. SIAM, 2012.

[12] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-constrained multimodal route planning. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments*. SIAM, 2012.

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

[14] Robert Geisberger. *Advanced Route Planning in Transportation Networks*. PhD thesis, Karlsruher Institut für Technologie, 2011.

[15] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 319–333. Springer-Verlag, 2008.

[16] Thomas Misa and Philip France. An interview with Edsger W. Dijkstra. *Communications of the ACM*, 53(8):41–47, 2010.

[17] Thomas Pajor. Multi-modal route planning. Master's thesis, Karlsruher Institut für Technologie, 2009.

[18] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12, 2008.

[19] Haicong Yu and Feng Lu. Advanced multi-modal routing approach for pedestrians. In *Proceedings of the 2nth International Conference on Consumer Electronics, Communications and Networks*, pages 2349 –2352, 2012.