Karlsruhe Institute of Technology

# Engineering Multimodal Transit Route Planning

Master Thesis of


## Huyen Chau Nguyen

At the Department of Informatics
Institute of Theoretical Computer Science


Reviewers: Prof. Dr. Dorothea Wagner
Prof. Dr. Peter Sanders
Advisors: Moritz Baum, M.Sc.
Tobias Zündorf, M.Sc.

Time Period: May 2017 – October 2017

**Statement of Authorship**

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

Karlsruhe, 31st October 2017

## Abstract

Public transit route planning that finds shortest paths in a public transit network that includes footpaths is a challenge. It faces the difficulty of handling data of a footpath network and data of scheduled public transit data. It is a shortest path problem in a graph but travel times in a public transit network are time-dependent because they have to represent time tables of connections. Finding shortest paths in a graph with constant travel times such as in footpath networks and shortest paths in a time-dependent graph are conceptually very different. This thesis focuses on finding routes in such a multimodal graph with constant and time-dependent travel times. Our aim is to compute travel time profiles which means that we find all shortest paths from a start to a goal for any departure time within a given time window. We utilize Dijkstra's algorithm to solve this problem as our baseline algorithm. We implement A* Search as a speed-up technique. A* Search that computes profiles in a multimodal network loses the label-setting property which impacts its performance. Furthermore, it is difficult to find suitable potentials in a time-dependent graph that accelerate computation. At last, we propose our Multilevel A* algorithm. We divide the graph into several levels. On the resulting multilevel graph we can make use of the levels by generating different potentials that aid us in computing shortest paths in a multimodal graph.



## Deutsche Zusammenfassung

Routenplanung im öffentlichen Verkehrsnetz, welches auch Fußwege berücksichtigt, ist schwierig. Eine solche Routenplanung muss sowohl den Fußwegegraphen als auch das öffentliche Verkehrsnetz berücksichtigen. Es ist eine Suche nach kürzesten Wegen in einem Graph. Allerdings sind Reisezeiten im öffentlichen Verkehrsnetz zeitabhängig, denn sie spiegeln Fahrpläne wieder. Die Suche nach kürzesten Wegen in konstanten Graphen wie in einem Fußwegegraphen unterscheidet sich aber grundsätzlich sehr von der Suche nach kürzesten Wegen in zeitabhängigen Graphen. Unser Ziel ist die Berechnung von kürzesten Wegen in solchen multimodalen Netzwerken, die sowohl konstante Reisezeiten als auch zeitabhängige Reisezeiten beeinhalten. Wir berechnen Reisezeitprofile, also alle kürzesten Wege von einem Start zu einem Ziel zu jeder Abfahrtszeit innerhalb eines gegebenen Zeitfensters. Wir untersuchen, wie man Dijkstras Algorithmus anpassen kann um solche Profile zu berechnen und nutzen Dijkstras Algorithmus als Referenz. Wir nutzen die A* Suche als Beschleunigungstechnik. Eine A* Suche welche Profile in einem multimodalen Netzwerk berechnet, besitzt nicht mehr die Label-Setting Eigenschaft. Dies beeinträchtigt die Laufzeit. Zudem ist es in multimodalen Netzwerken schwierig, geeignete Potenziale zu finden, die die Berechnung genügend schnell unterstützen. Schließlich stellen wir unseren neuen Algorithmus Multilevel A* vor. Wir führen Ebenen in einem Graph ein und ordnen jede Kante einer Ebene zu. Auf einem solchen Graph berechnen wir verschiedene Potenziale für verschiede Ebenen um kürzeste Wege zu berechnen.

# Contents

**1 Introduction** **1**
1.1 Related Work . . . . . 1
1.2 Contribution . . . . . 4
1.3 Outline . . . . . 4

**2 Preliminaries** **5**
2.1 Graph Theory . . . . . 5
2.2 Shortest Path Problem . . . . . 6
2.3 Public Transit Network Models . . . . . 6
2.4 Public Transit Routing . . . . . 9
2.5 Multi Modal Routing . . . . . 11

**3 Basic Approaches for Multi Modal Public Transit Routing** **13**
3.1 Dijkstra's Algorithm . . . . . 13
3.2 A* Search . . . . . 21

**4 Multilevel A*** **27**
4.1 Introducing Edge Levels . . . . . 28
4.2 A* Search on Levels . . . . . 29
4.3 Finding Level Assignments . . . . . 35

**5 Experimental Results and Evaluation** **37**
5.1 Experimental Setup and Input Data . . . . . 37
5.2 Experimental Results . . . . . 40
5.3 Evaluation . . . . . 44

**6 Conclusion** **53**
6.1 Summary . . . . . 53
6.2 Future Work . . . . . 53

**Bibliography** **55**

# 1. Introduction

Mobility and traveling have always been important and is becoming more and more essential in our globalized world. Navigation and routing without computer assistance are nowadays unimaginable. The popularity of applications such as Google Maps and mobile routing on the phone show this clearly.

Breakthroughs in research enabled this trend. Routing is fundamentally a shortest path problem in a road network modeled as graphs. Research has found many speed-up techniques to improve the runtime of finding shortest paths in road networks to support the need for fast and high-quality route generation greatly. Related, but still very different to routing in road networks is public transit routing. With progressing urbanization, public transit will become even more important in the future. Public transit networks can also be modeled as graphs. Graphs of public transit however, are very different to road network graphs. One of the reasons for this is the type of requests needed for public transit routing. In public transit routing we are not only interested in the shortest distance between two points at a certain time but we wish to know the next few connections or all connections within a time range. For car or pedestrian routing, we are mostly interested in optimizing travel time. Useful public transit routing however may optimize more than one criteria such as the number of transfers, ticket fees or reliability of the resulting route.

In public transit routing, only regarding public the transit network for routing is not sufficient in practical use. In most realistic cases, a trip using public transit starts or ends with walking to a station and from a station . Shortest public transit trips sometimes require multiple footpaths during the trip. This includes but is not limited to changing platforms or changing to a another nearby station. Most prior work does not address these footpaths or require exhausting data maintenance.

This work examines the A* algorithm to solve public transit routing with unrestricted walking and proposes an algorithm to find shortest paths. The work presented does not require special handling of footpaths. The algorithm works with a normal pedestrian walking graph merged with a public transit graph.

## 1.1 Related Work

Routing is a shortest path problem in a graph. The most important algorithm and the base of many other works is Dijkstra's Algorithm [Dij59]. Dijkstra's algorithm finds a shortest path in a graph $G = (V, E)$ with non-negative edge weights without any preprocessing in

$O(|E| + |V| \log |V|)$. Given a source node $s \in V$ and a target $t \in V$, Dijkstra's algorithm finds the shortest path between $s$ and $t$ by incrementally finding all shortest paths from $s$ in a circular search space around source $s$. When the circular search space finds $t$, it has found the shortest path to $t$. An improvement to Dijkstra's circular search space was A* Search that narrows the search space and makes it goal directed [HNR68] [HNR72].

However, in big graphs such as real world road networks Dijkstra's algorithm is too slow for practical use. To improve the query time, many works make use of expensive preprocessing of the data. Most of the algorithms are exploiting graph hierarchy, partitions or goal direction. A recent survey [BDG+15] gives an extensive overview of most important techniques.

**Algorithms for Road Network Routing.** Road networks are modeled as graphs by representing intersections as vertices and roads as edges. The weight or distance of an edge is the time needed to travel from one intersection to another using the road that is connecting them. In order to consider turn restrictions, road networks can alternatively be modeled by representing roads by vertices and representing intersection where it is possible to turn from one road to another by edges. A lot of prior work targets road network routing on models like these.

A* algorithm is an algorithm that modifies the search space. The search space depends on an estimation or *potential* of every vertex to target $t$. The challenge of implementing a fast A* algorithm depends heavily on finding good potentials for every vertex to every potential target $t$ that can be used in the query. The goal-directed algorithm *ALT* [GKW07] successfully implements A* Search by using landmarks and triangle inequality for road network routing. The potentials for A* are preprocessed and used during query time. An algorithm using graph hierarchy to achieve a speed up is *Contraction Hierarchies* [GSSD08]. The algorithm exploits the fact that there are vertices that are important and part of many shortest paths and other vertices are not. Contracting insignificant vertices, a hierarchy between nodes is built. Using that hierarchy the search space is reduced. Another algorithm is *Multi-Level Dijkstra* [DGPW11]. It uses partitions and preprocesses shortest paths within partitions. This process creates an overlay graph with shortcuts. Finding a shortest path between a source $s$ and a target $t$ is reduced to finding a shortest path between the partition of $s$ and the partition of $t$.

**Algorithms for Public Transit Routing.** Modeling a public transit network as a graph is not as intuitive as in road networks due to the fact that it has to represent time tables and connections. Public transit networks are mostly modeled as either *time-expanded* graphs, or *time-dependent* graphs, or variations of those two. More details about models will be discussed in Section 2.3.

Modeling public transit networks as graphs permits solving public transit routing by solving shortest path problems at a given departure time. However, most techniques used for road network routing do not yield comparable speed-ups on public transit routing. A summary by Bast [Bas09] lists five main key differences between car routing and public transit routing. These differences prevent known techniques that work well on road networks from yielding comparable results in public transit routing: A speed-up technique often used is *bidirectional search*. When finding a route to a target it is unknown at what time the shortest path will arrive. It may also be unknown which public transit stop is the nearest to the actual goal if walking to the target is needed. Without knowledge of the arrival time and the target stop, bidirectional search is not easily applied on public transit routing. Instead, a set of potential target nodes need to be considered which makes it less efficient. Furthermore, *hierarchy* is less easily found and detectable in public transit networks. The high node degree in public transit networks prevents good *contraction or*

*shortcuts* and the lack of efficient algorithms for local search make exploiting *goal direction* or *distance tables* hard.

Most prior work in public transit routing have achieved best results when searching for shortest paths by handling data rather as timetables than normal graphs. *Connection Scan Algorithm* [DPSW13] [SW] sorts all connections by departure time and finds trips that can be reached until the target is found. *Frequency-based search for public transit* [BS14] makes use of the periodicity of connections in common timetables to reduce the number of edges needed to consider. Other approaches make use of the structure of most shortest paths in public transit. RAPTOR [DPW12] is a round based search that finds routes with a fixed number of transfers in each round. This algorithm exploits the fact that in real networks all shortest paths only have few transfers. *Transfer Pattern* [BCE+10] is an algorithm that also exploits a characteristic of shortest paths in public transit. All shortest paths in public transit between two nodes have similar patterns. Even though there may be many shortest paths between two nodes throughout the day, most of these shortest paths will be the same when comparing where these routes start, transfer and end. These characteristics of a route, its transfers throughout the route are called *transfer pattern* of a source and a target. The algorithm precomputes main pattern of the network in such a way that combining all these patterns covers all existing transfer pattern of all shortest paths. During query time of a shortest path between two nodes, only a couple of transfer patterns need to be checked for shortest paths, massively reducing the search space. *Trip-Based* Routing [Wit15] focuses on trips themselves and the transfers between trips to compute shortest paths.

Often we do not wish to answer the question of the *earliest arrival* at a given departure time. Instead, queries might ask for a time *profile* with all shortest paths in a given time range. All of the mentioned algorithms except Transfer Pattern provide an earliest arrival as well as a profile search version. The transfer pattern algorithm is extensible to profile requests. In addition to having to serve profile queries, public transit routing also faces further challenges that do not exist in road network routing. This includes additional requirements such as reducing transfers, ticket costs, or reliability of routes.

**Algorithms for Multimodal Routing.** Multimodal routing means routing that uses multiple means of transportation such as walking, driving the car, or using buses and trains within the same trip. Even though walking and driving are two different modes in reality, we define a network as a multimodal networks when it uses both, time-dependent scheduled transportation such as trains and time-independent modes of transportation such as walking or driving the car. Especially in public transit routing the need for multimodal routing is crucial because it is almost always necessary to walk to the nearest station from the source and from the last station to the actual target. Often, transfers and short footpaths are needed as well. It is also shown that footpaths are essential for finding shortest paths with public transit [WZ17]. If ignoring footpaths, the quality of shortest paths that can be found differ a lot.

Multimodal routing can be solved with *Label-Constrained* algorithms [BBH+08]. *Access Node Routing* [DPW09] is an algorithm finding shortest multimodal paths with the restriction that only the beginning and the end of trip is using time-independent transport like walking. Another algorithm [DDP+13] computes pareto sets to take in account travel time, cost and convenience. Heuristics and fuzzy logic then score the solutions. The algorithm *SDALT* [**?**] solves multimodal routing by using A* Search on precomputed potentials using landmarks.

## 1.2 Contribution

In this thesis we address the problem of finding shortest paths in multimodal route planning. Even though focusing on time-scheduled public transit and unrestricted walking, it can support any kind of multimodal data. The network in which we solve the problem is a walking pedestrian network and a public transit network that contains the network of high-speed trains as well as local public transport.

We adapt Dijkstra's algorithm in such a way that it solves routing in multimodal networks. The adapted version of Dijkstra's algorithm will be the baseline algorithm of this work. We later discuss algorithms using A* that solve the earliest arrival problem as well as profile queries. We also discuss their implementations. We propose a novel algorithm *Multilevel A** that computes profile queries based on A* Search. Assigning edges of the graph to different levels, we are able to use different potentials in different stages of the search which allows us to improve our potentials. Furthermore, we show how to choose and compute edge levels of a graph.

Finally, we compare our implementations of A* Search and Multilevel A* to Dijkstra's algorithm as our baseline and evaluate their performances.

## 1.3 Outline

The following chapters in this work are structured as follows.

**Chapter 2** explains background knowledge of graph theory and route planning important to this work. Here, we introduce foundations of graph theory. We will present and discuss different models of public transit networks. Furthermore, we will focus on details and characteristics of shortest paths in road and public transit networks. Dijkstra's algorithm is introduced and explained in detail. The chapter also defines notations and definitions that will be used throughout this work. Finally, we will define our problem statement of this thesis.

**Chapter 3** discusses adaptations, variations and applications of how to use Dijkstra's algorithm and A* Search for multimodal public transit routing. We discuss necessary adaptations of Dijkstra's search that are needed to solve routing in multimodal networks. We explain in detail how we implemented different variations of Dijkstra's algorithm and A* Search. The Dijkstra's algorithm and A* Search of this chapter are our baseline algorithms used in this work.

**Chapter 4** introduces our novel approach of Multilevel A*. We explain the intuition and main idea of our algorithm. We define level edges in a graph and show in detail how the query algorithm can make use of a graph with different edge levels. We also describe the algorithm that preprocesses the data for usage with our algorithm.

**Chapter 5** evaluates the performance of Multilevel A* and compares its performance with the performance of Dijkstra's search and A* Search. We also thoroughly analyze the impact of Multilevel A*. Specifically, we have a look at the search space, the impact of using multiple layers and potentials, and how the layer size influences performance. We identify cases in which Multilevel A* performs well and cases in which A* Search perform well.

**Chapter 6** concludes our findings. We summarize our learnings and observations of this work. Based on our experiences, we discuss and present possible future work.

# 2. Preliminaries

This chapter defines notations of graph theory that are needed when examining shortest paths in graphs. We focus on foundation necessary for finding shortest paths for pedestrians, shortest paths in public transit routing, and shortest paths in multimodal networks.

## 2.1 Graph Theory

A *directed graph* $G = (V, E)$ is a set of *vertices* $V$ and a set of *edges* $E \subseteq V \times V$. Vertices can also be called nodes. An edge $e = (u, v)$ is a connection between two vertices $u, v \in V$. An edge $e = (u, v)$ has a direction and goes from vertex $from(e) = u$ to vertex $to(e) = v$. A *weighted graph* is a graph with a weight function $w : E \mapsto \mathbb{N}^0$. We refer to the weight of an edge $e = (u, v)$ as $dist(u, v)$ or $len(u, v)$. A graph is *symmetric* if edges between two vertices always exist in both directions $(u, v) \in E \rightarrow (v, u) \in E$ and if their weights are the same $dist(u, v) = dist(v, u)$. Graphs that are not symmetric are *asymmetric*. The number of edges going to or going from a vertex are its in-degree $deg_{in}(v) = |\{u \mid (v, u) \in E\}|$ and its out-degree $deg_{out}(v) = |\{u \mid (u, v) \in E\}|$. A *path* $P$ is a sequence of vertices $P = (v_0, v_1, \ldots, v_k)$ that are connected by edges $(v_i, v_{i+1}) \in E$. Sometimes it is easier to specify a path by a sequence of edges $P = (e_0, e_1, \ldots, e_k)$ that connect vertices $to(e_i) = from(e_{i+1})$. The weight of a path $P = (e_0, \ldots, e_k)$ is the sum of its edge weights $dist(P) = \sum_{i=0}^{k} dist(e_i)$. A graph is *planar*, if the drawing of the graph can be embedded in a plane without any intersecting edges. A graph is connected if there exists a path between any two vertices.

Graphs can model street networks by representing roads by edges and intersections by vertices. The weight of edges relate to the distance or travel time between two nodes. Therefore, in the context of street networks, the weight function is also often called a distance function. Routes in a street network are paths in the graph and the length or distance of a route is the weight of a path. Modeling a road network in such a way is intuitive, however it does not support turn restrictions. This can be solved by adding turn tables at every intersection that indicate which turns are allowed. Another, more implicit solution to the problem is to model the graph the other way around. Road networks can be modeled by representing roads by vertices we add edges between two vertices where an intersection makes it possible to turn from one road to another. This graph only permits allowed turns. Forbidden turns can not be taken because the edges to turn from one street to another are missing.

Street network graphs are asymmetric graphs due to one-way streets or asymmetric speed limitations. Because of bridges and other restrictions such graphs are also non-planar. However, due to the nature of street networks the graphs are nearly symmetric, nearly planar and most degrees are lower than five.

## 2.2 Shortest Path Problem

A graph can be used for pedestrian routing when it represents a street network with a weight function indicating the time needed to walk along the respective road. As the time needed to walk along a road is always a positive number of seconds, the weight function in the context of street networks is a function $dist : E \mapsto \mathbb{N}^0$.

The routing problem in street networks is therefore equivalent to the shortest path problem in connected, asymmetric, weighted graphs:

**Problem:** SHORTEST PATH PROBLEM
Given: Graph $G = (V, E)$, weight function $len : E \mapsto \mathbb{N}$, source node $s \in V$ and target node $t \in V$.
Task: Find the travel time needed to walk from source $s$ to target $t$ on the shortest path.

Variations of the problem additionally require the actual shortest path $P$ from node $s$ to $t$. The process of finding the actual path is referred to as *unpacking* the path. In this thesis we will be only discussing problems that find the shortest travel time. We assume that unpacking the path after having found the fastest travel time is just a matter of simple implementation. Therefore, we will use the terms of finding the fastest travel time and finding the shortest path interchangeably in this thesis.

The most popular algorithm to find shortest paths in graphs without negative edge weights is Dijkstra's Algorithm [Dij59]. The algorithm is shown in Algorithm 2.1.

Dijkstra's algorithm incrementally finds the shortest travel time to a vertex $v$ in its search space. When the target $t$ is found in the search space it finds the distance to $t$. In this work we will refer to the distance that the search finds from source $s$ to a vertex $v$ as the *travel time of vertex v*. The most crucial data structure in Dijkstra's Algorithm is the priority queue. It must be a data structure that sorts labels by their keys. In Dijkstra's algorithm the label of the queue is the vertex id and the key is its distance. Nodes and its distances can be inserted into the data structure and we need to be able to update the distance of a node that is already in the data structure. We need to be able to extract the node with the minimum distance. Usually, a min-heap is used.

When a pedestrian walking network or a car road network is modeled as a graph, Dijkstra's algorithm solves road network routing.

## 2.3 Public Transit Network Models

Routing in a public transit network solves the question of how to get from one place to another by taking public transportation such as busses and trains. Public transit routing can also be solved by finding shortest paths in a graph as well. Modeling a public transit network as a graph is not as straightforward as modeling road networks. Distances between two stops are not constant travel times because the time needed to travel from one stop to another depends on the departure time. For example, traveling from one stop to another at a time when a bus just departed at the station takes longer than traveling when a bus is just arriving. The travel time is different because the time necessary to wait for the bus is different. Buses might also be faster or slower during or outside rush hours. The

**Algorithm 2.1:** DIJKSTRA'S ALGORITHM

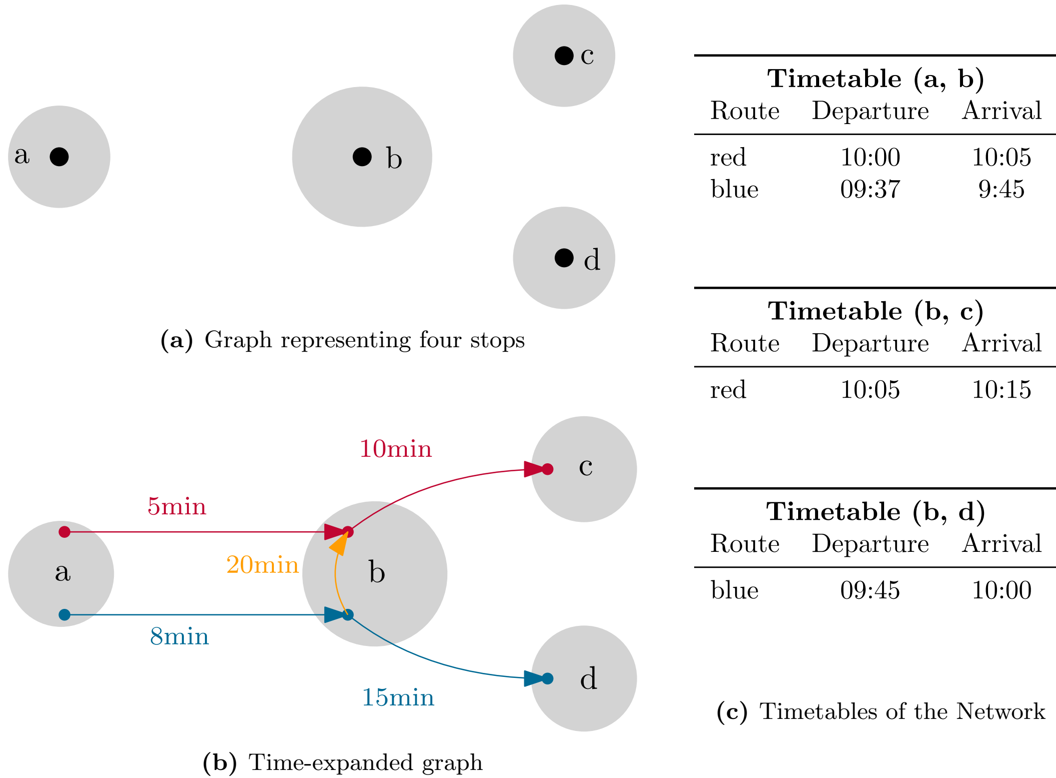
```
    Input: Graph G = (V, E), node s, node t
    Output: Distance from s to t
    Using: q: a min-queue with a vertex as label and a timestamp as key
           q.update(v, t): if label v exists in q, update its key; otherwise insert (v, t)
           len : E ↦ ℕ: returns the weight of an edge
           dist : V ↦ ℕ: used to store distances from s to a vertex
 1  forall the v ∈ V do
 2  |  dist[v] ← ∞
 3  end
 4  dist[s] ← 0;
 5  q.add(s, 0);
 6  while !q.empty() do
 7  |  u ← q.popMin();
 8  |  if u = t then
 9  |  |  return dist[t]
10  |  end
11  |  forall the edges e = (u, v) ∈ E do
12  |  |  if dist[u] + len(e) < dist[v]  then
13  |  |  |  dist[v] ← dist[u] + len(e) ;
14  |  |  |  q.update(v, dist[v])
15  |  |  end
16  |  end
17  end
```

model also needs to support transfer times at stops. Transferring from one transportation mode to another takes time. However, if a stop is reached and left by the same vehicle e.g. the same train, no transfer time is needed. The most common models for public transit networks are *time-expanded* graphs and *time-dependent* graphs. Before introducing models, we introduce some definitions in public transit routing.

**Definitions.** A *connection* is a tuple $c = (u, v, departure, arrival)$ that represents a vehicle leaving $from(c) = u$ at time *departure* and arrives at $to(c) = v$ at time *arrival*. A *route* is a sequence of connections that represents the tour of a vehicle. All connections of a route $(c_0, \dots, c_k)$ are reachable by the previous connections $to(c_i) = from(c_{i+1})$ and $arr(c_i) \leq dep(c_{i+1})$. A *line* is a set of routes that runs the same path, namely all routes of vehicles that run the same route. All routes of the same line have the *FIFO property* which is true when there is no route $R_i$ that overtakes another route $R_j$: $\forall(u, v, dep_i, arr_i) \in R_i \forall(u, v, dep_j, arr_j) \in R_j : dep_i < dep_j \Rightarrow arr_i < arr_j$.

**Time-Expanded Graphs** In a time-expanded graph, we add nodes associated to stops. For every connection, we add an additional departure node to the stop where the connection departs and we add an additional arrival node to the stop where the connection arrives. The added departure and arrival node are connected by an edge. The travel time is the travel time of the connection.

We then add edges from arrival and departure node associated to the same stop that belong to connections of the same route. Finally, we add transfer edges from arrival and departure nodes of the same stop if the arrival is earlier than the departure event and if the time between arrival and departure are longer than the minimal transfer time needed at that stop. This way we make sure that all transfers between different vehicles respect the minimal transfer time but arriving and leaving a stop with the same vehicle does not




(a) Graph representing four stops

(b) Time-expanded graph

| Timetable (a, b) | | |
|---|---|---|
| Route | Departure | Arrival |
| red | 10:00 | 10:05 |
| blue | 09:37 | 9:45 |

| Timetable (b, c) | | |
|---|---|---|
| Route | Departure | Arrival |
| red | 10:05 | 10:15 |

| Timetable (b, d) | | |
|---|---|---|
| Route | Departure | Arrival |
| blue | 09:45 | 10:00 |

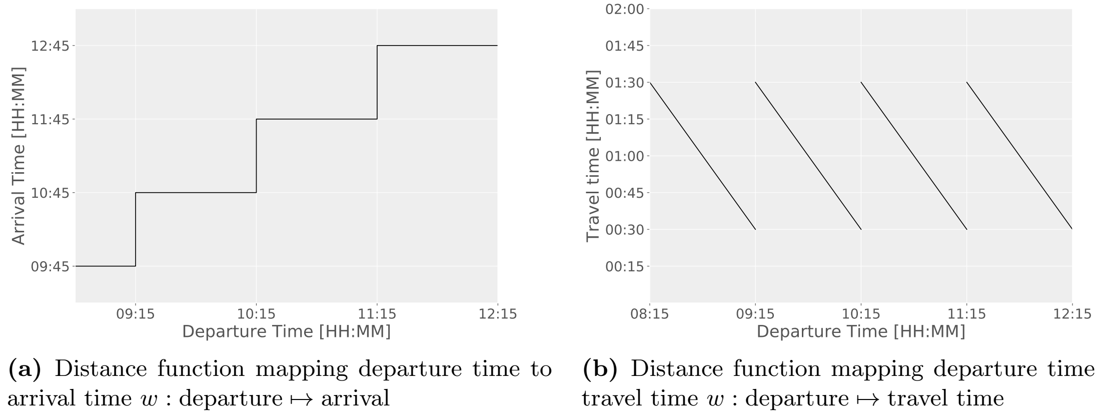
(c) Timetables of the Network

**Figure 2.1:** Figure (a) shows a graph with four stop nodes. Figure (b) shows an example that models four connections in the graph. The stop nodes have associated departure and arrival nodes and are connected by edges that represent connections. There is an additional transfer edge at stop node $b$ from the blue route to the red route. The minimal transfer time at stop $b$ is three minutes. This permits a transfer from the blue route to the red route. The travel time of the transfer edge is 20 minutes because arrival of the blue line is 9:45 and the departure of the red line at stop b is 10:05 which is 20 minutes later. The distances of the edges are scalar times.

need to regard the minimal transfer time. Figure 2.1 shows an example how departure and arrival nodes and edges are added in a time-expanded graph. We note that the resulting graph is a normal, familiar, scalar and directed graph with travel times. When solving an earliest arrival query at a certain departure time, we simply have to choose the right departure node at start point $s$ and stop our search when we find the shortest path to any arrival node associated with target $t$. As we add departure and arrival nodes for every connection, the resulting graph is very large.

**Time-Dependent Graphs** Time-dependent graphs also model stops as vertices. Vertices are connected by an edge if there exists an elementary connection between two vertices. The edge $e = (u, v)$ does not have a simple scalar weight but a time-dependent distance function that represents the set of connections that connect the two nodes. These functions $dist : \mathbb{N}^0 \mapsto \mathbb{N}^0$ map a departure time to the travel time that is needed to traverse the edge when leaving vertex $u$. The function can alternatively map a departure time at vertex $u$ to the arrival time at vertex $v$. The *minimum travel time* of an edge is the minimum travel time at an arbitrary departure time during the time range of the edge. The *maximum travel time* of an edge is the maximum travel time at a departure time during the time range. The *travel time variance* of a distance function is the difference of the minimum travel time and the maximum travel time.

Figure 2.2 shows an example of a time table and its resulting edge weight functions in two representations. One representation shows the travel time for any departure time. The




**(a)** Distance function mapping departure time to arrival time $w$ : departure $\mapsto$ arrival

**(b)** Distance function mapping departure time to travel time $w$ : departure $\mapsto$ travel time

| Station A | Station B |
|---|---|
| 09:05 | 09:35 |
| 10:05 | 10:35 |
| 11:05 | 11:35 |
| 12:05 | 12:35 |

**(c)** Time Table

**Figure 2.2:** An example of a time table of a connection between two stops and the resulting edge weight function that either maps departure time to arrival time or departure time to travel time.

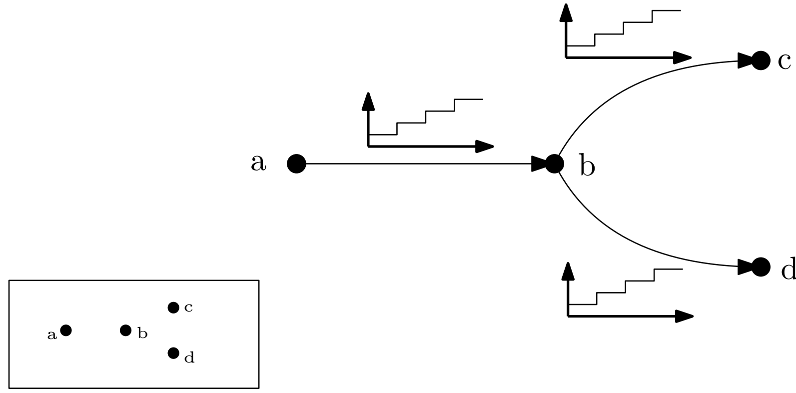
other representation shows the arrival time for any departure. All three representations can be easily converted to another. Throughout this work we will use all representations depending on which one is easier in that particular use case.

Time-dependent graphs do not add additional vertices and only connect stop nodes. Thus, the resulting graph is much smaller than time-expended graphs. However, minimal transfer times need to be considered in a non-trivial way. Figure 2.3 shows an example of a time-dependent graph.

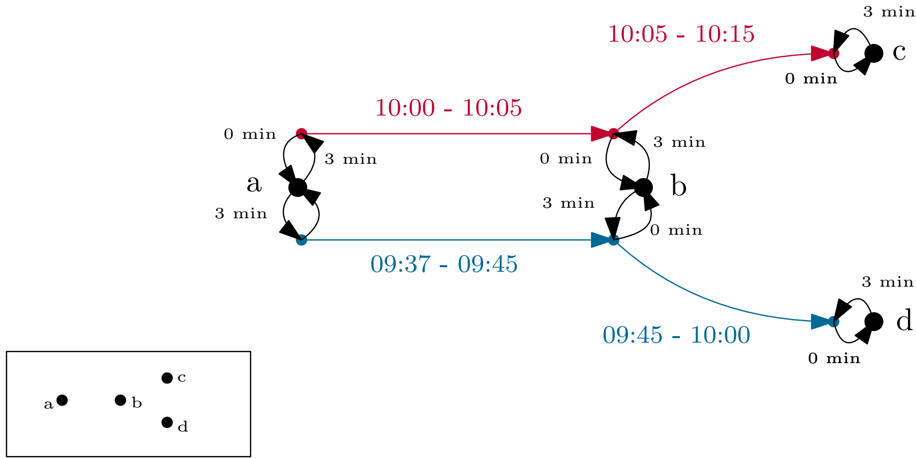
**Transfer Time Aware Time-Dependent Graphs** We extend time-dependent graphs to make them transfer time aware. Per definition, lines have the FIFO property. Therefore, no shortest path will ever transfer from a connection of a route $r_i$ to a connection of a later route $r_j$ of the same line. Transfers in shortest paths only make sense between different lines. Hence, when constructing a time-dependent graph we do not need to ensure that transfers between such connections of the same line abide by the minimal transfer time at the stop. A shortest path will not change from one of these connections to another anyways. We add additional *line nodes* to stop nodes and add edges from stops to lines with a travel time representing the time needed to board the train. We add edges from line nodes to stop nodes with travel time 0. If we transfer from a connection from a line $\ell_i$ to a connection of another line $\ell_j$ at stop $p$, we first use the edge from line node $\ell_i$ to the stop node $p$ and then the edge from $p$ to line node $\ell_j$. The edge from $p$ includes the minimal transfer time. Figure 2.3 shows an example of a time-dependent graph adjusted to be transfer time aware.

## 2.4 Public Transit Routing

In public transit routing we distinguish *earliest arrival requests* and *time profile requests*. Earliest arrival requests ask for the shortest path from vertex $s$ to vertex $t$ when start




**(a)** Time-dependent graph with connections




**(b)** Transfer Time Aware Time-dependent graph with connections

**Figure 2.3:** Figure (a) is a time-dependent graph that represents a network with four stops and adds time-dependent edges for every elementary connection. Figure (b) is a transfer time aware graph representing the same graph.

traveling at a given departure time. The result to such a query is a timestamp of the arrival at vertex $t$. Profile requests however ask for all shortest paths in a given time range. The response to such a query is a travel time function showing the travel time to target $t$ at any departure time within the given time range.

**Problem:** Time-dependent Shortest Path or Earliest Arrival Problem
Given: Graph $G = (V, E)$, distance function $dist : (E, \mathbb{N}) \mapsto \mathbb{N}$, source node $s$, target node $t$ and departure time $dep$.
Task: Find the length of the shortest path from $s$ to $t$ when departing at time $dep$ from $s$.

**Problem:** Shortest Paths Profile Search
Given: Graph $G = (V, E)$, distance function $dist : (E, \mathbb{N}^0) \mapsto \mathbb{N}^0$, source node $s$, target node $t$, time range $(from, till)$
Task: Find all shortest paths for every departure time in the range $(from, till)$ and return a travel time function with the travel time of the shortest path.

Most public transit routing algorithms solve one or both of these problems. Realistic public transit routing may optimize even more criteria. In addition to optimizing travel time, the number of transfers, ticket fees, or the reliability of a path are considered as well.

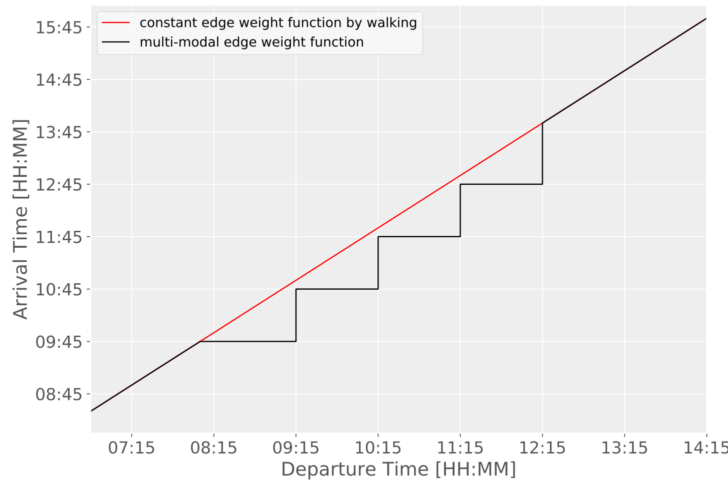



**Figure 2.4:** Example of an edge weight function in a multimodal graph with a time table as in Figure 2.2c and a walking time of 100min.

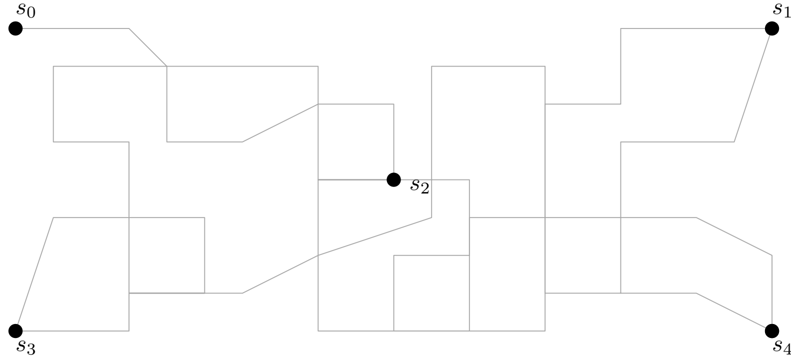
## 2.5 Multi Modal Routing

Routing in a multi modal setting means that more than one mode of transportation are available. The challenge of multi modal routing is that we have both, time-dependent data representing scheduled public transit with time tables and time-independent data for modes of transportation such as walking, riding the bike or driving a car. In the context of this work, we specifically focus on routing in a pedestrian walking network and a public transit network. Our data is therefore the merged data of a pedestrian's street network and a public transit network graph.

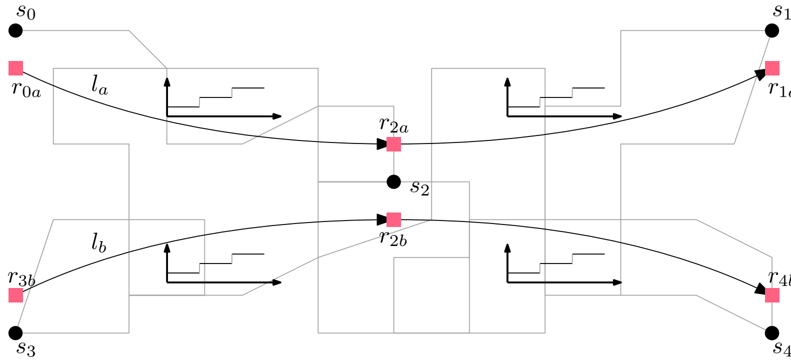
In this work we use a transfer time aware time-dependent graph that also has many walking edges with constant distances. Every distance function between two stops now has an upper bound which is the time needed to walk from one stop to another. Thus, distance functions are now not only step functions representing connections but can have linear sections too. Figure 2.4 shows an example of a time-dependent edge. The travel time functions of an edge $e = (u, v)$ is defined by a time-independent $walkingTime(e)$ and the travel times of its $k$ time-dependent connections $c_i = (u, v, dep_i(e), arr_i(e)) \quad, i \leq k$. Figure 2.5 illustrates how the final graph model for a multimodal street network is created by merging our two data sources.

This work focuses on profile routing requests in multimodal networks that regards minimal transfer times at stations.

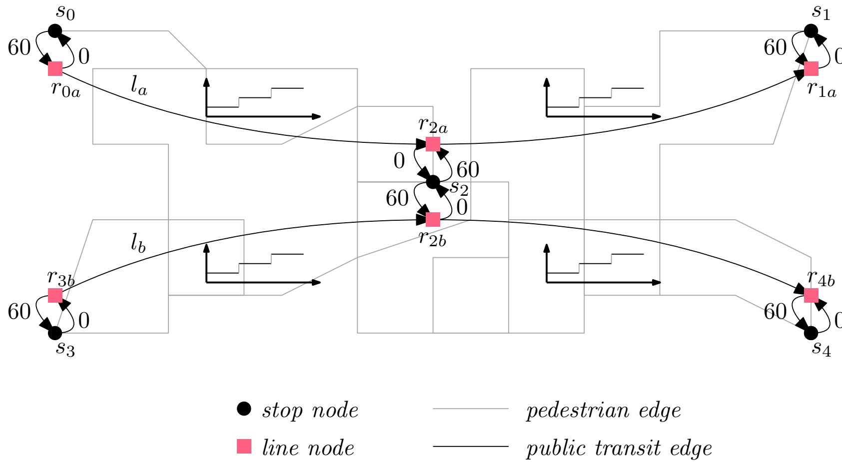
**Problem:** Multimodal Shortest Path Profile Search
Given: Transfer time aware time-dependent graph $G = (V, E)$, function $dist : (E, \mathbb{N}) \mapsto \mathbb{N}^0$, source $s$, target $t$, time range defined by $from$ and $till$
Task: Find all shortest paths from $s$ to $t$ for every departure time in the range $(from, till)$ and return a travel time function with the travel time of the shortest path.




**(a)** Time-independent pedestrian street network with highlighted vertices that are also transit stops.




**(b)** Street network that adds line nodes and adds time-dependent edges for each line.




**(c)** Multimodal street network that merges the pedestrian street network with the public transit graph by adding transfer edges between transit stops and line nodes with a boarding time of 60s.

**Figure 2.5:** Illustration of how we model a multimodal street network. We use a time-independent street network with a walking profile as the base (2.5a). We then all line nodes for each line that stops at the stop and add time-dependent edge weights between line nodes for every line (2.5b). At last we add transfer edges that connect transit stop nodes and the line nodes. Edges to line nodes include the transfer time. Edges from line nodes have zero weight.

# 3. Basic Approaches for Multi Modal Public Transit Routing

This chapter introduces and explains the baseline algorithms that we use as reference to our approach on solving multimodal shortest paths. The baseline algorithms solve the problem of earliest arrival as well as profile search in multimodal graphs.

## 3.1 Dijkstra's Algorithm

To solve time-dependent routing problem with Dijkstra's algorithm 2.1 the algorithm has to be adapted. We will show the adaptations needed in order for Dijkstra's algorithm to solve shortest path problems in multimodal networks.

### 3.1.1 Dijkstra's Algorithm in Time-Dependent Graphs

In case of the earliest arrival, adaptation is straight forward and mainly involves regarding departure time when using the travel time of an edge. Time-Dependent Dijkstra's Algorithm does not use the travel time needed to reach a vertex $v$ as its key in the queue. Instead, it uses the arrival time to a node $v$. The travel time to vertex $v$ is thus the difference of the departure time at source $s$ and the arrival time at target $t$. Consequently, the result value is not the travel time to $t$ but the arrival time at $t$ when departing at $s$ at departure time $d$. Algorithm 3.1 shows the modified time-dependent Dijkstra for earliest arrival requests.

#### Solving Profile Search with Dijsktra's Algorithm

Finding travel time profiles of shortest paths in a time-dependent graph can also be done by adjusting Dijkstra's algorithm. Instead of scalar edge weights, we now have travel time functions at edges. Therefore, we can not use Algorithm 2.1 as it is. More specifically, we need to define how to translate line 12 or line 13 of the algorithm, namely

$$\text{if}(dist[u] + len(u, v) < dist[v])$$

when checking whether we need to relax an edge and

$$dist[v] \leftarrow dist[u] + len(u, v)$$

when we update a travel time to vertex $v$ after we found a shorter path from source $s$ to the node $v$.

**Algorithm 3.1:** TIME-DEPENDENT DIJKSTRA'S ALGORITHM

**Input**: Graph $G = (V, E)$, node $s$, node $t$, departure $d$
**Output**: Distance from $s$ to $t$
**Using**: $q$: a min-queue with a vertex as label and a timestamp as key
$q.update(v, t)$: if label $v$ exists in $q$, update its key; otherwise insert $(v, t)$
$len : (e, \text{departure}) \mapsto \text{arrival}$: returns the arrival time when traveling e
$dist : v \mapsto \text{arrival}$: used to fill with arrival times from $s$ to vertex $v$

1 **forall the** $v \in V$ **do**
2 | $dist[v] \leftarrow \infty$
3 **end**
4 $dist[s] = d$;
5 $q.add(s, d)$;
6 **while** NOT $q.empty()$ **do**
7 | $u \leftarrow q.popMin()$;
8 | **if** $u = t$ **then**
9 | | **return** $dist[t]$
10 | **end**
11 | **forall the** *edges* $e = (u, v) \in E$ **do**
12 | | **if** $dist[u] + len(e, dist[u]) < dist[v]$ **then**
13 | | | $dist[v] \leftarrow dist[u] + len(e, dist[u])$ ;
14 | | | $q.update(v, dist[v])$
15 | | **end**
16 | **end**
17 **end**

Hence, we have to define

1. how we add two time-dependent travel time profiles (*linking edges*),
2. how we compare two time-dependent travel time profiles (*checking dominance*) and
3. how we update an existing travel time profile with a new one (*merging profiles*)

We will explain in detail how the adjustment for each of these operations are done.

**(1) Linking Time-dependent Travel Time Profiles.** When adding two scalar travel times, it represents the travel time needed when traversing one edge $e_0$ and another edge $e_1$ afterwards. In a time-dependent graph context, we denote this by *linking* two edges $e_0$ and $e_1$ to a linked travel time profile of the shortcut edge $e$. When traveling both edges there are four possibilities on how these edges might be traversed: both edges are walked, the first edge is walked and the second edge uses a time-dependent connection, or the first edge is traveled by a time-dependent connection and either walks or uses another connection at the second edge.

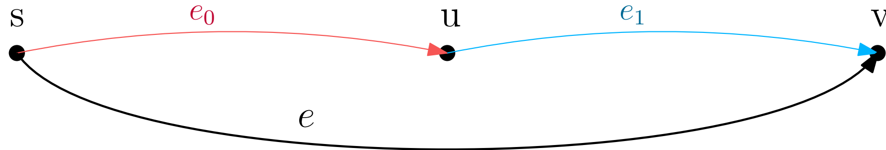
When computing the linked travel time profile, all these four possibilities have to be considered. The travel time when both edges are walked is easily computed: The walking time of the linked profile is the sum of the individual walking times. To find all connections that walk the first edge but use a time-dependent connection in the second edge, all connections $(dep_k(e_1), arr_k(e_1))$ of $e_1$ need to be regarded. The corresponding linked connection of $e$ is $(dep_k(e_1) - walkingTime(e_0), arr_k(e_1))$ because $e_0$ first needs to be walked in order to reach the connection of $e_1$. Finally, to regard all connections that traverse $e_0$ with a time-dependent connection and then traverse $e_1$, we compute for every connection $(dep_k(e_0), arr_k(e_0))$ the linked connection $(dep_k(e_0), e_1.nextArrival(arr_k(e_0)))$.

| Travel time (s, u) | |
|---|---|
| Departure s | Arrival u |
| 09:05 | 09:10 |
| 10:05 | 10:10 |
| 11:05 | 11:10 |
| Walking Time: 10min | |

**(a)** Travel time from $s$ to $u$

| Travel time (u, v) | |
|---|---|
| Departure u | Arrival v |
| 10:10 | 10:14 |
| 11:10 | 11:14 |
| 12:10 | 12:14 |
| Walking Time: 7min | |

**(b)** Travel time from $u$ to $v$




| Linked Travel Time (s, v) | | | | |
|---|---|---|---|---|
| Departure s | Arrival v | Mode | | |
| 09:05 | 09:17 | PT | + | walk |
| 10:05 | 10:14 | PT | + | PT |
| 11:05 | 11:14 | PT | + | PT |
| 12:00 | 12:14 | walk | + | PT |
| Walking Time: 17min | | | | |

**(c)** Linked traveltime from $s$ to $v$ of shortcut edge $e$

**Figure 3.1:** Table (c) is the linked travel time from $s$ to $v$ when linking $e_0 = (s, u)$ and $e_1 = (u, v)$ with the travel times given in Table (a) and Table (b). The linked travel time from $s$ to $v$ is achieved by adding the walking times to get the linked walking time. Also, for every connection in the travel time from $(s, u)$ the corresponding arrival time at $v$ is computed. Finally, every connection of edge $(u, v)$ can be reached by walking edge $(s, u)$. After sorting all resulting connections and omitting all dominated connections, linking finally results in the final travel time (c).

We do not need to distinguish between connections that either walk or use a connection at $e_1$ because we are only interested in shortest connections. Whether walking or using a time-dependent connection, one option will be faster than the other which makes the slower one superfluous. We will only keep the shortest connection. Considering all these connections results in the correct linked travel time profile. It is very likely that there will be many dominated connections when considering all three cases. Therefore, we need to sort all these connections and omit the ones that are dominated by others in the end. Figure 3.1 shows an example of a linked edge.

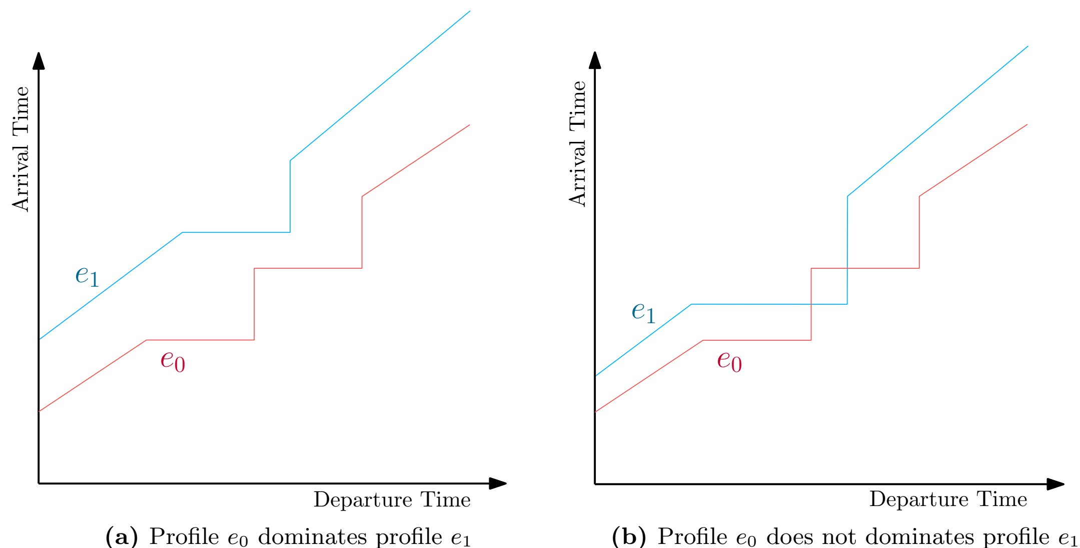
Thus, when we add two travel times in Dijkstra's original algorithm to traverse the second edge after traversing the first

$$dist[u] + len(u, v)$$

we need to translate this operation of adding the two travel times by linking the two travel time profiles

$$\textsc{link}(dist[u], len(u, v))$$

in the adjusted algorithm Profile A* to compute travel time profiles of shortest paths.




**(a)** Profile $e_0$ dominates profile $e_1$ **(b)** Profile $e_0$ does not dominates profile $e_1$

**Figure 3.2:** Showing two examples of profiles that either dominate each other or not dominate each other.

**(2) Comparing Time-dependent Travel Time Profiles.** In Dijkstra's algorithm we only relax an edge if taking the edge yields a faster travel time. When computing travel time profiles, we also only want to relax an edge if relaxing the edge will yield an improved travel time profile as well. An edge $e_1$ will not yield any faster connection than another profile of an edge $e_0$ if no travel time of $e_1$ is shorter than the travel time of $e_0$ *at any departure time*. This means that the profile of $e_0$ is *dominating* $e_1$. Therefore, we only need to relax an edge $e_1$ if the travel time profile that uses edge $e_1$ is not dominated by the travel time profile we have found until then. Whether a profile $e_0$ dominates another profile $e_1$ can be detected geometrically. If the travel time profile of $e_0$ is dominating the profile of $e_1$, the graph is always below (or on) the graph of $e_1$. If this is not the case, $e_0$ does not dominate $e_1$. When computing a dominance check, we need to check whether $walkingTime(e_0) \leq walkingTime(e_1)$ and whether for every connection $(dep(e_1)_k, arr(e_1)_k)$ of $e_1$, it is $e_0.nextArrival(dep(e_1)_k) \leq arr(e_1)_k$. Figure 3.2 shows examples of dominating and non-dominating travel time profiles. We note that with two profiles $e_0$ and $e_1$ it is possible that no profile is dominating another.

Thus, when we check whether we need to relax a path via vertex $v$ that we found by comparing the travel times

$$\text{if } (dist[u] + len(u, v) < dist[v])$$

in Dijkstra's original algorithm, we need to translate it to checking whether the travel time profile found dominates the new path

$$\text{if } (\text{NOT } dist[v].\text{DOMINATES}(\text{LINK}(dist[u], len(u, v))))$$

in the adjusted algorithm Profile A* to compute travel time profiles of shortest paths.

**(3) Merging Time-dependent Travel Time Profiles.** It is possible that there are two profiles that do not dominate each other. This means that when unpacking all shortest paths in a shortest travel time profile from $s$ to vertex $v$, there might not be a unique shortest path. Instead, the shortest profile might include multiple different unpacked shortest paths depending on the departure time. Thus, when we update a distance or an element in the queue in Dijkstra's algorithm we can not simply overwrite the old result with a new one. Rather, we need to update the distance by combining both profiles. This

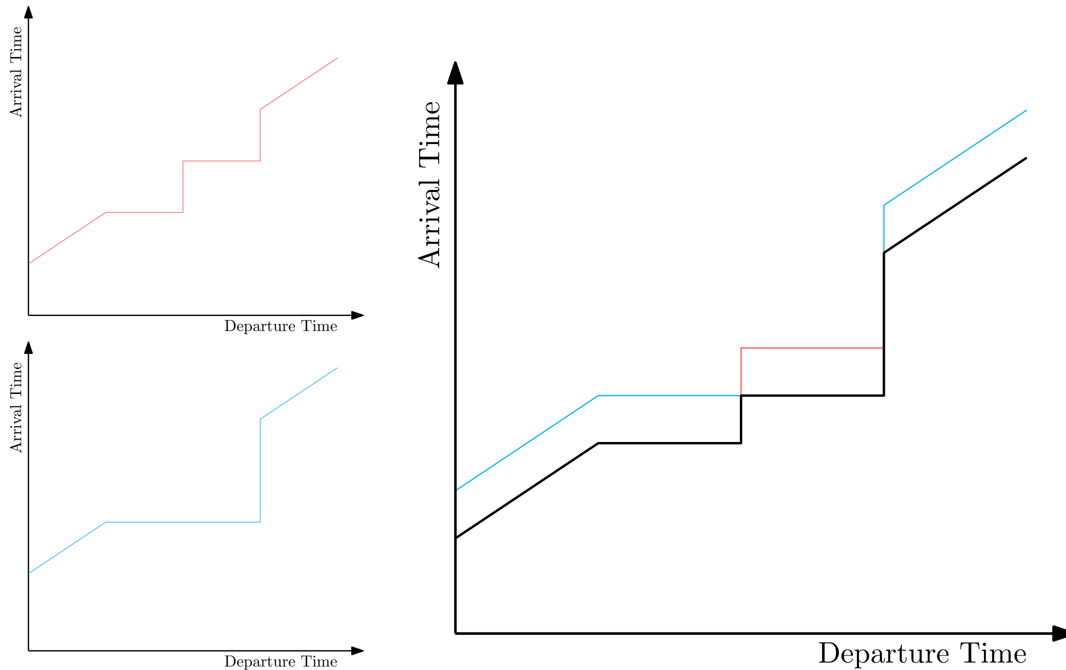



**Figure 3.3:** The travel time profile shown on the right is the merged travel time profile of the two travel time profiles on the left.

is done by *merging* two travel time profiles. We merge two profiles of edges $e_0$ and $e_1$ to a new profile of the shortcut edge $e$ by setting the walking time of $e$ to the minimum of $walkingTime(e_0)$ and $walkingTime(e_1)$. Merging all connections is done by using all connections of $e_0$ and $e_1$ in profile $e$. However, there might be many pairwise dominating connections. In order to validate all connections in $e$, the connections need to be sorted and dominated connections need to be removed. Geometrically, merging can be understood by using lower bound of both graphs. Figure 3.3 shows an example of a merged travel time profile.

Thus, when updating the travel time to a vertex $v$ when relaxing an edge $(u, v)$ yielded a faster travel time to $v$ by overwriting the travel time

$$dist[v] \leftarrow dist[u] + len(u, v)$$

in Dijkstra's original algorithm, we need to merge the travel time profile of the new path that we found with the travel time profile that we have found until then

$$dist[v].\textsc{merge}(dist[u].\textsc{link}(len(u, v)))$$

in the adjusted algorithm Profile A* to compute travel time profiles of shortest paths.

**Concluding Dijsktra's Profile Search.** When solving a profile search the desired result is a function that shows the time of arrival for any time of departure between source $s$ and target $t$. Similarly to Dijkstra's algorithm, we find the travel time profile of the shortest path from source $s$ to a vertex $v$ with the vertices within the search space. We refer to the travel time profiles from source $s$ to a vertex $v$ that we find while searching as *the travel time profile of vertex v*. One big difference between normal Dijkstra's algorithm and Dijkstra's Profile Search is the loss of the *label setting property*. An algorithm is label setting when by extracting a vertex $v$ from the queue and setting its distance, it sets the final travel time to $v$ and will not be updated again in the search. In Dijkstra's algorithm it means that when extracting a vertex from the queue, its distance to the source $s$ is the

minimal distance and will never be improved again. This allows us to stop the algorithm when we extract target $t$ as done in line 10 of Algorithm 2.1. This is not true for our Dijkstra's Profile Search algorithm. When removing target $t$ from the queue we know that we have found the minimum travel time to $t$ in the time range. We know this, because we use minimum travel time as the key in the queue and all following vertices will have a higher minimum travel time than all other vertices that were extracted before. However, by continuing our search after extracting target $t$ we can still improve the profile to $t$. Another vertex $v$ that was not extracted from the queue can improve the travel time to $t$ for a departure time that has a higher travel time than its minimum travel time but is the shortest travel time for certain departure time. Therefore, contrary to Algorithm 2.1 in line 9 we can not stop our search when extracting $t$. We can use a weaker stop criteria that is discussed later in this chapter. Algorithm 3.2 shows the resulting algorithm that is considering all these changes.

**Algorithm 3.2:** TIME-DEPENDENT PROFILE DIJKSTRA SEARCH

```
   Input: Graph G = (V, E), node s, node t
   Output: Time-dependent travel time from s to t
   Using: q: a min-queue with a vertex as label and using min-travel time as key
          dist : V ↦ Profile: used to fill with shortest profiles from s to a vertex
 1 forall the v ∈ V do
 2 |  dist[v] ← ∞
 3 end
 4 dist[s] ← 0;
 5 q.add(s, dist[s]);
 6 while NOT q.empty() do
 7 |  u ← q.popMin();
 8 |  forall the edges e = (u, v) ∈ E do
 9 |  |  suv ← LINK(dist[u], profile(u, v));
10 |  |  if NOT dist[v].DOMINATES(suv) then
11 |  |  |  dist[v].MERGE(suv);
12 |  |  |  q.update(v, minTravelTime(dist[v]))
13 |  |  end
14 |  end
15 end
16 return dist[t]
```

### 3.1.2 Timestamp Version of Dijkstra's Time-Dependent Algorithm

A travel time function is defined by the departure and arrival times of connections and the walking time. Thus, instead of relaxing and settling whole time-dependent edge profile during Dijkstra's search, it is also possible to settle individual connections. By finding all shortest paths for every possible departure timestamp from $s$ to $t$, we can gradually build the resulting travel time function. Doing this, we also restore an important property for Dijkstra's search: the *label setting property.* We do not achieve label setting property for each vertex but achieve label setting property for every connection. Thus, the timestamp version of the shortest path search is similar to a profile search in a time-expanded graph. In a time-expanded graph, each departure and arrival event has its own vertex and edges exist between connections that can be transfered to. In the timestamp version of Dijkstra's Profile Search this is implicitly done by checking the shortest path to another vertex depending on the departure time. Algorithm 3.3 shows the algorithm. It starts at source $s$ and takes all connections or walking paths possible and adds them to a queue. The queue

uses the travel time of a connection as key. Dijkstra's Timestamp Profile Search finds all shortest paths from source $s$ to target $t$ and solves Multimodal Shortest Path Profile Search.

**Algorithm 3.3:** DIJKSTRA'S TIMESTAMP PROFILE SEARCH

```
   Input: Graph G = (V, E), node s, node t
   Output: Time-dependent travel time from s to t
   Using: q: a min-queue with a connection (s, v, dep, arr, type) as label and using
          its travel time arr − dep as key
          dist : v ↦ profile: used to fill with shortest profiles from s to a vertex v
 1 forall the v ∈ V do
 2 |  dist[v] ← ∞
 3 end
 4 dist[s] ← 0;
 5 q.add(s, s, 0, 0, walking);
 6 while NOT q.empty() do
 7 |  (s, u, dep, arr, type) ← q.popMin();
 8 |  forall the edges e = (u, v) ∈ E do
 9 |  |  if type == walking then
10 |  |  |  walkingTraveltime ← arr;
11 |  |  |  forall the connections c ∈ e.connections do
12 |  |  |  |  newDeparture ← c.departure − walkingTraveltime;
13 |  |  |  |  if c.arrival < dist[v].nextArrival(newDeparture) then
14 |  |  |  |  |  dist[v].addConnection(newDeparture, c.arrival);
15 |  |  |  |  |  q.add(s, v, newDeparture, c.arrival, publicTransit);
16 |  |  |  |  end
17 |  |  |  end
18 |  |  |  if walkingTraveltime + e.walkingTime < dist[v].getWalkingTime()
   |  |  |  then
19 |  |  |  |  dist[v].setWalkingTime(walkingTraveltime + e.walkingTime);
20 |  |  |  |  q.add(s, v, 0, walkingTraveltime + e.walkingTime, walking);
21 |  |  |  end
22 |  |  else if type == publicTransit then
23 |  |  |  newArrival ← e.nextArrival(arr);
24 |  |  |  if newArrival < dist[v].nextArrival(dep) then
25 |  |  |  |  dist[v].addConnection(dep, newArrival);
26 |  |  |  |  q.add(s, v, dep, newArrival, publicTransit);
27 |  |  |  end
28 |  |  end
29 |  end
30 end
31 return dist[t]
```

### 3.1.3 Stop Criteria and Prunings for Speed-up in Dijkstra's Algorithm

As mentioned, Time-Dependent Dijkstra's Profile Search algorithm is not label-setting anymore. This means we lose the guarantee to have found the shortest path to vertex $t$ when we settle it. Instead, we have to settle all vertices pushed to the queue. However, losing the label-setting property does not mean we have to settle all vertices in the graph. There is still a weaker criteria to stop the search. *Pruning* a vertex means that we can skip processing a vertex that we just popped from queue and do not need to relax its edges.

A *stop criteria* defines a criteria which is true when we can prove that we have found the shortest path to our target $t$. We note that without a stop criteria, Dijkstra's Profile Search will not stop until it has found the shortest path from source $s$ to any other vertex $v \in V$. Therefore, a stop criteria as well as pruning are crucial to improve performance.

**Target Pruning.** When popping a vertex $u$ from the queue, we may check whether it is dominated by the shortest paths we already found to target $t$. If it is dominated, we may skip it and do not need to relax its outgoing edges. Thus, we add a target dominance check to Dijkstra's Profile Search after popping a vertex. This is denoted by *target pruning*. We add target pruning as follows.

```
[...]
u ← q.popMin();
if dist[t].dominates(dist[u]) then
    continue;
[...]
```

In the timestamp version of Dijkstra's algorithm, target pruning is done correspondingly as follows.

```
[...]
(s, u, dep, arr, type) ← q.popMin();
if type == walking then
    if arr > dist[t].getWalkingTime() then
        continue;
else if type == publicTransit then
    if arr > dist[t].nextArrival(dep) then
        continue;
[...]
```

**Stop Criteria.** The key of Dijkstra's queue is the minimum travel time to a vertex. The minimum travel time to a vertex is continuously growing during the search. Even if a popped vertex is dominated by target $t$, it is no indicator whether vertices popped at a later time may be part of the shortest path. However, there is a criteria that determines whether we can stop our search or not. Namely, if the minimum travel time of the vertex popped is bigger than the maximum travel time found to target $t$. When relaxing an edge from $u$, it may be part of the shortest path to $t$. When popping a vertex $v$, the travel time of every shortest path from source $s$ via $v$ to target $t$ will be equal or longer than the minimum travel time from source $s$ to vertex $v$. When we pop vertex $v$ and its minimum travel time is bigger than the maximum travel time to target $t$, we know that vertex $v$ and all vertices left in the queue can not improve the travel time profile from $s$ to $t$. We can stop the search at this point and add a stop criteria to our Dijkstra's algorithms as follows.

```
[...]
u ← q.popMin();
if dist[u].getMinimumTraveltime() > dist[t].getMaximumTraveltime then
    break;
[...]
```

In the timestamp version of Dijkstra's algorithm, the stop criteria is as follows.

$[\dots]$

$(s, u, dep, arr, type) \leftarrow q.popMin();$
**if** $(arr - dep) > dist[t].getMaximumTraveltime$ **then**
$\quad break;$

$[\dots]$

**Adjusting the Key.** The key of vertex $u$ used in the queue is its minimum travel time to $key(u) = dist[u].getMinTravelTime()$. We lost the label setting property, so we might relax an edge to a vertex $v$ and push $v$ to the queue multiple times because we improved a shortest path to $v$. When we relax an edge to a vertex $v$ and want to queue $v$ a second time to the queue, the travel time that was improved is at least equal or bigger than the minimum travel time when vertex $v$ was queued at an earlier time. In this case, we can find a more suitable key for vertex $v$. Instead of

$$key(v) = dist[v].getMinTravelTime()$$

we use the minimum travel time that was improved when we relaxed an edge to the vertex $v$

$$key(v) = minimumTravelTimeImproved(v).$$

We note that we only use the improved key if vertex $v$ has been in the queue before and there is no label of vertex $v$ in the queue anymore. If label $v$ is already in the queue, we do not update its key. By doing this, we make sure that our key labels that we extract from the heap is increasing over time and not aggressively jump back to vertices with a low minimum travel time.

## 3.2 A* Search

Dijkstra's algorithm uses a circular search space around source $s$ to search for shortest paths. This leads to a large search space with many unnecessary vertices. A* Search attempts to modifies the search space by making it more goal-directed, reducing the number of edges needed to relax to find $t$ from $s$.

A* Search is essentially Dijkstra's algorithm with a modification of the order in which vertices are settled. In Dijkstra's algorithm the order of vertices is determined by the distance of the vertices in the queue to the source node $s$ (see line 4 in Algorithm 2.1). In A* Search algorithm, a potential $\pi : V \to \mathbb{R}$ is assigned to every vertex $v$. The potential $\pi(v)$ is an estimation of the distance from $v$ to target $t$. The key of every vertex that defines the order in which a vertex $v$ is extracted from the queue is $dist(s, v) + \pi(v)$. Vertices that we estimate to be very far away from target $t$ have a bigger key than vertices that we estimate to be closer to target $t$. Vertices that we assume to be closer to $t$ and are more relevant for the shortest path have smaller keys.

The potential of a vertex determines the order in which a vertex is settled. Basic Dijkstra's algorithm settles vertices in the search space in a circle, increasing the radius until target $t$ is found. Dijkstra's algorithm can be seen as a specialized version of A* with the potential of all vertices being zero. With more appropriate potentials, we can influence the search space of A* Search and make it goal oriented. Figure 3.4 is a schematic showing the search space in Dijkstra's algorithm compared to A* Search.

**Algorithm 3.4:** A* Search

**Input**: Graph $G = (V, E)$, node $s$, node $t$, potentials $\pi$
**Output**: Distance from $s$ to $t$
**Using**: $q$: a min-queue with a vertex as label and a timestamp as key
$len : (E, \text{Departure}) \mapsto$ Arrival: returns the length of e
$dist : V \mapsto$ Arrival: used to fill with arrival times from $s$ to a vertex

```
1  dist[s] = 0;
2  q.add(s, 0);
3  while !q.empty() do
4      u ← q.popMin();
5      if u = t then
6          return dist[t]
7      end
8      forall the edges e = (u, v) ∈ E do
9          if dist[u] + len(e) < dist[v] then
10             dist[v] ← dist[u] + len(e);
11             q.update(v, dist[v] + π(v))
12         end
13     end
14 end
```

The choice of the potential used in A* Search is crucial to the query time of A* Search algorithm. In case of the label setting algorithm such as the time-dependent earliest arrival query in Algorithm 3.4 certain requirements have to hold true. This is because we modify the search space but we still stop our search when we settle vertex $t$. This can break optimality and correctness of the algorithm. However, A* Search is still optimal if every potential is admissible which means that it has to underestimate the distance $\forall v \in V : \pi(v) \leq dist(v, t)$. Furthermore, the resulting key values $key((u, v)) = dist(u, v) + \pi(v)$ must not be negative in order for the algorithm to work. Potentials that ensure positive key values are *feasible*.

If the potential of every vertex $v$ is $\pi(v) = 0$ the A* Search is equivalent to Dijkstra's algorithm. If $\pi(v) = dist(v, t)$, A* Search will only settle nodes on the shortest path between $s$ and $t$. Such potentials are called $perfect potentials$. The potential $\pi(s)$ of

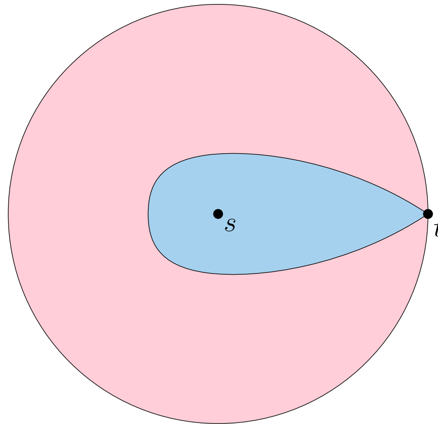
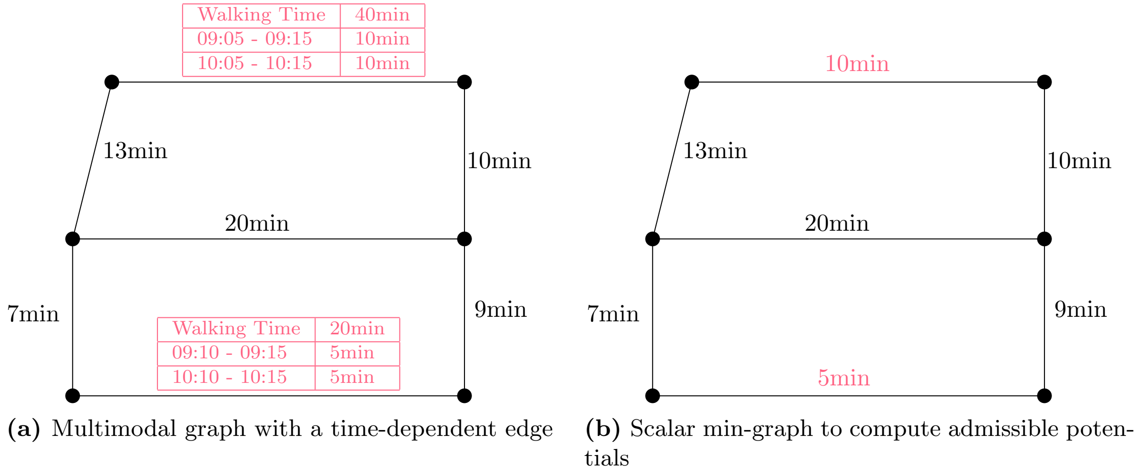



**Figure 3.4:** Schematic showing the circular search space of Dijkstra's algorithm (red) compared to a goal oriented search space of A* Search.




**(a)** Multimodal graph with a time-dependent edge

**(b)** Scalar min-graph to compute admissible potentials

**Figure 3.5:** Figure (a) shows a multimodal graph $G$ with time-dependent edges. Figure (b) shows a scalar min-graph $G'$ of graph $G$ in Figure (a). The scalar min-graph can be used to compute admissible potentials for $G$ with a backwards Dijkstra search.

source $s$ is the estimation of the travel time that we search. The bigger and closer $\pi(v)$ to $dist(v, t)$ while still preserving admissibility, the bigger is the speed-up of A* Search.

### 3.2.1 Time-Dependent Profile A* Search for Mutlimodal Routing

We can make use A* Search for routing in a multimodal public transit network $G = (V, E)$ as well. We define a scalar min-graph $G' = (V', E')$ that uses all minimum travel times of each time-dependent edge as the distance of the edge in $G'$ with the same vertices $V' = V$ and edges $E' = E$. The edge weights $dist(e') = minimumTravelTime(e), e \in E$ are the minimum travel times in the original graph. We refer to the minimum travel time of a travel time profile from a nodes $u$ to a node $v$ as $dist'(u, v)$. Figure 3.5 shows a multimodal graph and its corresponding scalar min-graph.

To achieve an admissible potential for graph $G$, we run a backward Dijkstra on the min-graph $G'$ from target $t$ to all other vertices in the graph. The potential $\pi(v)$ of a vertex $v$ is the distances from $t$ to another vertex $v$. This way it is guaranteed that $\pi(v)$ underestimates the $dist(v, t)$ in $G$ because it is the shortest path in a graph that only uses minimal travel times. We refer to this potentials as the *min-traveltime potential*. We note that these min-traveltime potentials are perfect potentials as well. If a potential were bigger than its min-traveltime potential, we can not ensure that the potential is admissible and not overestimating the real travel time without taking a closer look at all connections.

Algorithm 3.5 is a time-dependent A* Search algorithm that solves earliest arrival requests by using min-traveltime potentials. It is easy to see that the running time for computing the potentials $\pi$ for all vertices clearly dominates Time-Dependent Earliest Arrival A* Algorithm 3.5. Computing the perfect potentials requires a backwards search from target $t$ to all vertices in the graph. Thus, the A* Search approach proposed here is more interesting in the context of searching travel time profiles. For easier understanding of the algorithms, mechanisms and analysis of A* Search, we still take a look at how to compute time-dependent earliest arrival requests with perfect potentials.

Algorithm 3.6 shows an A* Search algorithm for travel time profile requests. The algorithm is very similar to our adjusted Profile Dijkstra Search 3.2 but uses potentials and the minimum travel time as keys. Profile computations are not label-setting. Limiting the search space can therefore have a bigger impact than just the difference of the different

**Algorithm 3.5:** TIME-DEPENDENT EARLIEST ARRIVAL A* SEARCH

```
   Input: Graph G = (V, E), node s, node t, departure d
   Output: Distance from s to t
   Using: q: a min-queue with a vertex as label and a timestamp as key
          len : (e, departure) ↦ arrival: returns the arrival time when traveling e
          dist : v ↦ arrival: used to fill with arrival times from s to vertex v
 1 π = getDistances(backwardsDijkstra(t));
 2 forall the v ∈ V do
 3 │ dist[v] ← ∞
 4 end
 5 dist[s] = d;
 6 q.add(s, d);
 7 while !q.empty() do
 8 │ u ← q.popMin();
 9 │ if u = t then
10 │ │ return dist[t]
11 │ end
12 │ forall the edges e = (u, v) ∈ E do
13 │ │ if dist[u] + len(e, dist[u]) < dist[v] then
14 │ │ │ dist[v] ← dist[u] + len(e, dist[u]);
15 │ │ │ q.update(v, dist[v] + π(v))
16 │ │ end
17 │ end
18 end
```

**Algorithm 3.6:** TIME-DEPENDENT PROFILE A* SEARCH

```
   Input: Graph G = (V, E), node s, node t
   Output: Time-dependent travel time from s to t
   Using: q: a min-queue with a vertex as label and using min-travel time as key
          dist : v ↦ profiles: used to fill with travel time profiles from s to v
 1 π = getDistances(backwardsDijkstra(t));
 2 forall the v ∈ V do
 3 │ dist[v] ← ∞
 4 end
 5 dist[s] = 0;
 6 q.add(s, dist[s]);
 7 while !q.empty() do
 8 │ u ← q.popMin();
 9 │ forall the edges e = (u, v) ∈ E do
10 │ │ suv ← LINK(dist[u], profile(u, v));
11 │ │ if NOT dist[v].DOMINATES(suv) then
12 │ │ │ dist[v].MERGE(suv);
13 │ │ │ q.update(v, dist[v] + π(v))
14 │ │ end
15 │ end
16 end
17 return dist[t]
```

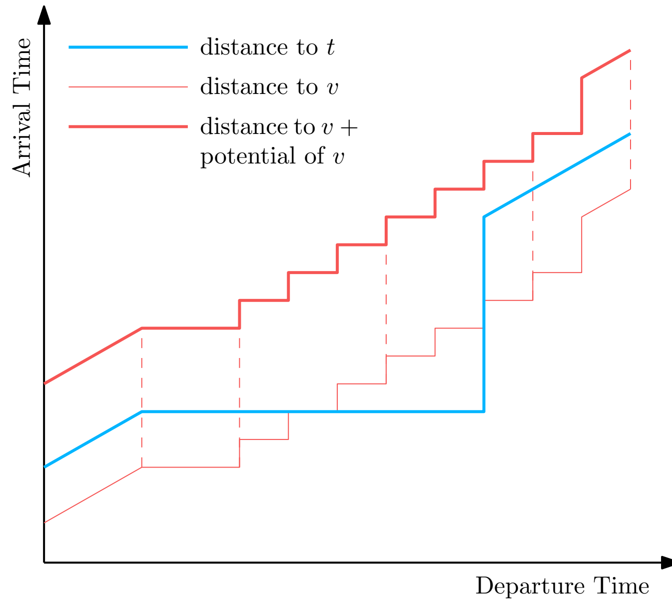



**Figure 3.6:** Figure showing the travel time profile from a source $s$ to a vertex $v$ and a travel time profile from $s$ to a target $t$. The travel time profile of $t$ does not dominate the distance to $v$ but it dominates the travel time profile of $v$ when adding its potential. Thus, $v$ can be pruned.

search space sizes. Moreover, the computations of LINK, MERGE, and DOMINATES are expensive. Computing and using the potentials $\pi$ can have a big speed-up and may be worth the computation.

### 3.2.2 Stop Criteria and Prunings for Speed-up in Profile A* Search

All stop criteria and pruning techniques discussed in 3.1 are applicable in Profile A* Search as well. Using the potentials accessible in A* Search, we can apply target pruning even better. If the travel time profile of a vertex $v$ plus its potential is dominated by the travel time profile of target $t$, we can prune the vertex. Figure 3.6 shows the travel time of a vertex $v$ and target $t$. The distance to $t$ does not dominate the distance to $v$ but the distance to $t$ dominates the distance to $v$ when adding its potential. Knowing the potential of $v$, we know that the best possible distance to $t$ via $v$ is the resulting distance of $dist(v) + potential(v)$ ad any departure time. We can therefore prune vertex $v$ in such a case. Target pruning in A* thus works as follows.

[...]
$u \leftarrow q.popMin()$;
**if** $dist[t].dominates(dist[u] + potential[u])$ **then**
  $continue$;
[...]

In the same way, the potential can also be used for the stop criteria. When extracting vertex $v$ and the sum of its minimum travel time and its potential is bigger than the

maximum travel time to target $t$, we can stop. Therefore, we modify the stop criteria in Profile A* as follows.

[...]

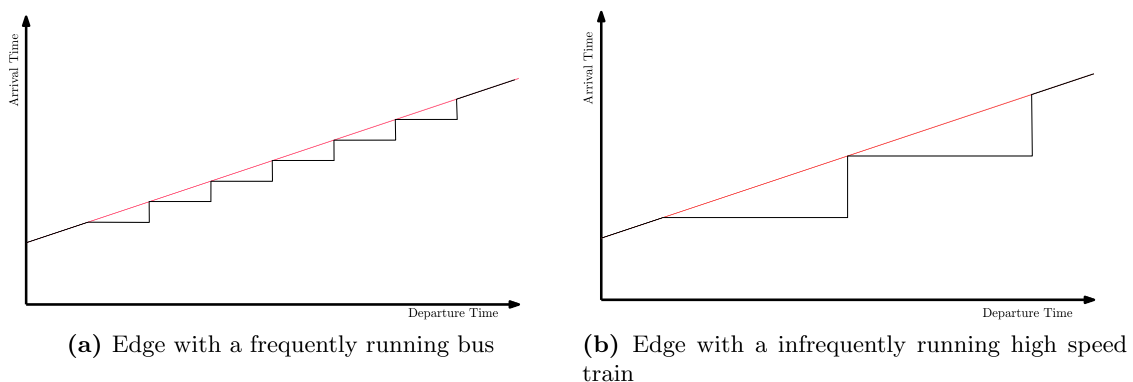
$u \leftarrow q.popMin();$
**if** $dist[u].getMinimumTraveltime() + potential(u) >$
$dist[t].getMaximumTraveltime()$ **then**
  $break;$

[...]

# 4. Multilevel A*

A* Search is an effective method to modify and reduce the search space. With a backwards Dijkstra's search as explained in the previous Section 3.2 we can even find perfect potentials in time-independent graphs. In a time-independent, scalar graph using perfect potentials means that *only* vertices that are part of the shortest path are settled. No other except those

1. Time-Dependent Profile A* Search is not label setting. Thus, we can not stop A* Search as soon as we find and settle target $t$.
2. When unpacking the actual paths, all shortest paths between two nodes in a time range might include multiple different paths.
3. We use perfect potentials in multimodal networks by using the minimum travel time of the edges. But the estimation computed by minimum travel times might not be a suitable estimation for most of the departure times. For most departure times the real travel time of the edge might be much longer. This especially affects high-speed trains. In cases of high-speed trains, it is very likely that such trains do not run often and the variance of travel times is high. vertices needed will be settled. As




**(a)** Edge with a frequently running bus

**(b)** Edge with a infrequently running high speed train

**Figure 4.1:** Figure (a) shows the travel time throughout a day of a connection with a frequently running bus resulting in a travel time with a low variance. Figure (b) on the other hand shows the travel time of a connection with a infrequently running high speed train resulting in a travel time with a high variance depending whether the train was just missed or not.

seen in the previous Chapter 3.2, perfect potentials in time-dependent graphs do not provide the same property. This is due to multiple reasons.

This shows that even perfect potentials can not reduce the search space enormously in multimodal networks as it can in scalar graphs. Finding admissible potentials usable for profile queries in time-dependent graphs is a challenge. In the following we will further investigate the importance of potentials in A* Search and its impact on the search space. We do this by observing how perfect potentials work in graphs with different travel time variances.

**Potentials in a Time-Dependent Graph with Low Variance of Travel Times.** Given is a time-dependent graph $G$ that only has edges with a low travel time variance of $var$. Let $P$ be the shortest path between $s$ and $t$ in the scalar min-graph $G'$. Using the same path in $G$, the real minimum travel time of that path $P$ is at best the minimum travel time of path $P$ $minTT(P) \geq dist'(s,t) \geq \pi(s)$ and the maximum travel time from $s$ to $t$ is at most $\pi(s) + (|P| \cdot var)$. When using Time-Dependent A* Profile Search Algorithm 3.6, the algorithm will find path $P$ very fast. The search space when searching the shortest path from $s$ to $t$ includes all vertices $\{v \mid minTT(s,v) + \pi(v) < maxTT(s,t)\}$. In graph $G$ this means that after finding path $P$, it will relax and extract all nodes $v$ with $minTT(s,v) + \pi(v) \leq maxTT(s,t) \leq \pi(s) + (|P| \cdot var)$ to search for possible shorter paths. Only after having searched all these nodes, it will stop searching.

This shows that the shorter $|P|$ and the smaller $var$, the smaller the search space. We conclude that in graphs with low variance, Time-Dependent A* Profile Search Algorithm 3.6 achieves a big speed-up.

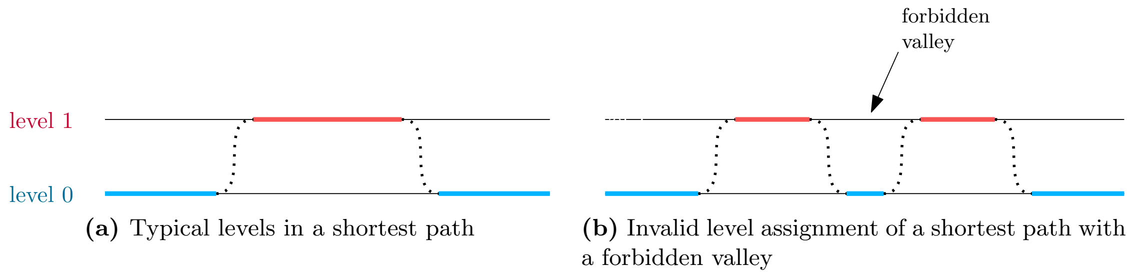
**Potentials in a Time-Dependent Graph with High Variance of Travel Times.** However, if $var$ is big, the algorithm will need to keep searching a lot of vertices until it can finally stop. This means that in graphs with a big variance of travel time, Profile A* will keep searching many vertices even if it has found the shortest paths to target $t$ already. Profile A* has to keep on searching and increasing its search space because we need to fulfill the stop criteria in order to prove the correctness of the travel time that we found.

Exploiting the fact that A* Search works well on graphs with low variance, we propose a multi-layered approach for the A* algorithm for public transit routing that tries to reduce the problem of big variance of the travel times of an edge. This approach relies on the assumption that connections of high speed trains are part of many shortest paths and that most shortest paths start with walking and possible busses and trams, uses a high speed train and again ends with busses, trams or walking.

With these observations in mind we propose a novel algorithm assigning levels to edges. The main idea is to identify a set of edges $E_1$ with the highest variance of travel time. In a subgraph $G_0 = (V, E \setminus E_1)$ that does not include edges with high travel time variance, using the minimum travel time of edges should yield decent potentials for routing in a graph. We introduce a concept of different levels that use different potentials.

## 4.1 Introducing Edge Levels

Our proposed algorithm works with multiple layers of arbitrary number. For simplicity's sake we will now limit the algorithm to be a bi-level A* Search. First, we introduce three different levels of edges: Edges of level 0, edges of level 1 and edges of level *. We set the level of edges with a high variance of travel time to level 1, edges with a low variance of travel time have level 0. We then need edges of level * to fulfill an important requirement for our algorithm: When ignoring edges of level *, every shortest path between two nodes




**(a)** Typical levels in a shortest path

**(b)** Invalid level assignment of a shortest path with a forbidden valley

**Figure 4.2:** Figure (a) shows a typical sequence of levels of shortest paths. Figure (b) shows an invalid level assignment of a shortest path with a forbidden valley which is a sequence of edges with level 1, level 0 and again of level 1. The path in Figure (b) violates the *non-valley requirement*.

starts with a sequence of edges on a certain level, might then raise and continue with a sequence of edges with level 1 and might end with a sequence of edges with level 0. It is important that no shortest path has a sequence of edges with level 1, then edges with level 0 and again with edges of level 1. We will define this as the *non-valley requirement* of all shortest paths.

### Notations and Definitions

- We define a level assignment $level : E \mapsto \mathbb{N} \cup \{*\}$ which assigns a level to every edge.
- The edges of a graph are divided into different levels that contain all edges with the same level $L_k = \{e \mid level(e) = k\}$.
- We denote the level of a path $P$ as $level(P) = k$ if all edges in the path have the same level $k$ or level $*$.
- A level assignment for graph $G$ is valid if the *non-valley requirement* is fulfilled for all shortest paths $P$ in $G$:
  - $P = \underline{P_s} + \overline{P} + \underline{P_t}$
  - $level(\underline{P_s}) = 0$
  - $level(\overline{P}) = 1$
  - $level(\underline{P_t}) = 0$

With these definitions and a level assignment as well as an admissible potential of all vertices, we can then use a *Multilevel A* Search* (MLA*) to find shortest paths in a multimodal graph.

## 4.2 A* Search on Levels

Given a valid level assignment, we know that all shortest paths look similar to Fig. 4.2a, starting at level 0, continuing on level 1 and ending on level 0. When we traverse a shortest path, we can therefore split the traversal of the shortest path into three different states:

1. *grey path*: when all edges are of level 0
2. *red path*: when currently traversing edges with level 1
3. *blue path*: when we have already traverse edges with level 1 but currently traverse edges with level 0

Using these states and our previous findings we can now define valid admissible potentials as follows.

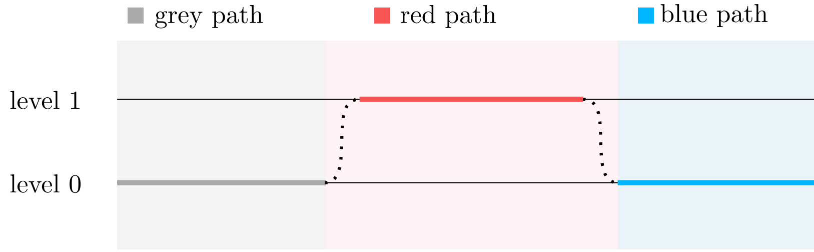



**Figure 4.3:** Different states of a shortest path

### 4.2.1 Valid Admissible Potentials in a Multimodal Multilevel Graph

As seen in Section 3.2, we find perfect potentials in graph $G$ by using a scalar min-graph. Per intuition, using the same method for potentials on a graph $G_0$ that only has edges with low variance of travel times should yield good potentials that do not increase the search space too much. In the following we will define $G_0$ to be the graph that only consists of edges with level 0 and level $*$. We then define two types of potentials in Multilevel A*:

- $\pi_{red}$ is the perfect potential on $G$ to a target $t$
- $\pi_{blue}$: is the perfect potentials on $G_0$ to a target $t$

We note that $\forall v \in V : \pi_{red}(v) \leq \pi_{blue}(v)$.

The potential $\pi_{blue}$ is obviously not always a correct admissible potential of graph $G$. However, using a valid level assignment and the fact that all shortest paths have to follow the *non-valley requirement*, we know that a shortest path, that has used edges of level 1 and then edges of level 0 afterwards will not contain edges of level 1 again. All following edges of the shortest path are in $G_0$. Therefore, we can skip relaxing all edges of level 0 if the current state is a blue path. Moreover, the potentials $\pi_{blue}$ are an admissible potential in graph $G_0$. Hence, we can use potentials of $\pi_{blue}$ during the blue state.

### 4.2.2 Designing Multilevel A* Search

The main idea to use Multilevel A* Search (MLA*) is that we can reduce the search space in two ways by using potential $\pi_{blue}$. If a target $t$ is far away and we need to use high-speed trains to get there, we do not want to visit vertices in cities where our high-speed train stops on the way. Potential $\pi_{blue}$ supports this behaviour. When we are searching for shortest paths and are currently using high-speed public transit on a red path and are still far away from the target $t$, we will use the bigger potentials $\pi_{blue}$ when we leave the path with level 1. As the potential is big, it will will keep us from visiting nodes with edges of level 0 and we will arrive at $t$ sooner. If a shortest path may require that we leave the path with level 1, the potential $\pi_{blue}$ will not be so high anymore. When we are close to the target $t$ in our search we do not want to take fast public transit to move further away from our target. The non-valley requirement can support this when we are in the blue state as we do not take any edges with level 1 anymore.

Combining all our findings, we can now define an A* Search using different levels and potentials as follows. We will start similar to Profile A* Search at source $s$ on a grey path and relax all outgoing edges. When relaxing the edge from a vertex $u$ to a vertex $v$, we will now need to regard the type of path how we reached vertex $u$ and how we found $v$. We know the type of the path how we reached vertex $u$ because we store it as part of the label in the queue. The path type how we reach vertex $v$ can be then deduced by the edge level of edge $(u, v)$ that is about to be relaxed. At this point, we can make use of our two different potentials and the non-valley requirement depending on the two path types as follows.

| **Relaxing an edge** $(u, v)$ | | | |
|---|---|---|---|
| Path Type to $u$ | Level of $(u, v)$ | Resulting Path Type to $v$ | Remarks |
| grey | $\{0, *\}$ | grey | use $\pi_{red}$ |
| grey | 1 | red | use $\pi_{red}$ |
| red | $\{1, *\}$ | red | use $\pi_{red}$ |
| red | 0 | blue | use $\pi_{blue}$ |
| blue | $\{0, *\}$ | blue | use $\pi_{blue}$ |
| blue | 1 | — | skip relaxing this edge |

Depending on the current state of the shortest path we use $\pi_{red}(v)$ or $\pi_{blue}(v)$ as potential for $v$. When adding a vertex to the queue after relaxing edge $(u, v)$, we add vertex $v$ with the corresponding path type. We make an exception for target $t$. If we relax an edge to $t$, we do not need to distinguish between the different path types to get to $t$ because we need to find all shortest paths to $t$ regardless of the path type. Algorithm 4.1 shows the complete algorithm. We note that the unique label in the queue is not only the vertex but the combination of vertex and the path state. In a bi-level version, a vertex $v$ can be queued in the queue three times with a different state. A shortest travel time profile can also include shortest paths via vertex $v$ in different states. Thus, it i possible that we reduce our search space using multiple levels, but we potentially increase the number of times we settle the same vertex during the A* Search run. However, we hope that thanks to potential $\pi_{blue}$, vertices that are not part of the shortest paths will be pruned and that we can skip many edges while being in the blue state.

### 4.2.3 Correctness of Multilevel A*

We change the label of the queue from a vertex $v$ to a tuple $(v, \mathit{pathType})$ with $v \in V$ and $\mathit{pathType} \in \{\text{grey}, \text{red}, \text{blue}\}$. This can also be seen as triplicating each vertex. Analyzing how we shift from one path type to another, we notice:

0. Per default, we start the search with a grey path at source $s$.
1. We ignore edges with level 1 when in state blue.
2. We shift from grey to red when an edge has level 1.
3. We shift from red to blue when an edge has level 0.

Thus, we can think of Multilevel A* as a search in a graph $G*$, which triplicates $G$ to three levels, one level for the grey, red, and blue state correspondingly. We do not need to triplicate vertex $s$ and its outgoing edges because we always start our search in grey. Thus, we can omit $(s, red)$ and $(s, blue)$ as well as how to get there. We do not need to triplicate vertex $t$ because we look at *any* shortest path to target $t$, we do not distinguish between paths to $(t, grey), or$(t, red), or $(t, blue)$. We then modify the graph as follows.

1. We remove all edges of level 1 in the blue level because we ignore those edges during the blue state anyways. We are allowed to do this thanks to the *non-valley* requirement which ensures us that there exists no shortest path with a sequence of level 1 edges, followed by level 0 edges and again level 1 edges.
2. We redirect all edges with level 1 from the grey level to the red level because whenever we traverse a level 1 edge in a grey state, we will move to the red state.
3. We redirect all edges with level 0 from the red level to the blue level because whenever we traverse a level 0 edge in a red state, we will move to the blue state.

**Algorithm 4.1:** MULTILEVEL A* PROFILE SEARCH

```
    Input: Graph G = (V, E), node s, node t
    Output: Time-dependent travel time from s to t
    Using: q: a min-queue using vertices and path types as labels, the travel time
              profile as element, and a given min-time as key
           level : e ↦ {0, 1, ∗}: returns the level of an edge e
           dist : v ↦ profile: used to fill with shortest profiles from s to a vertex
           pathType: path types can be grey, red or blue
 1  π_red = getDistances(backwardsDijkstra(G, t));
 2  π_blue = getDistances(backwardsDijkstra(G_0, t));
 3  forall the v ∈ V do
 4  |   dist[v] ← ∞
 5  end
 6  dist[s] = 0;
 7  q.add(s, grey, dist[s]);
 8  while !q.empty() do
 9  |   u, pathType ← q.popMin();
10  |   newPathType ← pathType;
11  |   forall the edges e = (u, v) ∈ E do
12  |   |   switch pathType do
13  |   |   |   case grey
14  |   |   |   |   potential ← π_red(v);
15  |   |   |   |   if level(e) == 1 then
16  |   |   |   |   |   newPathType ← red;
17  |   |   |   |   end
18  |   |   |   case red
19  |   |   |   |   if level(e) != 0 then
20  |   |   |   |   |   potential ← π_red(v);
21  |   |   |   |   else if level(e) == 1 then
22  |   |   |   |   |   potential ← π_blue(v);
23  |   |   |   |   |   newPathType ← blue;
24  |   |   |   |   end
25  |   |   |   case blue
26  |   |   |   |   if level(e) == 0 then
27  |   |   |   |   |   continue;
28  |   |   |   |   else
29  |   |   |   |   |   potential ← π_blue(v);
30  |   |   |   |   |   newPathType ← blue;
31  |   |   |   |   end
32  |   |   |   end
33  |   |   endsw
34  |   |   if v == t then
35  |   |   |   newPathType ← grey
36  |   |   end
37  |   |   suv ← LINK(dist[(u, pathType)], profile(u, v));
38  |   |   if NOT dist[(v, newPathType)].DOMINATES(suv) then
39  |   |   |   dist[(v, newPathType)].MERGE(suv);
40  |   |   |   q.update(v, newPathType, dist[v] + potential)
41  |   |   end
42  |   end
43  end
44  return dist[(t, grey)]
```

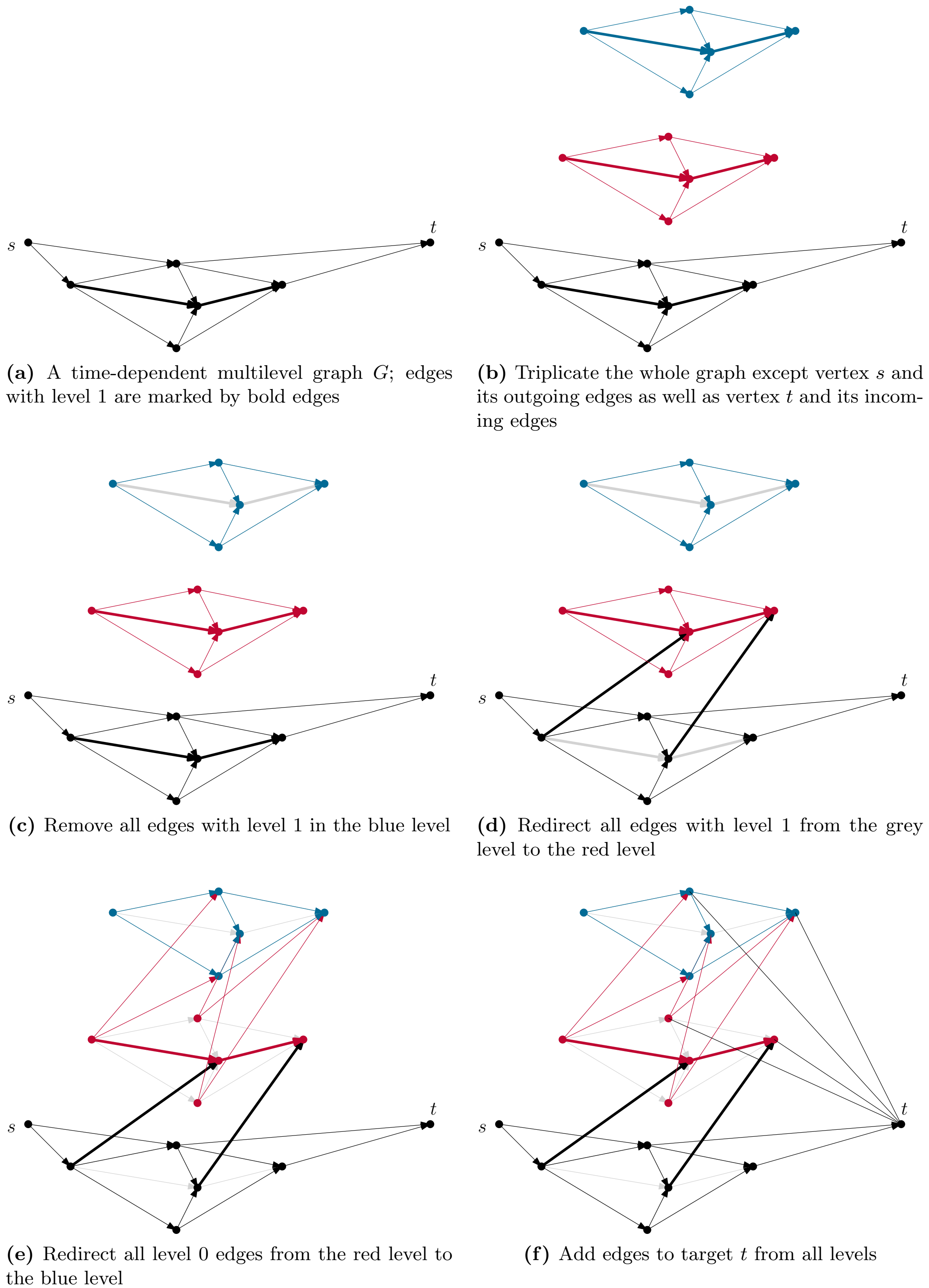
Finally, we add edges from vertex $v$ of any level to target $t$ if the edge $(v, t)$ exists in the original graph $G$. Thus, at target $t$, the levels merge together again.

We note that it is possible to shift from the grey state to the red state and from the red state to the blue state. However, being in the blue state, there is no path to another level except to target $t$. Running Profile A* on graph $G*$ is equivalent to running Multilevel A* on graph $G$. We can also see what the potentials $\pi_{red}$ and $\pi_{blue}$ represent. Using a backwards search on a scalar-min graph computes admissible potentials. If we use a backwards search on graph $G*$, the potentials that will be found for the blue level will equal $\pi_{blue}$. We also see, that we will find all shortest paths from $s$ to $t$. Shortest paths to $t$ can be in state *grey*, *red*, or *blue*. If it is a grey shortest path, it will find the shortest path by searching the grey level. If it is a red shortest path, it will find the shortest path by searching the grey and red level. And if it is a blue shortest path, it will find the shortest path by searching whole graph $G*$. Figure 4.4 shows how we construct graph $G*$ from the original graph $G$.

### 4.2.4 Stop Criteria and Prunings for Speed-up in Multilevel A*

All stop criteria and pruning techniques discussed in 3.2 are applicable in A* Search as well. Additionally, we can use the states to prune some more vertices. Progressing from state *grey*, to *red*, to *blue*, less paths to target $t$ are available. Label $(v, grey)$ can find more paths to $t$ than $(v, red)$ starting with the same departure time. Label $(v, red)$ can find more paths to $t$ than $(v, blue)$ starting with the same departure time. Thus, if the travel time profile from $(v, grey)$ dominates $(v, red)$ or $(v, blue)$ the label can be pruned. If $(v, red)$ dominates $(v, blue)$ we can prune it as well. This is denoted by *level pruning*. We add a level pruning to Multilevel A* as follows.

```
[...]
(u, pathType) ← q.popMin();
if pathType == blue and distance[(u, red)].dominates(distance[(u, blue)]) then
    continue;
if pathType == blue and distance[(u, grey)].dominates(distance[(u, blue)]) then
    continue;
if pathType == red and distance[(u, grey)].dominates(distance[(u, red)]) then
    continue;
[...]
```




**(a)** A time-dependent multilevel graph $G$; edges with level 1 are marked by bold edges

**(b)** Triplicate the whole graph except vertex $s$ and its outgoing edges as well as vertex $t$ and its incoming edges

**(c)** Remove all edges with level 1 in the blue level

**(d)** Redirect all edges with level 1 from the grey level to the red level

**(e)** Redirect all level 0 edges from the red level to the blue level

**(f)** Add edges to target $t$ from all levels

**Figure 4.4:** This figure shows how we construct graph $G*$ step-by-step from graph $G$ shown in (a). Edges of level 1 are marked with bolder edges whereas edges of level 0 have thinner edges. In Figure (b) we triplicate the graph. In Figure (c) we remove edges with level 1. In Figure (d), we redirect edges from grey level to red level. In Figure (e) we redirect edges from red level to blue level. Finally, we add edges to target $t$ from all levels.

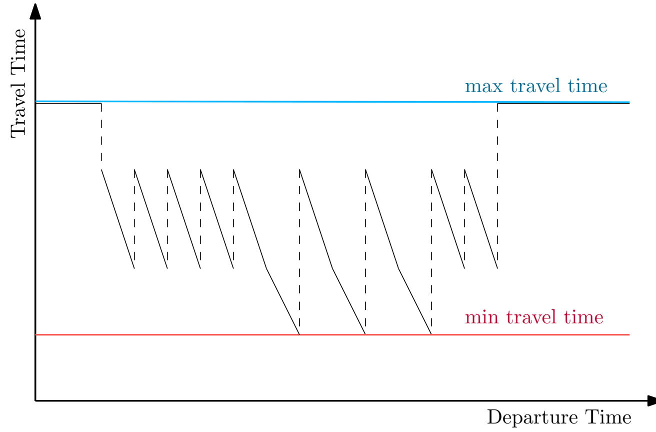



**Figure 4.5:** Showing the minimum and maximum travel time of a time-dependent edge.

## 4.3 Finding Level Assignments

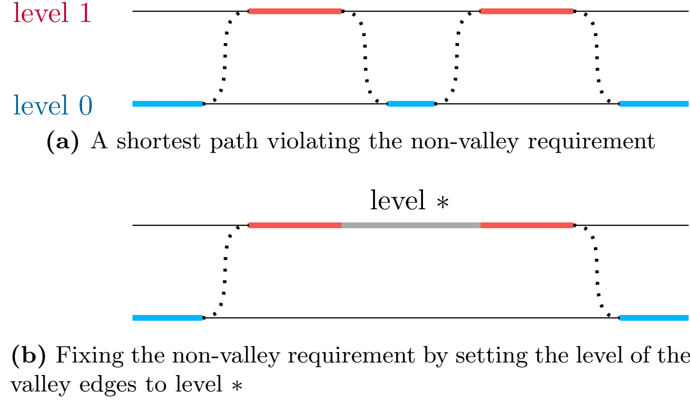
Multilevel A* Search only works with a valid level assignment. This section will explain how to generate a valid level assignment.

The main idea of Multilevel A* Search is to identify a set of edges $L_1$ with high variance of travel time such that perfect potentials in the graph $G_0$ without $L_1$ provide good potentials. Initially, all edges have level 0. To find suitable edges with level 1, we need to compute edges with a high variance. We achieve this by computing the difference of the maximum travel time of an edge $e$ and its minimum travel time. Figure 4.5 shows an example what the difference of travel time in a travel time profile is. Time-independent edges have a travel time difference of zero. After having computed all travel time differences of all edges, we set the level of the edges with the biggest differences to level 1. In our experiments in Section 5.1, we test different level assignment with varying numbers of edges with level 1.

Having defined all edges with level 1, the resulting level assignment might violate the *non-valley requirement* of some shortest paths. Here, we make use of the special edge level $*$ to fix the requirement. Edges with level $*$ do not violate the requirement as they act as if they have level 0 as well as level 1. We need to review all shortest paths and set the level of edges that violate the *non-valley requirement* to level $*$. Figure 4.6 shows an example of a shortest path that violates the requirement and is fixed by an edge with level $*$. We note that all shortest paths that violate the requirement and need to be fixed have a subpath from a vertex $u$ to a vertex $v$ and the edge $(u, v)$ has level 1. This edge is the reason why a shortest path may violate the requirement. In order to check all shortest paths, we therefore have to compute all pairwise shortest paths $\{v \mid (v, u) \in E \wedge level(v, u) = 1\} \times \{v \mid (u, v) \in E \wedge level(u, v) = 1\}$. Algorithm 4.2 shows how the levels are fixed.

### Alternative Assignment Criteria

The proposed level assignment maximizes the travel time variance in $L_1$. When removing all edges with level 1, a graph with lower travel time variance remains. We denote this by *high variance level assignment*. Using high variance level assignment is intuitive as our goal is to identify the most relevant high-speed connections that cause a high variance of travel time between two nodes. However, we might pick edges with this assignment that we would not necessarily have picked intuitively. Examples are ferries that just run a few times a day and thus have a very high travel time variance. However, such a ferry might have very a low importance for shortest paths because there is another bus connection that is similarly fast and runs more frequently. Ferry lines serving as sight seeing attractions are good examples for such a case. Thus, another possible criteria for picking edges with




**(a)** A shortest path violating the non-valley requirement

**(b)** Fixing the non-valley requirement by setting the level of the valley edges to level $*$

**Figure 4.6:** Figure (a) and Fig. (b) show how a shortest path that violates the non-valley requirement can be fixed by introducing edges with level $*$.

level 1 is choosing edges by their *speed-up*. This criteria uses the difference of the minimum travel time of an edge $(u, v)$ and the walking time needed to reach vertex $v$ from vertex $u$. Using a *speed-up level assignment* maximizes the number of high-speed train connections in $L_1$. However, an edge representing a high-speed train that runs very frequently might be chosen for level 1 even though it does not cause a higher travel time variance to the final travel time distance. Both assignments, *high variance level assignment* as well as *speed-up level assignment*, will be tested in our experiments in Section 5.1.

**Algorithm 4.2:** Fixing Level Assignment

**Input**: Graph $G = (V, E)$, Layer $L_1$
**Output**: Graph $G$ with valid level assignment
**Using**: $O2MDijkstra$: runs a one-to-many profile Search from a source to targets
$unpackPaths$: unpacks the actual paths of the shortest travel time profile
$violatesNonValleyRequirement$: checks validity of levels in a path
$setValleysToLevelStar$: marks all valleys and sets their levels to level $*$

```
1  sources ← fromVertices(L1);
2  targets ← toVertices(L1);
3  foreach s ∈ sources do
4  |   O2MDijkstra.run(s, targets);
5  |   foreach t ∈ targets do
6  |   |   shortestPaths ← O2MDijkstra.unpackPaths(s, t);
7  |   |   foreach sp ∈ shortestPaths do
8  |   |   |   if violatesNonValleyRequirement(sp) then
9  |   |   |   |   G.setValleysToLevelStar(sp);
10 |   |   |   end
11 |   |   end
12 |   end
13 end
```

# 5. Experimental Results and Evaluation

In this chapter we present experimental results of multimodal routing of the presented algorithms. We explain our test data and our test setup. Afterwards, we will evaluate the results and explain the outcome.

## 5.1 Experimental Setup and Input Data

We ran evaluation of multimodal routing in data of Switzerland and Germany. We conducted tests with varying distance and time ranges. To analyze and understand

### Test Setup

We implemented our algorithms in C++, compiled with clang++ version 3.8.0 on optimization level 3. Tests were run on quad core Intel Xeon E5-1630v3 clocked at 3.7GHz, with 128 GiB of DDR4-2133 RAM, 10 MiB of L3 cache, and 256 KiB of L2 cache. Tests were run exclusively on the machine, making sure no other computations or applications interfere with the computation.

### Data

The walking network of our data is provided by OpenStreetMap[1]. We extracted data of the road networks in Germany and Switzerland including pedestrian zones and stairs. OpenStreetMap data is intended for map rendering and contains many vertices with degree one or degree two. Edges connecting two of such vertices are irrelevant for routing. Therefore, consecutive edges were contracted. Vertices that are public transit stops were not contracted. We computed travel times of the walking network assuming a walking speed of 4km/h. The public transit network of Switzerland is based on a publicly available GTFS feed[2]. We extracted data of a business day (30th May 2017). The public transit network of Germany is provided by bahn.de of a day in winter 2011/2012. We then built our multimodal network by merging both data sources and constructing a transfer time aware time-dependent graph as explained in Section 2.3.

Details of the resulting graphs can be seen in the following table.

[1] http://download.geofabrik.de/
[2] http://gtfs.geops.ch/

| Data | #Vertices | #Edges | #Stops | #Walking Edges | #Connections |
|---|---|---|---|---|---|
| Germany | 10,825,575, | 33,057,041 | 245,690 | 29,281,552 (88.58%) | 45,540,983 |
| Switzerland | 771,375 | 2,336,436 | 24,048 | 2,178,686 (93.25%) | 1,950,880 |

**Classifying Query Difficulty**

Queries searching shortest paths are not equally "difficult". The search space needed to consider when searching the shortest paths has the biggest impact on run time. Keeping Dijkstra's algorithm in mind, it is evident that shortest paths between two nodes that are close to each other can be found faster than two nodes that are far away. Thus, we use the *Dijkstra rank* between two nodes as an indicator how difficult a query is. The Dijkstra rank $i$ of a source node $s$ to another node $v$ is the number of how many vertices need to be popped from the queue until $v$ is found with start node $s$ when using a normal Dijkstra's algorithm. Another way to explain Dijkstra rank is that vertex $v$ is the $i$-th farthest node from source $s$. In public transit network it is difficult to define the Dijkstra rank of two vertices due to time-dependency. In this thesis, we determine the Dijkstra rank of two vertices by running a walking Dijkstra's algorithm, namely Dijkstra's algorithm only using time-independent travel times. We group and average results of queries with the same logarithmic Dijkstra rank.

**Test Queries**

We ran test queries for Germany and Switzerland. We noticed that run times vary a lot even between queries of the same Dijkstra rank. Trips between nodes that are connected by a trip that does not require transfers where the estimation of the distance is close to the real distance are computed fast. Shortest paths between vertices where a transfer or more are needed, computation time is longer. Test queries in Switzerland that connect two big major cities such as Zurich and Bern are often computed faster than queries between smaller cities. For the final test query set we made sure to include queries between major cities as well as smaller cities. The test query set was determined as follows.

1. **Queries between big cities (`CITY`):** Queries between nodes of cities with stations serving high-speed trains. We identifies the biggest cities and picked nodes in the suburban areas of these cities. We build the test queries by running queries between the selected nodes of different cities at random.
2. **Queries between mid-sized cities (`MID`):** We identified smaller cities with stations that do not serve high-speed trains. We then built queries the same way we did for big cities.
3. **Queries within cities (`INNER`):** We used our nodes of cities already picked for the two other query sets and ran queries between all nodes of the same city. This results in a set of queries within suburban areas.

These queries also reflect the kind of queries that are most relevant in practical use. The resulting query sets include 720 queries for Switzerland and 580 queries in Germany. We ran tests in varying time ranges as well. We picked the following ranges.

1. **Night time 01:00 - 02:00 (60min):** We run queries with a 60min time range during night time. In many regions public transit service is paused during night time. Most of the shortest paths either walk distances that can be traveled with public transit during daytime or need to wait for public transit service to begin in the morning.

2. **Rush hour 07:00 - 08:00 (60min):** We run queries with a 60min time range during rush hour. The travel time variance of all edges are most likely the lowest in this short time range
3. **Main hours 07:00 - 17:00 (10h):** We run queries with a 10h time range. Most lines operate frequently in this time range.
4. **Whole day 00:00 - 24:00 (24h):** We run queries for the whole day. Many lines probably have a long break and do not operate at night. Thus, the travel time variance of shortest paths with a 24h profile is very big.

### Varying Level Assignments

Choosing a suitable level assignment for Multilevel A* is a difficult task as illustrated in Section 4.3. We ran experiments varying the number of edges with level 1 as well as the criteria of how to choose edges with level 1. Level assignments were preprocessed using the high variance criteria or the speed-up criteria.

Details of the resulting levels can be found in the following tables.

Level Assignment of Switzerland

| Data | Criteria Type | Level 1 Edges | Level * Edges | Criteria Bound |
|---|---|---|---|---|
| CH_HV_0 | High Variance | 204 | 6,653 | $\geq$ 10:40h |
| CH_HV_1 | High Variance | 1,383 | 49,952 | $\geq$ 3:17h |
| CH_HV_2 | High Variance | 2,929 | 101,381 | $\geq$ 1:12h |
| CH_HV_3 | High Variance | 6,385 | 278,603 | $\geq$ 0:28h |
| CH_SU_0 | Speed-up | 224 | 770 | $\geq$ 7:52h |
| CH_SU_1 | Speed-up | 1,381 | 21,457 | $\geq$ 3:03h |
| CH_SU_2 | Speed-up | 2,850 | 81,969 | $\geq$ 1:12h |
| CH_SU_3 | Speed-up | 6,659 | 264,468 | $\geq$ 0:30h |

Level Assignment of Germany

| Data | Criteria Type | Level 1 Edges | Level * Edges | Criteria Bound |
|---|---|---|---|---|
| GER_HV_0 | High Variance | 324 | 970 | $\geq$ 24:00h |
| GER_HV_1 | High Variance | 2,375 | 130,345 | $\geq$ 23:59h |
| GER_HV_2 | High Variance | 20,938 | 925,003 | $\geq$ 22:55h |
| GER_SU_0 | Speed-up | 331 | 55 | $\geq$ 1day 18:48h |
| GER_SU_1 | Speed-up | 2,727 | 8,016 | $\geq$ 20:53h |
| GER_SU_2 | Speed-up | 11,505 | 191,411 | $\geq$ 3:29h |

Figure 5.1 and Figure 5.2 show the level assignment of data `CH_HV_0` and `GER_SU_1` as examples.

### Key Parameters and Evaluation Criteria

The most important parameter is run time. We run performance tests of Dijkstra's algorithm, Profile A* Search and Multilevel A* Search. The A* algorithms use perfect potentials of the multimodal network. In order to get these perfect potentials, a backwards search from target $t$ to all vertices of the network is needed. Multilevel A* needs as many backward searches as levels in the network. In the case of bi-level A* Search, two backward searches are needed. The computation time of these backwards searches

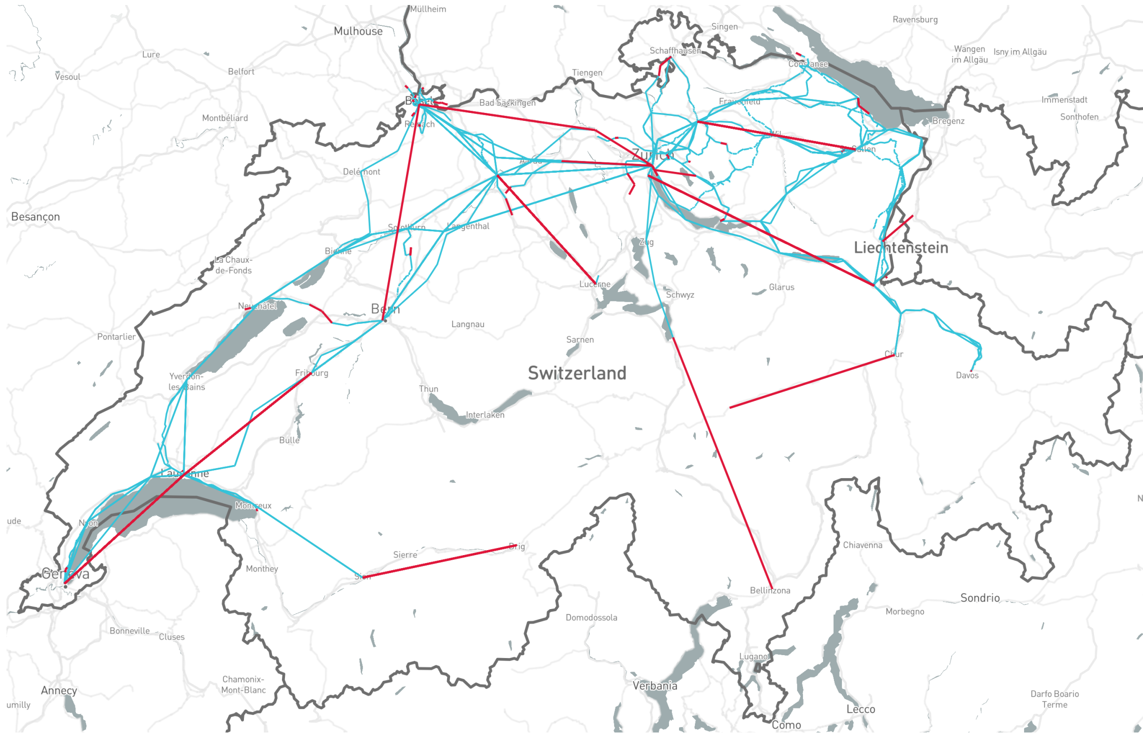



**Figure 5.1:** Shows the level assignment of `CH_HV_0`. Edges with level 1 are red. Edges with level * are blue. Edges of level 0 are omitted in favor of clarity but include all remaining edges of the street and public transit network.

often dominates the run time of the actual forward search. We denote the run time of the forward search as *forward run time*. Run time of the backward search is the *backward run time* respectively. In applications where routes of bigger networks than just Switzerland or Germany are needed, backward search would a long time. This shows that a backwards search as implemented here is not viable. In the scope of this work we evaluate and compare the capability of these algorithms. Therefore, rather than comparing their total run times we will take a detailed look at the run time of their forward searches.

Directly related to the run time is the size of the search space and the number how often we pop a vertex from the queue and visit it. The search space also enables us to understand and analyze how the algorithms work. In the search space we can see the effectiveness of potentials. The number of queue pops is important because profile searches are not label-setting. Our bi-level Multilevel A* triplicates the number of labels as every vertex can be visited by three path types. Analyzing the number of queue pops may show the impact of this decision.

## 5.2 Experimental Results

This section presents the final experimental results on data and with queries specified in the previous section. To assist the reader in this chapter it is noted that in all following plots properties of Dijkstra's algorithm are colored in dark grey, Profile A* and its timestamp version in blue, Multilevel A* with high variance level assignment in shades of orange, and Multilevel A* with speed-up level assignment in shades of red. All figures and networks in this section are made with OpenStreetMap and Mapbox.

### Comparing Dijkstra's Algorithm, Profile A* and Multilevel A*

Our experiments show that forward run time of both A* algorithms, Profile A* and Multilevel A*, clearly outperform Dijkstra's algorithm. For queries of higher Dijkstra ranks,

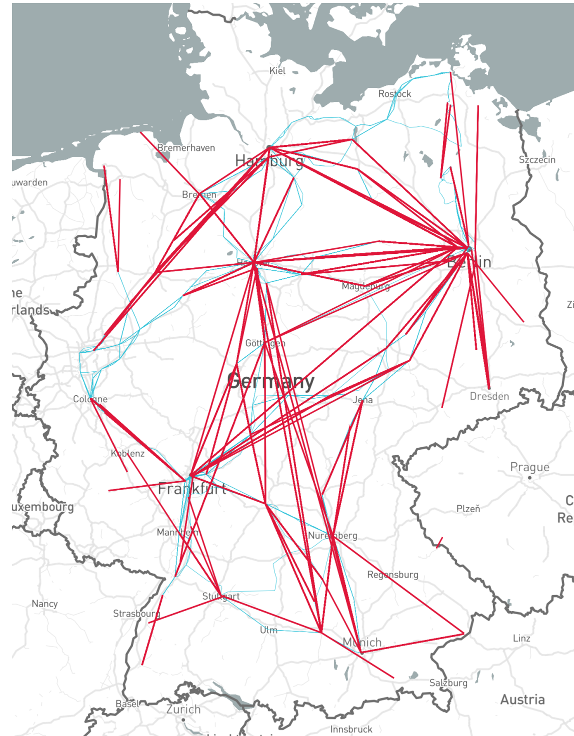



**Figure 5.2:** Shows the level assignment of `GER_SU_1`. Edges with level 1 and with level $*$ are red or blue respectively. Edges of level 0 are omitted in favor of clarity.

the speed-up of the forward search is big enough that the A* algorithms run faster than Dijkstra's algorithm even when including the computation time of the backwards search. Figure 5.3 shows the forward run time and search space sizes for queries in Switzerland from 7:00 – 17:00 to exemplify the observations. The results of experiments with the time ranges 7:00 – 8:00 and 7:00 – 17:00 are very similar. We will group them and refer to them as daytime queries in the evaluation. We observe that in many cases Multilevel A* achieves to reduce the search space compare to Profile A*. In most experiments the forward run time of Profile A* is faster than Multilevel A*. In queries during night time, Multilevel A* is able to reduce the search space significantly and outperforms Profile A* in regard of forward run time. The trends visible in experiments are visible in both datasets Switzerland and Germany.

### Experimental Results Regarding the Search Space

Multilevel A* reduces the search space up to 86% compared to Dijkstra's algorithm. Compared to the search spaces of Profile A*, Multilevel A* has a significantly smaller search space for queries in night-time and daytime. The search spaces of Multilevel A* in 24h queries is smaller than the search space of Profile A* but the difference is not very noticeable. Figure 5.4 shows the difference of search spaces of Switzerland in the morning and at night.

Running experiments of Multilevel A* with different level assignments show that the search spaces sizes are very similar regardless of which level assignment was used. As an example for this we show search spaces of experiments from 7:00 – 8:00 in Switzerland in the following table. This observations can be made for all time ranges and data sets.

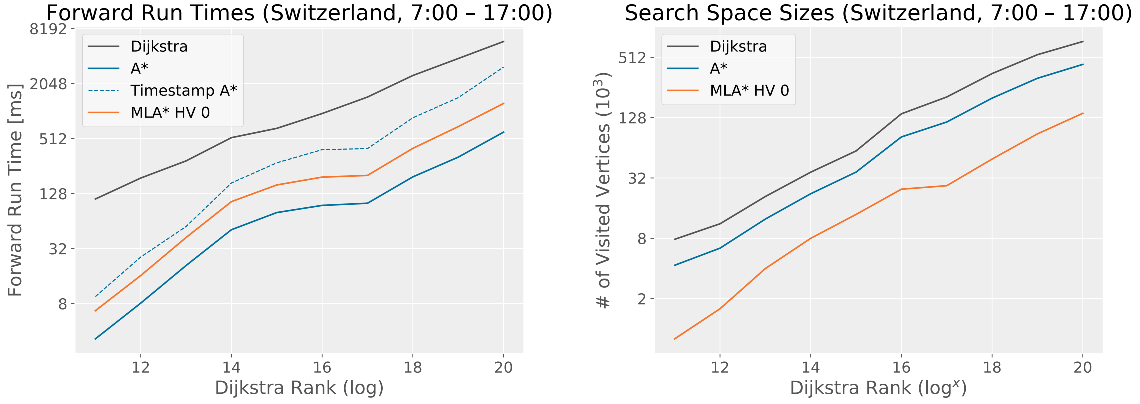



**Figure 5.3:** Run time of forward searches and search space sizes of queries in Switzerland from 7:00 – 8:00 with Dijkstra's algorithm, Profile A*, the timestamp version of Profile A* and Multilevel A*.

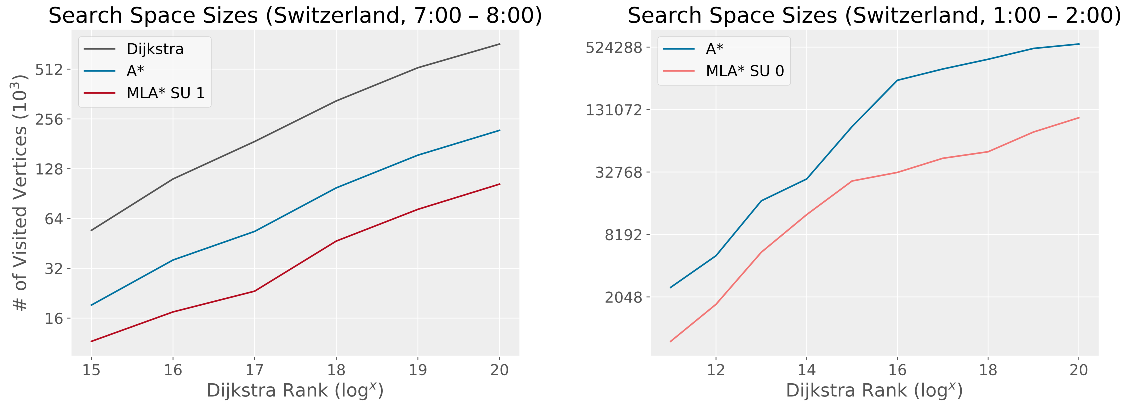



**Figure 5.4:** Search space sizes of queries in Switzerland from 7:00 – 8:00 and 1:00 – 2:00 with Dijkstra's algorithm, Profile A* and Multilevel A*.

**Search Spaces with Different Level Assignments (Switzerland)**

| Level Assignment | Time Range | Search Space with Dijkstra Rank | | |
|---|---|---|---|---|
| | | 18 | 19 | 20 |
| CH_HV_0 | 7:00 – 8:00 | 46,066 | 73,684 | 113,164 |
| CH_HV_1 | 7:00 – 8:00 | 45,568 | 73,940 | 108,977 |
| CH_HV_2 | 7:00 – 8:00 | 46,444 | 72,426 | 109,603 |
| CH_HV_3 | 7:00 – 8:00 | 46,829 | 74,185 | 110,142 |
| CH_SU_0 | 7:00 – 8:00 | 45,423 | 73,328 | 107,080 |
| CH_SU_1 | 7:00 – 8:00 | 46,758 | 72,737 | 103,304 |
| CH_SU_2 | 7:00 – 8:00 | 47,000 | 74,152 | 108,422 |
| CH_SU_3 | 7:00 – 8:00 | 46,651 | 74,679 | 110,537 |

### Experimental Results of the Run Time

The experiments show that forward run times of the A* algorithms are way faster than Dijkstra's algorithm. For longer distances with high Dijkstra ranks, the speed-up is even

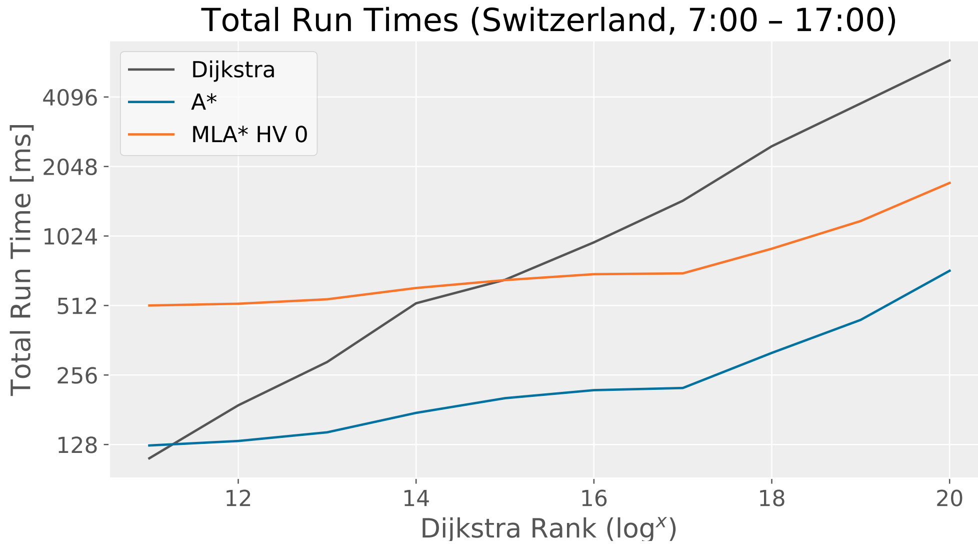



**Figure 5.5:** Total run times of Dijkstra's algorithm, Profile A* and Multilevel A*.

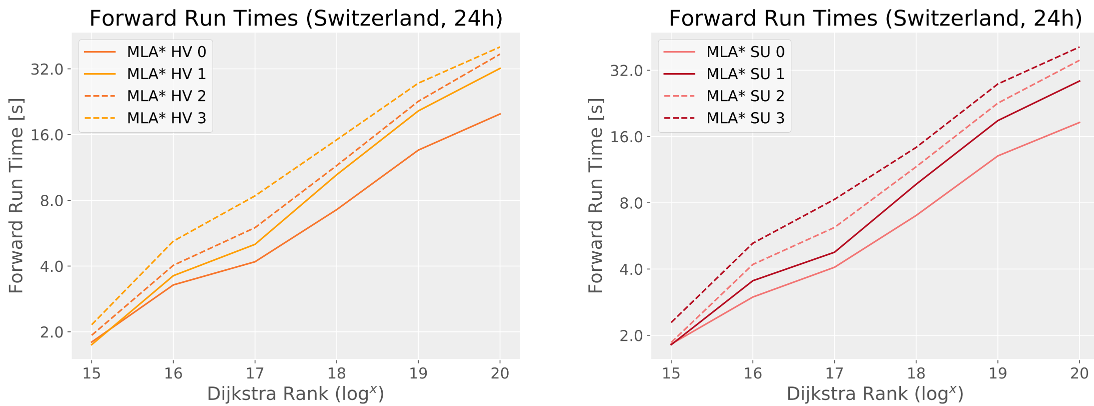



**Figure 5.6:** Forward run times with Multilevel A* for 24h queries in Switzerland with different level assignments.

big enough that it is worth to find the potentials in the expensive backwards search. This can be seen in Figure 5.5.

We took a closer look at forward run times of Multilevel A* with different level assignments in Switzerland and observed that the less edges with level 1 in the level assignment, the better the run time. This is linked to the number of queue pops that increases the more edges with level 1 exist. This behaviour is consistent for daytime and 24h queries. Figure 5.8 shows run times in Switzerland for 24h queries as examples. There is no visible run time difference during night-time. Queries in night-time run similarly fast regardless of level assignment. In Germany we do not see much differences in run time either. This may be due to the level assignments chosen in Germany to be too similar.

We observe that forward run times of MLA* are almost always slower than Profile A*. This is especially visible in 24h queries but also applies to daytime queries. In night-time queries Multilevel A* outperforms Profile A*'s forward run time. Figure 5.7 shows run times in Switzerland and Germany in daytime. Figure 5.8 shows run times in Switzerland for 24h queries and in night-time.

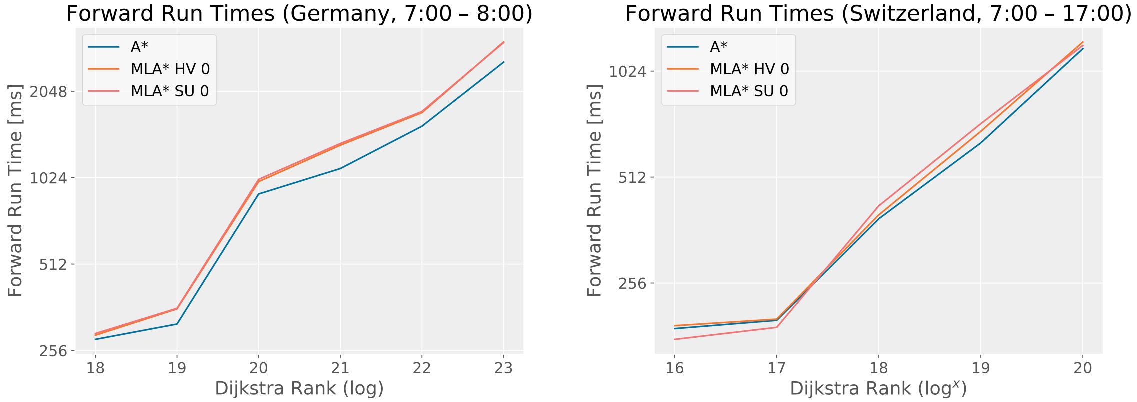



**Figure 5.7:** Forward run times of daytime queries in Switzerland and Germany.

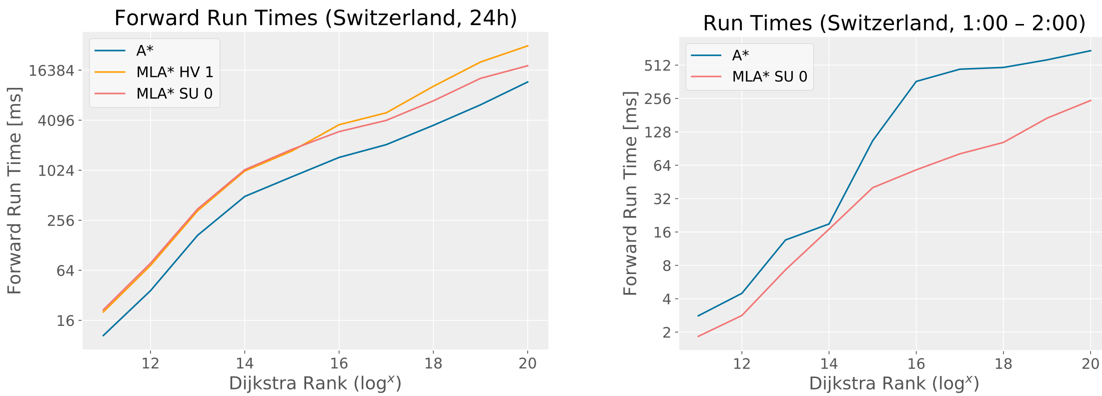



**Figure 5.8:** Forward run times with Multilevel A* for 24h queries and in night-time in Switzerland.

### Comparing Profile A* and its Timestamp Version

In addition to Profile A* we have implemented and tested the timestamp version of Profile A* (TSA*) as described in Section 3.2. Instead of whole profiles, we only process single timestamps and build travel time distances little by little. Expensive computation of merging, linking and dominance checks are easier in TSA*. The number of queue pops is a lot higher compared to normal Profile A* because every timestamp is processed and pushed to queue. When comparing run times of Profile A* and the TSA*, we observe that the timestamp version is slower or similarly fast to Profile A* for daytime and night-time queries. In 24h queries TSA* is faster than normal Profile A*. In queries with shorter distances, TSA* is often faster than Profile A* too. This is consistent whether for queries in the Germany or Switzerland and independent of the type of queries. Figure 5.9 show forward run times of daytime queries in Germany and 24h queries in Switzerland.

## 5.3 Evaluation

In this section we take a closer look at the experimental results. We explain the results and conclude the impact of Multilevel A*. In the end, we are most interested in the run time performance when solving multimodal routing. In a shortest path search, the search space and the number of how often a vertex is visited has the biggest impact on run time.

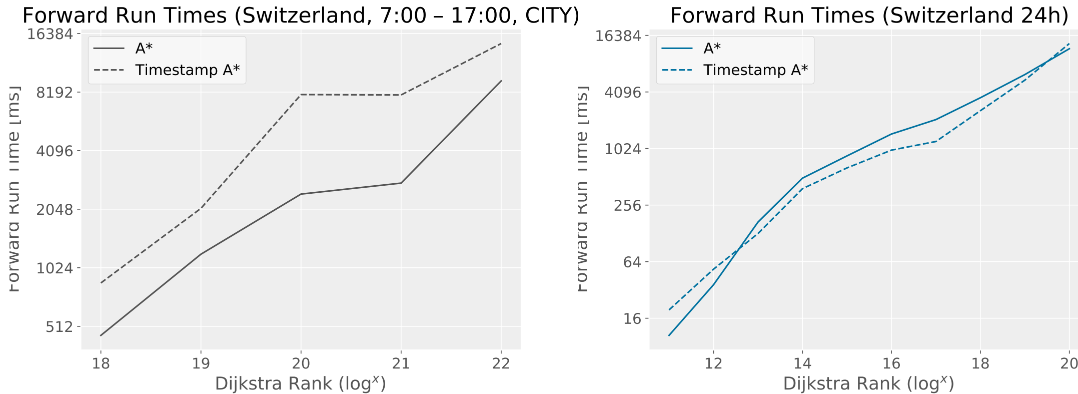



**Figure 5.9:** Forward run times of `CITY` queries from 7:00 – 17:00 in Germany and 24h queries in Switzerland with Profile A* and TSA*.

The experimental results show that performance of Multilevel A* is rather poor compared to Profile A* except for night-time queries. In night-time queries Multilevel A* easily outperforms Profile A* especially for queries of longer distances. The reason why the outcome for night-time queries is so different from the other queries is the minimum travel time of the distance from source $s$ to target $t$. In daytime queries or in 24h queries the potential of source $s$ and the real minimum travel time of the shortest paths between start and goal are close. When running Profile A*, using the minimum travel times as key will quickly lead to a shortest path to the target. During night-time however, the gap between our estimations and the real minimum travel times is much bigger. Profile A* is not able to reduce the search space effectively in such a scenario. In order to analyze the different outcomes in a meaningful way we will first discuss and evaluate results of daytime and 24h queries and afterwards analyze the results of the night-time queries separately.

### 5.3.1 Evaluating Daytime Queries and 24h Queries

This section will only evaluate and explain results of daytime queries and 24h queries. To understand the experimental results we will break it down by first evaluating search spaces, queue pops and finally the forward run times.

#### Evaluating Search Space

In order to reduce run time, reducing the search space is crucial. We will take a detailed look at the impact of Multilevel A* on the search space compared to Time-dependent Profile A*. With Multilevel A* we hope to reduce the search space by using $\pi_{blue}$. Potential $\pi_{blue}$ will be higher on paths that do not belong to the shortest paths. Such nodes will be visited later than others. Additionally, MLA* prunes vertices that are visited by paths that would violate the non-valley requirement. This reduces a lot of search space especially on long routes that use high speed trains. In cities around train stations where we could potentially leave the train, either potential $\pi_{blue}$ or the non-valley requirement prevents visiting unnecessary nodes. Figure 5.10a and Figure 5.11 show an comparison of search spaces using Profile A* and Multilevel A*. In Figure 5.10a the area around Bern shows clearly that Profile A* pops many vertices around Bern that are not needed in the shortest path. Multilevel A* however prunes most of these and hence achieves a much smaller search space.

However, we were surprised to find queries with the opposite effect as well. Some queries result in a bigger search space when using Multilevel A* than when using Profile A*.

An example can be seen in Figure 5.12. Multilevel A* never uses worse potentials than Profile A*. Thus, the search space of Multilevel A* should always be equal or smaller than Profile A*. This is true if we would not apply target pruning while searching. Using different potentials affects the order in which vertices are visited. This might also mean that the target $t$ is found at a different time. In the example in Figure 5.12 Multilevel A* finds some connections to target $t$ later than Profile A*. Both searches visit nodes in Zurich during their search but Profile A* visits fewer nodes in Zurich than Multilevel A*. This is because Profile A* has found a shortest path to target $t$ that prunes most of the nodes in Zurich at that point already. Multilevel A* has not found a shortest path to target $t$ that prunes these nodes yet. This case also shows how important a suitable level assignment is. The example shown in Figure 5.12 is a query run on the level assignment `CH_HV_1`. Tests of this queries with other level assignments but `CH_HV_1` do not have the same problem.

Presented here are cases with extreme differences in search space. However, there are mixed cases as well. Queries in which the size of the search space is similar, but the search space is still different.

Figure 5.13 shows an example of a query in which both algorithms have a similarly big search space but it still looks different. Again, one reason is that the difference in potentials changes the time when target $t$ is found and affects target pruning. However, it is noticeable that Profile A* visits many nodes at the beginning of the search whereas Multilevel A* visits more vertices at its end. MLA* prunes edges that would violate the non-valley requirement in the beginning. The difference between potential $\pi_{blue}$ and potential $\pi_{red}$ of a vertex $v$ is bigger, the further $v$ is away from target $t$.

As illustrated in Chapter 4 the maximum travel time from $s$ to $t$ and the size of the search space are highly related. This can clearly be seen in Figure 5.14. The search space grows when changing from a time range during rush hour from 7:00 – 8:00 to a query with a longer time range 7:00 – 5pm to a 24h query. When extending the time range to a 24h profile, it includes night time when hardly any trains run. The maximum travel time to the target is much higher in that time window and the search space increases because stop criteria is much harder to fulfill with a bigger maximum travel time. In average, 90% of the search space in a 24h query is processed after the search has found the shortest path to the target $t$. This means that 90% of the search space is only needed to prove correctness. Figure 5.10 shows the growing search space when the time range increases. In both cases MLA* achieves a better search space for queries with smaller time ranges. MLA* cannot reduce the search space by a lot in 24h queries. The reason is that the potential $\pi_{blue}$ does not help in reducing the search space in 24h queries. Even though $\pi_{blue}$ is bigger than $\pi_{red}$, the difference is not big enough to overcome the high maximum travel time to target $t$ in 24h queries. Thus, even if the potentials are bigger, the vertices are popped and processed anyways in 24h queries.

We conclude that, using two different potentials $\pi_{red}$ and potential $\pi_{blue}$ can affect the search space heavily. Depending on the query and the edges in level 1, the impact can be positive as well as negative.

### Evaluating Number of Queue Pops

Even more relevant for run time than the search space is the number of vertices popped from the queue and processed. In label-setting algorithms the number of popped vertices and the size of the search space is equal. In label-correcting algorithms such as Profile A* or Multilevel A* however, vertices can be popped multiple times. We observe that the number of pops and the size of search space are mostly proportional. Multilevel A* tends to need more queue pops than Profile A*. This is due to Multilevel A* considering a vertex

multiple times in different stages. A vertex $v$ can be in the queue up to three times either found by a grey path, a red path, and a blue path. As illustrated in Section 4.2 we can prune vertices of higher level when they are dominated. We can not prune vertices with a lower level. This potentially results in more queue pops. In order for MLA* to achieve less queue pops compared to Profile A*, the difference in search space must be very big.

#### Evaluating Run Time

We observe that even in some cases where MLA* has a smaller search space and fewer queue pops, the computation with Multilevel A* is slower than with Profile A*. The difference in run time is due to the overhead of checking edge levels, determining path states and testing level dominance. In average Profile A* performs better than MLA*.

### 5.3.2 Evaluating Night-Time Queries

The experiments show that Multilevel A* is a lot faster than Profile A* in night-time queries. Upon comparing the forward run times of Profile A* and Multilevel A* for the time-range 7:00 – 8:00 and 1:00 – 2:00, we observe that the run times of Multilevel A* do not change much whether it computes a 60 minute travel time profile of the night or the morning. Profile A* on the other hand is much slower during night-time. The main reason for Multilevel A* outperforming Profile A* at night is the poor performance of Profile A*. The comparison of the run times can be seen in Figure 5.15.

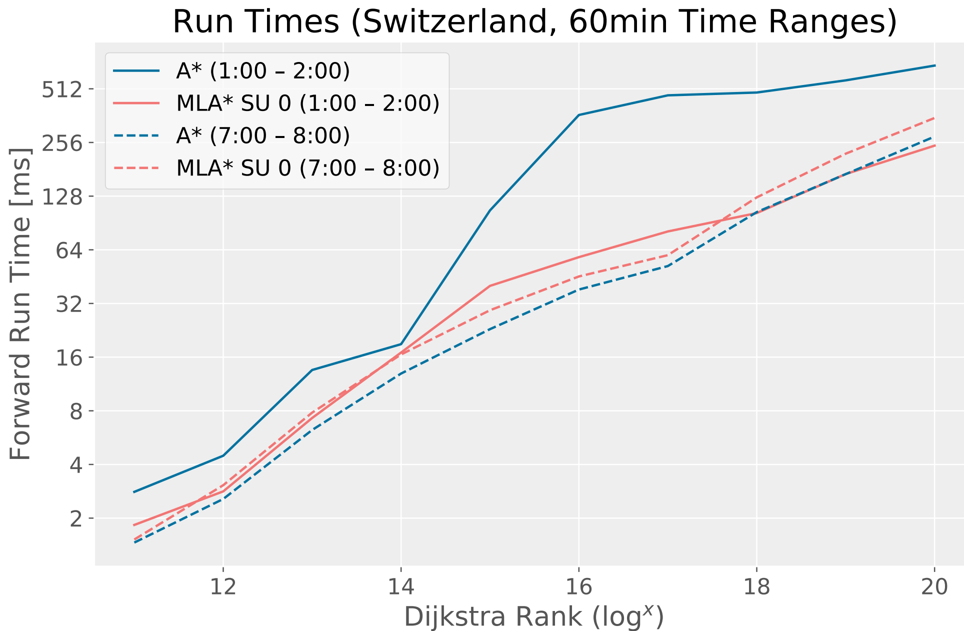
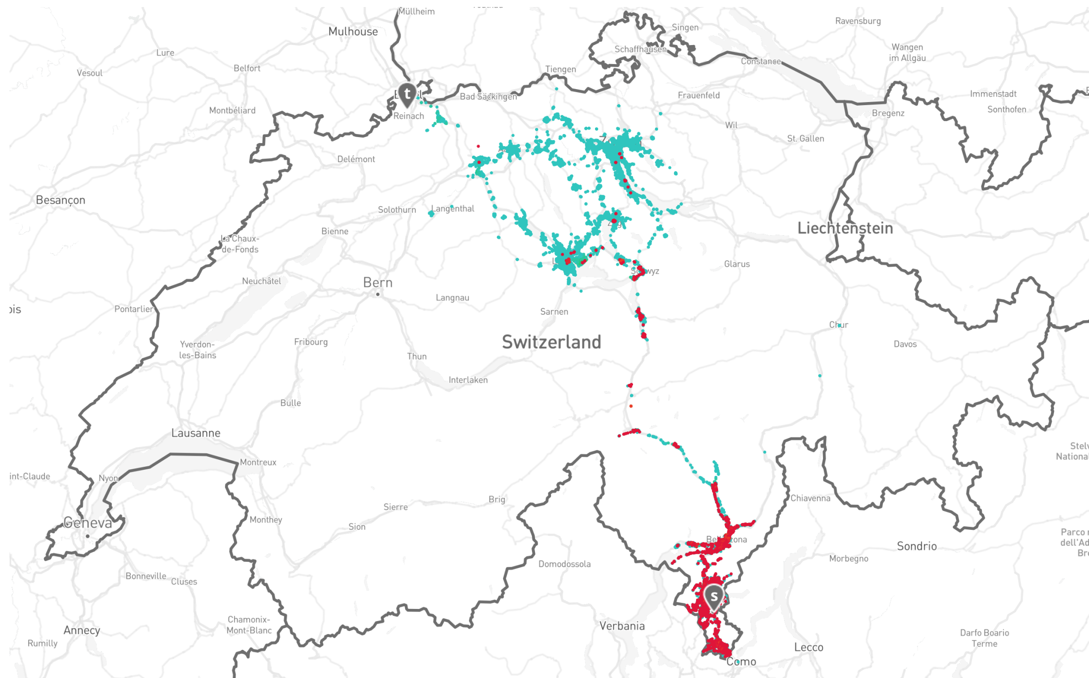
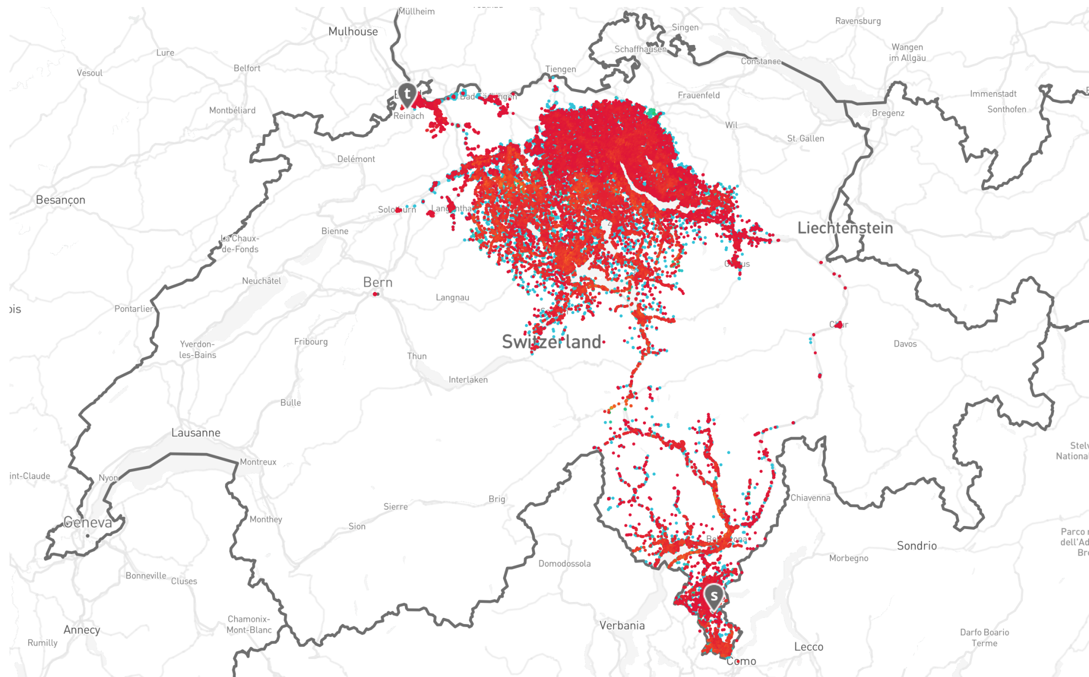



**Figure 5.15:** Forward run times of 60 minute time ranges in Switzerland in the morning (7:00 – 8:00) and in the night (1:00 – 2:00) run by Profile A* and Multilevel A*.

In fact, In some queries Profile A* even performs worse than Dijkstra's algorithm in terms of run time as well as search space. Profile A* uses minimum travel times as potentials. At night, the real travel time to the target might be very long. The longer the travel time the more vertices Profile A* is going to visit. This behaviour can be seen in Figure 5.16 which shows the comparison of search spaces of Profile A* in the morning and at night. Figure 5.17 shows the comparison between search spaces of Profile A* and Multilevel A* at night.




**(a)** Search space of Profile A* and MLA* in the morning. The search space of Profile A* is 99,380 vertices and of MLA* 22,829 vertices which is 23% of the search space of Profile A*.




**(b)** Search space of Profile A* and MLA* in a 24h query. The search space of Profile A* is 261,780 vertices and of MLA* 261,937.

**Figure 5.10:** Comparison of the search spaces of Profile A* in blue and MLA* in red. The Figure (a) is a query with time time range 7:00 – 8:00 and Figure (b) is a 24h query. In Figure (a) the search space of MLA* is smaller than the search space of Profile A*. In the 24h query in Figure (b) there is hardly any difference.

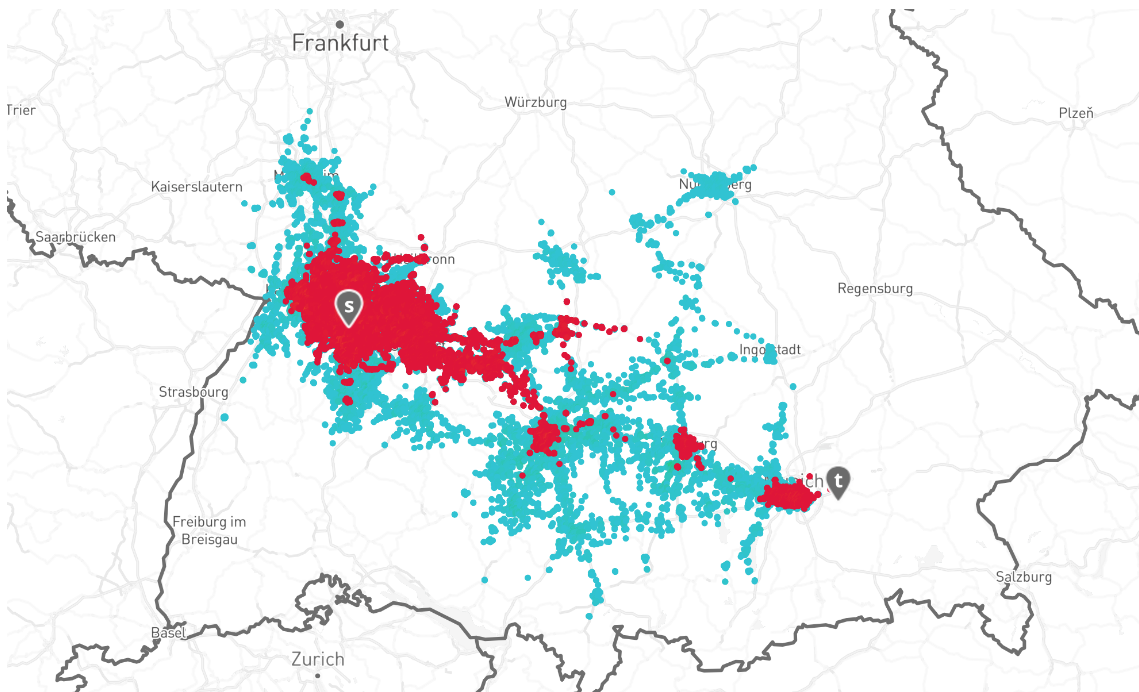



**Figure 5.11:** Search spaces of a query using Profile A* in blue and of MLA* in red from 7:00 – 17:00. The search space of Profile A* contains 939,499 vertices and MLA* 416,014 vertices which is 44% of Profile A*'s search space.

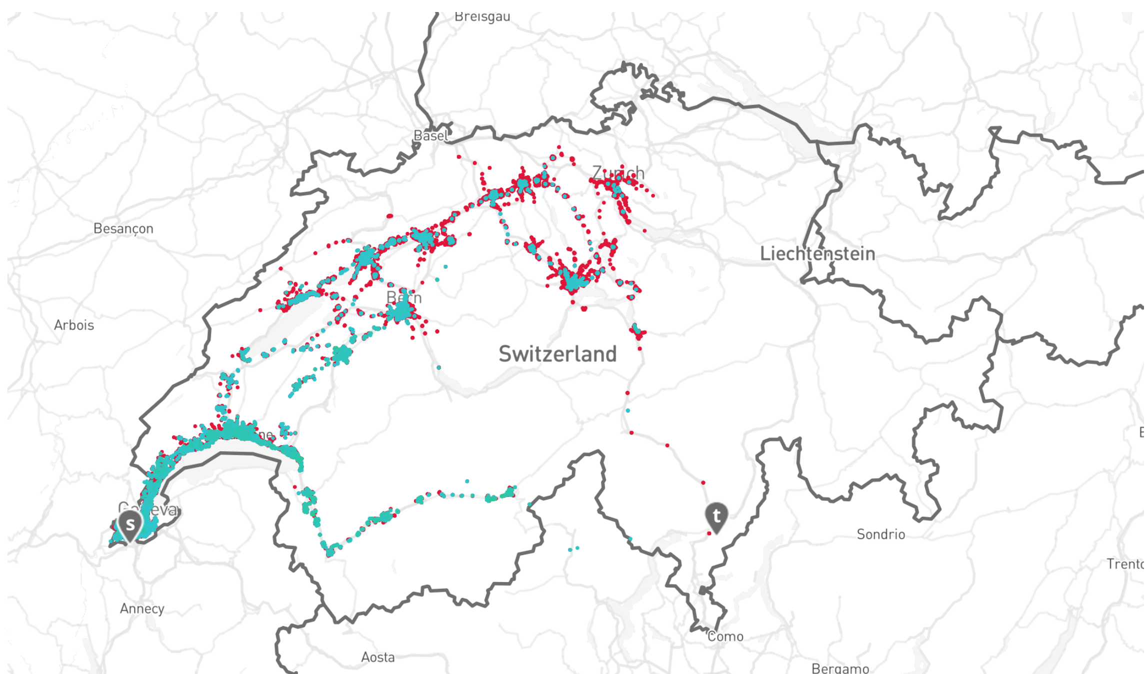



**Figure 5.12:** Search spaces of a query using from 7:00 – 8:00 where the search space of MLA* is bigger than the search space of Profile A*. Multilevel A* is red and contains 153,991 vertices which is 146% of the search space of Profile A* which is colored blue and contains 105,378 vertices

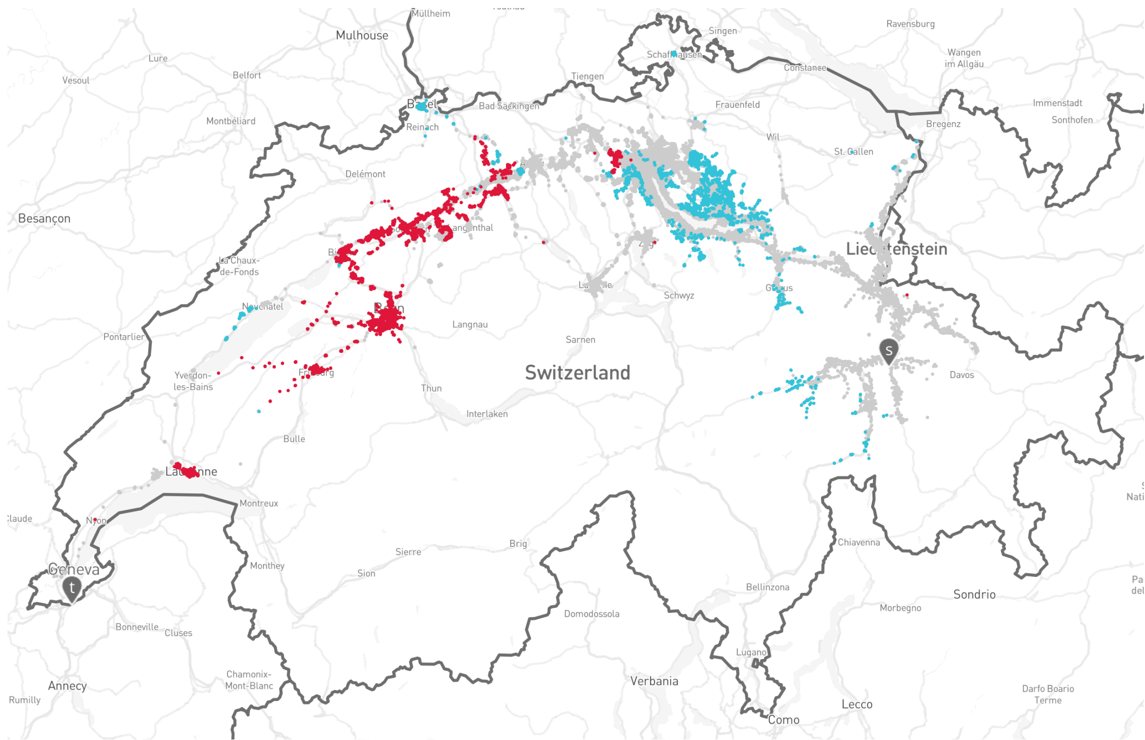
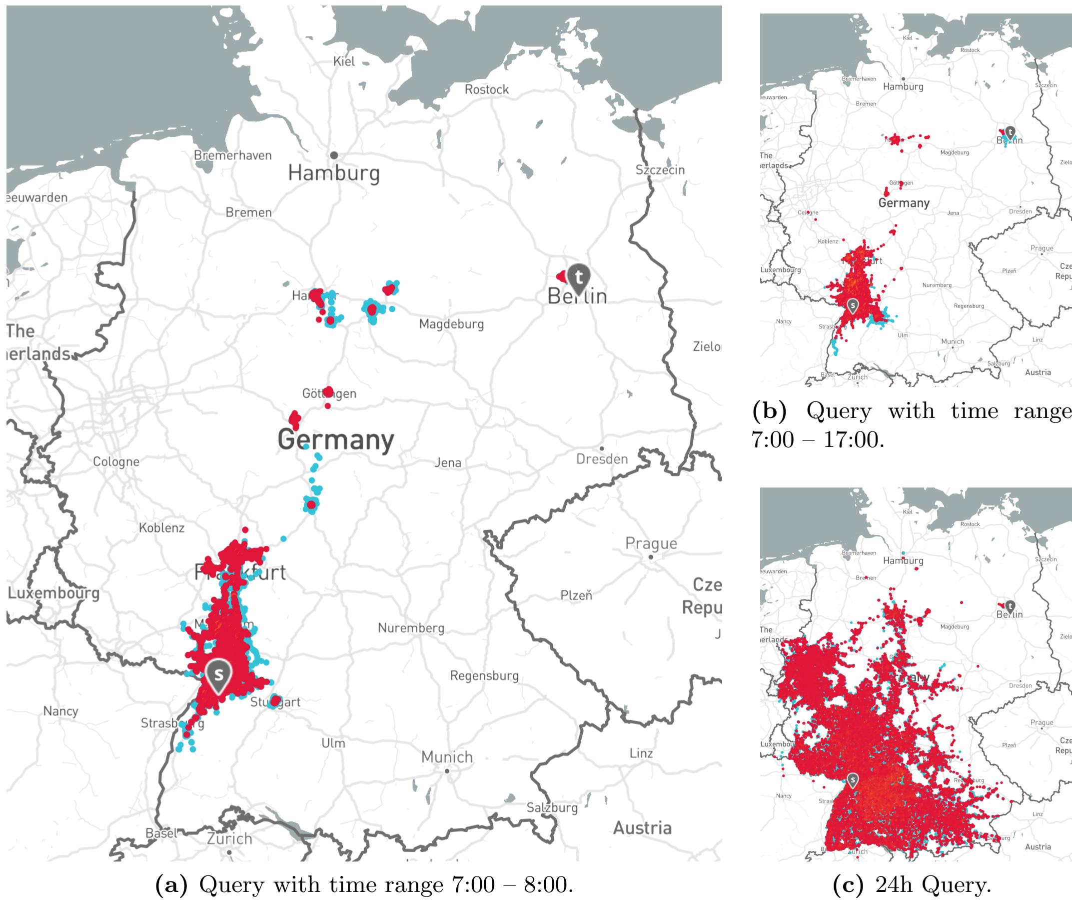



**Figure 5.13:** Search spaces of a query using Profile A* (blue and grey) and MLA* (red and grey). The query was run with time range 7:00 – 8:00. Blue vertices are only visited by Multilevel A* and red vertices only by Profile A*

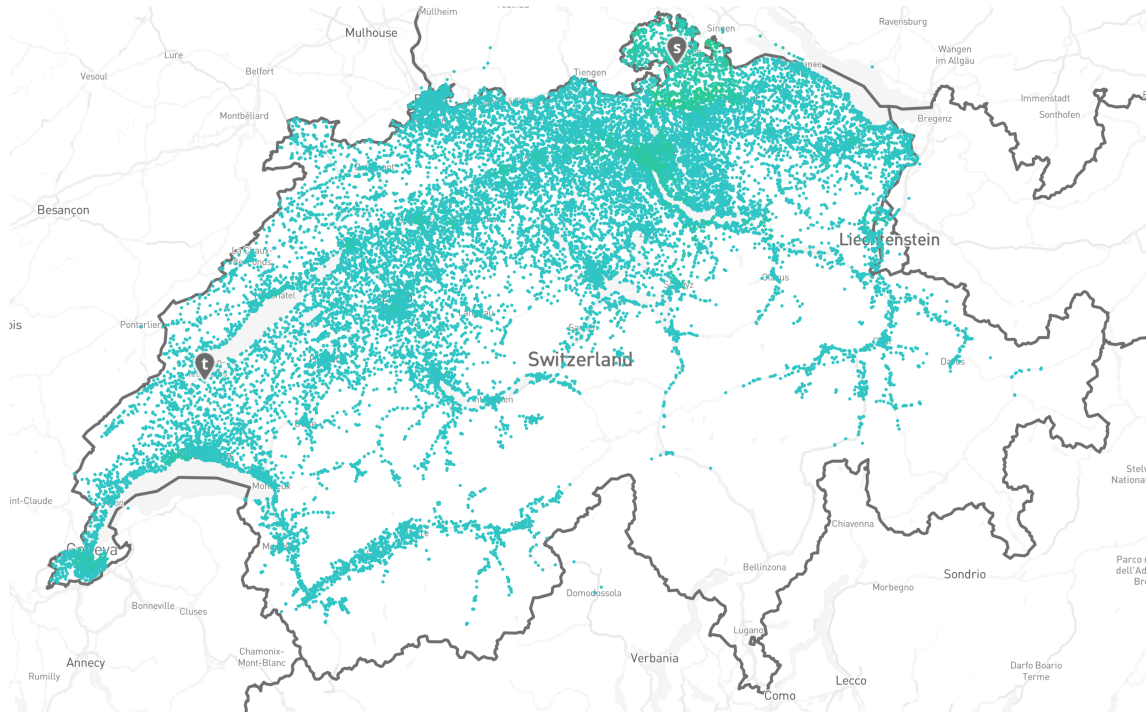



(a) Query with time range 7:00 – 8:00.

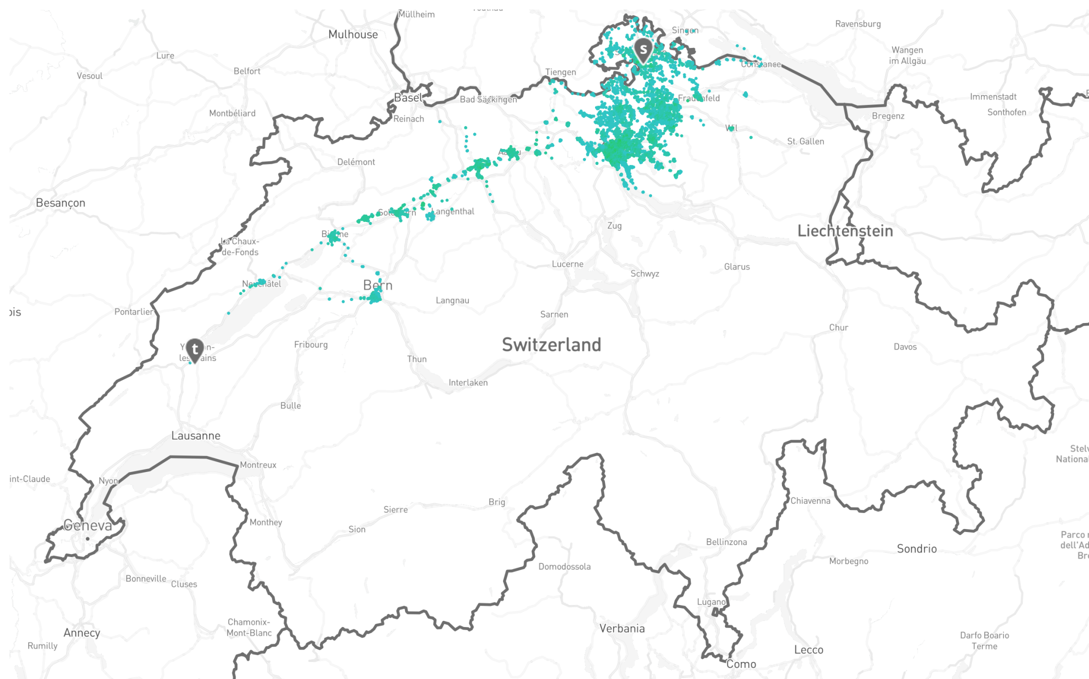
**(b)** Query with time range 7:00 – 17:00.

**(c)** 24h Query.

**Figure 5.14:** Search spaces of queries using Profile A* (blue) and Multilevel A* (red).




**(a)** Search space of a query from 1:00 – 2:00




**(b)** Search space of a query from 7:00 – 8:00

**Figure 5.16:** Search Space of a query from 7:00 – 8:00 and of the same query from 1:00 – 2:00 in Switzerland. This search space shows that the search space is much bigger during night-time.

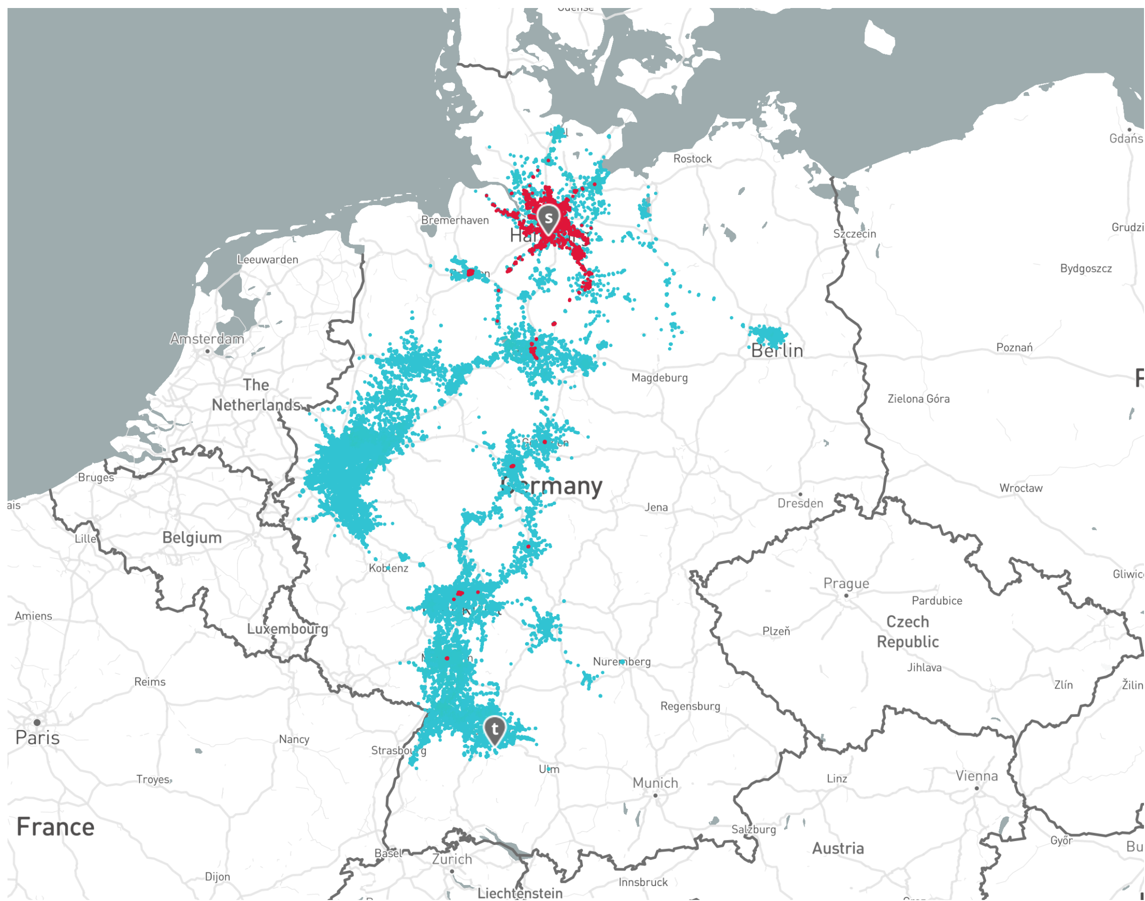



**Figure 5.17:** Search spaces of a night time query with Profile A* (blue) and Multilevel A* (red).

# 6. Conclusion

This final chapter concludes our findings in this thesis. We will summarize the results of the experiments and discuss future work to follow up this work.

## 6.1 Summary

Using A* Search with perfect potentials to find shortest paths provides a big speed-up compared to Dijkstra's algorithm. In time-independent, scalar graphs, perfect potentials exist such that during query time only relevant vertices are visited. Profile queries in multimodal graphs do not have such perfect potentials. Profile queries are more difficult because A* is not a label-setting but a label-correcting algorithm. We have observed that using the minimum travel time of each edge to compute perfect potentials in multimodal networks delivers a very good speedup in time ranges when public transit service operates frequently. More precisely A* Search with perfect potentials works well if the travel time variance on the shortest path profile from $s$ to $t$ is low and when the estimation of the distance from $s$ to $t$ is close to the real travel time needed.

Multilevel A* attempts to improve A* Search on multimodal graphs by assigning levels to edges. A graph $G$ then consists of subgraphs $G_i$ containing edges of level $i$. Multilevel A* shows great results on some queries reducing the search space. Especially in periods of low traffic intensity in public transit when Profile A* is not able to perform well, Multilevel A* performs much better than Profile A*. However, Multilevel A* fails to achieve this speed-up in queries during rush hour.

In conclusion, Profile A* Search in perfect potentials work well in networks with low travel time variance such as during rush hour. When the travel time variance is high and the potential differs much from the real travel time such as during night-time, Multilevel A* outperforms Profile A* Search.

## 6.2 Future Work

The speed-up by Multilevel A* compared to Profile A* is highly dependent on how well Multilevel A* can make use of the level assignment. The 24h experiments show that Multilevel A* fails to make use of potential $\pi_{blue}$ because the maximum travel time is too high. One way to approach this problem is to choose the edge level in such a way that the subgraph $G_0$ with low levels of the public transit network is decomposed into multiple

disconnected connected components. In such a case the potential $\pi_{blue}$ would be forced to use walking edges and the difference to potential $\pi_{red}$ is potentially maximized. It is desirable to research whether it is possible to find such a level assignment without having to many edges of level $*$ to make it work.

Computing travel time profiles over a time window of 24h is still very difficult with potentials. The search space increases a lot because of the high maximum travel time and with it increases run time. The main idea of Multilevel A* is to use different potentials for different subgraphs. The experimental results of 24h profile queries show that computation is slow because the maximum travel time is too big and do not match the perfect potentials. An approach using different potentials for different time windows and an algorithm that makes use of these might be able to solve this problem.

Another straight-forward next step to this work is extending a bi-level A* to a Multilevel A* using more than two levels. However, with rising number of levels we increase the number of different labels that we need to consider. Finding a modification of Multilevel A* that uses the level assignments as proposed in this thesis but does not need to triplicate every vertex might also improve run time.

This work also uses the potentials as if given and does not regard the run time needed to compute these. This is acceptable for research. For practical use the total run time and not only the run time used during forward search is important. Our research experiments run on small graphs such as Switzerland or Germany. When we compute the potentials we run a complete backwards Dijkstra's search to all vertices in the graph. The bigger the graph the longer it takes. If run in a graph of a whole continent or of the whole world, run time will be too long. Using the same preprocessing as in the ALT algorithm can provide admissible potentials for vertices without the need of a backwards search. We do not know exactly how Profile A* and Multilevel A* perform with such potentials. The experiments show that the speed-up of Multilevel A* to Profile A* is highest in cases when the difference between potential of $s$ and the real travel time from $s$ to $t$ are high. This raises hope that it might result in better run time when using preprocessed potentials as well.

At last, the weak point of A* Search is a big search space around the region of source $s$. Extending Multilevel A* to a bi-directional A* Search may reduce this.

# Bibliography


[Bas09] Hannah Bast. *Car or Public Transport—Two Worlds*, pages 355–367. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[BBH+08] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav Marathe, and Dorothea Wagner. *Engineering Label-Constrained Shortest-Path Algorithms*, pages 27–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[BCE+10] 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: Part I*, ESA'10, pages 290–301, Berlin, Heidelberg, 2010. Springer-Verlag.

[BDG+15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.

[BS14] Hannah Bast and Sabine Storandt. Frequency-based search for public transit. In *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '14, pages 13–22, New York, NY, USA, 2014. ACM.

[DDP+13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. *Computing Multimodal Journeys in Practice*, pages 260–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[DGPW11] Daniel Delling, Andrew Goldberg, Thomas Pajor, and Renato Werneck. Customizable route planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Springer Verlag, May 2011.

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

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. *Intriguingly Simple and Fast Transit Routing*, pages 43–54. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. *Accelerating Multimodal Route Planning by Access-Nodes*, pages 587–598. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[DPW12] Daniel Delling, Thomas Pajor, and Renato Werneck. Round-based public transit routing. Society for Industrial and Applied Mathematics, January 2012.

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. *Better Landmarks Within Reach*, pages 38–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[HNR72] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, pages 28–29, 1972.

[SW] Ben Strasser and Dorothea Wagner. *Connection Scan Accelerated*, pages 125–137.

[Wit15] Sascha Witt. Trip-based public transit routing. *CoRR*, abs/1504.07149, 2015.

[WZ17] Dorothea Wagner and Tobias Zündorf. Public Transit Routing with Unrestricted Walking. In Gianlorenzo D'Angelo and Twan Dollevoet, editors, *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.