# Multi-Modal Route Planning in Road and Transit Networks

**Master's Thesis**

Daniel Tischner

University of Freiburg, Germany,

daniel.tischner.cs@gmail.com

August 28, 2018

Supervisor: Prof. Dr. Hannah Bast
Advisor: Patrick Brosi

TODO: disable todos and macro highlights. Related Work, Future Work

# Contents

# Declaration

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

Place, Date

Signature

# Zusammenfassung

Wir präsentieren Algorithmen für multi-modale Routenplannung in Straßennetzwerken und Netzwerken des öffentlichen Personennahverkehrs (ÖPNV), so wie in kombinierten Netzwerken.

Dazu stellen wir das Nächste-Nachbar und das Kürzester-Pfad Problem vor und schlagen Lösungen basierend auf Cover Trees, ALT und CSA vor.

Des Weiteren erläutern wir die Theorie hinter den Algorithmen, geben eine kurze Übersicht über andere Techniken, zeigen Versuchsergebnisse auf und vergleichen die Techniken untereinander.

# Abstract

We present algorithms for multi-modal route planning in road and public transit networks, as well as in combined networks.

Therefore, we explore the nearest neighbor and shortest path problem and propose solutions based on Cover Trees, ALT and CSA.

Further, we illustrate the theory behind the algorithms, give a short overview of other techniques, present experimental results and compare the techniques with each other.

Section 1

# Introduction

Route planning refers to the problem of finding an *optimal* route between given locations in a network. With the ongoing expansion of road and public transit networks all over the world route planner gain more and more importance. This led to a rapid increase in research [16, 24, 36] of relevant topics and development of route planner software [30, 28, 47].

However, a common problem of most such services is that they are limited to one transportation mode only. That is a route can only be taken by a car or train but not by both at the same time. This is known as uni-modal routing. In contrast to that multi-modal routing allows the alternation of transportation modes. For example a route that first uses a car to drive to a train station, then a train which travels to a another train station and finally using a bicycle from there to reach the destination.

The difficulty with multi-modal routing lies in most algorithms being fitted to networks with specific properties. Unfortunately, road networks differ a lot from public transit networks. As such, a route planning algorithm fitted to a certain type of network will likely yield undesired results, have an impractical running time or not even be able to be used at all on different networks. We will explore this later in **Section 6**.

## 1.1 Contributions

Our main contribution to this research field is the development of Cobweb [44], which is an open-source framework for multi-modal route planning developed in the context of this thesis. Further, in **Section 6** we give a detailed evaluation of experiments demonstrating the effectiveness of our implementations for all algorithms explained in this thesis. Additionally, we give an overview over route planning and relevant approaches, as well as a thorough explanation for all used algorithms including examples illustrating them.

Cobweb is able to parse networks given in the OSM and GTFS format, which we will explore later in **Section 6.1**, as well as in compressed formats, such as BZIP2 [1], GZIP [25], ZIP [34] and XZ [10]. Networks are then represented in one of the models presented in **Section 3**. Metadata, like names of roads, are saved in an external database and retrieved again later.

The backend offers three REST-APIs [39] using a client-server-based structure com-

municating over the HTTP [29] which are written primarily in Java. One API is for planning journeys, one for searching nodes by their name and one for retrieving the nearest node to a given location.

The routing API answers journey planning requests from a given source to a destination. The answer contains multiple viable journeys. A request consists of

1. depTime, the departure time to start journeys at;
2. modes, transportation modes allowed for the journey. Applicable are car, bike, foot and tram;
3. from, the source node to departure from;
4. to, the destination node to travel to.

The server then computes journeys using the algorithms presented in **Section 5** and responds with a list of viable journeys. A journey mainly consists of geographic coordinates describing the path to travel and metadata, such as which transportation mode to use for which segment, names of roads and time information for each segment.

The name search API finds OSM nodes by their name. Therefore, we developed LexiSearch [43], an API for retrieving information in given datasets. It maintains names of OSM nodes in an inverted $n$-gram index [20, 22]. This makes it possible to efficiently retrieve nodes by an approximate name which is allowed to have errors, such as spelling mistakes. This is known as fuzzy search, or approximate string matching, see [37] for details. Further, nodes can be retrieved by prefixes, yielding search results *as-you-type*. For example, a request with the approximate prefix name *Freirb* would yield nodes with the name *Freiburg* and *Freiburg im Breisgau*.

The third API offers retrieval of the OSM node nearest to a given geographic coordinate. Making it possible for a client to plan a route from an arbitrary location to an arbitrary destination, for example by clicking on a map. Cobweb retrieves the nearest node by using a Cover Tree and solving the Nearest Neighbor Problem, as explained in **Section 4**.

Cobweb comes with a light web-based frontend (see **Fig. 1.1** for an image). Its interface is very similar to other route planning applications, providing input fields for a source and a destination, as well as a departure time and transportation mode restrictions. The frontend is primarily written in JavaScript and communicates with the backends REST-APIs using asynchronous method invocations. The resulting journeys are displayed on a map and highlighted according to metadata, such as the used transportation mode.

The source code of Cobweb, a release candidate, as well as a detailed description of the project, its APIs, an installation guide, the structure and its control flow, can be found at [44].

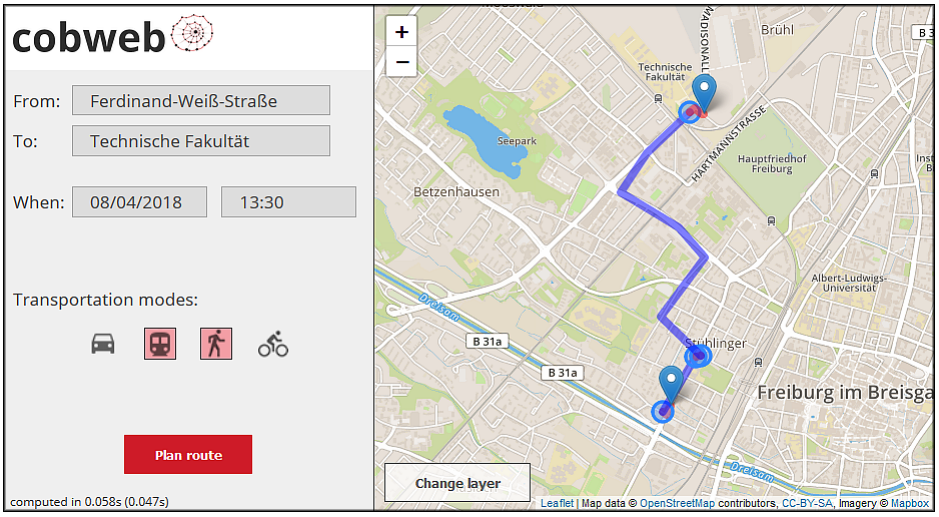


Fig. 1.1: Screenshot of COBWEBs [44] frontend, an open-source multi-modal route planner. It shows a multi-modal route starting from a given source, using the modes *foot-tram-foot-tram-foot* in that sequence to reach the destination.

## 1.2 Overview

In this thesis we explore a technique with which we can combine an algorithm fitted for road networks with an algorithm for public transit networks. Effectively obtaining a generic algorithm that is able to compute routes on combined networks. The basic idea is simple, given a source and destination, both in the road network, we select *access nodes* for both. This are nodes where we will switch from the road into the public transit network. A route can then be computed by using the road algorithm for the source to its access nodes, the transit algorithm for the access nodes of the source to the access nodes of the destination and finally the road algorithm again for the destinations access nodes to the destination. Note that this technique might not yield the shortest possible path anymore. Also, it does not allow an arbitrary alternation of transportation modes. However, we accept those limitations since the resulting algorithm is very generic and able to compute routes faster than without limitations. We will cover this technique in detail in **Section 5.3.2**.

Our final technique uses a modified version of ALT [31] as road algorithm and CSA [26] for the transportation network. The algorithms are presented in **Section 5.1.2** and **Section 5.2.1** respectively. We also develop a multi-modal variant of DIJKSTRA [21] which is able to compute the shortest route in a combined network with the possibility of changing transportation modes arbitrarily. It is presented in **Section 5.3.1** and acts as baseline to our final technique based on access nodes.

We compute access nodes by solving the NEAREST NEIGHBOR PROBLEM. For a given node in the road network its access nodes are then all nodes in the transit network which are in the *vicinity* of the road node. We explore a solution to this problem in **Section 4**.

**Section 3** starts by defining types of networks. We represent road networks by graphs only. For transit networks we provide a graph representation too. Both graphs can then be combined into a linked graph. The advantage of graph based models is that they are well studied and therefore we are able to use our multi-modal variant of DIJKSTRA to compute routes on them. However, we also propose a non-graph based representation for transit networks, a timetable. The timetable is used by CSA, an efficient algorithm for route planning on public transit networks. With that, our road and transit networks get incompatible and can not easily be combined. Therefore, we use the previously mentioned generic approach based on access nodes for this type of network.

Further, we implemented the presented algorithms in the COBWEB [44] project, which is an open-source multi-modal route planner. In **Section 6** we show our experimental results and compare the techniques with each other.

Section 2

# Preliminaries

Before we define our specific data models and problems we will introduce and formalize commonly reoccurring terms.

## 2.1 Graph

**Definition 1.** *A* graph $G$ *is a tuple* $(V, E)$ *with a set of nodes* $V$ *and a set of edges* $E \subseteq V \times \mathbb{R}_{\geq 0} \times V$*. An* edge $e \in E$ *is an ordered tuple* $(u, w, v)$ *with source node* $u \in V$*, a non-negative weight* $w \in \mathbb{R}_{\geq 0}$ *and a destination node* $v \in V$*.*

Note that **Definition 1** actually defines a *directed* graph, as opposed to an *undirected* graph where an edge like $(u, w, v)$ would be considered equal to the edge of opposite direction $(v, w, u)$ (compare to [27]). However, for transportation networks an undirected graph often is not applicable, for example due to one way streets or time dependent connections like trains which depart at different times for different directions.

In the context of route planning we refer to the weight $w$ of an edge $(u, w, v)$ as *cost*. It can be used to encode the length of the represented connection. Or to represent the time it takes to travel the distance in a given transportation mode.

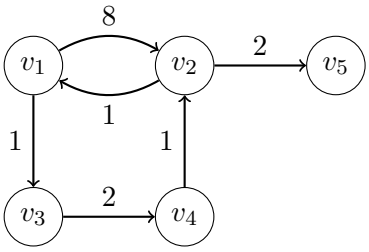


Fig. 2.1: Illustration of an example graph with five nodes and six edges.

As an example consider the graph $G = (V, E)$ with

$$V = \{v_1, v_2, v_3, v_4, v_5\} \text{ and}$$
$$E = \{(v_1, 8, v_2), (v_1, 1, v_3), (v_2, 1, v_1), (v_2, 2, v_5), (v_3, 2, v_4), (v_4, 1, v_2)\}.$$

which is illustrated by **Fig. 2.1**.

**Definition 2.** *Given a graph $G = (V, E)$ the function* src $: E \to V, ((u, w, v)) \mapsto u$ *gets the* source *of an edge. Analogously* dest $: E \to V, ((u, w, v)) \mapsto v$ *retrieves the* destination.

**Definition 3.** *A* path *in a graph $G = (V, E)$ is a sequence $p = e_1 e_2 e_3 \ldots$ of edges $e_i \in E$ such that*

$$\forall i : \text{dest}(e_i) = \text{src}(e_{i+1}).$$

*We write $e \in p$ if an edge $e$ is contained at least once in the path $p$. The* length *of a path is the amount of edges it contains, i.e. the length of the sequence. The* weight *or* cost *is the sum of its edges weights.*

*Let $k$ be the length of a path $p$, then we define:*

$$\begin{aligned} \text{src}(p) &= \text{src}(e_1) \\ \text{dest}(p) &= \text{dest}(e_k) \end{aligned}$$

*Given two paths $q_1 = e_1 \ldots e_k$ and $q_2 = e'_1 \ldots e'_l$ where $\text{dest}(e_k) = \text{src}(e'_1)$, the concatenation of both paths is a path*

$$p = e_1 \ldots e_k e'_1 \ldots e'_l$$

*with length $k + l$, also denoted by $p = q_1 q_2$.*

An example for a path in the graph $G$ would be

$$p = (v_1, 8, v_2)(v_2, 1, v_1)(v_1, 1, v_3).$$

Its length is 3 and it has a weight of 10.

## 2.2 Tree

**Definition 4.** *A* tree *is an graph $T = (V, E)$ with the following properties:*

1. *There is exactly one node $r \in V$ with no ingoing edges, called the* root*, i.e.*

$$\exists! r \in V \nexists e \in E : \text{dest}(e) = r.$$

2. *All other nodes $v$ have exactly one ingoing edge. The source $p$ of this edge is called* parent *of $v$ and $v$ is called* child *of $p$:*

$$\forall v \in V : v \neq r \Rightarrow \exists! e \in E : \text{dest}(e) = v.$$

**Definition 5.** *The* subtree *of a tree $T = (V, E)$ rooted at a node $r' \in V$ is a tree $T' = (V', E')$. $V' \subseteq V$ is the set of nodes that can be reached from $r'$. That is, all*

*nodes that are part of possible paths starting at $r'$. Likewise $E' \subseteq E$ is the set of edges restricted to the vertices in $V'$. The root of $T'$ is $r'$.*

**Definition 6.** *The* depth *of a node $v$ in a tree $T = (V, E)$, denoted by* $\text{depth}(v)$*, is defined as the amount of edges between $v$ and the root $r$. It is the length of the unique path $p$ starting at $r$ and ending at $v$.*

*The* height *of a tree is its greatest depth, i.e.*

$$\max_{v \in V} \text{depth}(v).$$

*And*

$$\text{children}(v) = \{c \in T | c \text{ child of } v\}.$$

Trees are hierarchical data-structures. Every node, except the root, has one parent. A node itself can have multiple children. Note that it is not possible to form a loop in a tree, i.e. a path that visits a node more than once. A node without children is called a *leaf*.

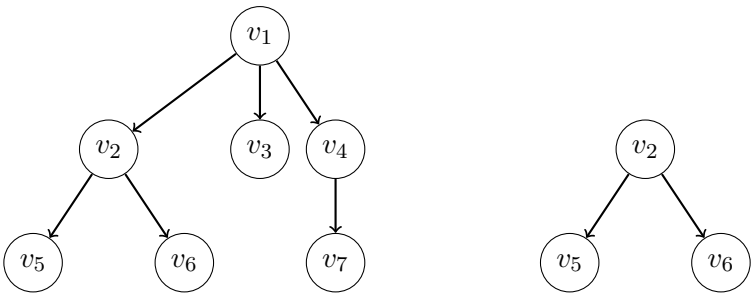


Fig. 2.2: Example of an unlabeled tree (left) and the subtree of $v_2$ (right).

**Fig. 2.2** shows a tree with 7 nodes. The node $v_1$ is the root; $v_5, v_6, v_3$ and $v_7$ are the leaves. The tree has a height of 2, the depth of $v_4$ is 1. The subtree rooted at $v_2$ only consists of the nodes $v_2, v_5$ and $v_6$.

## 2.3 Automaton

Automata are labeled graphs. They are used to represent states and the correlation between them.

**Definition 7.** *A* deterministic finite automaton *(DFA) A is a tuple $(Q, \sigma, \Delta, q_0, F)$ with*

*a set of states $Q$,*

*a set of labels $\sigma$, called* alphabet,

*a transition relation* $\Delta \subseteq Q \times \sigma \times Q$,

*an initial state* $q_0 \in Q$ *and*

*a set of accepting states* $F \subseteq Q$.

**Definition 8.** *A* word $w \in \Sigma^\star$ *is a finite sequence of letters*

$$w = a_0 a_1 a_2 \dots a_{k-1}$$

*with* $a_i \in \Sigma$ *and some* $k \in \mathbb{N}$. *The empty word is denoted by* $\varepsilon$.
*A word is called* accepted *iff*

1. $$\forall i : (q_i, a_i, q_{i+1}) \in \Delta,$$

   *for some* $q_i \in Q$,

2. $q_0$ *is the initial state of the automaton and*

3. *the last state is accepting, i.e.* $q_k \in F$.

*We say, the automaton* $A$ *accepts the word* $w$.

**Definition 9.** *The language* $\mathcal{L}(A)$ *of an automaton* $A$ *is defined as the set of accepted words:*

$$\mathcal{L}(A) = \{w \in \Sigma^\star | A \text{ accepts } w\}$$

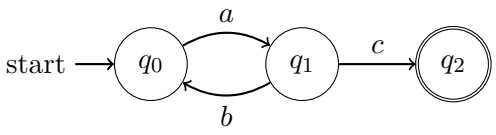


Fig. 2.3: Example of a deterministic finite automaton. $q_0$ is the initial state and $q_2$ is accepting.

For an example, refer to **Fig. 2.3** which accepts the language

$$(ab)^\star ac$$

denoting words with a finite sequence of $ab$, then one $a$ and one $c$. Such as:

$$ac$$
$$abac$$
$$ababac$$
$$abababac$$
$$\vdots$$

## 2.4 Metric

**Definition 10.** *A function $d : M \times M \to \mathbb{R}$ on a set $M$ is called a* metric *iff for all $x, y, z \in M$*

$$\begin{aligned} d(x,y) &\geq 0, && \textit{non-negativity} \\ d(x,y) = 0 &\Leftrightarrow x = y, && \textit{identity of indiscernibles} \\ d(x,y) &= d(y,x) \text{ and} && \textit{symmetry} \\ d(x,z) &\leq d(x,y) + d(y,z) && \textit{triangle inequality} \end{aligned}$$

*holds.*

**Definition 11.** *A* metric space *is a pair $(M, d)$ where $M$ is a set and $d : M \times M \to \mathbb{R}$ a metric on $M$.*

**Definition 12.** *Given a metric $d$ on a set $M$, the distance of a point $p \in M$ to a subset $Q \subseteq M$ is defined as the distance from $p$ to its nearest point in $Q$:*

$$d(p, Q) = \min_{q \in Q} d(p, q)$$

A metric is used to measure the distance between given locations. **Section 4** and **Section 5**, in particular **Section 5.1.2**, will make heavy use of this term.

There, we measure the distance between geographical locations given as pair of *latitude* and *longitude* coordinates. Latitude and longitude, often denoted by $\phi$ and $\lambda$, are real numbers in the ranges $(-90, 90)$ and $[-180, 180)$ respectively, measured in degrees. However, for convenience we represent them in radians. Both representations are equivalent to each other and can easily be converted using the ratio $360° = 2\pi$ rad.

A commonly used measure is the *as-the-crow-flies* metric, which is equivalent to the euclidean distance in the euclidean space. **Definition 13** defines an approximation of this distance on locations given by latitude and longitude coordinates. The approximation is commonly known as equirectangular projection of the earth [38]. Note that there are more accurate methods for computing the great-circle distance for geographical locations, like the haversine formula [40]. However, they come with a significant computational overhead.

**Definition 13.** *Given a set of coordinates* $M = \{(\phi, \lambda) | \phi \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right), \lambda \in [-\pi, \pi)\}$ *we define* asTheCrowFlies : $M \times M \to \mathbb{R}$ *such that*

$$((\phi_1, \lambda_1), (\phi_2, \lambda_2)) \mapsto \sqrt{\left((\lambda_2 - \lambda_1) \cdot \cos\left(\frac{\phi_1 + \phi_2}{2}\right)\right)^2 + (\phi_2 - \phi_1)^2} \cdot 6371000.$$

The value 6 371 000 refers to the approximate mean of the earth radius $R_\oplus$ in meters.

Section 3

# Models

This section defines the models we use for the different network types. We define a graph based representation for road and transit networks. Then both graphs are combined into a linked graph, making it possible to have one graph for the whole network. Afterwards an alternative representation for transit networks is shown.

## 3.1 Road graph

A road network typically is time-independent. It consists of geographical locations and roads connecting them with each other. We assume that a road can be taken at any time, with no time dependent constraints (see **Section 2** of [24]).

Modeling the network as graph is straightforward, **Definition 14** goes into detail.

**Definition 14.** *A* road graph *is a graph* $G = (V, E)$ *with a set of geographic coordinates*

$$V = \{(\phi, \lambda) | \phi \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right), \lambda \in [-\pi, \pi)\},$$

*for example road junctions. There is an edge* $(u, w, v) \in E$ *iff there is a road connecting the location* $u$ *with the location* $v$*, which can be taken in that direction. The weight* $w$ *of the edge is the average time needed to take the road from* $u$ *to* $v$ *using a car, measured in seconds.*

**Fig. 3.1** shows a contrived example road network with the corresponding road graph. Note that two way streets result in two edges, one edge for every direction the road can be taken.

Since edge weights are represented as average time it needs to take the road, it is possible to encode different road types. For example the average speed on a motorway is much higher than on a residential street. As such, the weight of an edge representing a motorway is much smaller than the weight of an edge representing a residential street.

While the example has exactly one node per road junction this must not always be the case. Typical real world data often consists of multiple nodes per road segment. However, **Definition 14** is still valid for such data as long as there are edges between the nodes if and only if there is a road connecting the locations.

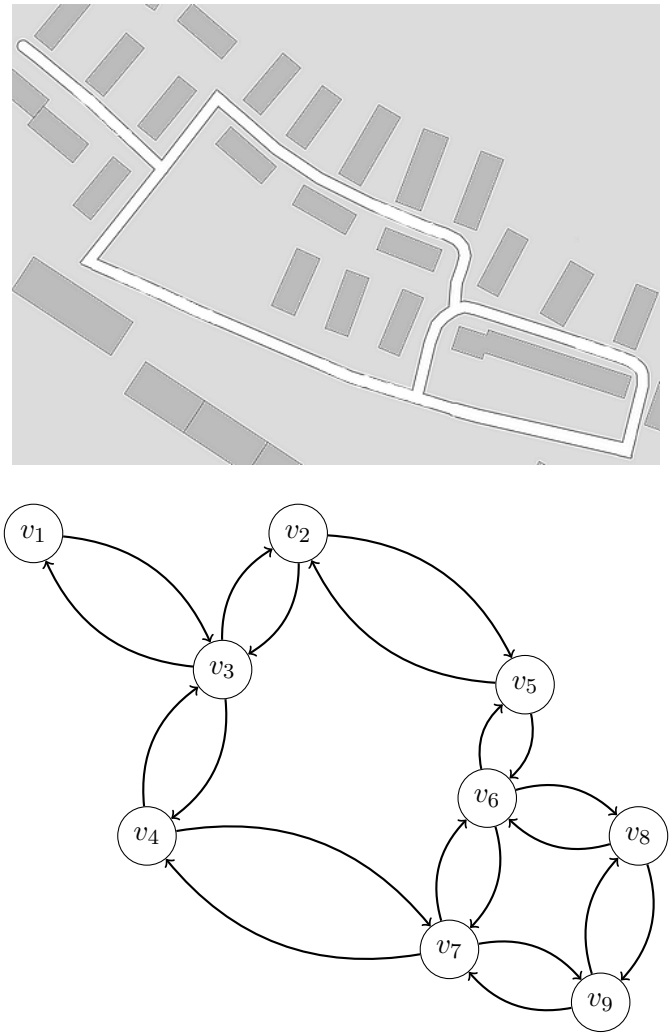


Fig. 3.1: Example of a road network with its corresponding road graph. White connections indicate roads, dark gray rectangles represent houses or other static objects. Geographical coordinates for each node, as well as edge weights are omitted in the graph illustration.

## 3.2 Transit graph

Transit networks can be modeled similar to road graphs. The key difference is that transit networks are time-dependent while road networks typically are not. For example an edge connecting *Freiburg main station* with *Karlsruhe main station* can not be taken at any time since trains and other transit vehicles only depart at certain times. The schedule might even change at different days.

The difficulty lies in modeling time dependence in a static graph. There are two common approaches to that problem (see [24, 36, 16]).

The first approach is called *time-dependent*. There, edge weights are not static numbers but piecewise continuous functions that take a date with time and compute the cost it needs to take the edge when starting at the given time. This includes waiting time. As an example assume an edge $(u, c, v)$ with the cost function $c$. The edge represents a train connection and the travel time is 10 minutes. However, the train departs at 10:15 *am*, while the starting time is 10:00 *am*. The cost function thus computes a waiting time of 15 minutes plus the travel time of 10 minutes. Resulting in an edge weight of 25 minutes.

The main problem with this model is that it makes pre-computations for route planning very difficult as the starting time is not known in advance.

The second approach, originally from [42], is called *time-expanded*. There, the idea is to remove any time dependence from the graph by creating additional nodes for every event at a station. A node then also has a time information next to its geographic location.

**Definition 15.** *A* time expanded transit graph *is a graph* $G = (V, E)$ *with a set of events at geographic coordinates*

$$V = \left\{ (\phi, \lambda, t) \middle| \phi \in \left( -\frac{\pi}{2}, \frac{\pi}{2} \right), \lambda \in [-\pi, \pi), t \text{ time} \right\},$$

*for example a train arriving or departing at a train station at a certain time.*

*For a node* $v \in V$, $v_\phi$ *and* $v_\lambda$ *denote its location and* $v_t$ *its time.*

*There is an edge* $(u, w, v) \in E$ *iff*

1. *there is a vehicle departing from* $u$ *at time* $u_t$ *which arrives at* $v$ *at time* $v_t$ *without stops in between, or*
2. $v$ *is the node at the same coordinates than* $u$ *with the smallest time* $v_t$ *that is still greater than* $u_t$*. This edge represents exiting a vehicle and waiting for another connection. That is*

$$\begin{aligned} \forall v' \in V \setminus \{v\} : v'_\phi = u_\phi \wedge v'_\lambda = u_\lambda \wedge v'_t \geq u_t \\ \Rightarrow v'_t - u_t > v_t - u_t. \end{aligned}$$

*The weight* $w$ *of an edge* $(u, w, v)$ *is the difference between both nodes times, that is*

$$w = v_t - u_t.$$

*Note that weights are still positive since* $v_t \geq u_t$ *always holds due to construction.*

**Definition 15** defines such a time expanded transit graph and **Fig. 3.2** shows an example. For simplicity it is assumed that the trains have no stops other than shown in the schedule. The schedule lists four trains:

| ⟶ | Freiburg Hbf | Offenburg | | Karlsruhe Hbf |
|---|---|---|---|---|
| | departure | arrival | departure | arrival |
| ICE 104 | 3:56 *pm* | 4:28 *pm* | 4:29 *pm* | 4:58 *pm* |
| RE 17024 | 4:03 *pm* | 4:50 *pm* | | |
| RE 17322 | | | 4:35 *pm* | 5:19 *pm* |
| ⟵ | arrival | departure | arrival | departure |
| ICE 79 | 8:10 *pm* | | | 7:10 *pm* |

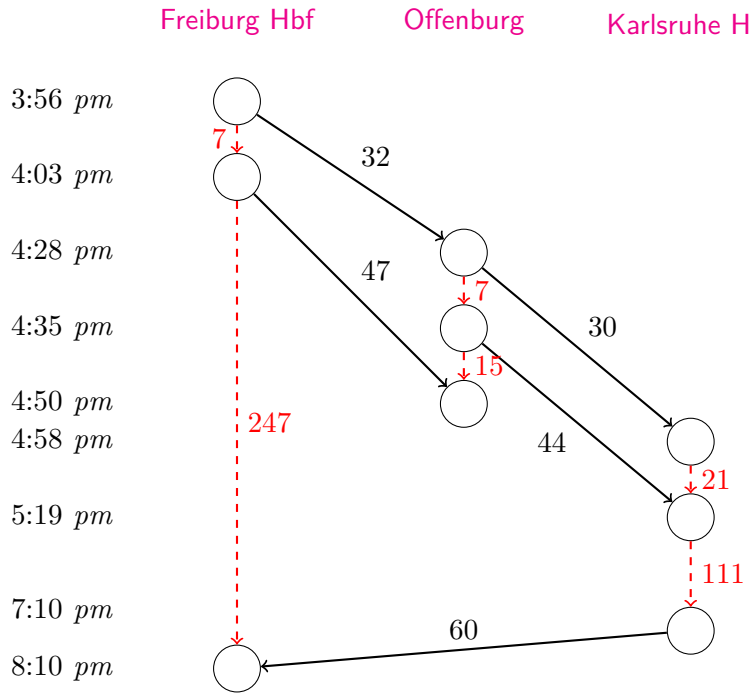


Fig. 3.2: Example of a transit network with its corresponding time expanded transit graph. The table shows an excerpt of a train schedule. Regular edges indicate a train connection and dashed edges waiting edges. Edge weights are measured in minutes.

1. The ICE 104 which travels from Freiburg Hbf to Karlsruhe Hbf via Offenburg,
2. the RE 17024 connecting Freiburg Hbf with Offenburg,
3. the RE 17322 driving from Offenburg to Karlsruhe Hbf and
4. ICE ICE 79 which travels in the opposite direction, connecting Karlsruhe Hbf with Freiburg Hbf without intermediate stops.

As seen in the example, the resulting graph has no time dependency anymore and is static, as well as all edge weights. The downside is that the graph size dramatically

increases as a new node is introduced for every single event. In order to limit the growth, we assume that a schedule is the same every day and does not change. In fact, most schedules are stable and often change only slightly, for example on weekends or at holidays. In practice hybrid models can be used for those exceptions.

However, the model still lacks an important feature. It does not represent *transfer buffers* [36, 16] yet. It takes some minimal amount of time to exit a vehicle and enter a different vehicle, possibly even at a different platform.

We model that by further distinguishing the nodes by arrival and departure events. In between we can then add transfer nodes which model the transfer duration. Therefore, the previous definition is adjusted and **Definition 16** is received.

**Definition 16.** *A* realistic time expanded transit graph *is a graph* $G = (V, E)$ *with a set of events at geographic coordinates*

$$V = \{(\phi, \lambda, t, e) | \phi \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right), \lambda \in [-\pi, \pi), t \text{ time}, e \in \{\textit{arrival}, \textit{departure}, \textit{transfer}\}\},$$

*for example a train arriving at a train station at a certain time.*

*A node* $(\phi, \lambda, t, e) \in V$ *is an* arrival node *if* $e = \textit{arrival}$*, analogously it is a* departure node *for* $e = \textit{departure}$ *and a transfer node for* $e = \textit{transfer}$*. For a node* $v \in V$*,* $v_\phi$ *and* $v_\lambda$ *denote its location,* $v_t$ *its time and* $v_e$ *its event type.*

*For every arrival node* $n$ *there must exist a transfer node* $m$ *at the same coordinates such that* $m_t = n_t + d$ *with* $d$ *being the average transfer duration at the corresponding stop.*

*There is an edge* $(u, w, v) \in E$ *iff*

1. $u_e = \textit{departure} \wedge v_e = \textit{arrival}$ *such that there is a vehicle departing from* $u$ *at time* $u_t$ *which arrives at* $v$ *at time* $v_t$ *without stops in between; or*

2. $u_e = \textit{arrival} \wedge v_e = \textit{departure}$ *such that* $u$ *and* $v$ *belong to the same connection. For example a train arriving at a station and then departing again; or*

3. $u_e = \textit{arrival} \wedge v_e = \textit{transfer}$ *such that* $v$ *is the first transfer node at the same coordinates whose time* $v_t$ *comes after* $u_t$*. That is*

$$\forall v' \in V \setminus \{v\} : v'_\phi = u_\phi \wedge v'_\lambda = u_\lambda \wedge v'_e = \textit{transfer} \wedge v'_t \geq u_t$$
$$\Rightarrow v'_t - u_t > v_t - u_t.$$

   *Such an edge represents exiting the vehicle and getting ready to enter a different vehicle; or*

4. $u_e = \textit{transfer} \wedge v_e = \textit{transfer}$ *such that* $v$ *is the first transfer node at the same coordinates whose time* $v_t$ *comes after* $u_t$*, representing waiting at a stop; or*

5. $u_e$ = *transfer* $\wedge$ $v_e$ = *departure* *such that* $u$ *is the last transfer node at the same coordinates whose time* $u_t$ *comes before* $v_t$*, i.e.*

$$\forall u' \in V \setminus \{u\} : u'_\phi = v_\phi \wedge u'_\lambda = v_\lambda \wedge u'_e = \textit{transfer} \wedge u'_t \leq v_t \\ \Rightarrow v_t - u'_t > v_t - u_t.$$

*An edge like this represents entering a different vehicle from a stop after transferring or waiting at the stop.*

*The weight* $w$ *of an edge* $(u, w, v)$ *is the difference between both nodes times, that is*

$$w = v_t - u_t.$$

**Fig. 3.3** shows how the transit graph from **Fig. 3.2** changes with transfer buffers.

The weight of edges connecting arrival nodes with transfer nodes is equal to the transfer duration, 5 minutes in the example. The transfer duration can be different for each edge. A transfer is now possible if the departure of the desired vehicle is after the arrival of the current vehicle plus the duration time. As seen in the example, edges connecting transfer nodes with departure nodes are present exactly in this case. A transfer from ICE 104 to RE 17322 in Offenburg is indicated by taking the edge to the first transfer node in Offenburg and then following the edge with cost 2 to the departure node of the train.

## 3.3 Link graph

In this section we examine how a road and a transit graph can be combined into a single graph such that all connections of the real network are preserved.

The approach is simple, selected nodes in the road network are connected to nodes of a certain stop in the transit network and vice versa. Since starting time is not known in advance, the graph must connect a road node to all arrival nodes of a stop (compare to [23]).

In order to not miss a connection, the transit graph must ensure that every connection starts with an arrival node. In **Fig. 3.3** this is not the case and all four trains start at a departure node. However, this is easily fixed by adding an additional arrival node to the beginning of every connection not starting with an arrival node already. The arrival nodes time is the same as the time of the departure node and both are connected by an edge with a weight of 0. **Definition 17** formalized the model.

**Definition 17.** *Assume a road graph* $R = (V_R, E_R)$*, a realistic time expanded transit graph* $T = (V_T, E_T)$ *where every connection in* $T$ *starts by an arrival node and a partial function* link : $V_R \nrightarrow M$ *where* $M$ *contains subsets* $S \subseteq V_T$*. For every element* $S \in M$ *with an arbitrary element* $s \in S$ *the following properties must hold:*

1. *All contained elements must be arrival nodes and have the same location than* $s$*,*

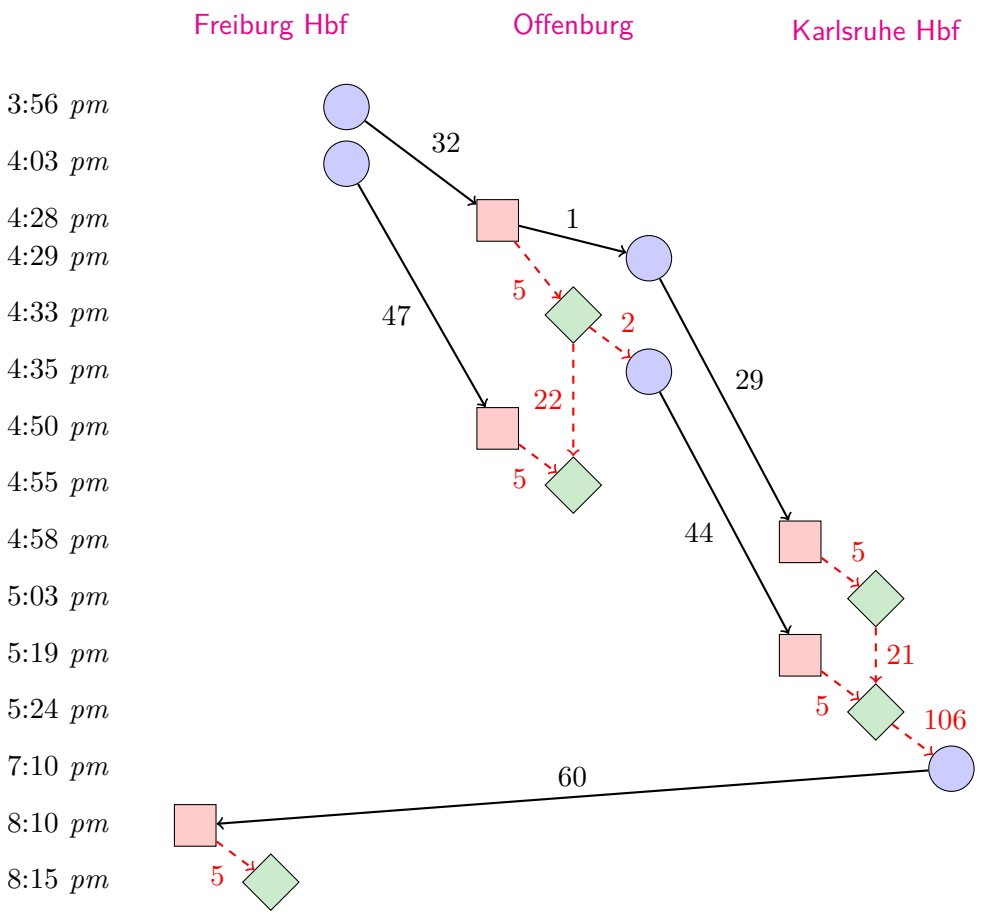


Fig. 3.3: Illustration of a realistic time expanded transit graph representing the schedule from **Fig. 3.2**. A transfer duration of 5 minutes is assumed at every stop. Rectangular nodes are arrival nodes, circular nodes represent departure nodes and diamond shaped nodes are transfer nodes. Regular edges indicate a train connection and dashed edges involve transfer nodes. Edge weights are measured in minutes.

*i.e.*

$$\forall s' \in S : s'_e = \textit{arrival} \wedge s'_\phi = s_\phi \wedge s'_\lambda = s_\lambda.$$

2. *The set must contain all arrival nodes at the location of $s$, i.e.*

$$\nexists v \in V_T \setminus S : v_e = \textit{arrival} \wedge v_\phi = s_\phi \wedge v_\lambda = s_\lambda.$$

*Then, a* link graph *is a graph $L = (V_R \cup V_T, E_R \cup E_T \cup E_L)$ with an additional set of link edges $E_L = V_R \times \mathbb{R}_{\geq 0} \times V_T$.*

*There is an edge* $(u, 0, v) \in E_L$ *iff* $\text{link}(u)$ *is defined and* $v \in \text{link}(u)$.

The function link can be obtained in different ways. For example by creating a mapping from a road node $u$ to a stop $S$ if $u$ is in the vicinity of $S$ according to the asTheCrowFlies metric.

Another straightforward possibility is to always connect a stop with the road node nearest to it. We will explore this problem in **Section 4**. An obvious downside of this approach is that the nearest road node might not always have a good connectivity to the road network. A solution consists in creating a road node at the coordinates of the stop as representative. The node can then be connected with all road nodes in the vicinity.

## 3.4 Timetable

Timetables [16] are non-graph based representations for transit networks. They consists of stops, trips, connections and footpaths.

**Definition 18.** *A* timetable *is a tuple* $(S, T, C, F)$ *with stops* $S$*, trips* $T$*, connections* $C$ *and footpaths* $F$*.*

A stop is a position where passengers can enter or exit a vehicle, for example a train station or bus stop. It is represented as geographical coordinate $(\phi, \lambda)$ with $\phi \in (-\frac{\pi}{2}, \frac{\pi}{2}), \lambda \in [-\pi, \pi)$.

A trip is a scheduled vehicle, like the ICE 104 in the example schedule of **Fig. 3.2** or a bus.

In contrast to a trip, a connection is only a segment of a trip without stops in between. For example the connection of the ICE 104 from Freiburg Hbf at 3:56 *pm* to Offenburg with arrival at 4:28 *pm*. It is defined as tuple $c = (s_{\text{dep}}, s_{\text{arr}}, t_{\text{dep}}, t_{\text{arr}}, o)$ with $s_{\text{dep}}, s_{\text{arr}} \in S$ representing the departure and arrival stop of the connection respectively. Analogously $t_{\text{dep}}$ is the time the vehicle departs at $s_{\text{dep}}$ and $t_{\text{arr}}$ when it arrives at $s_{\text{arr}}$. And $o \in T$ is the trip the connection belongs to.

Footpaths represent transfer possibilities between stops and are formalized as ordered tuple $(s_{\text{dep}}, d, s_{\text{arr}})$ with $s_{\text{dep}}, s_{\text{arr}} \in S$ being the stops the footpath connects. The duration it needs to take the path by foot is represented by $d$, measured in seconds. Together with the set of stops $S$ the footpaths build a graph $G = (S, F)$, representing directed edges between stops.

We require the following for the footpaths:

1. Footpaths must be transitively closed, that is

$$\exists (a, d_1, b), (b, d_2, c) \in F \Rightarrow (a, d_3, c) \in F$$

for arbitrary durations $d_1, d_2, d_3$.

2. The triangle inequality must hold for all footpaths:

$$\exists(a, d_1, b), (b, d_2, c) \in F \Rightarrow \exists(a, d_3, c) \in F : d_3 \leq d_1 + d_2$$

3. Every stop must have a self-loop footpath, i.e.

$$\forall s \in S \Rightarrow (s, d, s) \in F.$$

   The duration $d$ models the transfer time at this stop, as already introduced in **Section 3.2**.

The first property can easily make the set of footpaths huge. However, it is necessary for our algorithms that the amount of footpaths stays relatively small. In practice, we therefore connect each stop only to stops in its vicinity and then compute the transitive closure to ensure that the model is transitively closed.

To familiarize more with the model, we take a look at the schedule from **Fig. 3.2** again. The corresponding timetable consists of:

$$S = \{f, o, k\},$$

where $f, o, k$ represent Freiburg Hbf, Offenburg and Karlsruhe Hbf respectively;

$$T = \{t_{104}, t_{17024}, t_{17322}, t_{79}\},$$

representing the four trains ICE 104, RE 17024, RE 17322 and ICE 79; the connections

$$\begin{aligned}
&(f, o, 3{:}56\ pm, 4{:}28\ pm, t_{104}),\\
&(o, k, 4{:}29\ pm, 4{:}58\ pm, t_{104}),\\
&(f, o, 4{:}03\ pm, 4{:}50\ pm, t_{17024}),\\
&(o, k, 4{:}35\ pm, 5{:}19\ pm, t_{17322}),\\
&(k, f, 7{:}10\ pm, 8{:}10\ pm, t_{79})
\end{aligned}$$

and at least the footpaths

$$\begin{aligned}
&(f, 300, f),\\
&(o, 300, o),\\
&(k, 300, k)
\end{aligned}$$

for transferring at the same stop with a duration of 300 seconds (5 minutes).

If we would decide that Offenburg is reachable from Freiburg Hbf by foot, and analogously Karlsruhe Hbf from Offenburg, we would also need to add a footpath connecting Freiburg Hbf directly with Karlsruhe Hbf. Else the footpaths would not be transitively closed anymore.

Section 4

# Nearest neighbor problem

In this section we introduce the Nearest Neighbor Problem, also known as nearest neighbor search (NNS). First, we define the problem. Then a short overview of related research is given, after which we elaborate on a solution called Cover Tree [18].

**Definition 19.** *Given a metric space* $(M, d)$ *(see* ***Definition 11****) with* $|M| \geq 2$ *and a point* $x \in M$*, the* nearest neighbor problem *asks for finding a point* $y \in M$ *such that*

$$y = \underset{y' \in M \setminus \{x\}}{\arg\min}\, d(x, y').$$

*The point* $y$ *is called* nearest neighbor *of* $x$*.*

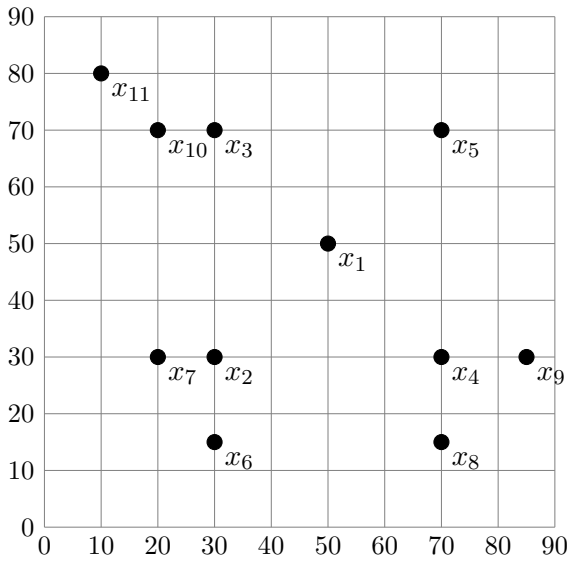


Fig. 4.1: Grid showing eleven points in the cartesian plane $\mathbb{R}^2$.

For following examples the toy data set shown in **Fig. 4.1** is introduced. It consists of

the points

$$\begin{aligned} x_1 &= (50, 50), \\ x_2 &= (30, 30), \\ x_3 &= (30, 70), \\ x_4 &= (70, 30), \\ x_5 &= (70, 70), \\ x_6 &= (30, 15), \\ x_7 &= (20, 30), \\ x_8 &= (70, 15), \\ x_9 &= (85, 30), \\ x_{10} &= (20, 70), \\ x_{11} &= (10, 80). \end{aligned}$$

All points are elements of the cartesian plane $\mathbb{R}$. The euclidean distance $d$ is chosen as metric on this set. For two dimensions it can be defined as:

$$d : \mathbb{R}^2 \times \mathbb{R}^2, ((x_1, y_1), (x_2, y_2)) \mapsto \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Informally $d$ computes the *ordinary* straight-line distance between two points.

The nearest neighbor of $x_5$ is $x_1$, as

$$\begin{aligned} d(x_5, x_1) &= \sqrt{(50 - 70)^2 + (50 - 70)^2} \\ &= \sqrt{800} \end{aligned}$$

is smaller than all other distances to $x_5$, like

$$\begin{aligned} d(x_5, x_4) &= \sqrt{(70 - 70)^2 + (30 - 70)^2} \\ &= \sqrt{1600}. \end{aligned}$$

On the other hand, $x_1$ has four smallest neighbors:

$$d(x_1, x_2) = d(x_1, x_3) = d(x_1, x_4) = d(x_1, x_5)$$

Any of them is a valid solution to the nearest neighbor problem for $x_1$.

The search for a nearest neighbor is a well understood problem [12, 11] and has many applications. Without restrictions, solving the problem on general metrics is proven to require $\Omega(n)$ time [12], where $n$ is the amount of points.

Typical approaches divide the space into regions, exploiting properties of the metric space. Common examples include K-D TREEs [17], VP TREEs [46], BK-TREEs [19] and COVER TREEs [18].

The problem also has a lot of variants. We elaborate on two of them:

**Definition 20.** *The* k-nearest neighbors *of a point $x \in M$ are the $k$ closest points $\{y_1, y_2, \ldots, y_k\} \subseteq M$ to $x$. That is*

$$\begin{aligned} y_1 &= \operatorname*{arg\,min}_{y' \in M \setminus \{x\}} d(x, y'), \\ y_2 &= \operatorname*{arg\,min}_{y' \in M \setminus \{x, y_1\}} d(x, y'), \\ &\vdots \\ y_k &= \operatorname*{arg\,min}_{y' \in M \setminus \{x, y_1, \ldots, y_{k-1}\}} d(x, y'). \end{aligned}$$

**Definition 21.** *The* k-neighborhood *of a point $x \in M$ is the set*

$$\{y \in M \setminus \{x\} | d(x, y) \leq k\}.$$

## 4.1 Cover tree

**Definition 22.** *A* cover tree *$T$ on a metric space $(M, d)$ is a leveled tree $(V, E)$.*

*The root is placed at the greatest level, denoted by $i_{max} \in \mathbb{Z}$. The level of a node $v \in V$ is*

$$\mathrm{lvl}(v) = i_{max} - \mathrm{depth}(v).$$

*The lowest level is denoted by $i_{min}$. Every node $v \in V$ is associated with a point $m \in M$. We write $\mathrm{assoc}(v) = m$. Nodes of a certain level form a cover of points in $M$. A cover for a level $i$ is defined as*

$$C_i = \{m \in M | \exists v \in V : \mathrm{lvl}(v) = i \wedge \mathrm{assoc}(v) = m\}.$$

*The following properties must hold*

1. *For a level $i$ there must not exist nodes which are associated with the same point $m \in M$:*

   $$\nexists v, v' \in V : i = \mathrm{lvl}(v) = \mathrm{lvl}(v') \wedge v \neq v' \wedge \mathrm{assoc}(v) = \mathrm{assoc}(v')$$

   *So each point can at most appear once per level.*

2. *$C_i \subset C_{i-1}$. This ensures that, once a point was associated with a node in a level, it appears in all lower levels too.*

3. *Points are covered by their parents:*

   $$\forall p \in C_{i-1} \exists q \in C_i : d(p, q) < 2^i$$

*and the node* $v_p$ *with* $\text{lvl}(v_p) = i \wedge \text{assoc}(v_p) = p$ *is the parent of the node* $v_q$ *with* $\text{lvl}(v_q) = i - 1 \wedge \text{assoc}(v_q) = q$.

4. *Points in a cover* $C_i$ *have a separation of at least* $2^i$, *i.e.*

$$\forall p, q \in C_i : p \neq q \Rightarrow d(p, q) > 2^i.$$

A cover tree [18] has interesting distance properties on its nodes which allows for efficient retrieval of nearest neighbors. The general approach is straightforward. Given a node $v$ in the tree placed at level $i$, we know that all nodes of the subtree rooted at $v$ are associated with points inside a distance of at most $2^i$. This means that, if we search for a nearest neighbor, and traverse to a node $v$ in the tree, all nodes underneath $v$ are relatively close to $v$. So, if we already have a candidate for a nearest neighbor, with a distance of $d$ and $v$ is already further away than $d + 2^i$; $v$ and all nodes in its subtree can not improve the distance.

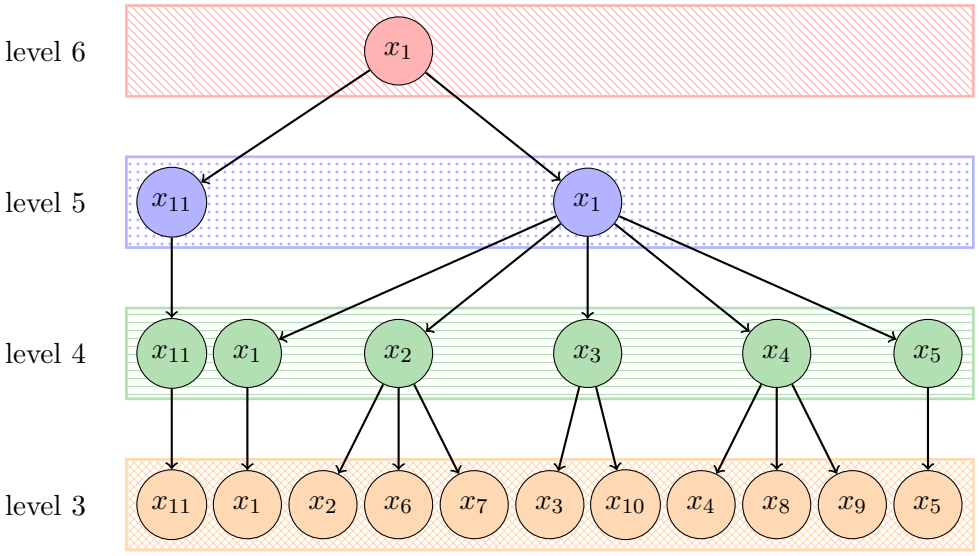


Fig. 4.2: Cover tree for the data set of **Fig. 4.1**. Nodes are vertically grouped by their levels and highlighted accordingly.

**Fig. 4.2** shows a valid cover tree for the toy example illustrated by **Fig. 4.1**. The covers are

$$\begin{aligned}
C_6 &= \{x_1\}, \\
C_5 &= \{x_1, x_{11}\}, \\
C_4 &= \{x_1, x_2, x_3, x_4, x_5, x_{11}\}, \\
C_3 &= \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}\}.
\end{aligned}$$

Clearly the first property holds, there is no level where a $x_i$ is associated with a node

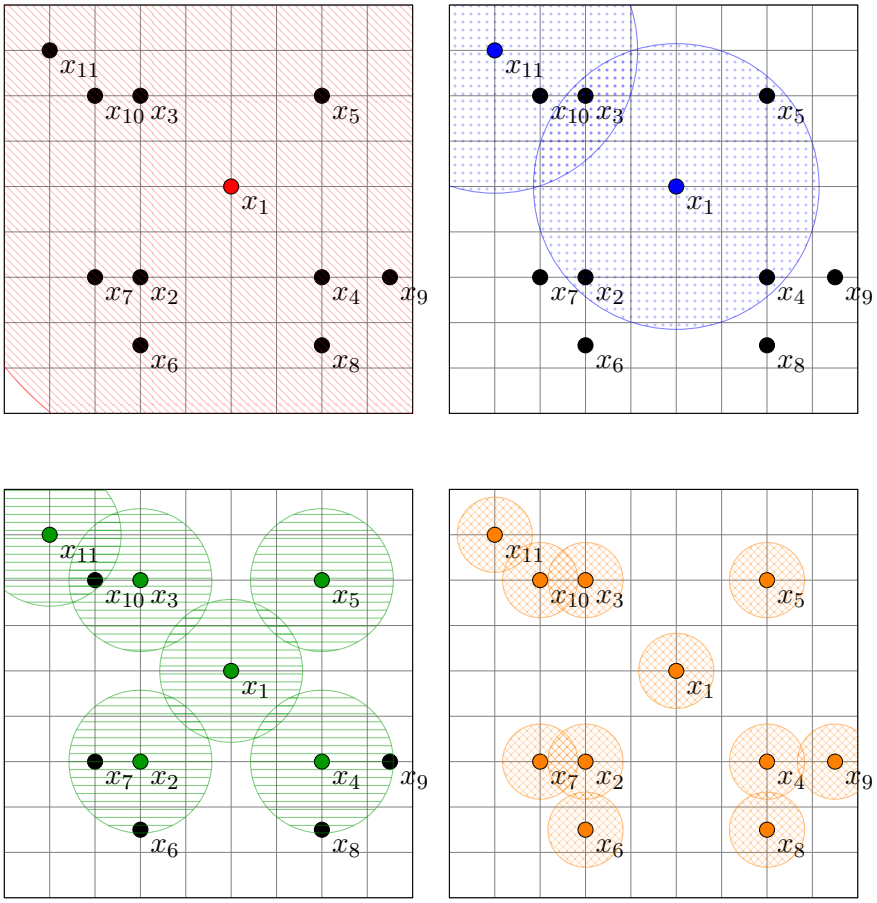


Fig. 4.3: Figure that shows the separation property for each level of the cover tree shown by **Fig. 4.2**. The levels are highlighted in the same manner than in the previous example. The levels are $6, 5, 4$ and $3$ from top left to bottom right. The radii around the points have a size of $2^6, 2^5, 2^4$ and $2^3$.

more than once. The second property holds too, it is

$$C_6 \subset C_5 \subset C_4 \subset C_3.$$

For the last two properties we take a look at **Fig. 4.3**. It illustrates the fourth property. The property states that all points in a cover $C_i$ must have a distance of at least $2^i$ to each other. For level 6 this is trivial since the set only contains $x_1$. For level 5 it must hold that

$$d(x_1, x_{11}) = 50 > 32 = 2^5,$$

which is true. If this would not be the case, the figure would show the nodes included inside the circle around the other node. Analogously all nodes in $C_4$ and $C_3$ are separated enough from each other.

The third property can easily be confirmed using the figure too. It states that a node in level $i-1$ must be closer than $2^i$ to its parent. Obviously this holds for $x_1$ and $x_{11}$ in level 5, as a radius of $2^6$ around their parent $x_1$ covers all nodes. Likewise are $x_1, x_2, x_3, x_4$ and $x_5$ included in the circle around their parent $x_1$ with radius $2^5$.

Note that it is not necessary that a node covers its whole subtree in its level. As example, we refer to $x_1$ in level 5 which does not cover $x_{10}$, as $d(x_1, x_{10}) > 2^5$, though it is part of the subtree rooted at $x_1$. The third property only demands that a parent covers all its direct children, not grandchildren or similar.

**Algorithm 1:** Inserting a point into a cover tree operating on a metric space $(M, d)$.

**input :** point $p \in M$, candidate cover set $Q_i \subseteq C_i$, level $i$
**output:** true if $p$ was inserted at level $i-1$, false otherwise

```
1  Q ← {children(q) | q ∈ Q_i};
2  if d(p, Q) > 2^i then
3      return false ;                                  // Check separation
4  else
5      Q_{i-1} ← {q ∈ Q | d(p, q) ≤ 2^i};              // Covering candidates
6      if ¬insert(p, Q_{i-1}, i − 1) ∧ d(p, Q_i) ≤ 2^i then
7          pick any q ∈ Q_i : d(p, q) ≤ 2^i;
8          append q as child to q;
9          return true;
10     else
11         return false;
```

The cover tree is constructed using **Algorithm 1** with the maximal level $i_{\text{max}}$ and the cover set $C_k$ which only consists of the root. The algorithm is stated recursively, but can easily be implemented without recursion by descending the levels and only following relevant candidates.

A point $p$ can be appended in level $i-1$ to a parent $q$ in level $i$ if the point has enough separation to all other nodes in this level, meaning more than $2^{i-1}$, and is covered by the parent, that is a distance of less than $2^i$. The algorithm searches such a point by descending the levels, computing the separation and appending it to a node if it also covers the point.

A search for a nearest neighbor follows a similar approach. **Algorithm 2** starts at the root and traverses the tree by following the children. The candidate set is refined by

**Algorithm 2:** Searching a nearest neighbor in a cover tree operating on a metric space $(M, d)$.

**input :** point $p \in M$
**output:** a nearest neighbor to $p$ in $M$

$Q_{i_{max}} \leftarrow C_{i_{max}}$;
**for** $i$ *from* $i_{max}$ *to* $i_{min}$ **do**
  $Q \leftarrow \{\texttt{children}(q) | q \in Q_i\}$;
  $Q_{i-1} \leftarrow \{q \in Q | \texttt{d}(p, q) \leq \texttt{d}(p, Q) + 2^i\}$;
**return** $\arg\min_{q \in Q_{i_{min}}} \texttt{d}(p, q)$;

only following children which are closer than

$$d(p, Q) + 2^i.$$

There, the distance to the set represents the distance of the currently best candidate. Nodes in the subtree rooted at a child can maximally be $2^i$ closer than the child itself. Therefore, take a look at **Fig. 4.3** where $x_2$ is maximally $2^5$ closer to $x_7$ than $x_1$, else it would not be covered by its parent $x_1$. Because of that the algorithm only follows children which can have nodes in their subtree that improve over the currently best candidate. Other children are rejected.

Note that the algorithm must track down all levels, as another node could show up in the lowest level because of the separation property.

**Algorithm 3:** Searching the $k$-nearest neighbors in a cover tree operating on a metric space $(M, d)$.

**input :** point $p \in M$, amount $k \in \mathbb{N}$
**output:** $k$-nearest neighbors to $p$ in $M$

$Q_{i_{max}} \leftarrow C_{i_{max}}$;
**for** $i$ *from* $i_{max}$ *to* $i_{min}$ **do**
  $Q \leftarrow \{\texttt{children}(q) | q \in Q_i\}$;
  perform a $k$-partial sort of $Q$, ascending in $\texttt{d}(p, q)$;
  let $q'$ be the $k$-th element of $Q$;
  $Q_{i-1} \leftarrow \{q \in Q | \texttt{d}(p, q) \leq \texttt{d}(p, q') + 2^i\}$;
perform a $k$-partial sort of $Q_{i_{min}}$, ascending in $\texttt{d}(p, q)$;
**return** first $k$ elements of $Q_{i_{min}}$;

The cover tree can also be used to efficiently compute the $k$-nearest neighbors or the

**Algorithm 4:** Computing the $k$-neighborhood by using a cover tree which operates on a metric space $(M, d)$.

**input :** point $p \in M$, radius $k \in \mathbb{R}_{\geq 0}$
**output:** $k$-neighborhood of $p$ in $M$

$Q_{i_{max}} \leftarrow C_{i_{max}}$;
**for** $i$ *from* $i_{max}$ *to* $i_{min}$ **do**
  $Q \leftarrow \{\texttt{children}(q) | q \in Q_i\}$;
  $Q_{i-1} \leftarrow \{q \in Q | \texttt{d}(p, q) \leq k + 2^i\}$;
**return** $\{q \in Q_{i_{min}} | \texttt{d}(p, q) \leq k\}$;

$k$-neighborhood. In order to compute the $k$-nearest neighbors, **Algorithm 3** extends the range bound from the currently best candidate to the $k$-th best candidate. Likewise does **Algorithm 4** extend the bound to the given range $k$ instead of involving candidate distances.

For other operations and a detailed analysis of the cover tree, as well as its complexity and a comparison against other techniques, refer to [18].

Section 5

# Shortest path problem

For route planning, routes through a network must be optimized in regards to one or even many criteria. A common criteria is the *travel time*. Others include cost, number of transfers or restrictions in transportation types.

In this chapter, we will first give an informal description of the Earliest Arrival Problem. Followed by the Shortest Path Problem, which is equivalent to the Earliest Arrival Problem for our graph based network representations.

Then, we introduce algorithms for solving the problem. First, for time-independent networks, then for time-dependent. Afterwards, we explain two solutions for combined networks, using multiple transportation modes. There, the problem description slightly changes by adding transportation mode restrictions.

**Definition 23.** *The earliest arrival problem asks for finding a* route *in a network with following properties*

1. *The route must start at $s$ and end at $t$.*
2. *The departure time at $s$ is $\tau$.*
3. *All other applicable routes must have a greater travel time, i.e. arrive later at $t$.*

*Points $s$ and $t$ are given source and target points in the network respectively. $\tau$ is the desired departure time, it may be ignored for a time-independent network.*

**Definition 24.** *Given a graph $G = (V, E)$, source and target nodes $s, t \in V$ and a desired departure time $\tau$, the shortest path problem asks for a path $p$ (see **Definition 3**) which*

1. *begins at $s$ and ends at $t$,*
2. *has the smallest weight of all applicable paths.*

*The arrival time at $t$ is $\tau$ plus the weight of $p$. In a time-dependent graph $\tau$ must be used to ensure correct edge weights. The path $p$ is called* shortest path.

Additionally, we consider a special variant of the shortest path problem:

**Definition 25.** *The many-to-one shortest path problem is a variation of the shortest path problem where the source consists of a set of source nodes $S \subseteq V$.*

*The problem asks for the path $p$ that starts at the source $s \in S$ which minimizes the path weight.*

## 5.1 Time-independent

Route planning in time-independent networks is a very well studied problem. Many efficient solutions to the shortest path problem exists. We introduce a very basic algorithm, Dijkstra and a simple improvement based on heuristics, A$^\star$.

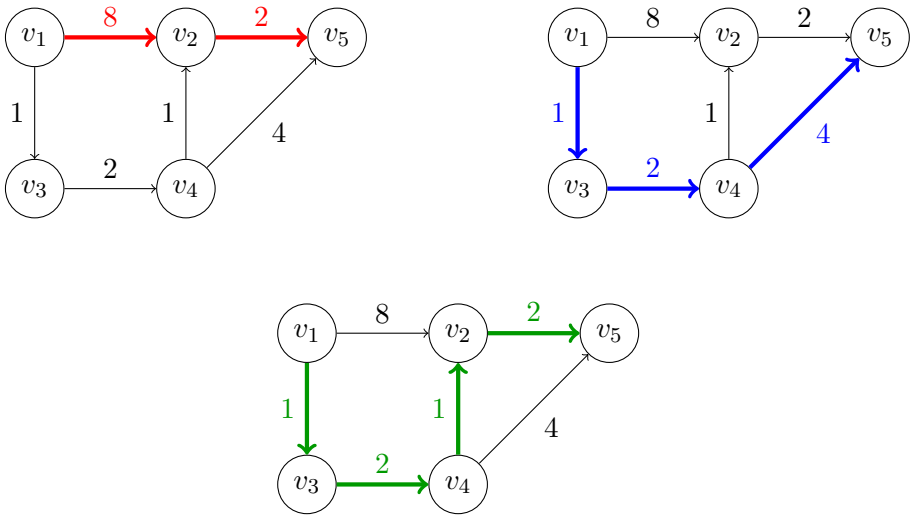


Fig. 5.1: Example for a time independent network, represented by a road graph. The figure shows three paths from $v_1$ to $v_5$. From top left to bottom right, the path weights are 10, 7 and 6. The last example represents the shortest path from $v_1$ to $v_5$.

The network shown by **Fig. 5.1** acts as toy example for this section.

### 5.1.1 Dijkstra

Dijkstra [21] is a simple approach to solving the shortest path problem. It can be viewed as the logical extension of breadth-first search (BFS) [21] in weighted graphs. The algorithm revolves around a priority queue where it stores neighboring nodes, sorted by their shortest path cost. In each round, the node with the smallest shortest path cost is *relaxed*. That is, all its neighboring, not already relaxed, nodes are added to the queue. The algorithm terminates as soon as the target node has been relaxed. **Algorithm 5** gives a formal description.

To familiarize with the algorithm, we step through the execution for the graph shown by **Fig. 5.1**, with $v_1$ as source and $v_5$ as target node.

The `dist` function, often implemented as array, stores the tentative shortest path weight to the given node. `prev` is used for path extraction at the end, it stores the parent nodes used for the shortest paths represented by `dist`. The algorithm starts by initializing both collections with default values. Initially, the distance to all nodes, except the source, is unknown. Thus, $\infty$ is used for them. $Q$ represents the list of nodes

**Algorithm 5:** Dijkstra's algorithm for computing shortest paths in time-independent graphs.

**input :** graph $G = (V, E)$, source $s \in V$, target $t \in V$
**output:** shortest path from $s$ to $t$

```
// Initialization
for v ∈ V do
    dist(v) ← ∞;
    prev(v) ← undefined;
dist(s) ← 0;
Q ← {s};
// Compute shortest paths
while Q is not empty do
    u ← arg min_{u'∈Q} dist(u');
    Q ← Q \ {u};
    if u == t then
        break;
    // Relax u
    for outgoing edge (u, w, v) ∈ E do
        currentDist ← dist(u) + w;
        if currentDist < dist(v) then
            // Improve distance by using this edge
            dist(v) ← currentDist;
            prev(v) ← u;
            Q ← Q ∪ {v};
// Extract path by backtracking
p ← empty path;
u ← t;
while prev(u) ≠ undefined do
    w ← dist(u) − dist(prev(u));
    prepend (prev(u), w, u) to p;
    u ← prev(u);
prepend s to p;
return p;
```

that need to be processed, usually implemented as priority queue. Initially, it only holds the source node $s$.

In the example $Q$ starts as $\{v_1\}$. The algorithm then relaxes $v_1$ and stores distances to its neighbors:

$$\begin{aligned} \texttt{dist}(v_2) &= 8 \quad \texttt{prev}(v_2) = v_1, \\ \texttt{dist}(v_3) &= 1 \quad \texttt{prev}(v_3) = v_1 \end{aligned}$$

Additionally, the queue $Q$ is updated, it is

$$Q = \{v_2, v_3\}.$$

The next iteration of the loop starts and the node with the smallest distance is chosen, i.e. $v_3$. The node is relaxed and we receive

$$\begin{aligned} \texttt{dist}(v_4) &= 3 \quad \texttt{prev}(v_4) = v_3, \\ Q &= \{v_2, v_4\}. \end{aligned}$$

The next node is $v_4$, yielding

$$\begin{aligned} \texttt{dist}(v_2) &= 4 \quad \texttt{prev}(v_2) = v_4, \\ \texttt{dist}(v_5) &= 7 \quad \texttt{prev}(v_5) = v_4, \\ Q &= \{v_2, v_5\}. \end{aligned}$$

Note that $v_4$ improves the distance to $v_2$. The previous values for $v_2$ are overwritten and the tentative shortest path to $v_2$ uses $(v_4, 1, v_2)$ and not $(v_1, 8, v_2)$ anymore. In the next round $v_2$ is relaxed which improves the distance to $v_5$:

$$\begin{aligned} \texttt{dist}(v_5) &= 6 \quad \texttt{prev}(v_5) = v_2, \\ Q &= \{v_5\}. \end{aligned}$$

The only node left is the target node $v_5$ now. It is relaxed and the loop terminates. The algorithm backtracks the parent pointers

$$\begin{aligned} \texttt{prev}(v_5) &= v_2, \\ \texttt{prev}(v_2) &= v_4, \\ \texttt{prev}(v_4) &= v_3, \\ \texttt{prev}(v_3) &= v_1, \\ \texttt{prev}(v_1) &= \text{undefined} \end{aligned}$$

and constructs the shortest path

$$p = (v_1, 1, v_3)(v_3, 2, v_4)(v_4, 1, v_2)(v_2, 2, v_5)$$

which is the path shown by the last example in the figure.

### 5.1.2 A* and ALT

An important observation of Dijkstra is that, if it settles the shortest path distance to a node, then, all nodes which are closer to the source, were already settled in a previous round.

Moreover, the algorithm explores the graph in all directions equally. It has no sense of *goal direction*.

The A* algorithm [31] is a simple extension of Dijkstra which improves its efficiency by steering the exploration more towards the target. **Fig. 5.2** illustrates this by comparing the *search space* of both algorithms. The search space of A* is smaller and much more directed to the target node $t$.

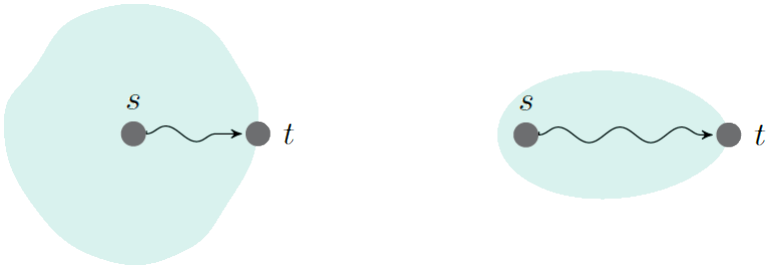


Fig. 5.2: Schematic illustration of a query processed by Dijkstra (left) and A* (right). The highlighted areas indicate the *search space*, i.e. the nodes the algorithm has explored already. The illustration is from [16].

Unfortunately, computing the exact goal direction is as hard as computing the shortest path to the target. Therefore, a heuristic is used to approximate the direction. The choice of the heuristic heavily depends on the underlying network. In the worst case, a heuristic may not improve over Dijkstra and the same search space is received. In the best case, the algorithm explores only the nodes on the shortest path.

Such a heuristic must fulfill two properties, formulated by **Definition 26**.

**Definition 26.** *Given a graph $G = (V, E)$, a metric* `dist` *on $V$ (see **Definition 10**), a* heuristic *is a function $h : V \times V \to \mathbb{R}_{\geq 0}$ which approximates* `dist`*. The heurstic $h$ must be*

1. admissable, *i.e. never overestimate:*

$$\forall u, t \in V : h(u, t) \leq \texttt{dist}(u, t)$$

2. monotone, *i.e. satisfy the triangle inequality:*

$$\forall t \in V \, \forall (u, w, v) \in E : h(u, t) \leq w + h(v, t)$$

Given such a heuristic $h$, the A* algorithm is received by adjusting **line 7** of **Algorithm**

5 to

$$u \leftarrow \arg\min_{u' \in Q} \texttt{dist}(u') + h(u', t).$$

This will prefer nodes that are estimated to be closer to the target before others. By that, the algorithms search space first expands into a direction that minimizes the distance according to the heuristic $h$.

A common choice for a simple heuristic is the *as-the-crow-flies* metric (see **Definition 13**). The properties are easily verified. A theoretically shortest path has the shortest possible distance and uses the fastest available transportation mode. This is exactly the path represented by the *straight-line* distance, computed by the *as-the-crow-flies* metric. It can thus never overestimate. It is also trivially monotone since it is a metric, i.e. the triangle inequality holds for all elements.

A heuristic is a good choice if it approximates the actual shortest path distance well. As such, the *as-the-crow-flies* heuristic works well on networks with a high connectivity in all directions. For example a residential area of a city without one way streets. Unfortunately, in road networks, the common case is to first drive into the opposite direction in order to reach a fast highway. This even gets worse on networks where the importance of nodes heavily differ, such as public transit networks. For train networks, the typical case is that one first needs to travel to a main station. This is obviously due to a main station having a much better connectivity and faster trains available. Because of that, the effectiveness of *as-the-crow-flies* is very limited on such networks.

The *landmark heurstic* partially solves the issue. An A$^\star$ algorithm using the landmark heuristic is called ALT [31], which stands for *landmarks and triangle inequality*.

The heuristic provides a more generic approach by approximating the distance between nodes $u$ and $v$ by using precomputed distances with pre-determined nodes $l$, called *landmarks*.

**Definition 27.** *Given a set of landmarks $L \subseteq V$, the heuristic* landmarks *is defined by*

$$\text{landmarks}(u, v) = \max_{l \in L} \left( \max\{\texttt{dist}(u, l) - \texttt{dist}(v, l), \texttt{dist}(l, v) - \texttt{dist}(l, u)\} \right).$$

Obviously, the heuristic improves if the set of landmarks is increased. However, actual shortest path distances from all landmarks to all other nodes in the graph must be precomputed. With an increasing amount of landmarks the precomputation might not be feasible anymore because it takes too long or consumes too much space. Note that if $L = V$, the heuristic becomes the actual shortest path distance function, i.e. landmarks $= \texttt{dist}$.

In practice, an amount between 20 and 50 randomly chosen nodes seems to be a good compromise. Refer to [31] for a detailed analysis.

The computation of the actual shortest path distances, to and from the landmarks, can be done by using Dijkstra. But, instead of running the algorithm for all pairs of

nodes, the distances can be obtained with two runs only. Therefore, the algorithm is slightly modified by dropping **lines 9** and **10**, such that the algorithm relaxes the whole network. By that, a single run of DIJKSTRA with a landmark $l$ as source, computes the distances $\texttt{dist}(l, v)$ to all nodes $v$ in the network. By reversing the graph, i.e. edges $(u, w, v)$ become $(v, w, u)$, the distances to the landmarks can be obtained analogously with $l$ as source again. Depending on the graph implementation, reversal can be done in $\mathcal{O}(1)$ by only implicitly reversing the edges.

## 5.2 Time-dependent

Approaches designed for time-independent networks, such as ALT, have an important drawback. Optimization is always done on assuming that edge costs are constant. However, in a time-dependent network, this is not the case. The weight of an edge is dependent on the departure time, which is not known in advance.

DIJKSTRA and its variants A$^\star$ and ALT can easily be adapted to also work in time-dependent networks by taking the departure time into consideration when computing the weight of an edge. However, their effectiveness is very limited. Nonetheless, they where used for a long time for time-dependent networks too. With increasing research on route planning in time-dependent networks, more effective algorithms, such as TRANSFER PATTERNS [15] and CSA [26], were developed. Many of them do not use graphs and prefer data-structures that are designed for time-dependent data, such as *timetables* (see **Section 3.4**).

### 5.2.1 Connection scan

Connection scan (CSA) [26] is an algorithm for route planning specially designed for time-dependent networks, such as public transit networks. It processes the network represented as timetable, as defined by **Definition 18**.

The algorithm is very simple. All connections of the timetable are sorted by their departure time. Given a query, connections are explored increasing in their departure time. The algorithm is fast primarily due to the fact that connections can be maintained in a simple array. In contrast to DIJKSTRA, it does not need to maintain a priority queue or other more complex data-structures. Arrays are heavily optimized and benefit from a lot of effects, like cache locality [32].

**Algorithm 6** shows the full connection scan algorithm. The array $S$ stores for each stop the currently best arrival time. $T$ associates for each trip the first connection it is taken with. $J$ is used for path extraction and memorizes for each stop a segment of a trip, consisting of enter and exit connections $c_{\mathsf{enter}}$ and $c_{\mathsf{exit}}$ respectively, and a footpath $f$:

$$(c_{\mathsf{enter}}, c_{\mathsf{exit}}, f)$$

**Algorithm 6:** Connection scan algorithm for computing shortest paths in time-dependent networks, represented by timetables.

**input :** timetable $(S, T, C, F)$, source $s \in S$, target $t \in S$, departure time $\tau$
**output:** shortest path from $s$ to $t$

```
// Initialization
for u ∈ S do S[u] ← ∞ ;
for o ∈ T do T[o] ← undefined ;
for u ∈ S do J[u] ← (undefined, undefined, undefined) ;
for f = (u_dep, d, u_arr) ∈ F : u_dep = s do
    S[u_arr] ← τ + d;
    J[u_arr] ← (undefined, undefined, f);
// Explore connections increasing in departure time
c_0 ← arg min_{(u_dep,u_arr,τ_dep,τ_arr,o)∈C:τ_dep≥τ} τ_dep;
for c = (u_dep, u_arr, τ_dep, τ_arr, o) ∈ C increasing by τ_dep, starting from c_0 do
    if τ_dep ≥ S[t] then
        break;
    if T[o] ≠ undefined ∨ τ_dep ≥ S[u_dep] then
        if T[o] == undefined then
            T[o] ← c;
        if τ_arr < S[u_arr] then
            for f = (v_dep, d, v_arr) ∈ F : v_dep = u_arr do
                if τ_arr + d < S[v_arr] then
                    S[v_arr] ← τ_arr + d;
                    J[v_arr] ← (T[o], c, f);
// Extract path by backtracking
p ← empty path;
u ← t;
while c_enter ≠ undefined : (c_enter, c_exit, f) = J[u] do
    prepend f to p;
    prepend the part of the trip between c_enter and c_exit to p;
    u ← v_dep : (v_dep, v_arr, τ'_dep, τ'_arr, o) = c_enter;
prepend f : (undefined, undefined, f) = J[s] to p;
return p;
```

It represents a path which takes the segment of the trip starting at $c_{\text{enter}}$, ending at $c_{\text{exit}}$ and then taking the footpath $f$ from the arrival stop of $c_{\text{exit}}$. Such an entry is associated to the arrival stop of the footpath $f$, always representing the parent path that results in the current best arrival time for the corresponding stop.

The algorithm starts by initializing the arrays with default values and relaxing all initial footpaths. Connections are then explored increasing in their departure time, starting from the first connection $c_0$ that starts after the departure time $\tau$. **Line 7** is typically implemented as *binary search* [35] on a sorted array of connections $C$.

**Line 9** is the stopping criteria, which lets the algorithm terminate once a connection departs after the current best arrival time at the target $t$. Since connections are explored increasing in time, it is impossible that a connection can improve on the arrival time anymore.

**Line 11** will only explore a connection if a previous connection of the same trip was already used, indicating traveling without a transfer; or if it was already possible to arrive at the stop earlier with a previous connection, indicating a transfer at this stop.

A connection is then only relaxed if it improves the arrival time at its arrival stop, represented by **line 14**. If so, all outgoing footpaths are explored. A footpath represents exiting the vehicle, walking to the arrival stop of the footpath ready for entering another vehicle. Note that self-loop footpaths must be contained in timetables (compare to **Definition 18**), making it possible to transfer at one stop.

**Line 16** only considers footpaths that improve the arrival time at the corresponding stop. **Line 18** stores the path represented by taking this connection and the footpath.

For an example, we refer to the schedule of **Fig. 3.2** again. The corresponding timetable is explained in **Section 3.4**, we use the same notion again. It consists of five connections, denoted by $c_1, c_2, c_3, c_4$ and $c_5$, sorted by departure time. We assume only the three self-loop footpaths on the stops $f$, $o$ and $k$.

Assume a query from Freiburg Hbf, represented by stop $f$ to Karlsruhe Hbf, represented by $k$, with a departure time of $\tau = 3{:}50\ pm$. The initial configuration after **line 3** is

$$\begin{aligned} S[f] = S[o] = S[k] &= \infty, \\ T[t_{104}] = T[t_{17024}] = T[t_{17322}] = T[t_{79}] &= \text{undefined}, \\ J[f] = J[o] = J[k] &= (\text{undefined}, \text{undefined}, \text{undefined}). \end{aligned}$$

Then the footpath $(f, 300, f)$ departing at Freiburg Hbf is relaxed, resulting in

$$\begin{aligned} S[f] &= 3{:}55\ pm, \\ J[f] &= (\text{undefined}, \text{undefined}, (f, 300, f)). \end{aligned}$$

Connections are now explored increasing in departure time, starting with

$$c_1 = (f, o, 3{:}56\ pm, 4{:}28\ pm, t_{104}).$$

The connection is considered since we already arrived at Freiburg Hbf before $3{:}56\ pm$.

The trip is set and the footpath at Offenburg is relaxed, yielding

$$\begin{aligned} T[t_{104}] &= c_1, \\ S[o] &= 4{:}33\ pm, \\ J[o] &= (c_1, c_1, (o, 300, o)). \end{aligned}$$

The next connection is

$$c_2 = (f, o, 4{:}03\ pm, 4{:}50\ pm, t_{17024}).$$

However, it induces no changes, as the previous connection already arrived at Offenburg earlier. The algorithm continues by exploring

$$c_3 = (o, k, 4{:}29\ pm, 4{:}58\ pm, t_{104}).$$

The connection is considered because the trip $t_{104}$ was used before already, indicating that the trip can be taken without transferring. Else it would not be applicable, since the current best arrival time at Offenburg, including the transfer duration of 5 minutes, is 4:33 *pm*, which is after the departure time of $c_3$. The changes are

$$\begin{aligned} S[k] &= 5{:}03\ pm, \\ J[k] &= (c_1, c_3, (k, 300, k)). \end{aligned}$$

In the next iteration

$$c_4 = (o, k, 4{:}35\ pm, 5{:}19\ pm, t_{17322})$$

is considered, again inducing no changes. The algorithm then terminates exploration since the last connection

$$c_5 = (k, f, 7{:}10\ pm, 8{:}10\ pm, t_{79})$$

departs after the current best arrival time at Karlsruhe Hbf, which is $S[k] = 5{:}03\ pm$.

Path construction is straightforward, it is

$$\begin{aligned} J[k] &= (c_1, c_3, (k, 300, k)), \\ J[f] &= (\text{undefined}, \text{undefined}, (f, 300, f)), \end{aligned}$$

which yields the path which takes

- the footpath from Freiburg Hbf to Freiburg Hbf,
- $t_{104}$ starting with $c_1$ to $c_3$, which is using the ICE 104 from Freiburg Hbf to Karlsruhe Hbf,
- and a final footpath from Karlsruhe Hbf to Karlsruhe Hbf.

The earliest arrival time at Karlsruhe Hbf is $S[k] = 5{:}03\ pm$.

## 5.3 Multi-modal

So far, all presented route planning algorithms are limited to networks only consisting routes of one transportation mode, for example a train network. We only distinguished between time-independent and time-dependent networks. However, in practice we want to plan routes involving multiple transportation modes. For example using a bicycle to drive to the next train main station, using the road network, and then entering a train.

To represent transportation mode possibilities in the networks, we slightly modify our models. All edges in graph based models get transportation mode labels, formalized by **Definition 28**.

**Definition 28.** *Given a set of transportation mode labels $M$, a* multi-modal graph $G = (V, E)$ *is a graph with a label function*

$$\text{mode} : E \to \{S \subseteq M\}$$

*that assigns to each vertex a set of available transportation modes.*

In our implementation in COBWEB we use the modes

$$M = \{\text{car}, \text{bike}, \text{foot}, \text{tram}\}.$$

The *timetable* model is adjusted by assigning all connections the mode tram and all footpaths foot.

Another difficulty of multi-modal routing is that, in practice, it is usually not be applicable to change transportation modes arbitrarily. User have different requirements and preferences regarding the change of modes. For example, it might not be possible to use a car right after traveling with a tram and then leaving it at a train station before continuing the journey using a train. If the model does not account for this, the algorithm should not be allowed to pick such a route.

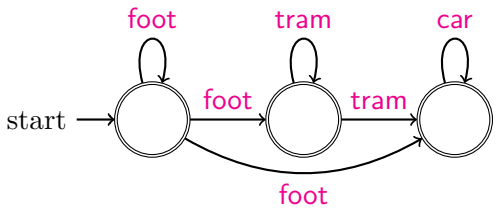


Fig. 5.3: Automaton representing transportation mode constraints.

Applicable transportation mode sequences are typically represented as languages of automata (see **Section 2.3**) [14]. **Fig. 5.3** shows an example. The automaton accepts words consisting of routes that

1. are empty,
2. only use foot,
3. use the tram after walking to a stop,
4. use the car after walking to a stop and using the tram, and
5. use the car directly after walking.

A route that takes the tram after using a car is not accepted by the automaton and thus, not applicable.

The search of shortest paths, restricted to such transportation mode automata, is called the Label-Constrained Shortest Path Problem [14] (LCSPP). Common algorithms, like Dijkstra, A$^\star$ and ALT, were adapted and analyzed with respect to the LCSPP [14, 31, 45].

However, we will study two algorithms that are restricted to fixed languages, not accepting arbitrary automata. First, we show a simple extension of Dijkstra and its variants that adapts the algorithm for multi-modal route planning. Afterwards, we present a generic approach to combine any uni-modal algorithms for limited multi-modal route planning.

### 5.3.1 Modified Dijkstra

In order to adapt Dijkstra and its variants A$^\star$ and ALT for multi-modal graphs (see **Definition 28**), the algorithm needs to account for the labels at edges.

Given a multi-modal graph, a source $s$ and a target $t$, and a set of available transportation modes

$$S \subseteq \{\mathsf{car}, \mathsf{bike}, \mathsf{foot}, \mathsf{tram}\} = M$$

The modified Dijkstra computes a shortest path $p$ from $s$ to $t$ which does only use edges labeled with available modes, i.e.

$$\forall e \in p : \text{mode}(e) \subseteq S.$$

Therefore, we adjust **line 11** of **Algorithm 5** to only consider outgoing edges such that

$$e = (u, w, v) \in E : \text{mode}(e) \subseteq S.$$

When multiple transportation modes are available, such as $\{\mathsf{bike}, \mathsf{car}\}$, the edge weight is not static anymore, as a car can travel the distance faster than a bike. To break the ties, we always choose the fastest transportation mode, referring to the order

$$\mathsf{foot} \sqsubset \mathsf{bike} \sqsubset \mathsf{tram} \sqsubset \mathsf{car}.$$

The edge weight $w$ in **line 11** is then computed as if the fastest, on this edge available, transportation mode is used:

$$\max_{\sqsubset} \text{mode}(e)$$

The modified DIJKSTRA accepts the transportation mode model shown by **Fig. 5.4**.

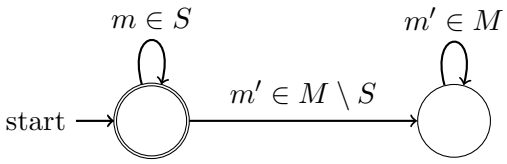


Fig. 5.4: The transportation mode constraints of DIJKSTRA, adapted to multi-modal routing.

While this modification works perfectly fine for DIJKSTRA, it does impair the effectiveness of A$^\star$ and ALT. The problem is that the heuristic of A$^\star$ can not know the transportation mode restrictions $S$ beforehand. Because of that, a heuristic must always assume that the fastest possible transportation mode is chosen. Else, it might be possible that the actual shortest path uses a faster mode than the heuristic assumed, in which case the heuristic would overestimate the travel time and violate **Definition 26**.

For asTheCrowFlies this means that it must assume that the straight-line distance is traveled using a car, or more general:

$$\max_{\sqsubset} M$$

For ALT all precomputation must be done under the assumption that, at query time, there are no transportation mode restrictions, i.e.

$$S = M.$$

The actual impact on the effectiveness heavily depends on the type of network. It has no effect at all, if all edges on the shortest path for $S = M$ can also be taken with the actual restricted version of $S$. It gets worse if edges are not available anymore, for example a highway that can not be taken for $S = \{\text{foot}\}$, although the heuristic assumed it can be taken using a car.

In a typical road network most edges support all road-type transportation modes, i.e. $\{\text{foot}, \text{bike}, \text{car}\}$. The most common exceptions are highways, pedestrian zones and bikeways. However, the latter two do typically not cover big distances and a regular road connecting the same locations is often available too. Because of that A$^\star$ and ALT typically perform worse only on long-distance routes, which make heavy usage of highways, if the transportation modes are restricted to modes not available on highways. A similar observation can be done for combined networks, like a link graph (see **Section 3.3**).

For ALT this problem can be tackled by precomputing the distances to the landmarks for every possible transportation mode restriction $S$ individually. However, this results in

$$|\mathcal{P}(M)| = 2^{|M|}$$

combinations, which is usually not feasible.

### 5.3.2 Access nodes

Often, combining multiple networks of different types into one representation, such as a graph, is not appropriate. We have seen that graph representations for public transit networks dramatically scale in size, due to representing time information. A timetable is more suited for such a network type and algorithms optimized to a specific network type, such as CSA, perform much better than a generic approach like Dijkstra.

In this section, we elaborate on a generic technique that allows to combine any networks with corresponding algorithms for a restricted variant of the Shortest Path Problem. We describe the algorithm by combining a road with a public transit network, using the multi-modal variant of ALT and CSA respectively. The general technique is known as Access-Node Routing (ANR) [**23**, **16**].

Given a source and a destination node in the road network, we first compute *access nodes*. Those are nodes where we will switch from the road into the public transit network. Therefore, the access nodes are computed as the k-nearest neighbors (see **Definition 20**) for both, the source and the destination node, in the public transit network. The amount $k$ should be kept small in order to keep query time low, we use 3 in our implementation.

In the best case, the access nodes are *important*, i.e. they maximize the amount of shortest paths, from the source to the destination, of which they are part of. Because of that, typically they are precomputed, using a ranking among the nodes. For example, a train main station is preferred over a small tram stop. The computation can be optimized further by using heuristics and techniques like ALT were some paths are already precomputed. See [**23**] for details on how to obtain *good* access nodes.

Given the access nodes for source and destination, a path is computed piecewise, by computing shortest paths from

1. the source to all its access nodes,
2. the access nodes of the source to all access nodes of the destination, and
3. the access nodes of the destination to the destination.

We denote the corresponding sets of paths by $P_s$, $P_{st}$ and $P_t$ respectively. The resulting path is chosen as the concatenation of paths from those sets, such that the cost is minimized. That is, we receive a path

$$p = p_1 p_2 p_3$$

with $p_1 \in P_s, p_2 \in P_{st}$ and $p_3 \in P_t$ such that

$$\begin{aligned} \text{dest}(p_1) &= \text{src}(p_2), \\ \text{dest}(p_2) &= \text{src}(p_3). \end{aligned}$$

Of all paths satisfying this constraints, $p$ is chosen as the path with the smallest cost. Additionally, we consider the shortest path $q$ between the source and destination that only uses the road network. The final path is again the one with the smaller cost. **Fig. 5.5** illustrates the scheme of this approach.

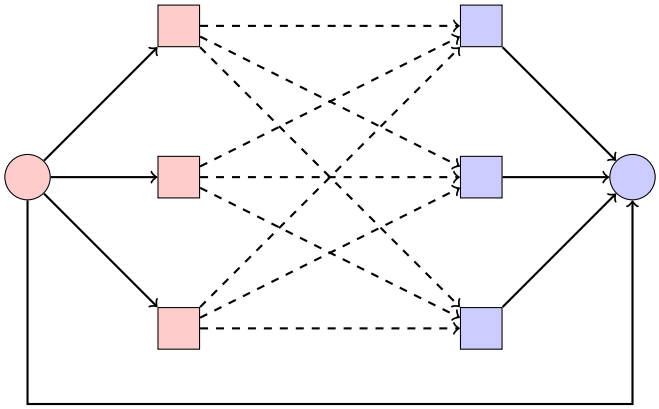

Fig. 5.5: Scheme of Access-Node Routing. Circular nodes represent the source and destination node, rectangular nodes are their corresponding access nodes. Solid edges indicate shortest paths in the first network, dashed lines are in the second network.

The accepted transportation mode model is shown by **Fig. 5.6**.

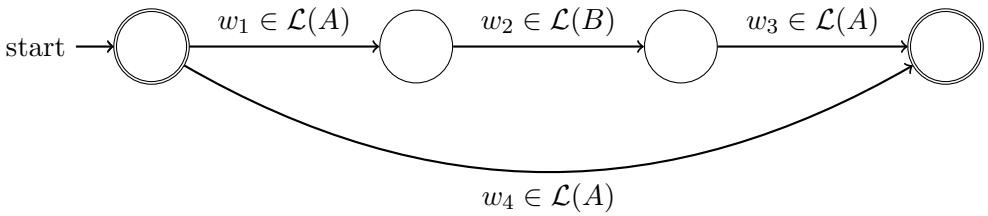


Fig. 5.6: The transportation mode constraints of Access-Node Routing with two networks. $A$ represents the transportation mode model accepted by the algorithm on the first network, $B$ refers to the automaton of the algorithm on the second network.

Note that the resulting path is not necessarily a valid solution to the Shortest Path Problem anymore. A correct solution may not even contain any of the used access nodes. However, if access nodes are chosen well, the resulting path is likely to be appropriate and a good approximation to the actual solution.

Section 6

# Evaluation

In this section we report on our experimental results for the presented algorithms on three data sets of increasing size. Therefore, we first give insights on the data sets and how the network models are obtained. Afterwards we evaluate Cover Trees, Dijkstra, A$^\star$ (with asTheCrowFlies), ALT, CSA and multi-modal methods such as the adopted Dijkstra and our simplified version of ANR on the given data sets.

When evaluating shortest path queries on randomly chosen source and target nodes, the resulting paths tend to be long-range. However, in practice most queries are only local and algorithms like Dijkstra do not scale well with increasing range. To overcome this measurement problem, we introduce the notion of a *Dijkstra rank* [41].

**Definition 29.** *Given a graph $G = (V, E)$, the* Dijkstra rank *of a node $v \in V$ is the number of the iteration in which, when running Dijkstra on the graph, it is polled from the priority queue (see **line 7** of **Algorithm 5**).*

*That is the position $i$ for $v_i$ in the order of vertices when sorted ascending by their distance to the source, i.e.*

$$v_1, v_2, \ldots, v_{|V|}$$

*with* $\texttt{dist}(v_i) \leq \texttt{dist}(v_{i+1})$ *for all $i$.*

Instead of choosing queries randomly, we only choose source nodes randomly and then select targets by their *Dijkstra rank* to the source. Queries can then be sorted by the *Dijkstra rank* and, by that, evaluated in terms of increasing range.

## 6.1 Input data

We consider three data sets, consisting of road and public transit data. The road network is extracted from OSM [33] formatted data and transit data is given in the GTFS [13] format.

Our data sets represent the region around the german cities Freiburg and Stuttgart. Their road network is of similar size, while our transit data for Freiburg only includes tram data, whereas the data for Stuttgart also includes train and bus connections. The size of our transit network for Stuttgart is about ten times the size of the network for Freiburg.

Furthermore, we include a road and transit network for the country Switzerland. The transit data consists of train, tram and bus connections. Both networks are about three times the size of the Stuttgarts.

We obtain our road networks from [4, 6, 8] and our transit networks from [3, 7]. The transit data used for Stuttgart is under restricted public access (refer to [9]).

### 6.1.1 OSM

OSM [33] (OpenStreetMap) data is represented in a XML structure describing

1. *nodes*, with an unique identifier and a coordinate given as pair of latitude and longitude;
2. *ways*, also with an unique identifier, consisting of multiple nodes referenced by their identifier;
3. *relations*, consisting of nodes, ways and other relations, representing relationships between the referenced data;
4. *tags* as key-value pairs, storing metadata about the other items.

A small OSM example data set is shown by **Lst. 6.1**. Ways are used to represent roads consisting of nodes. Tags are used to describe metadata like speed limits for a road or whether it is a one-way street or not. However, the format also contains a lot of data not directly relevant for route planning, like shapes of buildings and outlines of public parks. Therefore, we filter OSM data and only keep relevant information.

As we are only interested in the road network itself, we start by reading the ways. We filter them based on the tags described by **Lst. 6.2**. Ways having at least one of the key-value pairs described under $--KEEP$ and none of the pairs under $--DROP$ are kept, as they represent roads of the network. All other ways are rejected, as well as all relations. After that, we read the nodes and only keep nodes that occurred at least once in any of the ways that passed the filter. Our road network is then build using the remaining nodes as graph nodes, translating the ways into edges between the nodes.

Ways with a positive *oneway* tag are translated into edges only going into the given direction, else we generate both edges for both directions. The cost of an edge is computed as the time it needs to travel the direct distance between the source and destination coordinates (see **Definition 13**) with a certain speed. The speed is determined either by a given *maxspeed* tag or the average speed for the road type defined by the *highway* tag. Therefore, we use the average speed references shown by **Table 6.1**.

The size of the resulting road graphs (see **Section 3.1**) for all three data sets is reported in **Table 6.2**. As seen, filtering the OSM data sets beforehand reduces the size of data that is to be processed by 95% to 97%. The road graphs have approximately two edges per node. This is due to most streets being two-way streets, thus generating

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version="0.6">
  <bounds minlon="7.253190" minlat="47.299090" maxlon="9.246965" maxlat="
      48.751520"/>
  <node id="29764598" lat="47.8512831" lon="7.9230269"/>
  <node id="669209525" lat="47.8513215" lon="7.9231227"/>
  <node id="3993821274" lat="47.8513342" lon="7.923183"/>
  <node id="832450227" lat="47.8157938" lon="8.8487527">
    <tag k="highway" v="motorway_junction"/>
    <tag k="name" v="Kreuz Hegau"/>
  </node>
  <node id="100036455" lat="47.5728421" lon="8.0365409">
    <tag k="name" v="Niederhof"/>
    <tag k="traffic_sign" v="city_limit"/>
  </node>
  <way id="29764598">
    <nd ref="669209525"/>
    <nd ref="3993821274"/>
    <tag k="highway" v="motorway"/>
    <tag k="oneway" v="yes"/>
  </way>
  <relation id="56688">
    <member type="node" ref="29764598" role=""/>
    <member type="node" ref="669209525" role=""/>
    <member type="way" ref="29764598" role=""/>
    <tag k="name" v="Bus line 1"/>
    <tag k="network" v="VVW"/>
    <tag k="ref" v="1"/>
    <tag k="route" v="bus"/>
    <tag k="type" v="route"/>
  </relation>
</osm>
```

Listing 6.1: OSM example data set, derived from [5].

```
--KEEP

#highways
highway=motorway
highway=trunk
highway=primary
highway=secondary
highway=tertiary
highway=residential
highway=living_street
highway=unclassified
highway=cycleway

#highwaylinks
highway=motorway_link
highway=trunk_link
highway=primary_link
highway=secondary_link
highway=tertiary_link
highway=residential_link

#non-standard
way=primary
way=seconday

--DROP

area=yes
train=yes
access=no
type=multipolygon
railway=platform
railway=station
highway=proposed
highway=construction
building=yes
building=train_station
```

Listing 6.2: Tag filter for OSM ways.

| tag value | ∅ km/h |
|---|---|
| motorway | 120 |
| trunk | 110 |
| primary | 100 |
| secondary | 80 |
| tertiary | 70 |
| motorway_link | 50 |
| trunk_link | 50 |
| primary_link | 50 |
| secondary_link | 50 |
| residential | 50 |
| unclassified | 40 |
| unsurfaced | 30 |
| road | 20 |
| cycleway | 14 |
| living_street | 7 |
| service | 7 |

Table 6.1: Average speed in km/h for a OSM way with the corresponding value for the *highway* tag.

two edges per connection between two nodes. Obviously, road junctions are, compared to the amount of nodes, rare and thus, multiple edges do only rarely share the same node. The in- and outdegree of nodes is extremely low, mostly 2 ($\approx 80\%$), as seen in **Table 6.3**.

### 6.1.2 GTFS

GTFS [13] is short for General Transit Feed Specification, it defines a common format for public transit schedules. It comes compressed as ZIP archive, consisting multiple text files formatted as CSV tables. The mandatory tables are

1. *agency.txt*, defining metadata about the transit agency;
2. *routes.txt*, containing information about complete routes, like all trips belonging to a bus line;
3. *trips.txt*, consisting of single trips, belonging to a route;
4. *stop_times.txt*, having departure and arrival times at the stops for all connections in the network;
5. *stops.txt*, providing metadata and coordinates of all stops;

| | data (MB) | | Road graph | |
|---|---|---|---|---|
| | raw | filtered | nodes | edges |
| Freiburg | 2 260 | 86 | 743 003 | 1 494 883 |
| Stuttgart | 2 420 | 118 | 973 142 | 1 950 978 |
| Switzerland | 5 530 | 279 | 2 627 645 | 5 226 060 |

Table 6.2: Size of the OSM data sets, in megabyte (MB) before and after filtering, and the size of the resulting road graphs in amount of nodes $|V|$ and edges $|E|$.

| | indegree $\deg^-$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Freiburg | 90 | 64 990 | 611 055 | 59 751 | 7 057 | 58 | 2 |
| Stuttgart | 145 | 109 808 | 759 157 | 93 354 | 10 599 | 76 | 3 |
| Switzerland | 325 | 235 069 | 2 201 945 | 174 333 | 15 767 | 202 | 4 |

| | outdegree $\deg^+$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Freiburg | 105 | 65 336 | 610 353 | 60 059 | 7 088 | 60 | 2 | 0 |
| Stuttgart | 162 | 110 002 | 758 740 | 93 545 | 10 607 | 83 | 3 | 0 |
| Switzerland | 328 | 235 255 | 2 201 711 | 174 247 | 15 884 | 215 | 4 | 1 |

Table 6.3: Table showing the number of nodes of the corresponding road graph that have a certain in- or outdegree. That is, the number of ingoing and outgoing edges respectively.

6. *calendar.txt* defining the service pattern at which routes are available.

Furthermore, there are a couple of optional tables of which we are only interested in

7. *transfer.txt*, provides transfer possibilities between stops and their duration.

An example feed can be seen in **Lst. 6.3**. The format is similar to our definition of timetables (see **Section 3.4**), with the difference that connections are not directly given as edges departing from one stop and another, but as pair of arrival and departure time at stops. Also, it contains a lot of metadata which we do not process.

Construction of a realistic time expanded transit graph (see **Definition 16**) is straightforward and mainly involves around parsing *stop_times.txt*. We build two nodes for every entry, one representing the arrival event at the stop and another for the departure.

```
1  // agency.txt
2  agency_id, agency_name, agency_url, agency_timezone, agency_phone,
       agency_lang
3  FunBus, The Fun Bus, , (310) 555-0222, en
4
5  // routes.txt
6  route_id, route_short_name, route_long_name, route_desc, route_type
7  A, 17, Mission, From lower Mission to Downtown., 3
8
9  // trips.txt
10 route_id, service_id, trip_id, trip_headsign, block_id
11 A, WE, AWE1, Downtown, 1
12 A, WE, AWE2, Downtown, 2
13
14 // stop_times.txt
15 trip_id, arrival_time, departure_time, stop_id, stop_sequence,
       pickup_type, drop_off_type
16 AWE1, 0:06:10, 0:06:10, S1, 1, 0, 0
17 AWE1, 0:06:20, 0:06:30, S3, 3, 0, 0
18 AWE1, 0:06:45, 0:06:45, S6, 5, 0, 0
19 AWD1, 0:06:10, 0:06:10, S1, 1, 0, 0
20 AWD1, 0:06:20, 0:06:20, S3, 3, 0, 0
21 AWD1, 0:06:45, 0:06:45, S6, 6, 0, 0
22
23 // stops.txt
24 stop_id, stop_name, stop_desc, stop_lat, stop_lon, stop_url,
       location_type, parent_station
25 S1, Mission St. & Silver Ave., , 37.728631, -122.431282, , ,
26 S3, Mission St. & 24th St., , 37.75223, -122.418581, , ,
27 S6, Mission St. & 15th St., , 37.766629, -122.419782, , ,
28
29 // calendar.txt
30 service_id, monday, tuesday, wednesday, thursday, friday, saturday,
       sunday, start_date, end_date
31 WE, 0, 0, 0, 0, 0, 1, 1, 20060701, 20060731
32 WD, 1, 1, 1, 1, 1, 0, 0, 20060701, 20060731
33
34 // transfers.txt
35 from_stop_id, to_stop_id, transfer_type, min_transfer_time
36 S3, S6, 2, 300
37 S6, S3 3, 180
```

Listing 6.3: GTFS example data set, inspired by [2].

Furthermore, we create a transfer node for every arrival node, indicating a transfer at the given stop. Each arrival node is then connected by an edge with its corresponding departure and transfer node.

After parsing all data, we connect departure nodes with the arrival nodes at the next stop in a trip. Therefore, we process each trip and follow the *stop_times.txt* entries belonging to that trip in the order defined by the *stop_sequence* field.

As a next step, waiting edges are created by sorting transfer nodes of a stop ascending in time and then creating edges connecting them in that order. Finally, every departure node is connected to its previous transfer node. We find the transfer node by using a *binary search* [35] on the sorted list of transfer nodes for this stop.

Timetables (see **Definition 18**) are received similar. But simpler, as transfer nodes are not present. We process all stops and trips defined in *stops.txt* and *trips.txt* and obtain the sets $S$ and $T$ respectively. Connections are created by again processing entries in *stop_times.txt*, belonging to one trip, in the sequence defined by *stop_sequence*. We create one connection for every departure node with the corresponding next arrival node.

For the footpaths, we initially take the transfers given in *transfers.txt*. In order to increase the quality of our footpath model, we also connect stops with footpaths if they are within 600 meters to each other.

However, our footpaths need to fulfill strong properties (see **Section 3.4**), which the given transfers usually not obey. Therefore, we have to add self-loop footpaths, if not present. And we need to compute the transitive closure of the given footpaths in order to ensure that they are transitively closed. Therefore, it is crucial that the range, for which close stops are connected, is kept low. Else, the amount of footpaths dramatically increases due to the transitive closure.

The *triangle inequality* property is ensured by rejecting given transfer durations and approximating all durations by using asTheCrowFlies. Additionally, all footpath durations must not be lower than the transfer buffer used for the self-loop footpaths. We do so by taking the max of the transfer buffer and the calculated duration.

**Table 6.4** reports the size of the feed and the resulting network. It can be clearly seen that a timetable has a much smaller amount of objects, compared to a realistic time expanded transit graph. In particular compared to the size of a road graph (see **Table 6.2**). This even becomes worse if we use it to construct a link graph, as seen in **Section 3.3**, as we need to add an incoming and outgoing edge for each arrival node, in order to connect it with the road graph. **Table 6.5** reports the exact amount of added link edges.

## 6.2 Experiments

This section shows our experimental results for the algorithms presented in **Section 4** and **Section 5**. The algorithms are implemented in the context of the Cobweb [44]

| | data (KB) | Transit graph | |
|---|---|---|---|
| | | nodes | edges |
| Freiburg | 1 713 | 613 329 | 1 006 862 |
| Stuttgart | 32 213 | 4 517 511 | 7 415 894 |
| Switzerland | 75 477 | 32 688 498 | 53 370 236 |

| | Timetable | | | |
|---|---|---|---|---|
| | stops | trips | connections | footpaths |
| Freiburg | 713 | 13 249 | 191 194 | 255 495 |
| Stuttgart | 7 877 | 90 475 | 1 415 362 | 1 926 611 |
| Switzerland | 30 227 | 1 014 699 | 9 881 467 | 3 793 581 |

| | Footpaths | | | |
|---|---|---|---|---|
| | given | self-loops | close | closure |
| Freiburg | 0 | 713 | 9 008 | 245 774 |
| Stuttgart | 6 080 | 7 877 | 73 730 | 1 838 924 |
| Switzerland | 22 402 | 30 227 | 174 698 | 3 566 254 |

Table 6.4: Size of the GTFS feeds, in kilobyte (KB) and the size of the resulting realistic time expanded transit graphs in amount of nodes $|V|$ and edges $|E|$. Also, the size of the obtained timetable and details about the footpath generation. The column *given* denotes the amount of footpaths already given in the *transfer.txt* file, *self-loops* represents how many missing self-loop paths were added. Likewise does *close* report how many footpaths we added for connecting close stops with each other. And *closure* denotes the amount of paths added to ensure that the model is transitively closed.

project, which is an open-source multi-modal route planner written in Java.

Results are measured from a sequential execution on a 6-core Intel Xeon E5649 machine running at 2.53 GHz. The maximal heap size of Javas virtual machine is restricted to 85 GB.

### 6.2.1 Nearest neighbor computation

For solving the Nearest Neighbor Problem we implemented a Cover Tree data-structure with corresponding retrieval methods, as explained in **Section 22**. It operates on nodes of the road network obtained from the data sets Freiburg, Stuttgart and Switzerland, using asTheCrowFlies as metric on the nodes.

The experiment consists of continuous insertion of nodes, for each of the three networks respectively, and then measuring random nearest neighbor queries, i.e. the execution

| | link edges |
|---|---|
| Freiburg | 306 906 |
| Stuttgart | 1 944 388 |
| Switzerland | 19 584 786 |

Table 6.5: Amount of link edges that are added when combining road with transit graphs to create a link graph.

time of **Algorithm 2**. Measurements are done for tree sizes of 1, 10 000 and then in steps of 10 000. Each measurement is averaged over 1 000 queries using randomly selected nodes.

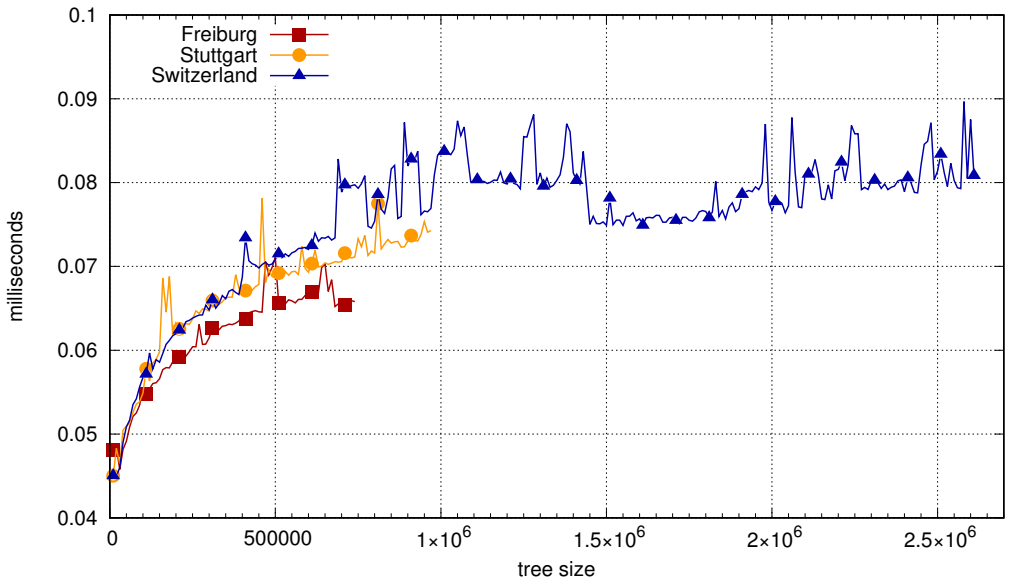


Fig. 6.1: Query durations for **Algorithm 2** in a Cover Tree with increasing size, for three road networks respectively. Measurements are done at a size of 1, 10 000 and then in steps of 10 000, averaged over 1 000 random queries. Running time is stated in milliseconds.

**Fig. 6.1** shows the results of the experiment. The method is comparably fast, even for large road networks like Switzerland. The graph appears to be similar for all three data sets. This is obviously due to the fact that they all represent the same type of network,

with a similar distribution of nodes.

In a road network, nodes are typically close to each other and appear in local groups, representing cities and structured road segments. In particular, they are not uniformly distributed. A Cover Tree benefits from this, as a node can be the parent of many other, locally close nodes. And as such, the tree is balanced well, resulting in efficient queries that are able to quickly find the correct path in the tree that leads to the nearest neighbor.

Due to the same reason, the running time scales approximately logarithmic with increasing size. Queries take longer if the depth of the tree increases. In a well balanced Cover Tree the depth is logarithmic to its size.

### 6.2.2 Uni-modal routing

The first experiment for uni-modal routing compares time-independent methods for solving the Shortest Path Problem. It measures an implementation of Dijkstra (see **Algorithm 5**), the A$^\star$ algorithm (see **Section 5.1.2**) using asTheCrowFlies as heuristic and ALT with the precomputed heuristic shown in **Definition 27**.

Queries are performed on the road graphs obtained by the data sets Freiburg, Stuttgart and Switzerland. We choose 50 random source nodes and then determine the *Dijkstra rank* (see **Definition 29**) for the source nodes to all other nodes in the graph. Source nodes with a bad connectivity are rejected and exchanged against another random source node. This is determined by a source node having no node in the graph with a rank of at least $2^{15}$ which is only rarely the case for randomly chosen nodes. We then choose nodes as destinations that have a Dijkstra rank of

$$2^0, 2^1, \ldots, 2^k$$

where $k \geq 15$ is the maximal rank all source nodes have in common. By that, the queries cover all types of ranges, highlighting how well the algorithms scale with queries of increasing ranges. By that, we receive for every rank $2^i$ in total 50 different queries of which we average the measured running time over.

**Fig. 6.2** shows the results of the experiment. First of all, it can be seen that all three methods do not scale well with queries of increasing ranges. Long range queries, like for a rank of $2^{20}$ or $2^{21}$ range from 1 to 10 seconds. In fact, the running time scales exponentially for increasing ranges. Further, A$^\star$ and ALT are slower than Dijkstra for short range queries. This is due to the increased overhead of the modified Dijkstra variants. Both need to additionally evaluate their corresponding heuristic on every relaxed edge. However, for mid and, in particular, for long range queries, A$^\star$ performs similar to Dijkstra and ALT even is about twice as fast. At this point the additional overhead is negligible and the benefit of a good heuristic pays off. It can also be seen that asTheCrowFlies, which is used by A$^\star$, is not a good heuristic for road networks and does not improve over the ordinary Dijkstra approach, as already explained in **Section 5.1.2**.

Furthermore, if ALT is implemented very carefully and optimized, it can outperform

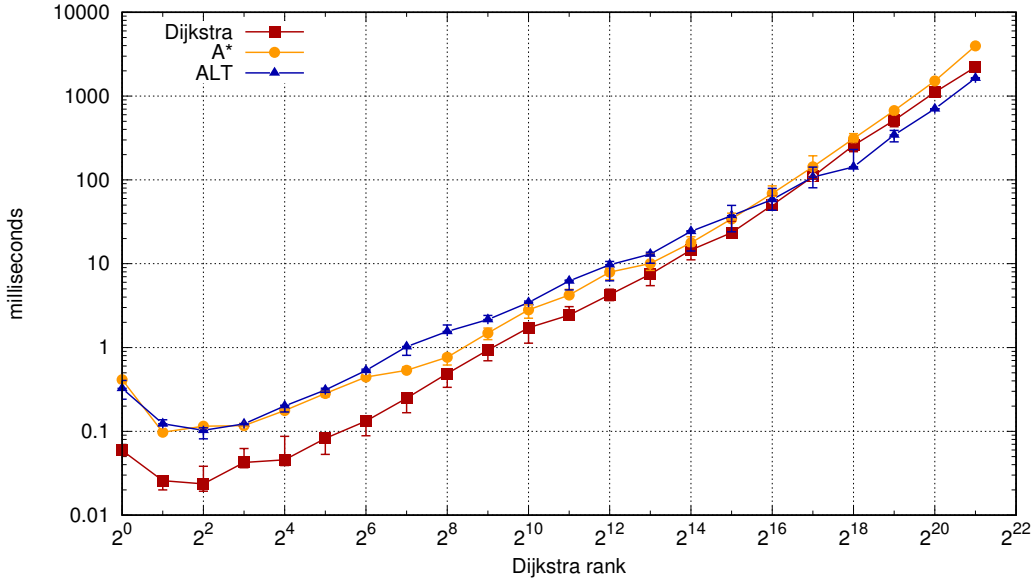


Fig. 6.2: Query durations for uni-modal time-independent route planning algorithms computing shortest paths. Running time is measured in milliseconds, presented on a logarithmic scale. Every point represents 50 queries from a random source to a random target with the given *Dijkstra rank* over which the measurement is averaged over. *Errorbars* indicate the results on the three data sets. The upper end of the bar represents Switzerland, the dot Stuttgart and the lower end Freiburg.

Dijkstra earlier. For a comparison, we include the results from [16] of similar measured experiments for highly optimized variants of Dijkstra and other techniques for uni-modal time-independent route planning in **Fig. 6.3**. The results show that Dijkstras performance can be increased by approximately a factor of 1 000, compared to our implementation, if heavily optimized. However, the running time for long range queries is still not feasible. Fortunately, there exist other approaches which tackle this problem, like seen in the figure. The presented algorithms are referenced and briefly explained in [16].

Additionally, they give a general overview of uni-modal time-independent route planning techniques, comparing their average query time and their necessary preprocessing time. We include their overview in **Fig. 6.4**.

The second experiment compares time-dependent solutions to the Shortest Path Problem. We measure the performance of an adopted Dijkstra variant (see **Section**

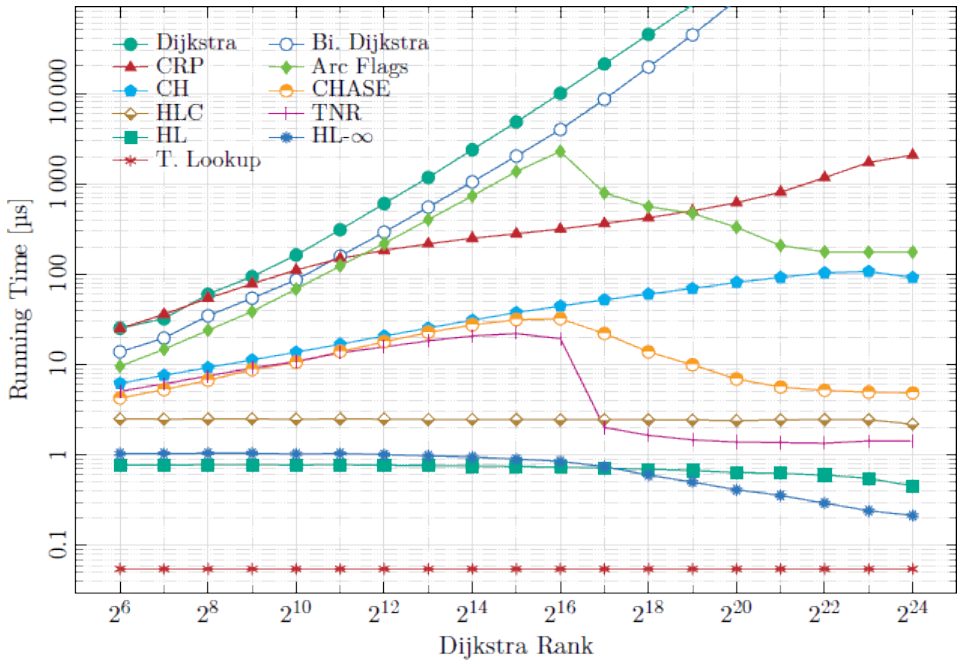


Fig. 6.3: Experimental results from [16] measured similar to **Fig. 6.2** for carefully implemented uni-modal time-independent route planning algorithms.

5.2) against CSA (using **Algorithm 6**) over the duration of one day, with changing time. The experiment is measured for the 10.10.2018, which is a Wednesday, representing an average day in the schedule of the transit network. DIJKSTRA runs on a realistic time expanded transit graph (see **Definition 16**) and CSA on a timetable (see **Definition 18**), both obtained from the public transit data of Freiburg, Stuttgart and Switzerland.

Measurements are taken in steps of 10 minutes over the whole day, averaged over 50 randomly chosen queries. The only exception is DIJKSTRA for Switzerland, which is done in steps of 30 minutes, due to very long running times.

The algorithms are compared in **Fig. 6.5**, with their single performance highlighted by **Fig. 6.6**.

Both algorithms perform worse if the size of the time schedule increases, roughly increasing by a factor of 10 for all three data sets. However, CSA runs on Switzerland 10 times faster than DIJKSTRA on the small schedule of Freiburg, where CSA even performs better by a factor of 1 000. Clearly, CSA outperforms DIJKSTRA for time-dependent routing, making it a very viable choice. CSA can even successfully compete against other approaches designed especially for time-dependent route planning, as shown by [26].

It can also be seen that CSA is subject to the traffic congestion of the time schedule. Yielding better running times in the evening and night from 6:00 *pm* to 6:00 *am*, than in the morning, noon and afternoon from 6:00 *am* to 6:00 *pm*. This is due to the fact that

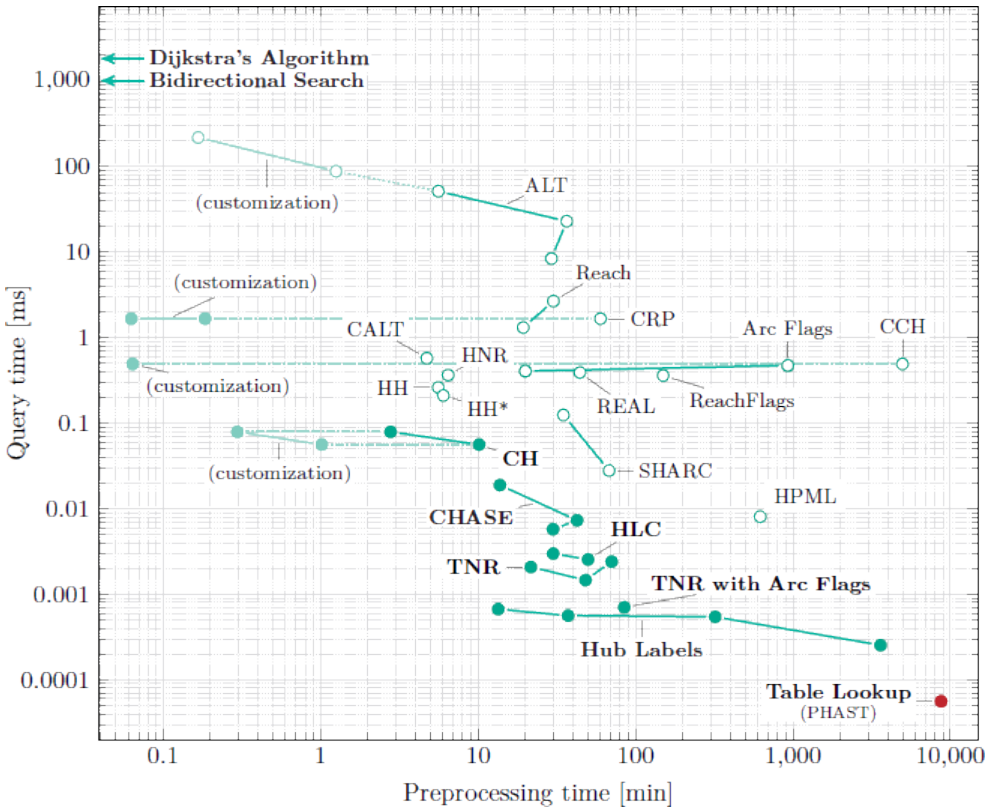


Fig. 6.4: Overview from [16] of uni-modal time-independent route planning techniques, comparing their average query time in milliseconds and their necessary preprocessing time in minutes.

CSA needs to iterate all connections from a given time, not only relevant connections. In a rush hour, the schedule has way more connections that need to be processed, leading to a worse performance.

Dijkstra, on the other hand, only needs to scan connections available from the already processed routes. Thus, it is not affected by traffic congestion as much as CSA and is still more subject to the range of queries, which is not captured by this experiment.

### 6.2.3 Multi-modal routing

For multi-modal routing we compare a modified Dijkstra (see **Section 5.3.1**), running on a link graph (see **Definition 17**), with our simplified version of ANR (refer to **Section 5.3.2**. ANR runs on a road graph and a timetable, using an ordinary Dijkstra for the road and CSA for the transit network. For a given query, it computes the three nearest neighbors to the source and destination as access nodes, using a Cover Tree, then it runs Dijkstra to compute the shortest paths from the source and destination to their access nodes. After that, CSA is used to compute the shortest paths between the sources and destinations access nodes. Additionally, one shortest path query from

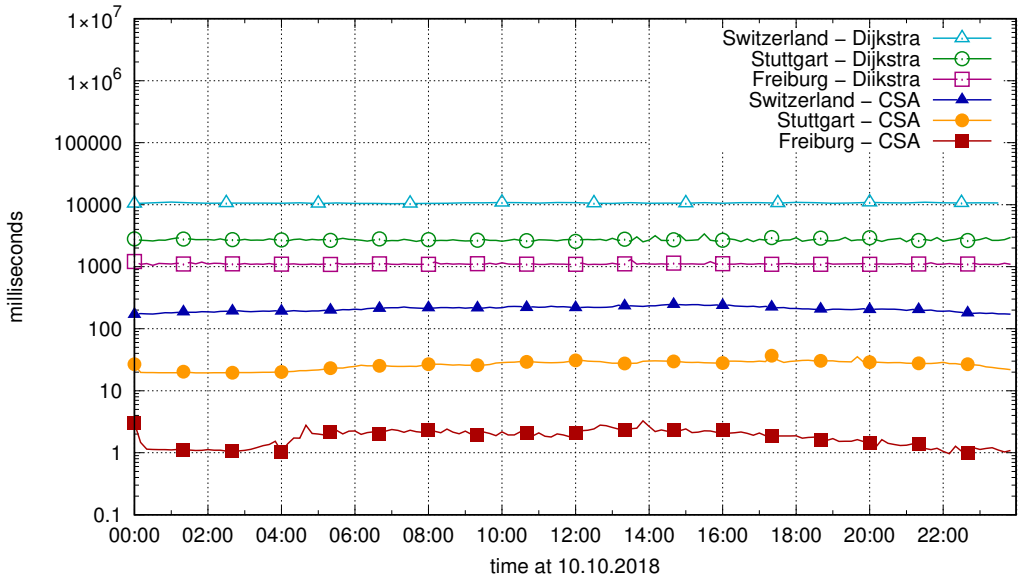


Fig. 6.5: Query durations of an time-independent variant of Dijkstra and CSA for three data sets, measured in milliseconds on a logarithmic scale. Measurements are done for every 10 minutes of the 10.10.2018, averaged over 50 random queries. Dijkstra for Switzerland is measured in steps of 30 minutes.

the source to the destination, limited to the road network, is run. In total this makes

2× 3-nearest neighbor queries from source and destination,

6× Dijkstra from source and destination to access nodes,

9× CSA between access nodes,

1× Dijkstra from source to destination, limited to the road graph.

The measurement is done similar to the experiments for uni-modal time-independent routing, as seen in **Fig. 6.2**, measuring for specific increasing *Dikstra rank*s. Additionally, the measurement is fixed to the 10.10.2018 at 12:00 *pm*. The first experiment has no limitations on the transportation modes. All modes of the set

$$\{\text{car}, \text{bike}, \text{foot}, \text{tram}\}$$

are available, while the second experiment limits the available modes to

$$\{\text{bike}, \text{tram}\}.$$

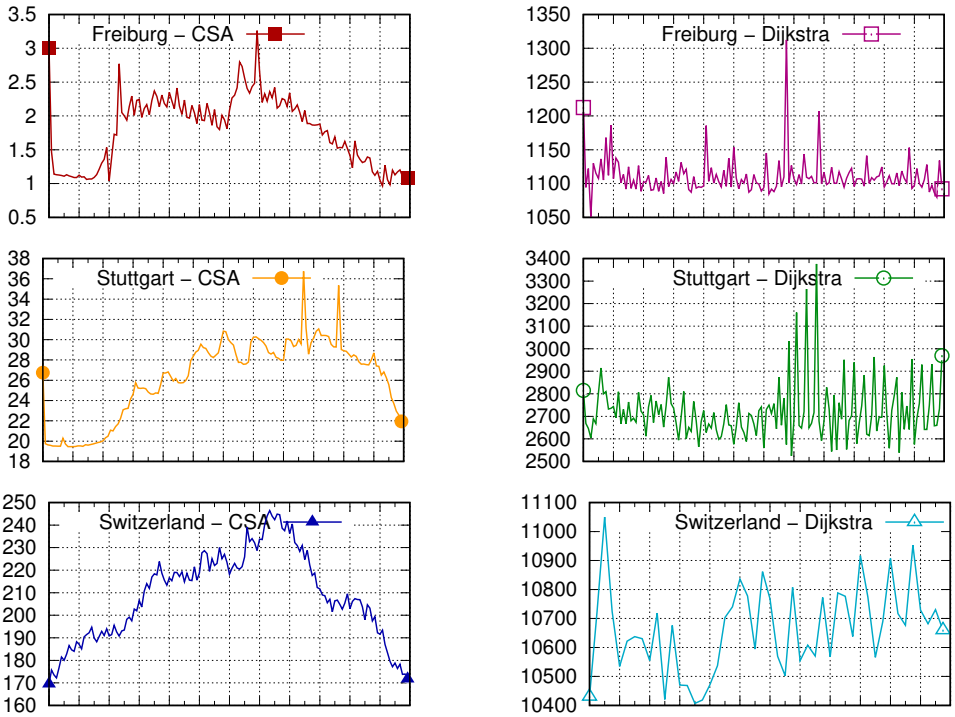


Fig. 6.6: Results from **Fig. 6.5**, but isolated and with a linear scale for the query duration. The duration is measured in milliseconds and all graphs range from 12:00 *am* to 11:59 *pm* for the 10.10.2018.

The results are given by **Fig. 6.7** and **Fig. 6.8** respectively.

Transportation mode restrictions do not impair the running time of DIJKSTRA or ANR. Which is due to DIJKSTRA not using any optimizations relying on transportation modes. Computation is done on the fly, without using precomputed results. The same holds for the simplified ANR, which uses ordinary DIJKSTRA and CSA. Unfortunately, optimizations like ALT do not adapt well to multi-modal route planning, since the precomputation must be done under the assumption of specific transportation modes restrictions, which might be different at query time.

A key problem of DIJKSTRA on link graphs is that its running time is not applicable for long range queries and that a link graph scales very bad in space consumption. In our experiments, the link graph for Switzerland consumes approximately 75 GB, while ANR allocates only about 15 GB for the road graph and the timetable.

As expected, the simplified version of ANR does not beat the ordinary DIJKSTRA, as it still needs to compute long range routes on the road graph using DIJKSTRA. The key problem of our approach is that access nodes, which are chosen as nearest neighbors, might be far away or not even be reachable when using the road network. Geographic

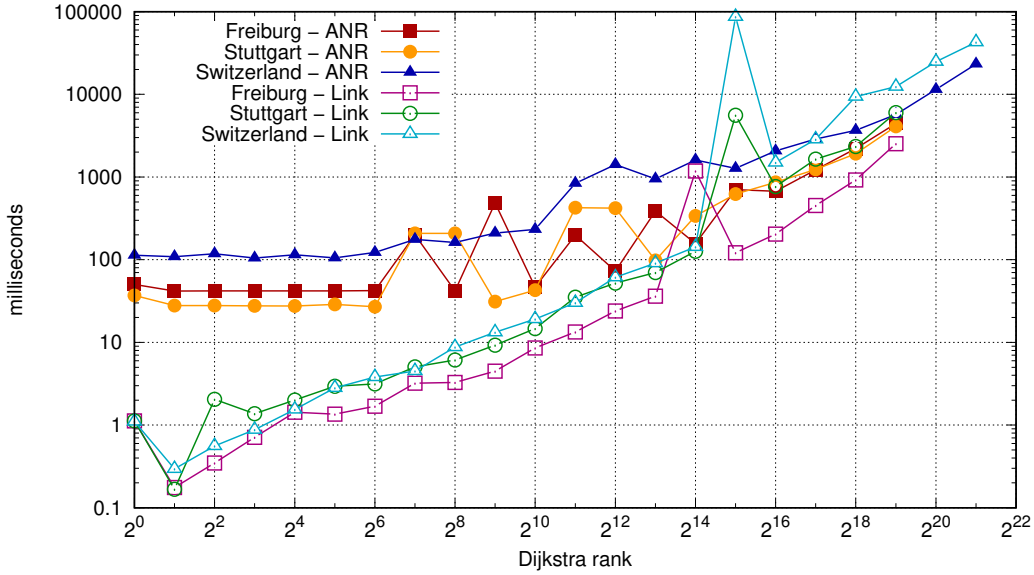


Fig. 6.7: Results of multi-modal route planning with transportation modes {car, bike, foot, tram}. Query duration of Dijkstra on a link graph and a simplified version of ANR using Dijkstra on a road graph and CSA on a timetable are shown. Measurements are averaged over 50 random queries with the specified Dijkstra rank. Query duration is measured in milliseconds, presented on a logarithmic scale.

proximity does not necessarily imply short travel times. In this case, the 6 short range Dijkstra computations are actually long range computations, for which Dijkstra scales bad.

However, ANR has one major advantage over Dijkstra. It can use any algorithm that computes shortest paths on a road network. This stands in contrast to the link graph approach which needs an algorithm that is able to route on a combined network, containing road and transit data. Because of that, a well implemented ANR uses a fast algorithm for road networks (compare to **Fig. 6.4**) and selects access nodes more sophisticated. Which leads to ANR easily beating the query time of Dijkstra on link graphs, making it a feasible approach for multi-modal route planning (see [**23**]).

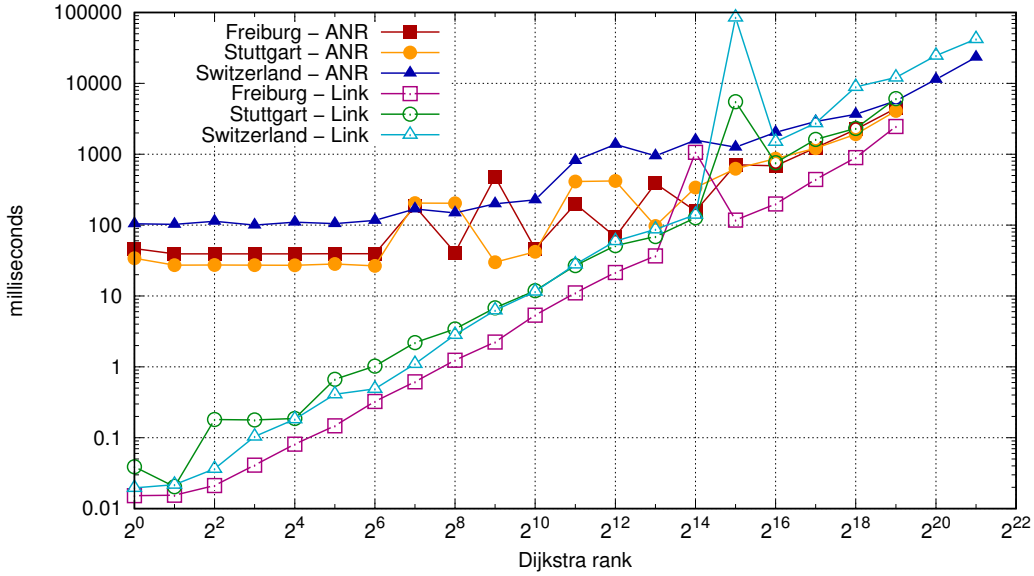


Fig. 6.8: Experiment from **Fig. 6.7**, but restricted to the transportation modes {bike, tram}.

Section 7

# Conclusion

Route planning is a problem of much interest that gained a lot of interest in the last decades. Problem settings like uni-modal route planning are well researched, efficient solutions were developed. Corresponding research is now focused on multi-modal routing and other difficult problems occurring in practice, such as turn penalties and multi criteria routing.

We have presented common and established models for road and transit networks. Graph based solutions are straightforward representations of the network, but can not easily adapt to time dependent data, such as transit networks. Timetables are non-graph based alternatives for public transit networks which fit their structure better than static graphs. Additionally, a link graph can be used to combine graph based models for multiple networks in a straightforward manner. While it might not necessarily be an effective approach, it makes route planning on combined networks for graph based algorithms possible.

In order to explain more sophisticated route planning approaches, we presented the Nearest Neighbor Problem and thoroughly discussed an efficient solution to the problem and various variants, using Cover Trees.

We covered basic route planning algorithms, such as Dijkstra and common optimizations like A$^\star$. The effectiveness of A$^\star$ heavily relies on the chosen heuristic, which depends on the underlying structure of the network. ALT was presented as a solution to this problem, providing a general applicable heuristic which is based on the actual shortest path distances to chosen landmarks. For an overview of more sophisticated uni-modal time-independent algorithms, we refer to [16].

CSA was introduced as an efficient approach for time-dependent route planning on timetables. The approach is very simple, it just processes all connections available after the initial departure time. CSA is fast because it heavily exploits cache locality [32] and other low-level optimizations for arrays.

For multi-modal route planning we showed how Dijkstra can be adapted to run on a link graph, representing a combined network. Further, we presented the general concept of ANR and proposed a simplified variant of it, generalized to an arbitrary algorithm for road networks and another algorithm for transit networks. This makes it possible to combine a graph based solution like Dijkstra, or even more sophisticated approaches, for the road network, with a timetable based approach for transit networks, such as CSA.

Further, we presented experimental results of implementations in the Cobweb project [44] and discussed them. For the experiments three data sets are used, Freiburg,

Stuttgart and Switzerland. The setup, as well as the structure of the input data, was thoroughly explained. Cover Trees and CSA turned out to be a a very efficient solution to their corresponding problems. Dijkstra works well for short range queries, but scales bad for increasing ranges. Further, it lacks behind more sophisticated approaches as seen in **Fig. 6.4**. A$^\star$ using asTheCrowFlies does not perform well on networks used for road planning. While ALT, if carefully implemented, typically beats Dijkstra, especially for mid to long range queries. In practice, link graphs are often not feasible due to extreme demands on space capacity. For multi-modal routing Dijkstra performs similar to uni-modal routing, being feasible for short range queries, but scaling bad for increasing ranges. ANR, if paired with efficient algorithms for both networks, is a promising approach to multi-modal route planning, as seen in [**23**].

Route planning, in particular in practice, is a complex topic. A typical application needs to account for more than just finding a route with the shortest travel time. Turn penalties and multi-criteria routing, such as the cost of a trip, are important factors for a client and need to be considered. A similar observation is done for multi-modal routing, where transportation mode restrictions, in practice, are not just a set of available modes, but rather a complex model with multiple states depending on previous states, as explained in **Section 5.3**.

Most algorithms do not adapt well to such restrictions, leading to the development of many very specialized solutions. Because of that, existing approaches, such as ANR, rather try to combine multiple algorithms, all suited well for their own specialized type of network. In particular for multi-modal routing, including common restrictions occurring in practice, there does not yet exist a feasible solution for networks of a large scale, such as big countries or even continents. However, with increasing research in the last decade, many promising approaches were developed and a solution does not seem too far.

# References


[1] Bzip2 specification. https://github.com/dsnet/compress/blob/master/doc/bzip2-format.pdf. Accessed: 2018-08-28.

[2] Example gtfs feed | static transit | google developers. https://developers.google.com/transit/gtfs/examples/gtfs-feed. Accessed: 2018-08-22.

[3] Freiburg (gtfs): Fritz informationsportal - open data - vag fahrplandaten. https://fritz.freiburg.de/csv_Downloads/VAGFR.zip. Accessed: 2018-08-22.

[4] Freiburg (osm): Geofabrik download server. http://download.geofabrik.de/europe/germany/baden-wuerttemberg/freiburg-regbez.html. Accessed: 2018-08-22.

[5] Osm xml - openstreetmap wiki. https://wiki.openstreetmap.org/wiki/OSM_XML. Accessed: 2018-08-22.

[6] Stuttgart (osm): Geofabrik download server. http://download.geofabrik.de/europe/germany/baden-wuerttemberg/stuttgart-regbez.html. Accessed: 2018-08-22.

[7] Switzerland (gtfs): Fahrplan 2018 (gtfs) - daten | open-data-plattform öv schweiz. https://opentransportdata.swiss/dataset/timetable-2018-gtfs. Accessed: 2018-08-22.

[8] Switzerland (osm): Geofabrik download server. http://download.geofabrik.de/europe/switzerland.html. Accessed: 2018-08-22.

[9] Vvs verkehrs- und tarifverbund stuttgart. http://www.vvs.de/. Accessed: 2018-08-22.

[10] Xz specification. https://tukaani.org/xz/xz-file-format.txt. Accessed: 2018-08-28.

[11] Mohammad Reza Abbasifard, Bijan Ghahremani, and Hassan Naderi. A survey on nearest neighbor search methods. 2014.

[12] Alexandr Andoni. Nearest neighbor search : the old , the new , and the impossible by alexandr andoni. 2009.

[13] Aaron Antrim, Sean J Barbeau, et al. The many uses of gtfs data–opening the door to transit and multimodal applications. *Location-Aware Information Systems Laboratory at the University of South Florida*, 4, 2013.

[14] Chris Barrett, Riko Jacob, and Madhav Marathe. Formal-language-constrained path problems. *SIAM J. Comput.*, 30(3):809–837, May 2000.

[15] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, pages 290–301, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[16] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*, pages 19–80. Springer International Publishing, Cham, 2016.

[17] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[18] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 97–104, New York, NY, USA, 2006. ACM.

[19] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, April 1973.

[20] William B Cavnar, John M Trenkle, et al. N-gram-based text categorization. *Ann arbor mi*, 48113(2):161–175, 1994.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[22] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*, volume 283. Addison-Wesley Reading, 2010.

[23] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating multi-modal route planning by access-nodes. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 587–598, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[24] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. *Engineering Route Planning Algorithms*, pages 117–139. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[25] L. Peter Deutsch, Jean-Loup Gailly, Mark Adler, L. Peter Deutsch, and Glenn Randers-Pehrson. Gzip file format specification version 4.3. RFC 1952, RFC Editor, May 1996. http://www.rfc-editor.org/rfc/rfc1952.txt.

[26] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm. *CoRR*, abs/1703.05997, 2017.

[27] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[28] Wei Dong. An overview of in-vehicle route guidance system. In *Australasian Transport Research Forum*, volume 2011. Citeseer, 2011.

[29] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing. RFC 7230, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7230.txt.

[30] R. L. French. Historical overview of automobile navigation technology. In *36th IEEE Vehicular Technology Conference*, volume 36, pages 350–358, May 1986.

[31] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[32] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *ACM SIGPLAN Notices*, volume 28, pages 177–186. ACM, 1993.

[33] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *Ieee Pervas Comput*, 7(4):12–18, 2008.

[34] Phillip Katz et al. Zip file format specification, 2014. version 6.3.4. url: https://pkware.cachefly.net/webdocs/casestudies/appnote.txt.

[35] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[36] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In Frank Geraets, Leo Kroon, Anita Schoebel, Dorothea Wagner, and Christos D. Zaroliagis, editors, *Algorithmic Methods for Railway Optimization*, pages 67–90, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[37] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001.

[38] John P. Snyder. Flattening the earth: Two thousand years of map projections. 10 1994.

[39] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[40] C. C. Robusto. The cosine-haversine formula. *The American Mathematical Monthly*, 64(1):38–40, 1957.

[41] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Algorithms – ESA 2005*, pages 568–579, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[42] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *J. Exp. Algorithmics*, 5, December 2000.

[43] Daniel Tischner. Lexisearch. https://github.com/ZabuzaW/LexiSearch, 2017.

[44] Daniel Tischner. Cobweb. https://github.com/ZabuzaW/Cobweb, 2018.

[45] Dirck Van Vliet. Improved shortest path algorithms for transport networks. *Transportation Research*, 12(1):7 – 20, 1978.

[46] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[47] Yilin Zhao. *Vehicle location and navigation systems*. 1997.