Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Bc. Jan Hroník

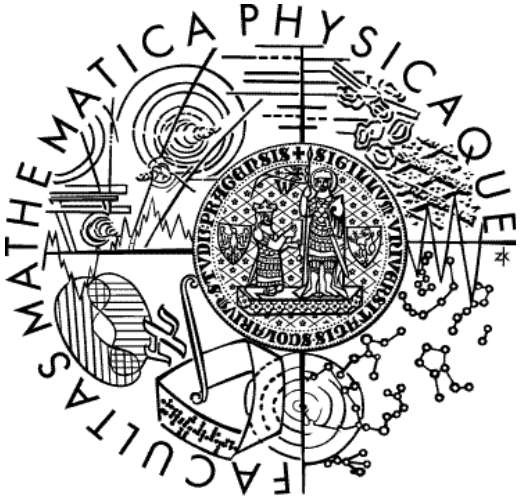
Nejkratší cesty při vyhledávání dopravního spojení

Katedra aplikované matematiky

Vedoucí diplomové práce:	*Mgr. Petr Kolman, Ph.D.*
Studijní program:	*Informatika*
Studijní obor:	*Teoretická informatika*

Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS



Bc. Jan Hroník

Shortest paths when searching for travel connections

Department of Applied Mathematics

Supervisor:	*Mgr. Petr Kolman, Ph.D.*
Program of study:	*Computer Science*
Specialization:	*Theoretical Computer Science*

Děkuji vedoucímu diplomové práce Mgr. Petru Kolmanovi, Ph.D., za ochotné vedení mé diplomové práce a za jeho četné připomínky, které přispěly ke zkvalitnění výsledného textu. Zároveň děkuji Mgr. Robertu Babilonovi za poskytnutí cenných informací v počáteční fázi práce a Dominiku Schultesovi za pohotové poskytnutí důkazů ke své práci. Rovněž bych chtěl poděkovat svým rodičům a přítelkyni Lucy za projevenou podporu.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 9. srpna 2007 Jan Hroník

Table of Contents

Název práce:	**Nejkratší cesty při vyhledávání dopravního spojení**
Autor:	**Bc. Jan Hroník**
Katedra:	**Katedra aplikované matematiky**
Vedoucí diplomové práce:	**Mgr. Petr Kolman, Ph.D.**
e-mail vedoucího:	**kolman@kam.mff.cuni.cz**

Abstrakt: Zabýváme se algoritmy pro hledání nejlepšího spojení podle jízdního řádu, přičemž pojmem nejlepší myslíme nejkratší vzhledem ke zvolenému ohodnocení cest (např. nejrychlejší, nejkratší na počet ujetých km, spojení s nejmenším počtem přestupů). Problém nejkratšího spojení v dopravní síti je formalizován a převeden na problém nejkratší cesty v grafu. K tomu je navržena reprezentace dopravní sítě pomocí orientovaného grafu. Dále je popsáno několik standardních algoritmů pro hledání nejkratších cest v grafu a jejich optimalizace pro použití při hledání dopravních spojení. Nakonec je porovnána výkonnost jednotlivých algoritmů při jejich použití na (1) vlakovou síť pro Českou republiku a (2) na náhodně vygenerovaný graf.

Klíčová slova: dopravní spojení, nejkratší cesty v orientovaném grafu, Dijkstrův algoritmus, obousměrný Dijkstrův algoritmus, algoritmus A* (A-Star), silniční hierarchie

Title:	**Shortest paths when searching for travel connections**
Author:	**Bc. Jan Hroník**
Department:	**Department of Applied Mathematics**
Supervisor:	**Mgr. Petr Kolman, Ph.D.**
Supervisor's e-mail address:	**kolman@kam.mff.cuni.cz**

Abstract: We deal with algorithms for finding cheapest connections in a transportation network with timetables where a cheapest connection is one with the lowest value given some evaluation function. A problem of cheapest connection in a transportation network is introduced and formalized, and is reduced to a shortest path problem in a directed graph. A representation of a transportation network by a directed graph is thereafter given, and several standard algorithms for the shortest path problem are described in turn. Several optimizations of the algorithms for their application to the transportation network are proposed. Eventually, performance results of the algorithms are presented along with their comparison. The algorithms were tested on (1) the train network of the Czech Republic, and on (2) a randomly generated graph.

Keywords: travel connections, shortest paths in a directed graph, Dijkstra's Algorithm, bidirectional Dijkstra's Algorithm, A* (A-Star) algorithm, highway hierarchies

1 Introduction

All of us have already used some search engine to find a suitable train, bus or air plane connection, when we were planning an itinerary for a trip. There are a lot of web-based on-line search engines on the World Wide Web, and some major carriers provide their custom search for travel connections. Yet there is still some space for improvements, especially in the area of search criteria.

Ideally, we would like a system that answers any query quickly and allows for a complex criteria specification – a user should be able to specify what a good connection is. Whereas one user looks for a fast comfortable connection without changes, another may prefer a low fare connection and does not care about the travel time much.

A useful feature of a search engine may be to allow a user to specify a source and a destination *along with* a search radius for both, the source and the destination station, so that a connection *from* a station within the radius of the source station *to* a station within the radius of the destination station may be returned by the system. This feature could be especially useful in a city transportation network where many stations are often packed into a small area so we often do not care from (to) which of them the bus departs (arrives).

In this work, we propose a travel network representation that will allow us to find travel connections which are shortest in time, have lowest fare or encompass a reasonable combination of both the travel time and the travel fare. It will also allow for a search radii to be specified and will allow the user to pose additional constraints. For instance, the user may want to travel with their bicycle, so the system will be able to limit the search only to trains that have the possibility of carriage of large luggage.

The approach used in this work was to exploit the well explored shortest path problem and apply it in the setting of travel connection search. Therefore, we will describe several general algorithms for the shortest path problem and compare their performance in the setting of travel connection search.

2 Shortest Paths and Train Connections

2.1 Definitions

Before we proceed further, we need to introduce a couple of basic definitions. The purpose is to establish basic notation used throughout this work. We are going to present the definition of a directed graph and that of a shortest path. For more details, refer to [2] or any introductory text to graph theory.

Definition A *graph* (also a *directed graph*) G is an ordered pair $G = (V, E)$, where V is a non-empty finite set and $E \subseteq V \times V$. The elements of V are called *nodes*, and the elements of E are called (directed) *edges*.

Definition Let $G=(V, E)$ be a graph. We define the *reverse graph of* G, as a graph $\bar{G}=(V, \bar{E})$, where for each $(v, u) \in V \times V$ we have $(v, u) \in \bar{E} \Leftrightarrow (u, v) \in E$.

Definition Whenever $(u, v) \in E$, we call v to be an *adjacent node of* u. Furthermore, for a node v, each edge (u, v) will be called an *incoming edge to* v, and each edge (v, w) will be called an *outgoing edge from* v. The sum of the number of incoming edges to v and the number of outgoing edges from v is referred to as the *degree of* v.

Definition We will need a value assigned to each edge (this value will represent the cost of choosing the edge when traversing a graph). Therefore, we define the *weight function* $w : E \to \mathbb{R}^+$, where E is a set of edges[1].

Definition A *path* P from a node $u \in V$ to a node $v \in V$, is a sequence of nodes u_1 ..., u_n such that $n \geq 0$, $u_1 = u$, $u_n = v$, and for each $1 \leq i \leq n-1$ we have $(u_i, u_{i+1}) \in E$ and $u_i = u_j \Rightarrow i = j$, that is, there are edges connecting each subsequent pair of nodes on the path, and each node can lie on a path at most once. We write $P = \langle u_1, ..., u_n \rangle$.

Definition Let $P = \langle u_1, ..., x, ..., y, ..., u_n \rangle$ be a path. A *sub-path of path* P, denoted by

1 Usually a more general definition $w : E \to \mathbb{R}$ is used, but for the purposes of this work, only non-negative weight functions will be needed.

$P|_{x \to y}$, is a sequence of nodes x, ... ,y. Note that a sub-path is also a path.

Definition *Length of a path* $P = \langle u_1, ..., u_k \rangle$, denoted by $l(P)$, is defined as the number of nodes on P, that is, $l(P) = k$

Definition *Weight of a path* $P = \langle u_1, ..., u_k \rangle$, denoted by $\boldsymbol{w}(P)$, is defined as the sum of the weights of the edges on this path:

$$\boldsymbol{w}(P) = \sum_{i=1}^{k-1} c(u_i, u_{i+1})$$

Definition A *shortest path* $P = \langle u, ..., v \rangle$ *from* $u \in V$ to $v \in V$ is a path, such that $\boldsymbol{w}(P) = min\{\boldsymbol{w}(Q) \mid Q \text{ is a path from } u \text{ to } v\}$. Note that a shortest path from u to v does not exist if and only if there is no path from u to v.

Definition Let $s \in V$ and $u \in V$ be nodes. We define the *distance* δ *from s to u* to be the weight of the shortest path from s to u, or ∞ if no path exists from u to v.

Definition Let $S \subseteq V$ and $T \subseteq V$ be non-empty sets of nodes. We define the *distance from S to T*, denoted by $\delta(S, T)$, to be the minimum of the weights of all shortest paths from any $s \in S$ to any $t \in T$:

$\delta(S, T) = min\ \{\delta(s, t) \mid s \in S,\ t \in T\}$

For the special case $T = \{v\}$, we simply write $\delta(S, v)$.

Definition We say that a path $P = \langle u, ..., v \rangle$ from $u \in S$ to $v \in T$ is a *shortest path from S to T* if and only if $\delta(u, v) = \delta(S, T)$.

2.2 Shortest path problems

From this section onwards, when we refer to a graph (unless otherwise specified) we are always referring to a directed graph $G = (V, E)$ and consider a weight function $w : E \to \mathbb{R}^+$ for this graph.

We introduce the shortest path problem, which is the most important problem for us, as finding connections in a transportation network can be easily reduced to the shortest path problem, as we will see in the next section.

We can distinguish among the following variants of the shortest path problem:

- *The single-pair shortest path problem* is usually defined as the problem of finding a single shortest path from a node (source) to a node (destination). For the purposes of this work, when we refer to the single-pair shortest path problem, we mean a more general problem of finding a *single* shortest path

between a pair of node sets, that is, we are trying to find a shortest path from $S \subseteq V$ to $T \subseteq V$. Notice that this problem differs from the following two in that its solutions is at most one shortest path, whereas in the two below, it is some finite number of shortest paths.

- *The single-source shortest path problem* is the problem of finding a shortest path from a given (source) node to all the nodes in the graph.
- *The all-pairs shortest path problem* is concerned with finding a shortest path between any two nodes in a graph, i.e., the goal is to find a path from *u* to *v* for all $u, v \subseteq V$.

We will be concerned exclusively with the first from the three problems, that is, with finding a single shortest path between a pair of node sets. Thus, whenever we talk about the shortest path problem, we mean the single-pair shortest path problem, as described above.

2.3 Train connections

2.3.1 Definitions

Let us first formalize relevant aspects of some of the real-world notions, every traveller knows like the palm of their hand.

Definition Let *St* denote some finite set of stations $\mathbb{N} \ni n \geq 2$. A *line* $l = \langle d_1, ..., d_n \rangle \in L$ is a sequence of *stop descriptors*, where

- for all $1 \leq i \leq n$ we have $d_i = (s_i, T_i)$, where $s_i \in St$ is a station and $T_i \in \{0, 1, ..., 1339\}$ represents a time in minutes relative to 12 am
- $s_1 \neq s_2$, $s_{n-1} \neq s_n$
- for all $2 \leq i \leq n-2$ we have $s_i = s_{i+1} \Leftrightarrow s_{i+1} \neq s_{i+2}$.

The meaning of this definition is that whenever a train stops at a station s_i, then $s_i = s_{i+1}$, T_i represents the arrival time to s_i, and T_{i+1} represents the departure time from s_i.

Definition A *transportation network* is a triplet $N = (St, L, c)$, where *St* represents a set of stations, L is a set of lines and $c : L \times \mathbb{N} \rightarrow (\mathbb{R}^+)^a$, for some $a \in \mathbb{N}$, is a cost function.

For a line $\langle d_1, ..., d_n \rangle$, $c(\langle d_1, ..., d_n \rangle, i)$ represents the cost of travelling from a station

s_i to a station s_{i+1} if $s_i \neq s_{i+1}$ or the cost of waiting at $s_i = s_{i+1}$. Instead of choosing a set of scalar values, a more general approach was needed here. The reason for this is that the cost of a trip may have various attributes, such as the time spent, price of the ticket, etc. The convention is that the *first coordinate of the vector is the time portion* of the cost.

We require that whenever $s_i = s_{i+1}$ for a line $l = \langle d_1, \ldots, d_n \rangle$ (train is waiting at a station $s = s_i = s_{i+1}$), we have $c(l,i) = v(T) = \langle T, 0, \ldots, 0 \rangle$, where T represents the waiting time at station *s*.

In addition to this, we define a weight function $w : (\mathbb{R}^+)^a \to (\mathbb{R}^+)$ for a transportation network N as a *linear* mapping from costs to scalar values, that is, $w(\langle a_1, \ldots, a_n \rangle) = k_1 \cdot a_1 + \ldots + k_n \cdot a_n$ for some $k_1, \ldots, k_n$. Whereas the cost function represents costs of connections in a given transportation network, the weight function represents rather a traveller's preference. For instance one traveller may prefer cheap fare over comfort, while another one may value short trips more, so we can represent each traveller's preference by a weight function.

Moreover, the weight function will allow us to weight whole trips in a more straightforward way.

Definition We define the *weight of a line* $l = \langle d_1, \ldots, d_n \rangle$ as follows:

$$w(l) = \sum_{i=1}^{n} w(c(l,i))$$

Definitions For a line $l = \langle d_1, \ldots, d_n \rangle$ and $1 \leq i < j \leq n$, we denote a subsequence $\langle d_i, \ldots, d_j \rangle$ of l by $l(i,j)$.

Note that a subsequence of a line is also a line.

For a line $l = \langle d_1, \ldots, d_n \rangle$ and $1 \leq i \leq n$, we denote the *i*-th element of l by $l(i)$ and the *n*-th element of l by $l(\infty)$. that is, $l(i) = d_i$, and $l(\infty) = d_n$.

Furthermore, for a line $l = \langle (s_1, T_1), \ldots, (s_n, T_n) \rangle$ and $1 \leq i \leq n$, we denote the *i*-th station part of the stop descriptor d_i by $l(i).station$ and the *i*-th time part of the stop descriptor d_i by $l(i).time$. That is, $l(i).station = s_i$, and $l(i).time = T_i$.

Definition A *connection* C in a transportation network $N = (St, L, c)$ from a station $s \in St$ to a station $t \in St$, where $s \neq t$, is a sequence $C = \langle l_1, \ldots, l_n \rangle$ of lines, where $n \geq 1$ and the following properties hold:

- for each $1 \le i \le n$ we have that l_i is a subsequence of some line $\bar{l}_i \in L$, that is, $l_i = \bar{l}_i(j_i, k_i)$ for some valid j_i and k_i.
- for each $1 \le i \le n$ and $l_i = \bar{l}_i(j_i, k_i)$, we have $j_i < k_i$
- $s = l_1(1)$ and $l_n(\infty) = t$
- for each $1 \le i < n$, and $\bar{l}_i, \bar{l}_{i+1} \in L$, where $l_i = \bar{l}_i(j_i, k_i)$ and $l_{i+1} = \bar{l}_{i+1}(j_{i+1}, k_{i+1})$, we have $l_i(\infty).station = l_{i+1}(1).station$

That is to say, C is a connection from *s* to *t* if $\langle l_1, ..., l_n \rangle$ is a sequence of lines such that each of them is a subsequence of some line in *L*, l_1 departs from *s*, each l_i carries us to a successive station, where we can change to l_{i+1} and l_n stops at *t*.

Definition Let $C = \langle l_1, ..., l_n \rangle$ be a connection such that for each $1 \le i \le n$ we have $l_i = \bar{l}_i(j_i, k_i)$ for some $\bar{l}_i \in L$. We define the *length of a connection C*, denoted by $l(C)$, as the number of stations on *C*, that is, $l(C) = \left| \{ l_i(j).station \mid 1 \le i \le n \wedge j_i \le j \le k_i \} \right|$.

2.3.2 Cheapest connection problem

The problem we are concerned with is to find a connection with the smallest weight for given *s* and *t*, at a time *T*, in a given transportation network. As we stated in the introduction, we also want it to be possible to specify radii for *s* and *t*, so that a cheapest connection from a station within the radius of *s* to a station within the radius of *t* will be found. As we will see, we will solve this last requirement in a more general way, by considering a set of origin stations and a set of destination stations. But before we present a formal definition of the cheapest connection problem, we first have to define the weight of a connection.

It might sound reasonable to define the weight of a connection as the sum of the weights of all the lines in this connection, and in fact, it is not much far from the definition we are about to present. Nevertheless, it is not enough, as we should also consider the waiting times between the lines of the connection.

Definition We define a convenience operator for a correct subtraction of relative times $0 \le T_1, T_2 < 1440$:

$$T_1 \ominus T_2 = \begin{cases} T_1 - T_2, & \text{if } T_1 \ge T_2 \\ 1440 + T_1 - T_2, & \text{otherwise} \end{cases}$$

Definition We define the *weight* $w(C,T)$ *of a connection* $C=\langle l_1,...,l_n\rangle$ *at time* T in the following way:

$w(C,T)=\sum_{i=1}^{n} w(l_i)+\sum_{i=1}^{n} w(v(T_i))$, where

$T_1=l_1(1).time \ominus T$, and

for each $2\leq i\leq n$ we have $T_i=l_i(1).time \ominus l_{i-1}(\infty).time$.

Now we can give a definition of the cheapest connection problem. Let $N=(St,L,c)$ be a transportation network, $R\subseteq St$ a set of (origin) stations, $D\subseteq St$ a set of (destination) stations and a relative time T. The *cheapest connection problem* is a problem of finding a connection C in a transportation network N from an origin station $r\in R$ to a destination station $d\in D$ at time T, such that $w(C,T)\leq w(\bar{C},T)$, for all connections $\bar{C}$ such that $\bar{C}$ is a connection from $\bar{r}\in R$ to $\bar{d}\in D$ in N. In the rest of this work, *cheapest connection query* will refer to a given sets of stations R, D and a relative time T.

There are two possible approaches we can take to solve the cheapest connection problem. The first one is to use a different graph definition and/or cost function to the one presented earlier, so that we can represent the transportation network in a straightforward way. That is, we could use a multi-graph that would allow us to represent the different line edges that may go between any two stations or we could possibly use a time-dependent cost function to easily obtain the time cost of getting from a given node to another node at a given time.

The second approach is to make up a way of representing the transportation network by a directed graph, in such a manner that finding a shortest path in this graph will be equivalent to finding a cheapest connection in the transportation network.

We decided to go for the second of the two described approaches, for the following two reasons:

- Due to a long history of the shortest path problem, and its being a target of many researchers, there exist a lot of efficient algorithms for solving this problem. Thus we will be able to choose any of these algorithms and apply them indirectly on the cheapest connection problem by finding a shortest path in our graph representation of the transportation network.
- It is usual that for a given query actually more than one cheapest connection exists in a transportation network and we want our search algorithm to find the one of them that has some additional qualities in comparison to the other connections. For example, we would probably prefer a connection with the least changes, with the

latest departure from the origin station, etc. This seems to be achievable more easily in our graph representation of the transportation network by easy modifications to the search algorithm.

In the following section, a graph representation of the transportation network will be described.

2.3.3 Graph representation of a transportation network

Now that we decided to make up a graph representation for the transportation network, the cheapest connection problem boils down to finding one. Once we have it, we will be able to apply any of the existing algorithms for the shortest path problem on our graph, thus solving the cheapest connection problem.

So let us introduce the graph representation of a transportation network. For an intuitive understanding, let us first describe what the graph will look like informally, providing a formal description later on.

We begin with the description of the nodes and will continue by the description of the edges of the graph.

The graph will contain several nodes for each station. The nodes will be of two kinds. First there will be exactly one node for each arrival to each station. We will call these nodes the *arrival nodes*. Second, there will be a node representing a departure *time* from a station. We will call these nodes the *departure nodes*. One departure node may actually represent various departures from a given station, whenever all these departures happen at the same time.

Now we turn to the description of the edges of our graph. There will be four sorts of edges.

There will be two kinds of outgoing edges for each departure node. First, an edge will be added from a departure node to the *earliest following* departure node *at the same station*. When we say following departure, we mean the next departure in time. We will call these edges the *wait edges*. In other words, for each station there will be a succession of departure nodes, such that each of the departure nodes is connected just with the earliest following departure node.

The second kind of outgoing edge will be the *departure edge* which connects the given departure node for station s_1 with an arrival node in the following station s_2, whenever there is a line that goes from s_1 to s_2 directly (i.e., without intermediate stops). There may be as many departure edges from a departure node, as there are lines departing at the given time from the station.

We now turn to the description of the two remaining kinds of edges. These will correspond to the arrival nodes. The first one (a *change edge*) connects an arrival node with the next departure time at the same station and the second one (a *continuation edge*) connects the node with the arrival node at the following station corresponding to the same line. This way we represent the fact that when we arrive to a station, we can either choose to get off the train and wait for another train (change edge), or stay on the train and continue to the following station (continuation edge).

The last thing to mention is how to represent the travel costs of the transportation network *N*. We will consider a weight function w_N for *N* and will want to concoct a weight function w_G for our graph. We leave the details of this to the formal definition below, and just briefly summarize what it says. The weights for departure and continuation edges can be obtained directly from w_N, as each of these edges have their cost assigned in *N*. Less clear is how we should define the weights for the change edges and the wait edges, since these edges do not originate in the transportation network.

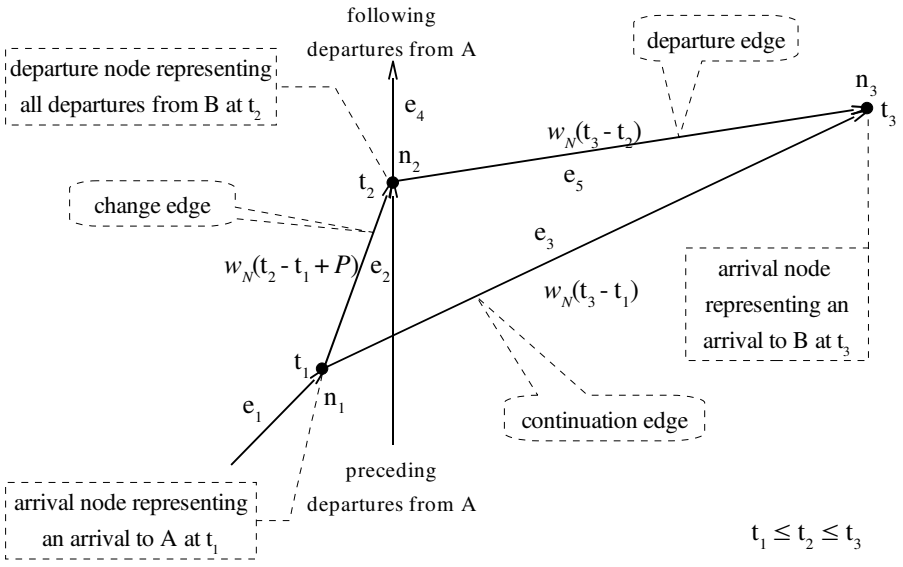


Fig. 2.3.1.: When we arrive at station A (node n_1), we can either get off at the station to opt for another train (change edge e_2), or we can stay on the train and continue to the next station (continuation edge e_3). Note that for the resulting graph to work correctly, it is necessary that n_2 is a departure node representing the *earliest* departure (not necessarily the departure of the same line as shown in this figure), such that $t_1 \le t_2$. The wait edge e_4 connects the departure node n_2 with the closest departure node at this station. Finally, there is just one departure edge in this example, namely the edge e_5. In this example we assume that the weight function is one-dimensional, i.e., $a = 1$, and $w_N : \mathbb{R}^+ \to \mathbb{R}^+$.

Nevertheless, we will solve it again using the weight function w_N, by applying it on a cost vector $v(T)$ for each edge representing a waiting time T, thus obtaining a real valued weight for the corresponding edge.

Note that the introduction of change edges makes it easy to charge a penalty for each change, which may make the algorithm prefer (sub-optimal) connections with fewer changes. This could be done by adding a value P to the first coordinate of the corresponding vector. See Fig. 2.3.1 on page 9 for an illustration of the graph construction considering just one line and Fig. 2.3.2 on page 10 for a more complete example encompassing various lines.

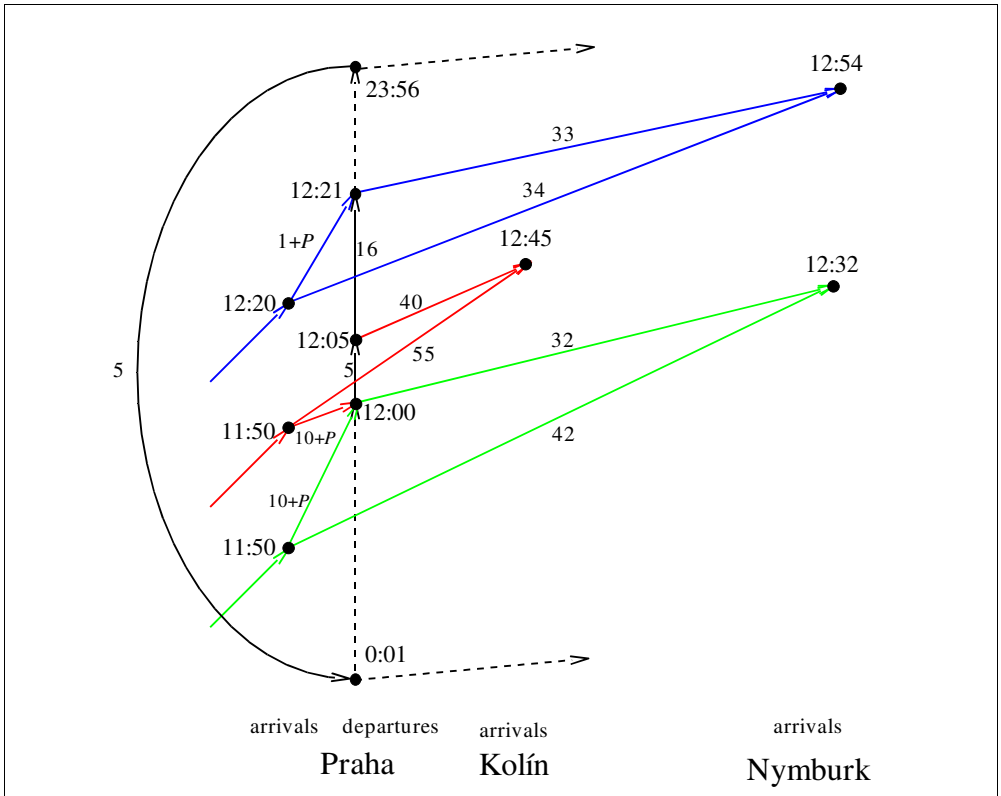


Fig. 2.3.2.: We again assume that w_N is a one-dimensional weight function (i.e., $a = 1$, and $w_N : \mathbb{R}^+ \to \mathbb{R}^+$) but for brevity leave out the function name w_N and just specify the function's arguments next to each edge. Note that the edge connecting the last departure of the day (23:59) with the first departure of the day (0:01) is necessary in order for the overnight connections to be considered in the search. In this example you can see the case, when a change edge is not connected to the departure node of the same line. This is the case of the red line. P is a change penalty parameter.

2.3.4 Transportation graph

Let us formalize what we described informally in the previous section. Let

$N=(St, L, c)$ be a transportation network and w_N a weight function for *N*. For a given change penalty *P*, we give a definition of a graph representation $G_N=(V_N, E_N)$ of the transportation network *N* and a weight function w_G for G_N. We will call G_N the transportation graph for *N*, or shortly, the *transportation graph*.

1) We define the set of nodes V_N of G_N as follows:

For each line $l_k=\langle(s_1, T_1), \ldots, (s_n, T_n)\rangle \in L$, and for each $1 \le i < n$ and each s_i such that $s_i \neq s_{i+1}$, we add the following nodes to V_N:

- $d[s_i, T_i] \in V_N$ (s_i-departure nodes)
- $a[s_{i+1}, T_{i+1}, k] \in V_N$ (s_{i+1}-arrival nodes)

Note, that it follows from this definition that whereas there is an arrival node for *each* arrival to a station, there may be *just one* departure node *for several* departures from a station whenever the departing time is the same for all these departures.

Before we proceed to the definition of E_N, we define an auxiliary function $ND_{G_N}(s, T_1)$ for G_N, station *s* and time T_1 that will give us a means of referring to the *s*-departure node most closely after the time T_1:

$$ND_{G_N}(s, T_1) = \begin{cases} \min_{T_2} \{d[s, T_2] \in V_N | T_2 \ge T_1\}, & \text{if } \{d[s, T_2] \in V_N | T_2 \ge T_1\} \neq \emptyset \\ \min_{T_2} \{d[s, T_2] \in V_N\}, & \text{otherwise} \end{cases}$$

2) We define the set of edges E_N of G_N as follows:

For each line $l_k=\langle(s_1, T_1), \ldots, (s_n, T_n)\rangle \in L$, and for each $3 \le i < n$ and s_i such that $s_i \neq s_{i+1}$ we add the following edges to E_N:

- $(d[s_1, T_1], a[s_2, T_2, k]) \in E_N$ (first departure edge for line *k*)
- $(d[s_i, T_i], a[s_{i+1}, T_{i+1}, k]) \in E_N$ (other departure edges)
- $(a[s_{i-1}, T_{i-1}, k], ND_{G_N}(s_{i-1}, T_{i-1})) \in E_N$ (change edges)
- $(a[s_{i-1}, T_{i-1}, k], a[s_{i+1}, T_{i+1}, k]) \in E_N$ (continuation edges)

Furthermore, for each station $s \in St$ and for each departure node $d(s, T) \in V_N$, we add

the following edge to E_N:

– $(d[s,T], ND_{G_N}(s,T+1)) \in E_N$, (wait edges)

3) We define the weight function w_G on the edges from 2).

Let $\mathbb{R} \ni P \geq 0$ be a change penalty. For each line $l_k = \langle (s_1,T_1),\ldots,(s_n,T_n)\rangle \in L$, $s_i \neq s_{i+1}$ and for each $3 \leq i < n$ and s_i such that $s_i \neq s_{i+1}$ we define w_G as follows:

– (first departure edge for line *k*) $w_G(d[s_{1,}T_1], a[s_{2,}T_{2,}k]) = w_N(c(l_k,1))$

– (other departure edges) $w_G(d[s_i,T_i], a[s_{i+1},T_{i+1},k]) = w_N(c(l_k,i))$.

– (change edges) $w_G(a[s_{i-1},T_{i-1},k], d[\bar{s}_{i-1},\bar{T}_{i-1}]) = w_N(v(T)) + P$, where $d[\bar{s}_{i-1},\bar{T}_{i-1}] = ND_{G_N}(s_{i-1},T_{i-1})$, and $T = \bar{T}_{i-1} \ominus T_{i-1}$.

– (continuation edges) $w_G(a[s_{i-1},T_{i-1},k], a[s_{i+1},T_{i+1},k]) =$, $= w_N(c(l_k,i-1)) + w_N(c(l_k,i))$.

For each station $s \in St$ and for each departure node $d(s,T) \in V_N$, we define w_G on the corresponding wait edge as follows:

– (wait edges) $w_G(d[s,T], d[\bar{s},\bar{T}]) = w_N(\vec{v}^T)$, where $d[\bar{s},\bar{T}] = ND_{G_N}(s,T)$, and $T = \bar{T} \ominus T$.

2.3.5 Shortest paths and cheapest connections

Our concern now will be to show that the graph representation of a transportation network is reasonable, that is, that a query for a cheapest connection in a transportation network *N* can be transformed into a shortest path query and that a shortest path found by any correct algorithm for the shortest path problem can be transformed into a connection in *N* and that this connection is a cheapest connection for the cheapest connection query. This last statement only holds when the change penalty parameter used in the construction of G_N is zero, as the change penalty parameter may force the algorithm to prefer a connection with fewer changes over a one connection with smallest weight.

Let us use the notation from the previous section, that is, we have a transportation network $N = (St, L, c)$, where $L = \{l_{1,}\ldots,l_z\}$ with a weight function w_N, and a transportation graph $G_N = (V_N, E_N)$ for *N* and a weight function w_G for G_N. Moreover, we will consider a cheapest connection query for a set of origin stations $R \subseteq St$, a set of destination stations $D \subseteq St$, where $R \cap D = \emptyset$, and time $\tilde{T}$ and when

we refer to a shortest path query, we mean a single-pair shortest path query, as defined earlier. Finally, let *A* denote an algorithm for the shortest path problem.

From cheapest connection query to shortest path query

For our cheapest connection query, we construct the following shortest path query:

$S = \{ND_{G_N}(r, \tilde{T}) \mid r \in R\}$ (source set)

and

$T = \{a[d, Y, k] \in G_N \mid d \in D \wedge 0 \leq Y < 1440 \wedge l_k \in L\}$ (target set)

That is to say, *S* is a set of all earliest *r*-departure nodes after time $\tilde{T}$ for all $r \in R$, and *T* contains all *d*-arrival nodes for all $d \in D$.

Please note that we use the words *origin* (station) and *destination* (station), when we talk about cheapest connections, whereas we use the words *source* (node) and *target* (node), whenever we talk about shortest paths.

Let $P = \langle d[r, T_r] = u_1, \ldots, u_n = a[d, T_d, k] \rangle$, for some $r \in R$ and some $d \in D$ be the shortest path found by algorithm *A*. First, let us show how we obtain a connection C_P from *r* to *d* based on *P* and then give a clarification on why this has to be the cheapest connection from *r* to *d* and therefore this connection is a solution to the cheapest connection problem.

From shortest path to a connection

Without loss of generality, we can assume that the algorithm *A* always finds a path that contains no sub-path $\langle a[s_h, \bar{T}_h, k], d[s_{h+1}, \bar{T}_{h+1}], a[s_{h+2}, \bar{T}_{h+2}, k] \rangle$. In other words, we do not get off and get on the same train. For an illustration, see Fig. 2.3.1, where we assume that A always prefers <n_1, n_3> over <n_1, n_2, n_3>. This assumption is justified by the fact that we could always replace each sub-path $\langle n_1, n_2, n_3 \rangle$ with $\langle n_1, n_3 \rangle$, as it follows from the construction of G_N that $w(\langle n_1, n_3 \rangle) \leq w(\langle n_1, n_2, n_3 \rangle)$.

It follows from the definition of G_N that whenever there are two adjacent arrival nodes $u_i = a[s_i, \bar{T}_i, k]$, $u_{i+1} = a[s_{i+1}, \bar{T}_{i+1}, l]$ on *P*, we have (1) $l = k$ since from each arrival node we can either reach a departure node or the next successor arrival node *for the same line*. Moreover, when there is a departure node $u_i = d[s_i, \bar{T}_i]$ followed by an arrival node $u_{i+1} = a[s_{i+1}, \bar{T}_{i+1}, k]$ on *P* then (2) $(s_i, \bar{T}_i)$ is also a direct predecessor of $(s_{i+1}, \bar{T}_{i+1})$ on line *k*. Therefore we can obtain C_P gradually by going through *P* from

the beginning to the end and for each sub-path $\bar{P}$ of *P*, which for some $j \geq 1$ has to have one of the following forms:

- $\bar{P} = \langle d[s_h, \bar{T}_h], a[s_{h+1}, \bar{T}_{h+1}, k], \ldots, a[s_{h+j}, \bar{T}_{h+j}, k], d[s_{h+j+1}, \bar{T}_{h+j+1}] \rangle$ (not the last "fragment"), OR
- $\bar{P} = \langle d[s_h, \bar{T}_h], a[s_{h+1}, \bar{T}_{h+1}, k], \ldots, a[s_{h+j}, \bar{T}_{h+j}, k] \rangle$ (the last "fragment")

and for some $b \geq 1$ we have

a) $d[s_h, \bar{T}_h] = l_k(b)$, and

b) $a[s_{h+m}, \bar{T}_{h+m}] = l_k(b + 2 \cdot m + 1)$ for all $0 \leq m \leq j$,

then we add the line $l_k(b, b + 2 \cdot j + 1)$ to C_P . Note that a line l_k satisfying the criteria a) and b) always exists. This follows from the remarks 1) and 2) stated above, and from the construction of the transportation graph.

C_P thus constructed is a valid connection in *N*, for it is a sequence of valid lines, each of length at least 2, and for every two immediately consequent lines $S_1 = \langle (s_h, \bar{T}_h), \ldots, (s_{h+k}, \bar{T}_{h+k}) \rangle$, and $S_2 = \langle (s_j, \bar{T}_j), \ldots, (s_{j+m}, \bar{T}_{j+m}) \rangle$, we have $s_{h+k} = s_j$, because all nodes in *P*, skipped after S_1 , but before S_2 was added into C_P , had to be departure nodes, and it holds that any two subsequent departure nodes $d[s_1, T_1]$, $d[s_2, T_2]$ on any path have to belong to the same station, that is, $s_1 = s_2$.

We have shown that from a shortest path *P*, we can obtain a valid connection $C_P = \langle l_1, \ldots, l_n \rangle$ in *N*. Moreover, it is a connection from $r \in R$ to $d \in D$, which follows from the way C_P was constructed, and the fact that the first node on *P* is $d[d_r, T_r]$, and the last node on *P* is $a[d, \bar{T}_d, k]$.

Furthermore, from the definition of w_G , it follows that $w_G(P) = w(C_P, T_r)$.

Optimality of found connection

In this section, we propose ourselves to give some insight into why the connection C_P constructed from *P* in the previous section has to be a solution to the cheapest connection problem[2].

We will proceed by contradiction. Let $C = \langle l_1, \ldots, l_n \rangle$ be a connection from $\bar{r} \in R$ to

2 As stated earlier, we have to assume that the change penalty parameter in the construction of G_N was zero.

$\bar{d} \in D$ such that $w_N(C_P, \tilde{T}) > w_N(C, \tilde{T})$. We will construct a path $\bar{P}$ in G_N from *S* to *T*, such that $w(P) > w(\bar{P})$ contradicting our assumption that *P* is a shortest path.

For the purposes of the construction, we define an auxiliary function *Dep* that for a given station $s \in St$ and given relative times T_s, T_e (i.e., $T_s, T_e \in \{0, 1, ..., 1339\}$) will allow us to obtain a sequence of *s*-departure nodes before T_s and after T_e:

$Dep(s, T_s, T_e) = \langle d[s, T_1], ..., d[s, T_p] \rangle$, where

1) for each $1 \le i < p$ we have $(d[s, T_i], d[s, T_{i+1}]) \in E_N$ and
2) for each $1 \le j \le p$ we have $T_j \ominus T_s \le T_e \ominus T_s$ and
3) no node is repeated in the sequence $\langle d[s, T_1], ..., d[s, T_p] \rangle$.

To prove our claim, we start with an empty path and gradually add sequences of nodes to $\bar{P}$, each such a sequence corresponding to a line of *C*, and we proceed from the beginning to the end of *C*. For $i = 1, 2, ..., n$, we process the sequence l_i, by adding all the (departure) nodes from $Dep(l_i(1).station, T'_i, l_i(1).time)$ to the end of $\bar{P}$, where

$T'_i = T_s$ for *i* = 1 and

$T'_i = l_{i-1}(\infty).time$, for $2 \le i \le n$

After that, for each $1 \le j \le m_i/2$, where m_i is the length of the sequence l_i, all the (arrival) nodes $a[l_i(2 \cdot j).station, l_i(2 \cdot j).time, k]$, are added to the end of $\bar{P}$. This concludes the processing of line *i*.

At the end, we obtain a sequence $\bar{P}$ of nodes from G_N. It is a path in G_N, which follows from the construction of G_N and the definition of $Dep(s, T_s, T_e)$. Moreover, it is a path from *S* to *T*. The first sequence added to $\bar{P}$ was $Dep(\bar{r}, T_{\bar{r}}, l_i(1).time)$, whose first node is $ND_{G_N}(\bar{r}, \tilde{T})$. The last node on $\bar{P}$ is $a[\bar{d}, X, y]$, as the last station of the last sequence of *C* is $\bar{d}$ (*C* is a connection from $\bar{r}$ to $\bar{d}$).

Let us now analyse the weight of $\bar{P}$. For each line, we need to consider the cost of the waiting time preceding the change to the corresponding line and the cost of the "travelling" on this line:

1) The cost for waiting time before line start:

$w_N(v(T'))$, where $T' = l_i(1).time \ominus T_{\bar{r}}$ for *i* = 1, and $T' = l_i(1).time \ominus l_{i-1}(\infty).time$ for $2 \le i \le n$.

2) The cost of the "travelling" on this line is $w_N(l_i)$.

The cost of the waiting time at $\bar{r}$ and the waiting time between every two lines on C are counted and the cost of each line is counted. From the definition of w_N, it follows that $w_G(\bar{P}) = w_N(C, T_{\bar{r}})$. Since we also know that $w_G(P) = w_N(C_P, T_r)$, we have $w_G(\bar{P}) = w_N(C, T_{\bar{r}}) < w_N(C_P, T_r) = w_G(P)$, a contradiction with our assumption that P is a shortest path from S to T.

2.3.6 Restricted timetables

There is one important thing that has not been considered in the definition of the transportation network. That thing is that there is no information in the transportation network indicating whether a given line actually operates on a given day. It is a ubiquitous practice that some lines operate just during the weekdays, others just on weekends, whereas some do not run on statutory holidays, etc. Nevertheless, this deficiency can be fixed easily in the search algorithm, by not traversing some of the edges. We will postpone the details of the fixes until we arrive at the descriptions of the particular algorithms.

3 Shortest Path Algorithms

3.1 Dijkstra's Algorithm

The Dijkstra's Algorithm is one of the best known algorithms for the single-source shortest path problem. There are many modifications of the algorithm and some of the recently published algorithms are based on the Dijkstra's Algorithm. We will first present the original version of the algorithm and some of its modifications later on in this chapter.

The algorithm works in the following way: it maintains a priority queue Q of nodes that have been visited so far and a tentative distance $\boldsymbol{d}[u]$ for each node u (we write it in bold not to confuse it with a departure node $d[s, T]$). The priority queue stores ordered pairs (*key*, u), where *key* is a numeric value (usually a distance) and u is a node. Furthermore, we define three basic operations on Q:

- DECREASE-KEY(Q, u, *key*) – changes the key under which u is stored in Q. For DECREASE-KEY to work, it is assumed that, at the time DECREASE-KEY is invoked, u is already a member of Q *and* the new *key* is smaller or equal to the one under which u is currently stored in Q.
- INSERT(Q, u, *key*) – adds a node u to Q with the specified *key*. It is assumed that u is not a node in Q when the INSERT operation is invoked.
- EXTRACT-MIN(Q) – removes a node with the smallest key from Q and returns this node as result.

Each of this operations returns the modified queue (EXTRACT-MIN returns an ordered pair $(\bar{Q}, u)$, where $\bar{Q}$ is the queue after the removal of the node u with the smallest key).

The search is initialized by inserting s into Q, setting $\boldsymbol{d}[s]$ to 0 and $\boldsymbol{d}[u]$ to ∞ for $u \in V \setminus \{ s \}$.

At each iteration of the algorithm, a node u with the smallest tentative distance is extracted from the queue and all outgoing edges are *relaxed* where relaxation of an edge

(u, v) means updating the tentative distance $\boldsymbol{d}[v]$, for the node v and inserting v into Q if it is not yet in Q, or doing nothing, if the new tentative distance is greater or equal to $\boldsymbol{d}[v]$.

The algorithm terminates when the priority queue Q is empty.

Let us introduce three useful terms that will be used not only in the context of the Dijkstra's Algorithm but also in all its modifications described further in this chapter.

A node is called a *reached node* if it is or was an element of Q. Furthermore, we call a node *settled* if it has already been extracted from Q. We call an edge (u, v) *relaxed* if u has been settled during the performance of the algorithm.

We can think of the queue as of a fringe of the search. At each iteration, the node u closest to s is extracted form Q and the fringe is extended to the nodes beyond u, that is, the adjacent nodes of u.

The crucial point about the algorithm is that when a node u is settled, it holds that $\boldsymbol{d}[u] = \delta(s, u)$, that is, the distance from s to u has been found. This statement will be proved in the next section for a generalized version of the Dijkstra's Algorithm (and thus for the original version as well).

Note that it follows that when a node is settled, it will not become an element of Q in the future, an important feature for the complexity of the algorithm. Also note that this only holds for non-negative weight functions, as can be seen in the example in Fig. 3.1.1.

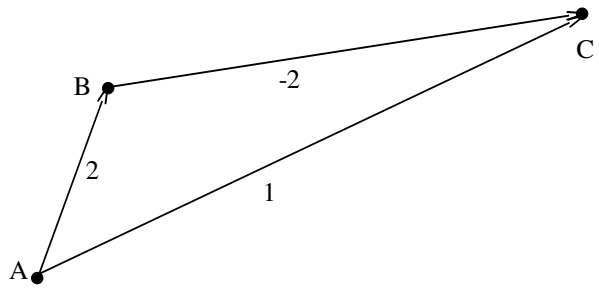


Fig. 3.1.1.: Graph with negative weight function. When searching from A, the settled elements will be (in this order) A, C, B, *C*

When the weight function is non-negative, a node u will not become an element of Q after it is settled, because it would have to be reached with a smaller tentative distance than $\delta(s, u)$ in order to become element of Q again.

After the Dijkstra's algorithm terminates, the distance from s to any node u will be

stored in $\boldsymbol{d}[u]$. But what about the actual shortest *paths*? Paths can be handled easily by maintaining a tentative parent $p[v]$ for each node v, initially set to N/A (no parent). Whenever v is reached from u and $\boldsymbol{d}[v]$ is decreased, we also update the parent $p[v]$ and set it to u. After the search has finished, we can traverse the path from s to v backwards by following the parents of v. ($p[v]$, $p[p[v]]$, etc.), until we reach s. See Fig. 3.1.2 for pseudo-code of the Dijkstra's algorithm and Fig. 3.1.3 for pseudo-code of the RELAX function.

```
DIJKSTRA(G, s, w)
1   d[s] = 0
2   foreach v ∈ V/{s} do
3           d[v] = ∞
4   Q ← S
5   while Q ≠ ∅ do
6           (Q, u) ← EXTRACT-MIN(Q)
7           foreach (u, v) ∈ E do
8                   Q ← RELAX(u, v, w, Q, d, p)
```

Fig. 3.1.2.: The original Dijkstra's Algorithm. G is a graph, s a source node and w a weight function for G.

```
RELAX(u, v, w, Q , d, p)
1   if d[u] + w(u, v) < d[v] then
2           d[v] ← d[u] + w(u, v)
3           p[v] ← u
4           if  v∈Q  then
5                   Q ← DECREASE-KEY(Q, v, d[v])
6           else
7                   Q ← INSERT(Q, v, d[v])
8   return Q
```

Fig. 3.1.3.: The RELAX function. u, v are nodes, w is a weight function, Q a priority queue, d a distance array and p a vector of predecessors. DECREASE-KEY decreases the key under which v is stored in Q. INSERT stores a node in Q with a specified key.

An easy modification to the Dijkstra's algorithm can yield an algorithm for finding a shortest path from a node s to a node t. This can actually reduce the area searched if we are only interested in finding the shortest path between two nodes s and t. We start the search from s as usual but terminate the search after t is settled (a shortest path to t has been found) or the priority queue is empty (there is no path from s to t). In the very

same way, we can modify the algorithm so that it finds a shortest path from a source node s to *any* node from a given set $T \subseteq V$ by terminating after any node from T is settled.

3.1.1 Time complexity of the Dijkstra's Algorithm

Let us analyse the upper bound time complexity of the Dijkstra's Algorithm. As discussed earlier, a node cannot become an element of Q more than once if there are no negative edges in the graph. Therefore the outer loop of the algorithm can be executed at most n times. This implies that an edge cannot be relaxed more than once, so the inner loop cannot be executed more than m times during the execution of the algorithm. This yields a time complexity of $O(n.T_1 + m.T_2)$, where T_1 represents the time complexity of one iteration of the outer loop not counting the inner loop, and T_2 represents the time complexity of one iteration of the inner loop. To define T_1 and T_2, we have to consider the three vital operations the priority queue implementation has to provide: INSERT(Q, u, *key*), DECREASE-KEY(Q, u, *key*), and EXTRACT-MIN(Q).

INSERT and DECREASE-KEY are used in the RELAX function. The EXTRACT-MIN is used in the outer loop to obtain the minimum element of Q and remove it from Q. INSERT inserts a new element into Q, and DECREASE-KEY decreases the key of a node under which it is stored in Q (possibly yielding a higher position in Q or even a new minimum).

A common implementation of a priority queue is the Fibonacci heap, which has the amortized complexities of $O(1)$, $O(1)$ and $O(log\ n)$ for INSERT, DECREASE-KEY, and EXTRACT-MIN, respectively. Thus, the time complexity of the Dijkstra's algorithm using the Fibonacci heap is $O(n.log\ n + m)$.

3.1.2 Multi-Source Dijkstra's Algorithm

Now we present a generalization of the Dijkstra's Algorithm which differs from the original version in that it finds the distances (and the corresponding shortest paths) from a given set $S \subseteq V$ of nodes to all nodes in the graph. In other words, for each node $s \in S$ and each node $v \in V$, the algorithm finds the distance from s to v. As we will see, this version of the Dijkstra's Algorithm can be easily modified to obtain an algorithm for the single-pair shortest path, which is the one of most interest to us.

For a pseudo-code of the algorithm, see Fig. 3.1.4. The only difference from the original version is that all the tentative distances for all nodes in S are initialized to 0. Note, that an algorithm for solving the single-pair shortest path problem can be obtained by adding an additional termination condition: if we consider a set T of target nodes, then we can

terminate the search after a node $u \in T$ is extracted on line 7 of the algorithm from Fig. 3.1.4.

```
MULTI-SOURCE-DIJKSTRA(G, S, w)
1   foreach s ∈ S do
2           d[s] = 0
3   foreach v ∈ V/S do
4           d[v] = ∞
5   Q ← S
6   while Q ≠ ∅ do
7           (Q, u) ← EXTRACT_MIN(Q)
8           Q ← Q\u
9           foreach (u, v) ∈ E do
10                  Q ← RELAX(u, v, w, Q, d, p)
```

Fig. 3.1.4.: Multi-source Dijkstra's Algorithm. G is a graph, S a set of source nodes and w a weight function for G.

The correctness proof for the Dijkstra's Algorithm will be a generalized version of the proof by T. H. Cormen et al. in [3]. But before presenting the correctness theorem for the Dijkstra's Algorithm and its proof, let us introduce a trivial property that will be used in the proof.

Upper-bound property *During the execution of the Dijkstra's Algorithm it holds that* $\mathbf{d}[v] \geq \delta(S, v)$ *for all* $v \in V$ *and once* $\mathbf{d}[v] = \delta(S, v)$, $\mathbf{d}[v]$ *never changes.*

Theorem *The multi-source Dijkstra's Algorithm, run on a directed graph (V, E) with non-negative costs, terminates and after it terminates,* $\mathbf{d}[u] = \delta(S, u)$ *holds for each* $u \in V$.

Proof It suffices to prove the following loop invariant:

After u is settled at line 7, it holds that $\mathbf{d}[u] = \delta(S, u)$. (3.1)

Note that the theorem follows from this invariant:

➔ **Termination***: The algorithm always terminates*. This can be seen by inspecting all operations that change the number of nodes in Q, that is, the lines 5, 7 and 10. On line 5, Q is initialized by a finite set of nodes. On line 7, exactly one node is removed from Q. As line 7 executes in each iteration of the while loop and decreases size of Q by one, it is enough to show that a node cannot be an element of Q more than once during the execution of the loop by inspecting the last remaining operation, which changes the number of nodes in Q. Line 10 is the only line which may increase the number of nodes in Q. The RELAX

function may only increase the size of Q if the condition on line 1 of the RELAX function holds. But for a node v that is no longer a member of Q this condition never holds, since $\boldsymbol{d}[u] + w(u, v) < \boldsymbol{d}[v]$ would contradict the invariant $\boldsymbol{d}[v] = \delta(S, v)$. Therefore a node cannot be a member of Q more than once.

➔ **Optimality***: $\boldsymbol{d}[u] = \delta(S, u)$* for each $u \in V$ after the while loop terminates. It can be easily shown by induction that any node $u \in V$ that is not reachable from S is never reached and therefore $\boldsymbol{d}[u] = \infty = \delta(S, u)$ after the loop terminates. Similarly, it can be shown that a node u that is reachable from S becomes a member of S during the execution of the while loop. It remains to note that $\boldsymbol{d}[u]$ will not be changed after it is extracted from Q on line 7. This follows from the invariant $\boldsymbol{d}[u] = \delta(S, u)$ and the upper-bound property.

Now by proving (3.1), we will conclude the proof of the theorem.

We want to prove that after a node u is settled on line 7, it holds that $\boldsymbol{d}[u] = \delta(S, u)$. It can be shown that any node $v \in V$ lying on a zero-weight shortest path is settled with $\boldsymbol{d}[v] = \delta(S, u) = 0$ *before* any node lying on a non zero-weight shortest path is settled. Namely this holds for each $s \in S$ as it lies on a zero-weight path $<s>$. Therefore, we only deal with nodes $u \in V$ such that $\delta(S, u) \neq 0$.

We will proceed by contradiction. Let us assume that u is the first node for which $\boldsymbol{d}[u] \neq \delta(S, u)$ just before u is settled on line 7. There must be a path from S to u for otherwise $\boldsymbol{d}[u] = \delta(S, u) = \infty$, which would violate our assumption. Since there is a path from S to u, there is a shortest path p from S to u. Let $p = <s, ..., u>$ for some $s \in S$ and let y be the first node on p such that y has not been settled and x be the predecessor of y, that is, x has been settled (there always exists such x and y, because there is at least one settled node, i.e., s and at least one unsettled node, i.e., u, on p).

We will show that $\boldsymbol{d}[y] = \delta(S, y)$ holds when u is about to be settled. Since x has been settled before u and u is the first node settled such that $\boldsymbol{d}[u] \neq \delta(S, u)$, we have $\boldsymbol{d}[x] = \delta(S, x)$ when x was settled. Since $\delta(S, y) = \boldsymbol{d}[x] + w((x, y)) \leq \boldsymbol{d}[y]$ must have held when x was settled (x and y lie on the shortest path from s to u), after the relaxation we got $\delta(S, y) = \boldsymbol{d}[x] + w((x, y)) = \boldsymbol{d}[y]$.

It is now easy to obtain a contradiction. Since y precedes u on p, the following holds:

$$\boldsymbol{d}[y] = \delta(S, y) \leq \delta(S, u) \leq \boldsymbol{d}[u] \tag{3.2}$$

We also have $\boldsymbol{d}[u] \leq \boldsymbol{d}[y]$ for neither u nor y have been yet settled and u will be settled next. Together we get $\boldsymbol{d}[u] = \boldsymbol{d}[y]$ and from (3.2) we get $\delta(S, u) = \boldsymbol{d}[u]$, which contradicts our assumption. This concludes the proof of (3.1) and consequently the proof of the theorem.

3.1.3 Bidirectional Dijkstra

A tempting modification of the Dijkstra's Algorithm used for the single-pair shortest path problem is to search from the source to the target and from the target to the source in the reversed graph *simultaneously*, thus possibly reducing the space searched. More precisely, at each iteration of the algorithm we check which of the two searches has a smaller key at the front of the priority and advance in that direction. We will refer to the two searches as the forward and the backward search.

In the case of bidirectional search, we can terminate the search as soon as the search scopes have met, that is, as soon as we settle a node in one direction that has already been settled in the other direction.

This common modification is called the bidirectional Dijkstra's Algorithm. The intuitive justification for this modification is illustrated in Fig. 3.1.5.

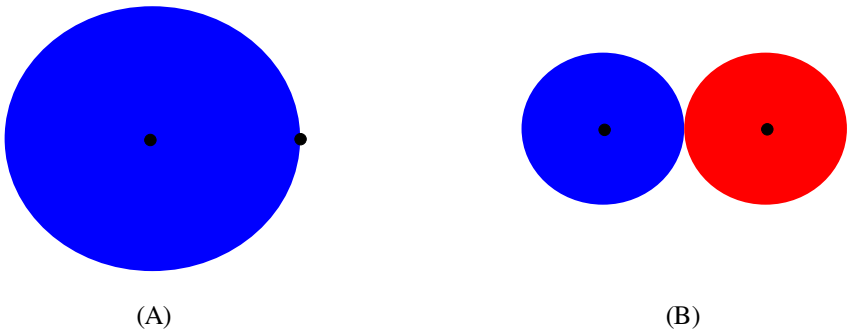


Fig. 3.1.5. Comparison of searches of the unidirectional (A) and the bidirectional Dijkstra's Algorithm (B). The area searched in (A) is $O(n^2)$, whereas the area searched in (B) is $O(2.(n/2)^2) = O(n^2/2)$.

The termination condition of the bidirectional algorithm has to be handled carefully. It is guaranteed that as soon as the search scopes meet at a node v a shortest path has been found. Nevertheless, there is no guarantee that node v lies on a shortest path. See Fig. 3.1.6 for an illustration.

To obtain the shortest path, we can maintain a list of meeting points ordered by the sum of distances from s and t. By a meeting point, we mean a node that has been reached in both, the forward and the backward search. When the search scopes meet, we terminate the search and extract the node from the head of the list of meeting points (the one with the shortest distance).

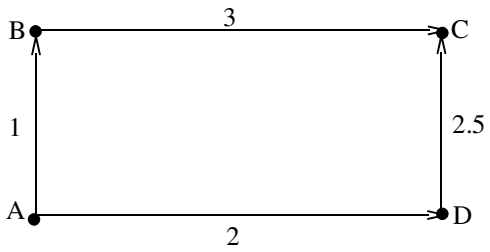


Fig. 3.1.6.: When searching from A (forward) and C (backward), D will be the first node settled in both, the forward and the backward search, even though it does not lie on the only shortest path, which is A-B-C.

3.1.4 Combining the modifications

For the transportation network application, we will need to combine the previous two modifications of the Dijkstra's Algorithm, yielding the multi-source, multi-target bidirectional Dijkstra's Algorithm. The modification is simple: just run the multi-source Dijkstra's algorithm from $s \in S$ and from $t \in T$ in the reverse graph simultaneously. Terminate, when the search scopes meet, just as in the case of the original bidirectional version.

There is an important performance issue and we will have to face it later on, when talking about the application to transportation networks. First, there is no guarantee that the bidirectional version of Dijkstra's algorithm outperforms the original version. In some cases the original version of the algorithm will find a shortest path faster, than the bidirectional one. This becomes even more likely in the case of the multi-source, multi-target bidirectional version. For an illustration, let us consider the case that $|S| = 1$ and

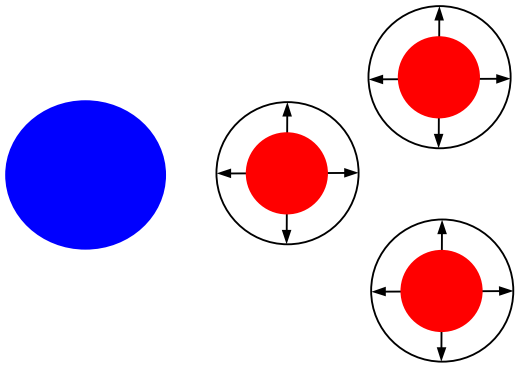


Fig. 3.1.7. If the unidirectional version of the Dijkstra algorithm were used, the search would finish easily by meeting t_1, but in the bidirectional search the nodes t_2 and t_3 yield a lot of unnecessary settled nodes.

$|T|=n>1$. Then whenever the forward search advances, the backward search has to advance n times to be aligned with the forward search (see Fig. 3.1.7). This makes the bidirectional terribly slow and usually the unidirectional multi-source algorithm performs better.

3.2 A* (A-Star) Algorithm

In this section, we introduce an algorithm that is a heuristic search algorithm. It originated in the field of Artificial Intelligence and is often stated in its more general version as a state space search. As we are only interested in the graph version of the algorithm, we will state it here. For the more general version, refer to [4]. Recall the pseudo code for the Dijkstra's Algorithm. At line 6, we used the operation EXTRACT-MIN on the priority queue to obtain the node closest to s. What if for each node u, we know an estimate of its distance to a target node? Wouldn't it be better to extract first the node that *we think* is closest to a target node? This is what A* does and its only difference from the Dijkstra's Algorithm.

When deciding which node should be settled next, the algorithm uses a function $f(u)$ that gives an estimate of the cost of a path from s to a target node through the node u. The function is defined as follows:

$f(u) = g(u) + h(u)$

where $g(u)$ is the cost of path from s to u (i.e., $g(u) = w(<s, ..., u>)$) and $h(u)$ is a *heuristic function*, which gives an estimate of the distance from u to a target node. $h(u)$ can be any totally computable function $h:\mathbb{R}\rightarrow\mathbb{R}$ with the only requirement that $h(t) = 0$ for any target node t. For an efficient implementation, it is clearly desirable that there be an *efficient* algorithm that computes h.

The important question is, whether A* always finds an optimal solution. The answer in general is no. It does though if we impose an additional requirement on the heuristic function.

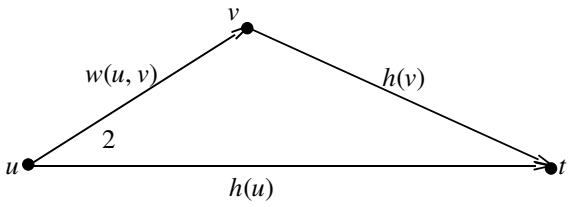
A heuristic function $h(u)$ is *consistent* if for every edge $(u, v)\in E$ the following property holds

$h(u)\leq w(u, v)+h(v)$

(recall triangle inequality) It says that an estimate of getting from u to a target node must not be greater than the actual cost of getting from u to v plus the estimate of getting from v to a target node. See Fig. 3.2.1.

Theorem *When a consistent heuristic function is used, the A* algorithm finds a shortest path.*

Proof It is sufficient to show that when a consistent heuristic function is used then the values of f along any path are non-decreasing. Then clearly (as the algorithm first extracts the node with the smallest $f(u)$) when a target node t is settled, a shortest path must have been found, because all subsequent nodes extracted from the queue will have a value equal or greater then $f(t)$.



$$h(u) \leq w(u, v) + h(v)$$

Fig. 3.2.1.: Consistent heuristic function

Let $p = \langle s, ..., u, v, ..., w \rangle$ be a path. Then $g(v) = g(u) + w(u, v)$ and we have

$f(v) = g(v) + h(v) = g(u) + w(u, v) + h(v) \geq g(u) + h(u) = f(u)$

Therefore the values of f along a path are non-decreasing. This concludes the proof.

A* can reduce the search space significantly in many applications. An example is finding shortest paths in a train network (which is of course the main interest of this work). In this instance, the Euclidean distances (i.e., straight line distances) can be used as basis for a consistent heuristic function.

Nevertheless, there is no guarantee that A* will perform better in any application and even in any instance of a given application.

The *bidirectional search* is also applicable to the A* algorithm. See [5] for details on the bidirectional version of A*.

3.3 Highway hierarchies

Finding shortest paths in a road network is an essential task for a lot of real word logistic and navigation systems. It is common that these systems exploit the property of a road network that there are different classes of roads. So if one wants to find a path between two distant cities, it would not be wise to consider every local road along the path, but we could just consider major highways and reduce the search time significantly. The disadvantage of this approach is though that leaving out some edges may actually lead to non-optimal solutions. Since we are only interested in shortest paths in this work, we will use Highway hierarchies, which does both, reduces search time considerably by

leveraging the mentioned property of road networks, and finds shortest paths. Apart from that, the awesome property of Highway hierarchies is that it finds the road types *implicitly*, that is, without outside guidance.

When talking about Highway hierarchies, we are actually talking about two algorithms. First, there is an algorithm used for the construction of a Highway hierarchy, based on an input road network (represented as a graph). Second, there is a search algorithm, which is, again, a modification of the Dijkstra's Algorithm applied on a Highway hierarchy. We will describe the two algorithms in turn.

3.3.1 Definition of Highway Hierarchy

Definition A *highway hierarchy* of graph $G = (V, E)$ consists of a sequence of level graphs $G_0 = (V_0, E_0)$, G_1, ..., $G_L = (V_L, E_L)$, where

1) The *level graph* G_0 for level 0 is defined as $G_0 = G$.
2) The *core* G'_0 of the level graph G_0 is defined as $G'_0 = G$.
3) The *level graph* G_l for level l, for each $0 < l \leq L$, is a sub-graph of the core G'_{l-1} of level $l - 1$. See below for details.
4) The *core* G'_l of the level graph G_l, for each $0 < l \leq L$, is obtained from the level graph G_l of level l. See below for details.

Before we proceed to the detailed definitions of a core of a level and a level graph, we need to define some additional notions used in the definitions.

Definition For a node each node $u \in V$ and a each level $0 \leq l \leq L$, we choose some non-negative *neighbourhood radius* of a node u in the level graph for level l, denoted by $r_l^{\rightarrow}(u)$.

Similarly, we choose a non-negative *neighbourhood radius* of a node u in the reverse graph of the level graph for level l, denoted by $r_l^{\leftarrow}(u)$.

Definition For each node $u \in V$ and each $0 \leq l \leq L$, we define the *neighbourhood* of a node u in the level graph for level l, as the set of all nodes with the distance from u smaller or equal to the neighbourhood radius $r_l^{\rightarrow}(u)$, and we denote the neighbourhood of node u by $N_l^{\rightarrow}(u)$. In other words, we define the neighbourhood of a node u as follows: $N_l^{\rightarrow}(u) = \{v \in V_l' | \delta(u, v) \leq r_l^{\rightarrow}(u)\}$.

Similarly, we define the neighbourhood of a node u in the reverse graph of the level

graph for level l: $N_l^{\leftarrow}(u)=\{v\in V_l' \mid \delta(u,v)\le r_l^{\leftarrow}(u)\}$.

Definition (Canonical shortest path) In order to preserve the correctness of the search algorithm, the construction algorithm cannot use just any shortest path when choosing edges to be added to the next level. The problem is illustrated in figure 3.3.1.

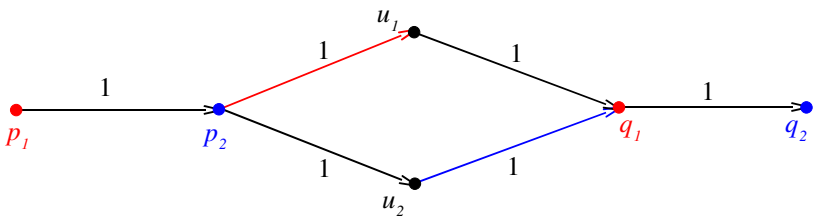


Fig. 3.3.1.: Importance of canonical shortest paths. This figure illustrates what might happen if just any path were used during the construction of level graphs. The neighbourhood radii of all nodes is chosen to be 1, so $N_l^{\rightarrow}(p_1)=\{p_2\}$, $N_l^{\rightarrow}(p_2)=\{p_1,u_1,u_2\}$, $N_l^{\leftarrow}(q_2)=\{q_1\}$, and $N_l^{\leftarrow}(q_1)=\{q_2,u_1,u_2\}$. Let us assume that the preferred shortest path between p_1 and q_1 is P_1 = <p_1, p_2, u_1, q_1> and between p_2 and q_2 it is P_2 = <p_2, u_2, q_1, q_2>. Therefore for the path P_1 the red edge (p_2, u_1) is added to G_l ,whereas for the path P_2 the blue edge (u_2, q_1) is added to G_l, thus breaking the level graph G_l, as there will not be the necessary path between p_2 and q_1.

Let SP_G be a set of shortest paths in graph $G = (V, E)$. Then we say that SP_G is a set of canonical shortest paths for G if

i. for each $u,v\in V$ such that there exists a path from u to v in G, we have $!\exists P=\langle u,\dots,v\rangle\in SP_G$ (i.e., there exists exactly one path from u to v in SP_G).

ii. for each path $P=\langle u,\dots,v\rangle\in SP_G$ and each sub-path $\bar{P}=P|_{\bar{u}\to\bar{v}}$ of P we have $\bar{P}\in SP_G$.

iii. nothing else is a member of SP_G

Let be a path P in G. Then we say that P is a *canonical shortest path*, iif $P\in SP_G$.

Definition of G_l

For each $0<l\le L$ we give a definition of $G_l=(V_l,E_l)$:

$G_l = (V_l, E_l)$ is obtained from $G'_{l-1}=(V'_{l-1},E'_{l-1})$ in the following way: (1) E_l is

obtained by taking just such edges (u, v) from E'_{l-1} that there exist $p, q \in V'_{l-1}$ and a *canonical* shortest path $<p, ..., u, v, ..., q>$ in G'_{l-1}, and the following properties are satisfied: $u \notin N_l^{\leftarrow}(q)$ and $v \notin N_l^{\rightarrow}(p)$. Notice that this condition tries to exploit the mentioned property of road networks: for nodes p and q that are distant enough, we add the highway edge (u, v), because there is a shortest path containing the edge (u, v) between p and q. It is likely that this same edge will lie on the shortest paths between the nodes in the proximity of p and those in the proximity of q. (2) V_l contains exactly the nodes u and v for each edge (u, v) added to E_l: $u \in V_l \Rightarrow \exists v \in V_{l-1}' : (u, v) \in E_l$.

Definition of G'_l

For each $0 < l \leq L$, we give a definition of $G'_l = (V'_l, E'_l)$:

Let B_l be a set of nodes. We call the nodes from B_l the *bypassable nodes*. We define the set of edges $S_l \subseteq V_l \times V_l$ (called the *shortcut edges*) in the following way: Let $u, v \in V_l / B_l$, $b_1, ..., b_n \in B_l$ and $P = <u, b_1, ..., b_n, v>$ be a path in G_l. Then $(u, v) \in S_l$ and we define the weight of the edge (u, v) as the weight of the path P: $w(u, v) = w(P)$.

We are now poised to define the core of level l: $G_l' = (V_l', E_l')$, where $V_l' = V_l / B_l$ and $E_l' = (V_l' \times V_l' \cap E_l) \cup S_l$.

An additional requirement on the neighbourhood radii will be needed: for each $0 \leq l < L$ and each $u \in B_l$ we require that $r_l^{\rightarrow}(u) = r_l^{\leftarrow}(u) = \infty$, and for each $u \in V_L$ we require that $r_L^{\rightarrow}(u) = r_L^{\leftarrow}(u) = \infty$.

3.3.2 Construction

The construction algorithm for highway hierarchy proposed by D. Schultes in [6] and adjusted for directed graphs by P. Sanders and D. Schultes in [7] will also be briefly described here. More details can be found in [6] and [7].

In this section, $\delta_l(u, v)$ will denote the distance from u to v in G_l'.

Neighbourhood radii

For each $0 \leq l < L$ and each $u \in V_l'$ and a parameter n_l, we determine the node radius for u in the following way: let $u(n_l) \in V'_l$ be a node such that it has the n_l-th smallest distance from u. Then we set $r_l^{\rightarrow}(u) = r_l^{\leftarrow}(u) = \delta(u, u(n_l))$. This can be achieved by

running the Dijkstra's algorithm from every node $u \in V_l'$ and counting the number of nodes visited. After a node v is settled and the counter exceeds n_l, we terminate the Dijkstra search and set $r_l^{\rightarrow}(u) = r_l^{\leftarrow}(u) = d[v]$.

Furthermore we set $r_l^{\rightarrow}(w) = r_l^{\leftarrow}(w) = \infty$, for each node $w \in V / V'_l$ and we set $r_L^{\rightarrow}(v) = r_L^{\leftarrow}(v) = \infty$ for each node $v \in V$.

Fast construction of a level graph

We assume that G_l' has already been created and will describe how G_{l+1} is constructed. For each node $u \in V_l'$ two phases are performed. First, a Dijkstra search is executed in G_l' from node u and a shortest path tree is built (a modification of Dijkstra's algorithm that always finds canonical shortest paths has to be used; see [6] for such a modification). Second, paths in the shortest path tree are traversed backwards and for each edge on each path it is considered whether it should be added to V_{l+1}. So let us assume that $u \in V_l'$. A more detailed description on how both these phases are performed on u follows.

Construction of the shortest path tree D. Schultes proposed in [6] an important optimization for the construction of the shortest path tree. Instead of constructing the complete shortest path tree by letting the Dijkstra's Algorithm to compute all shortest paths from every node u to any other node, a *partial* shortest path tree is constructed in such a way that no important edge is left out when adding edges to G_{l+1} in the second phase. The optimization lies in a specialized abort criterion, which limits the search to some close neighbourhood of u, thus speeding up the construction considerably.

The abort criterion is the following: during the Dijkstra search, a state of either active or passive is maintained for each node. Initially, u's state is set to active. When a node is about to be reached, it adopts the state of its predecessor. The search is aborted, as soon as there are no unsettled active nodes.

It remains to specify when a state of a node can be set to passive: let P be a tentative shortest path from u to z found by the Dijkstra's Algorithm for node z, where $P = \langle u, v, ..., w, x, y, ..., z \rangle$, and $\delta_l(v, x) \leq r_l^{\rightarrow}(v) < \delta_l(v, y)$. When a node z is settled then z's state can be set to passive if $y \neq z$, and $\delta_l(w, z) > r_l^{\leftarrow}(z)$. See Fig. 3.3.2 for an illustration.

During the second phase, each path on the partial shortest path tree is traversed backwards from a leaf z to u, and an edge (x, y) lying on the shortest path is added to

G_{l+1}, whenever $x \notin N_l^{\leftarrow}(z)$ and $y \notin N_l^{\rightarrow}(u)$.

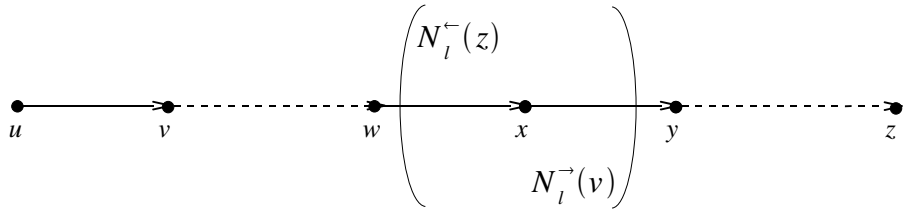


Fig. 3.3.2.: Abort criterion: *z*'s state can be set to passive.

Contraction of a level

To obtain the core G_l' of a graph G_l, we have to determine the set of bypassable nodes B_l and add shortcut edges corresponding to every bypassed node $u \in B_l$. This process is called the contraction of a level graph.

The contraction brings several benefits. First, the search gets faster as only nodes that are not bypassed are considered and the construction of subsequent levels gets faster for the same reason. Besides, smaller neighbourhood radii can be used without compromising the shrinking of the level graphs. Nevertheless, the selection of bypassable nodes should be done carefully as the addition of shortcut edges may increase the average degree of the graph considerably.

The iterative algorithm proposed in [7] is the following: a queue of prospect bypassable nodes is maintained, initially containing all nodes from V_l. At each iteration, a node *u* is removed from the queue and if $(\text{number of shortcuts}) \leq degree(u)$ is satisfied then *u* is actually bypassed, that is, it is added to B_l and the appropriate shortcuts are created for *u*.[3] After the node is bypassed, all adjacent nodes that have not been bypassed yet and are bypassable now have to be added into the queue, as the creation of shortcuts may change the degree of the adjacent nodes.

3.3.3 Search

It's time to introduce the search algorithm for highway hierarchies. We are going to present the directed version, as stated in [7], but generalized for our version of the single-pair shortest path problem. As mentioned earlier, it is again a modification of the

3 *degree*(*u*) is the sum of the input and output degree of *u*.

Dijkstra's algorithm. It introduces a local search where only a neighbourhood of a node is searched. When this neighbourhood is about to be left, the search is not continued in the current level and is switched into a higher level.

Apart from the distance, we maintain the search level and a gap for each node in Q. The search level is initialized to 0 for all source and target nodes (i.e., we start the search in level 0) and the gap is initialized to $r_0^{\rightarrow}(u)$ for every source node u and $r_0^{\leftarrow}(v)$ for every target node v. When an edge $e = (u, v)$ is relaxed, the gap of u minus $w(e)$ becomes the gap of v. When e is about to be relaxed and the resulting gap for v is smaller than 0 (i.e., we are about to leave the neighbourhood) then the search level is increased by 1 and v is added to Q with the new search level and a new gap $r_1^D(u) - w(e)$, where $D \in \{\rightarrow, \leftarrow\}$ denotes the current search direction. In that case, we call u the *entrance point* to the new level. But v is actually added to Q only if $v \in V_1$. Otherwise the edge (u, v) is not relaxed. This is the first one of the two important restrictions that account for the speed-up in comparison with the original Dijkstra's Algorithm. We will refer to this restriction as *restriction 1*. The algorithm described for level 0 applies in a similar fashion to the following levels, that is, when the neighbourhood of a node in level 1 is about to be left, search switches to level 2, etc.

The second restriction, *restriction* 2, is that after we settle a node $u \in V_l'$ in the *core* of level l, and are about to relax an edge (u, v), such that $v \in B_l$, we skip relaxing the edge. In other words, after we enter the core of level l, we do not leave it by reaching a bypassed node v.

For the search to be correct, the queues are prioritized by a key $k = (d, level, gap)$, where for two keys $k_1 = (d_1, level_1, gap_1)$, $k_2 = (d_2, \text{level}_2, gap_2)$ we have

$$k_1 < k_2 \Leftrightarrow d_1 < d_2 \vee$$
$$\vee (d_1 = d_2 \wedge level_1 > level_2) \vee$$
$$\vee (d_1 = d_2 \wedge level_1 = level_2 \wedge gap_1 < gap_2)$$

The correctness proof for the search algorithm for highway hierarchies is quite long with a lot of technical details and it will not therefore be stated here. Rather, we refer an interested reader to the appendix of [7], where the detailed proof can be found.

4 Cheapest Connection Search

In this chapter, we revisit the algorithms introduced in the previous chapter and describe some modifications of these algorithms for the use in our context of transportation networks.

There are four types of modifications dealt with in this chapter, logically separated into the four sections of this chapter.

First, we go through some performance improvements that can be applied to the algorithms.

We also show how simple modifications can improve the connections found. More precisely, when there are more than one cheapest connection, we want our algorithm to prefer one over another given some additional criteria such as the number of changes or the time spent on the trip (i.e., the time of the trip *excluding* the waiting time at the origin station).

Afterwards, we introduce two modifications that allow to constrain the connections found by limiting the maximum number of changes and by specifying some criteria on the trains used.

At the end of this chapter, we revisit the issue stated at the end of the chapter 2, that is, the problem that the definition of a transportation network does not handle restricted timetables. In this chapter, we see how restricted timetables can be handled by a modification to the search algorithm.

For this chapter, we consider a transportation network $N=(St, L, c)$, a corresponding transportation graph $G_N=(V_N, E_N)$, a cheapest connection query for a set of (origin) stations $R \subseteq St$, a set of (destination) stations $D \subseteq St$, a relative time $\tilde{T}$, and a shortest path query corresponding to the cheapest connection query, where *S* will denote the set of source nodes and *T* will denote the set of target nodes.

Before we proceed to the first of the four anticipated sub-chapters, we state a useful definition that will be referred to throughout this chapter.

Definition We consider a run of a search algorithm. Let *time*(*P*) denote the time-weight of a path *P*. We define the time portion of a distance of a node *u*, denoted by *TPD*(*u*), as

$$TPD(u) = \begin{cases} time(P) + T_r - \tilde{T}, \text{ where } P = \langle d[r, T_r] = u_1, ..., u_n = u \rangle \text{ is the tentative} \\ \text{shortest path found by the search algorithm (i.e., } r \in R \text{), or} \\ \infty \text{ if no path to } u \text{ has been found so far.} \end{cases}$$

In other words, the time portion of a distance of a node is the time distance of *u* from the time $\tilde{T}$ on the tentative shortest path found by the search algorithm.

4.1 Performance improvements

4.1.1 Earliest arrival optimization

In this section, we present a performance optimization applicable to the unidirectional version of the Dijkstra's and the A* algorithms. It also applies to the bidirectional versions of the two algorithms but only makes sense for the forward search.

Let us consider the following situation during the Dijkstra's Algorithm: we are about to reach an *s*-arrival node *v* (for some $s \in St$), and we know that some other *s*-departure node *u* had already been reached with a smaller or equal time-distance. Then any node reachable from *v* can also be reached from *u* with a smaller or equal time-distance. See Fig. 4.1.1 on page 35 for an example of this situation.

Therefore, we can consider skipping the relaxation of any edge that leads to a node *v* as depicted in Fig. 4.1.1, possibly leading to a speed-up of the search since the queue used during the search would shrink. Nevertheless, we cannot skip such an edge unless we are sure that any node reachable from *v* can also be reached from a node *u* with a smaller distance. This leads us to the following general rule:

Let T_s denote the time portion of the distance with which a node $a[s, T_2, x]$ is about to be reached. Then we can skip reaching the node $a[s, T_2, x]$ if there exists a node $d[s, T_1]$ such that the following two properties hold:

1) $TPD(d[s, T_1]) \leq T_s$

2) $\boldsymbol{d}[a[s, T_2, x]] \geq \boldsymbol{d}[d[s, T_1]] + w_N(v(\bar{T}))$, where
$\bar{T} = TPD(a[s, T_2, x]) - TPD(d[s, T_1])$.

Note that the fulfilment of the first property is crucial. If that did not hold, an important connection might be missed leading to a suboptimal path. This situation is depicted in Fig. 4.1.2 on page 36.

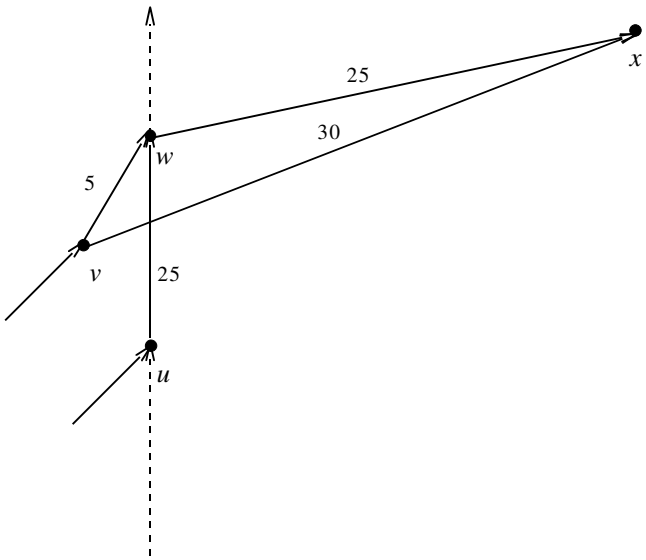


Fig. 4.1.1.: If node *u* was reached with time-distance 10, and node *v* is about to be reached with time-distance 40, then the nodes directly reachable from *v* (*w* and *x*) can both be reached via *u* with a smaller distance then if they were reached via *v*. Namely, *w* can be reached with the time-distance of 35, and *x* can be reached with the time-distance of 60, compared to 45 and 70 if both nodes were reached via *v*. Consequently, any node reachable from *v* can be reached from *u* with a smaller distance.

The implementation of this optimization can be done like this: in addition to the prospect distance ***d*** from *S* for each node, in the same way we maintain and update the time portion of the distance for each node, each initialized by an infinity-replacement value. Moreover, we have to maintain the smallest time portion of the distance of an already reached *s*-departure node for each station $s \in St$, also each initialized by an infinity-replacement value. Then we can check the above criterion and will not relax any edge that satisfies this criterion.

4.1.2 Lazy backward search for highway hierarchies

Here we revisit the performance issue of the bidirectional Dijkstra search. The problem we are facing is that when the bidirectional version of the Dijkstra's Algorithm is used, and $|S| \ll |T|$ then – as the fringe of the backward search is huge – it is very costly for the backward search to keep aligned with the forward search. See section 3.1.4 and Fig. 3.1.7 for details.

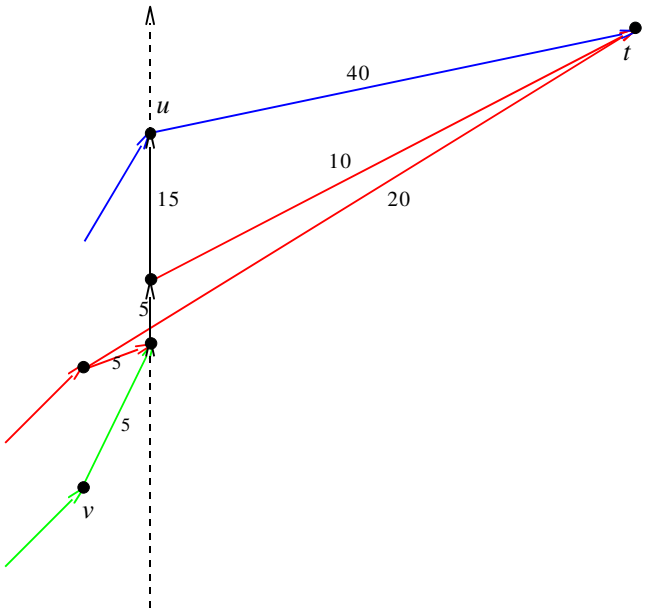


Fig. 4.1.2.: Let us assume that u is reached with distance 30 and v is reached with distance 40. The time portions of their distances are 30 and 5, respectively. The weights of the edges are displayed next to the corresponding edges. Then $\boldsymbol{d}[v]=30\geq 40+(5-30)=\boldsymbol{d}[u]+w_N(\vec{v}^T)$, therefore v is skipped and a suboptimal path through u with a weight of 70 is found, though the optimal one is the one going through v with a weight of 60.

In the case $|S|\ll|T|$ (which is usually the case of the search in a transportation graph, as T contains *all* d-arrival nodes for all $d\in D$) it turns out (and is backed up by the results in the next chapter) to be better not to use the bidirectional search whatsoever and use the unidirectional search instead. Nevertheless, we have to deal with this issue for the sake of the search in highway hierarchies, where a bidirectional search is indispensable.

The improvement of the bidirectional search in a highway hierarchy for the case $|S|\ll|T|$ will be similar to the one mentioned by S. Knopp et al. in [8], where it was applied to a version of the all-pairs shortest path problem.

The idea is not to search further than necessary in the backward direction. We know that we do not need to advance in the backward direction, whenever a node is about to be reached in the backward search with a level that the forward search has not yet get at. This is justified by the fact that in that case the search scopes cannot meet in that level. So we only have to search in the backward direction till we reach the core of the level reached by the forward search.

For the purposes of this section, let l_u^F denote the search level of node u in the forward

search and let l_u^B denote the search level of node u in the backward search. The optimization is the following: do not continue the backward search from a node v that is not bypassed in the level l_v^B if

$$l_v^B \geq max\{ l_u^F | u \text{ is reached but not settled in the forward search} \}$$

We denote the right part of the above inequality by L. To implement the described functionality, we maintain two priority queues for the backward search, Q_1 and Q_2. Q_1 is initialized as usual and Q_2 is set empty. During the search if the above criterion holds for a node v when it is extracted from Q_1, we insert v into Q_2 and do not continue backward search from v. Whenever L increases, we switch from Q_1 to Q_2 as long as there are any nodes in Q_2 and settle the nodes we had left out.

This modification does not impair the correctness of the search algorithm, since the termination condition used in the Dijkstra search for highway hierarchies guarantees that the forward search has enough time to settle the nodes that were not settled appropriately in the backward search.

Note that it is crucial that any node v that is skipped according to the presented criterion must not be bypassed. This is due to the restriction 2 of the search algorithm (see section 3.3.3) which could restrict the forward search from reaching the bypassed node v leading possibly to a suboptimal path.

4.2 Search results improvements

4.2.1 Stay at source station as long as possible

So far, we have seen modifications that dealt with the performance of the search algorithms. Now we turn to modifications that have to do with the quality of the corresponding connections, rather than with the speed of the algorithms.

What often happens in a transportation network is that for a given query, there are several connections with the same cost, and still it may be reasonable to prefer one over another. For an example see, Fig. 4.2.1.

When there are several connections with the same cost, it is reasonable to prefer the one with the shortest total travel time. Two adjustments to the Dijkstra's Algorithm have to be done in order to achieve the desired behaviour.

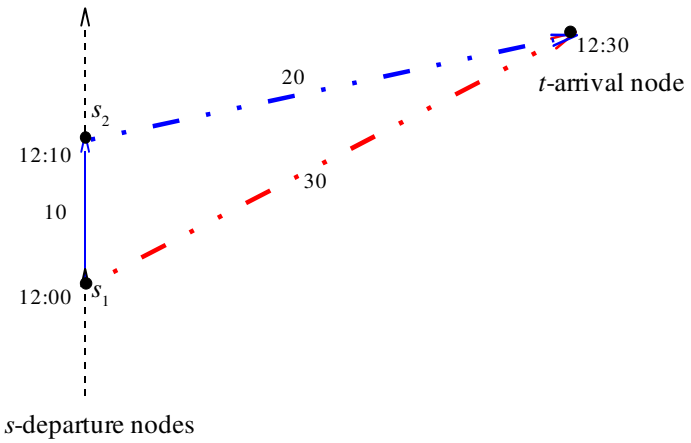


Fig. 4.2.1.: In this example, we consider a transportation network with time costs and a query for stations s and t and time T_s = 12:00, i.e $S=\{s_1\}$, and $t\in T$. We have two possible shortest paths: first one avoiding the node s_2 (depicted in red) and a second one, going through s_2 (depicted in blue). Though the search algorithm would prefer any one of the two paths (since both are shortest path) we should choose the second (blue) option, as the total travel time is 10 minutes shorter.

Definition Let $u\in V_N$ be a node such that a tentative shortest path $P=\langle d[r,T_r]=u_1,\ldots,u_n\rangle$ has been already found by the Dijkstra's Algorithm. The *time gap at origin for a node u* is defined as $TPD(u_k)$, where either $k=n$, or $1\le k<n$, $u_k=d[r,\bar{T}_r]$ for a relative time $\bar{T}_r$, and $u_{k+1}=a[s,T_s,l]$, for some $s\in St$.

Now we describe the modifications needed to implement the functionality described earlier in this section. The first modification is to maintain a time gap at origin $og[u]$ for each reached node u. Then, we modify the RELAX function so that it performs the update whenever the new distance is smaller then the old one, or whenever the distances are equal *and* the new (tentative) time gap at origin is *greater* than the old one. See Fig. 4.2.2 for the modified version of the RELAX function.

This will guarantee that whenever there are several shortest paths to a node v, the one yielding the greatest time gap at origin will be the one that will be continued from v.

There are two subtleties that will be addressed by the second adjustment. First, when there are several target nodes with the same distance from the source set then one with a smaller time gap at origin may be extracted first, thus terminating the search before the other nodes are considered. The second subtlety is the case of zero-weight edges when there are two zero-weight edges (u, t) and (v, t) for $t\in T$. Then, (after v is settled) u and t become members of the priority queue with the same key, which may again lead to a premature termination of the search, before u is considered.

```
RELAX(u, v, w, Q , d, p, og)
1   if d[u] + w(u, v) < d[v] ||
2   (d[u] + w(u, v) = d[v] && og[u] > og[v]) then
3           d[v] ← d[u] + w(u, v)
4           p[v] ← u
5           if u∈S then
6                   og[u] ← og[u] + (v.time ⊖ u.time)
7           else
8                   og[u] ← og[u]
9           if v∈Q then
10                  Q ← DECREASE-KEY(Q, v, d[v])
11          else
12                  Q ← INSERT(Q, v, d[v])
13  return Q
```

Fig. 4.2.2.: The RELAX function modified so that nodes with a greater time gap at origin are preferred. *u.time* denotes the time corresponding to this node.

These two issues can be fixed by adjusting the order in which the priority queue used in the Dijkstra's Algorithm extracts nodes. We extend the key used by the queue to be an ordered pair $k = (d, T)$, where d is the original key (i.e., the tentative distance of the node), and T is the time gap at origin for the node. For keys $k_1 = (d_1, T_1)$ and $k_2 = (d_2, T_2)$, we define the operator $k_1 < k_2$ used by the queue as follows:

$$k_1 < k_2 \Leftrightarrow d_1 < d_2 \vee (d_1 = d_2 \wedge T_1 > T_2)$$

The described modifications can be applied to the unidirectional versions of the Dijkstra and the A* algorithms. It can also be applied to the forward search of the bidirectional versions of the two algorithms. Nevertheless, the modification will not work for the highway hierarchy search where the departure edge from a node with greatest time gap at origin may be missing in higher levels.

4.2.2 Prefer fewer trains

Similarly to the previous section, also in this section we want to improve the quality of the connections by choosing a better from among all the cheapest connections. In this section, we will be concerned with the number of changes. By a change we mean getting off a train at a station and getting on another train on the same station. Usually, we want as few changes as possible in our trip, so whenever there are two connections with an equal distance, we prefer the one with fewer changes.

The modifications to the Dijkstra's algorithm we described in the previous section apply

in this scenario as well, as the assignment is just the same, only the criteria changed. Therefore we will not go into detail and just briefly describe the differences from the adjustments we introduced in the previous section.

In this case, we maintain a set of lines $LS[u]$ for each reached node *u*. The RELAX function performs an update whenever the new distance is smaller then the old one, or whenever the distances are equal *and* the cardinality of the new set of lines is *greater* than the old one. In addition, whenever we perform an update of an arrival node $a[s,T,k]$ reached from a node *q*, we also update *LS*: $LS[a[s,T,k]] \leftarrow LS[q] \cup \{k\}$ and whenever we perform an update of a departure node $d[s,T]$ reached from a node *q*, we set $LS[d[s,T]] \leftarrow LS[q]$.

The extraction order of the priority queue is modified as follows: we extend the key used by the queue to be an ordered pair $k = (d, LS)$, where *d* is the original key (i.e., the tentative distance of the node), and *LS* is the line set for the given node. For keys $k_1 = (d_1, LS_1)$ and $k_2 = (d_2, LS_2)$, we define the operator $k_1 < k_2$ as follows:

$$k_1 < k_2 \Leftrightarrow d_1 < d_2 \vee (d_1 = d_2 \wedge |LS_1| < |LS_2|)$$

The described modifications can be applied to the unidirectional versions of the Dijkstra and the A* algorithms. It can also be applied to the forward search of the bidirectional versions of the two algorithms. Nevertheless, the modification will not work for the highway hierarchy search as some edges from the path with fewer trains may be missing in a higher level.

There may be actually several cheapest connections with the fewest number of trains, in which case we likely want to choose one with the greatest time gap at origin. We can combine the modification from this section with the one from the previous section to achieve the described property of the algorithm. In order to do that, the following two modifications have to be done:

1) A key of the priority queue will be a 3-tuple (*d*, *LS*, *TGO*), where *d* is the tentative distance, *LS* is the set of trains and *TGO* is the time gap at origin. For keys $k_1 = (d_1, LS_1, \mathrm{TGO}_1)$ and $k_2 = (d_2, LS_2, TGO_2)$, we define the operator $k_1 < k_2$ as follows

   $$k_1 < k_2 \Leftrightarrow d_1 < d_2 \vee ((d_1 = d_2 \wedge |LS_1| < |LS_2|) \vee (|LS_1| = |LS_2| \wedge |TGO_1| > |TGO_2|))$$

2) Adjust the condition at lines 1-2 of the RELAX function (see Fig. 4.2.2) in a similar way to 1).

4.3 Search constraints

In this section, we introduce two modifications that substantially differ from the other modifications introduced so far. The major difference is in that these modifications constrain the search results in some way and therefore in general yield suboptimal connections because all the cheapest connections may be ruled out by the constraints.

4.3.1 Limit number of changes

It is often desirable to filter out connections with too many changes.

In this section, we introduce another modification of the search algorithm, which permits the Dijkstra's Algorithm find a shortest path with at most *M* changes.

The basic idea is to maintain for each node a list of prospect shortest paths with exactly 0, 1, 2, ... etc. changes. Then, whenever a node lying on a path with a number of changes exceeding *M* is about to be reached, we discard that path.

For the purposes of this section, *changes*(*P*), where *P* is a path, denotes the number of changes on a connection corresponding to *P*.

For each node *u* we maintain the weight $\boldsymbol{d}_i[u]$ of the prospect shortest path with exactly *i* changes. When we are about to relax an edge *e* = (*u*, *v*), lying on a path $P=\langle u_1, u_2, \ldots, u_n, u, v\rangle$, we actually relax the edge *e* if the following criterion **(1)** holds:

1) $changes(P) \le M$
2) for every *k* such that $0 \le k \le changes(P)$ we have that $\boldsymbol{d}_j[u]+w(u,v)<\boldsymbol{d}_k[v]$, where $j=changes(\langle u_1, u_2, \ldots, u_n, u\rangle)$.

In other words, we relax *e* if we do not exceed the maximum number of changes (1) and all the paths to *v* found so far with the number of changes smaller or equal to the number of changes to the new path are more expensive (2).

If the above property does NOT hold, we discard and do not relax the edge *e*.

To implement this modification, we have to make the following changes to the Dijkstra's Algorithm:

- Initialization: for all $u \in S$ and for all *i* > 0, set $\boldsymbol{d}_0[u] \leftarrow 0$, $\boldsymbol{d}_i[u] \leftarrow \infty$. Moreover, for all $v \in V_N \setminus S$ and for all $i \ge 0$, set $\boldsymbol{d}_i[v] \leftarrow \infty$. Similarly, for all $u \in S$, insert $(\boldsymbol{d}_0[u], u, 0)$ into *Q* (see below for details on *Q*).
- Queue *Q* contains additional information about each node: the number of changes a

node was reached with when it was added to Q, i.e., Q contains tuples (*key*, u, i), where i is the number of changes. Q remains prioritized by *key*, i.e., the distance from S.

- We maintain a set of lines $LS_i[u]$, which contains all the lines on the tentative shortest path corresponding to $\boldsymbol{d}_i[u]$. For all $u \in V_N$, and for all $i \geq 0$, $LS_i[u]$ is initially set to $\varnothing$. See below for details on how $LS_i[u]$ is updated.
- Let (*key*, u, j) be a tuple extracted from Q by the EXTRACT-MIN operation, $e=(u,v) \in E_N$ and

$$i = \begin{cases} j, & \text{if } v \text{ is a departure node, or } v = a[s,T,k] \text{ is an arrival node such that } k \in LS_j[u] \\ j+1, & \text{otherwise} \end{cases}$$

  If the above criterion **(1)** holds for e then we set $\boldsymbol{d}_i[v] \leftarrow \boldsymbol{d}_j[u] + w(u,v)$ and insert $(\boldsymbol{d}_i[v], v, i)$ into Q (or decrease key if a tuple (d, v, i) already exists in Q for some $\boldsymbol{d}_i[v] < d$).
- If the criterion **(1)** holds for $e=(u,v) \in E_N$, where $v = a[s,T,k]$, we perform the following update of $LS_i[v]$: $LS_i[v] \leftarrow LS_i[u] \cup \{k\}$.

In addition, we have to maintain predecessors for each prospect shortest path going through each node so that we can reconstruct the path found. Therefore for a node u and some i, we store the predecessor $p_i[u]$ of u on a path with i changes.

- $p_i[v] \leftarrow (u, j)$

The described modification applies to the unidirectional versions of the Dijkstra's and the A* algorithms.

4.3.2 Train criteria

Let us assume that additional information is associated with each train. For example, there may be an indication whether a train allows the carriage of large equipment and bicycles. And we want the search algorithm to only return connections containing trains meeting given criteria. An easy modification of the search algorithm would be to discard every departure and every continuation edge not meeting the given criteria. This could be achieved by not calling the RELAX function on every such edge (for a definition of departure and continuation edges, please refer to chapter 2).

This modification cannot applied to the search for highway hierarchies, as an edge

missing in a higher level may result in that no connection will be found even though a connection exists in the original graph.

4.4 Restricted timetables

Here we revisit an important property of transportation networks stated at the end of chapter 2, namely that a given train operates only on some days. We left this property out of the definition of a transportation network and decided to treat it in the search algorithm. In this section, we describe the modifications needed for an algorithm to respect the restricted timetables.

We alter the definition of a shortest path query by replacing the time parameter $\tilde{T}$ with a parameter $DT \in \mathbb{N}$ that will represent an *absolute time*. We need this parameter to be counted in minutes (starting at some initial date in the past). This will allow us to calculate the absolute time of a node u on a path as the sum of the departure time and the time portion of the distance of a node u.

In addition, we need a predicate that allows us to find out whether a given train is operating at a given station at a given time. For a line k, a station s and an absolute time DT, we define a predicate $Op(k, s, DT) \in \{false, true\}$ that has the value *true* if and only if the line k departs from station s at the absolute time DT.

Furthermore, we define a function *dep_time*(k, s, DT), as the absolute time of the next departure of a given line at or after a given absolute time:

$$dep\_time(k, s, DT) = min\{DT' \mid DT' \geq DT \wedge Op(k, s, DT') = true\}.$$

Thus armed, the modification to the Dijkstra's Algorithm is quite simple. We just maintain the current absolute time $dt[u]$ for each node u. Then, whenever an edge (u, v) leading to an arrival node is relaxed, *dep_time*(k, s, $dt[u]$) - $dt[u]$ is added to the cost of the edge. See Fig. 4.4.1 for the resulting RELAX function.

```
RELAX(u, v, w_N, w_G, Q , d, p, dt, Op)
1   if x = d && y = (a, k) then
2           T ← dep_time(k, s, dt[u]) - dt[u]
3           W = w_N(v(T)) + w_G(u, v)
4   else
5           W = w_G(u, v)
6   if d[u] + W < d[v] then
7           d[v] ← d[u] + W
8           p[v] ← u
9           dt[v] ← dt[u] + (v.time ⊖ u.time)
10          if v∈Q then
11                  Q ← DECREASE-KEY(Q, v, d[v])
12          else
13                  Q ← INSERT(Q, v, d[v])
14  return Q
```

Fig. 4.4.1.: The RELAX function modification for restricted timetables. w_N represents the weight function for the underlying transportation network *N*. *u.time* denotes the time corresponding to this node, whether it is an arrival or a departure node.

5 Performance Results

In order to evaluate the performance of the different algorithms, a series of tests was performed on a transportation graph representing a real transportation network and on a randomly generated graph. In this chapter we present the results of the tests and their comparison.

First, we describe the contents of a result of a test and the test data that were used. Then we present the results of the tests on each of the algorithms in turn. Finally, we compare the performance of the algorithms at the end of this chapter.

5.1 The tests

All the tests were performed on a laptop computer with an Intel Celeron processor clocked at 1.3 GHz, with 2GB system memory and Red Hat Linux operating system with the release number 2.6.16-1.2115_FC4.

The C++ sources were compiled with the g++ compiler version 4.0.2 with optimization level of 3.

By a *test run*, we mean a sequence of N cheapest connection queries all executed with a given search algorithm. After each query is executed, some information necessary for the evaluation of the algorithm's speed is logged. The information gathered is the following:

- *Search time*: The time spent by the algorithm to perform the search. In milliseconds.
- *Settled nodes*: Number of nodes settled during the query.
- *Reached nodes*: Number of nodes reached during the query.
- *Relaxed edges*: Number of edges relaxed during the query.
- *Connection length*: The number of stations on the cheapest connection found by the algorithm.

Whereas the search time is related to the system the query is run on, the numbers of

settled nodes, reached nodes and relaxed edges depend only on the search algorithm. Naturally, the smaller these numbers, the shorter the search time, but obviously the average unit time cost of each of the three figures will vary from algorithm to algorithm, as their asymptotic complexities vary.

The last value, the connection length (i.e., the length of a connection that was defined in 2.3.1 as the number of stations the connection traverses), is related mainly to the query itself, as the query actually implies a set of connections. Nevertheless, the connection length is also related to the search algorithm, as, for some queries, different algorithms may return different connections.

The connection length is the only value that directly reflects the "difficulty" of the query. The connection length can be therefore used to roughly classify the queries and compare not only the overall performance of the different algorithms but also their performance on queries with a connection length from a given range. This will give us insight into how well an algorithm performs on queries with the difficulty of "low", "medium" or "high" in comparison to the other algorithms.

For each algorithm, two test runs were performed: one on a transportation graph representing a train network of the Czech Republic and one on a randomly generated graph $G=(E,V)$ (see next section for more details).

The sequence of queries for each test run was of length N = 10 001, where each of the N queries was generated randomly. More precisely, each query was concocted by randomly choosing stations $s \neq t$ and time $0 \leq T_s < 1440$ for the case of the transportation network. For the case of the random graph, nodes $V \ni u \neq v \in V$ were chosen randomly for each query.

5.2 Test data

The algorithms were tested on two kinds of graphs: on a transportation graph for the train network of the Czech Republic, and on a random graph.

The train network has 4 138 stations and 11 813 trains. The resulting transportation graph has 241 263 nodes and 489 734 edges which yields an average output degree of slightly over 2.0. The huge number of nodes in comparison with the number of stations is the consequence of a high number of trains in the network.

The algorithms were tested on a directed random graph as well. A random graph G = (V, E) of the size of the transportation graph (i.e., n = 241 263 nodes and m = 489 734 edges) was generated as follows. A set of nodes $V=\{u_1,\dots,u_n\}$ was chosen and the edge set was constructed in such a way that a path exists for every pair of nodes: an edge (u_i, u_{i+1}) for each $1 \leq i < n$ was added and an additional edge (u_n, u_1) was added to G,

thus creating a circle in *G*. Each of the additional *m* – *n* (i.e., 248 471) edges (u_i, u_j) were added by iteratively choosing a random pair of distinct nodes $u_i, u_j \in V$, such that (u_i, u_j) is not yet a member of *E*.

For each node u_i, coordinates (x_i, y_i) were chosen randomly from a range of natural numbers and the weight of an edge $w(u_i, u_j)$ was calculated as the euclidean distance between u_i and u_j.

For both, the selection of graph edges, and the generation of coordinates of the nodes, a pseudo-random generator with a uniform distribution was used.

The maximum connection length for the transportation network was of 141 stations and the average standard deviation of the connection length derived from the standard deviations of connection lengths of all the algorithms was 6.75.

The queries for the transportation graph were classified by the resulting connection length as follows:

Query class	Connection length range
0-42	0-42
43-59	43-59
≥ 60	< 60
N/A	No connection exists

Table 5.2.1.: Classification of queries by connection length for the case of the transportation network.

The queries for the random graph were classified by the resulting path length as follows:

Query class	Connection length range
0-18	0-18
19-26	19-26
≥ 27	≥ 27
N/A	No connection exists

Table 5.2.2.: Classification of queries by connection length.

For an illustration, you can see the distribution of connection length for the Dijkstra's Algorithm in Fig. 5.2.1.

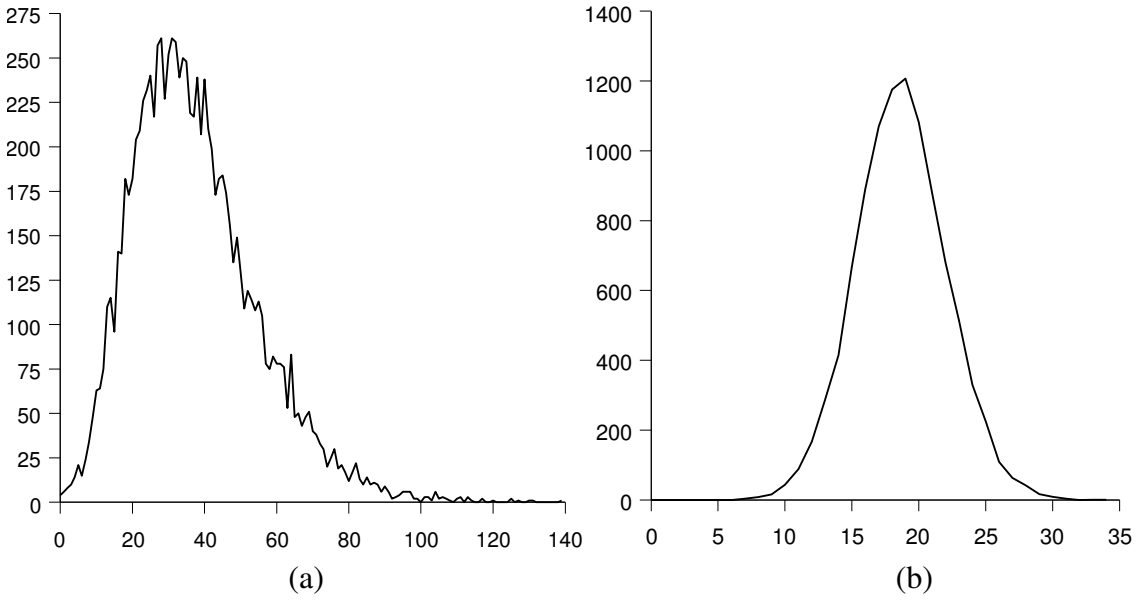


Fig. 5.2.1.: The number of queries as a function of the connection length and the path length for the test run of the Dijkstra's Algorithm for the transportation network (a) and the random graph (b).

5.3 The results

In the result tables below, we present the results of queries hitting the corresponding connection length ranges (given by the *connection length range* column), as well as the overall results for each test run. Each column represents one of the values described in section 5.1. Each of the rows 1 to 3 represents only those queries that had the resulting connection length from the range stated in the first column.

The fourth row represents queries for which no connection was found (i.e., such that no path exists in the graph).

The last row contains the overall results for a test run.

Two tables are presented for each algorithm except for the highway hierarchy search: one table for the test run on the transportation network and one table for the test run on the random graph. For highway hierarchies only a test run on the transportation graph was performed. This was due to the fact that the principal goal of this work was to examine the use of the algorithms on a transportation network, which was where most of the efforts concentrated.

5.3.1 Dijkstra's Algorithm

Unidirectional version of Dijkstra's Algorithm

In this section we present the results of the unidirectional version of the Dijkstra's algorithm. The earliest arrival optimization was used for the test on the transportation graph.

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	55.33	13 786	15 455	29 514	27.64	6 636
43-59	109.6	26 001	28 464	55 792	49.72	2 186
≥ 60	159.98	37 046	39 896	79 576	71.63	1 091
N/A	159.32	34 340	34 340	74 134	N/A	88
overall	79.52	19 173	21 130	41 112	37.35	10 001

Table 5.3.1.: Detailed results of unidirectional Dijkstra's search for the train network of the Czech Republic.

Path length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-18	866.70	85 313	118 763	173 291	15.88	4 838
19-26	1 569.40	153 791	186 346	312 341	21.13	5 026
≥ 27	2 006.85	198 501	218 349	403 069	27.96	137
overall	1 245.14	121 277	154 091	246 318	18.68	10 001

Table 5.3.2.: Detailed results of unidirectional Dijkstra's search for a random graph

Bidirectional version of Dijkstra's Algorithm

In this section we present the results of the bidirectional version of the Dijkstra's algorithm. The earliest arrival optimization in the forward search was used for the test on the transportation graph.

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	290.38	50 303	54 863	102 875	28.36	6 143
43-59	441.13	75 431	81 053	154 256	49.76	2 494
≥ 60	577.73	97 953	104 026	200 344	72.87	1 276
N/A	1 174.18	201 249	201 249	413 360	N/A	88
overall	372.64	64 015	68 991	130 935	38.3	10 001

Table 5.3.3.: Detailed results of bidirectional Dijkstra's search for the train network of the Czech Republic.

Path length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-18	17.67	2 279	4 564	4 627	15.88	4 837
19-26	33.22	3 985	7 863	8 086	21.13	5027
≥ 27	99.45	10 456	18 675	21 212	27.96	137
overall	26.61	3 248	6 416	6 593	18.68	10 001

Table 5.3.4.: Detailed results of bidirectional Dijkstra's search for a random graph

5.3.2 A* search

Unidirectional version of A Algorithm*

In this section, we present the results of the unidirectional A* algorithm. As the A* algorithm resulted to have the best overall performance, we present also two additional test results for the unidirectional version on the transportation graph, each corresponding to a modification presented in the previous chapter. The first will be the results for a modification of the algorithm, where the connections with fewer trains and with the largest stay at the origin station were searched, that is, both, the prefer fewer changes and the stay as long as possible at the origin station modifications were implemented. We refer to this version of the A* algorithm as the *version 1*. The second test run was performed on an algorithm with both the modifications from version 1, and with an additional modification to limit the number of changes. The number of changes was limited to 1 in all queries of this test run. We refer to this version of the A* algorithm as the *version 2*.

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	39.26	8 994	10 136	19 246	27.65	6 402
43-59	90.94	19 817	21 706	42 473	49.87	2 304
≥ 60	137.5	29 240	31 642	62 710	72.75	1 207
N/A	173.69	34 706	34 706	74 881	N/A	88
overall	64.21	14 157	15 613	30 332	38.3	10 001

Table 5.3.5.: Detailed results of unidirectional A* search for the train network of the Czech Republic.

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	135.51	27 823	29 903	57 099	27.57	7 480
43-59	268.18	53 020	55 998	108 721	49.17	1 899
≥ 60	309.68	60 872	64 118	124 850	68.58	534
N/A	634.02	118 290	118 290	241 098	N/A	88
overall	174.39	35 168	37 463	72 138	33.62	10 001

Table 5.3.6.: Detailed results of version 1 of the A* algorithm for the train network of

the Czech Republic.

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	28.15	4 530	4 626	8 981	21.67	1 096
43-59	42.57	7 085	7 189	14 072	48.74	86
≥ 60	38.36	6 754	6 850	13 420	64.73	11
N/A	36.95	6 604	6 604	13 028	N/A	8 808
overall	36.03	6 381	6 392	12 594	24.02	10 001

Table 5.3.7.: Detailed results of version 2 of the A* algorithm for the train network of the Czech Republic.

Path length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-18	560.24	56 106	85 143	113 977	15.88	4 839
19-26	1 188.49	117 047	153 904	237 736	21.13	5 025
≥ 27	1 726.49	171 413	198 738	348 108	27.96	137
overall	891.88	88 305	121 248	179 367	18.68	10 001

Table 5.3.8.: Detailed results of unidirectional A* search for a random graph

Bidirectional version of A Algorithm*

In this section we present the results of the bidirectional version of the A* algorithm. The earliest arrival optimization was used for the tests on the transportation graph.

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	253.86	41 273	45 086	84 504	28.81	5 913
43-59	428.58	67 438	72 609	138 178	49.91	2 575
≥ 60	590.37	91 512	97 194	187 438	72.77	1 425
N/A	1 268.37	208 921	208 921	428 854	N/A	88
overall	355.98	56 686	61 080	116 108	40.61	10 001

Table 5.3.9.: Detailed results of bidirectional A* search for the train network of the Czech Republic.

Path length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-18	18.69	2 318	4 640	4 707	15.88	4833
19-26	33.71	3 915	7 722	7 944	21.12	5031
≥ 27	95.91	9 760	17 688	19 802	27.97	137
overall	27.30	3 223	6 369	6 542	18.68	10 001

Table 5.3.10.: Detailed results of bidirectional A* search for a random graph

5.3.3 Highway hierarchies search

In this section we present the results of the highway hierarchies search based on both, the Dijkstra's and the A* algorithms. No tests on a random graph were executed for highway hierarchies, therefore only tests on the transportation graph for the transportation network of the Czech Republic are presented.

The configuration of the highway hierarchy was the following:

Parameter	**Value**
Number of levels	5
Neighbourhood	25 closest stations
shortcut factor	1.3

Table 5.3.11.: Parameters for highway hierarchy construction.

The resulting highway hierarchy had the following properties:

Level	**Number of nodes**	**Number of bypassed nodes**	**Number of edges**
0	241 263	N/A	489 734
1	224 150	147 212	729 200
2	75 606	38 716	334 615
3	36 837	11 640	212 554
4	25 191	3 874	157 463

Table 5.3.12.: Level sizes of the highway hierarchy. Number of nodes denotes the total number of nodes in a level (including bypassed nodes). Number of edges denotes the total number of edges in a level (including shortcut edges added during the graph contraction).

A final note, before we present the test results, both the algorithms had implemented the lazy backward search modification described in the previous chapter.

Dijkstra based search

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	69.42	5 861	9 987	34 001	28.59	5 816
43-59	96.41	7 898	12 417	54 313	49.98	2 598
≥ 60	125.11	10 336	14 616	75 041	73.58	1 499
N/A	60.24	5 439	5 579	39 461	N/A	88
overall	84.7	7 057	11 273	45 477	40.64	10 001

Table 5.3.13.: Detailed results of Dijkstra based HH search for the train network of the Czech Republic.

A based search*

Connection length range	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*	*Number of queries*
0-42	56.55	4 939	8 348	27 132	28.75	5 769
43-59	79.48	6 409	10 507	42 770	50.24	2 609
≥ 60	105.36	8 514	12 660	60 805	73.61	1 535
N/A	60.91	5 461	5 606	39 577	N/A	88
overall	70.06	5 876	9 549	36 489	40.99	10 001

Table 5.3.14.: Detailed results of A* based HH search for the train network of the Czech Republic.

5.4 Comparison

5.4.1 Transportation graph

The following table compares the *overall results* of all the algorithms on the train network of the Czech Republic. The lowest values are highlighted in each column.

Algorithm	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Connection length*
Unidirectional Dijkstra	79.52	19 173	21 130	41 112	37.35
Bidirectional Dijkstra	372.64	64 015	68 991	130 935	38.3
Unidirectional A*	64.21	14 157	15 613	30 332	38.3
A* version 1	174.39	35 168	37 463	72 138	33.62
A* version 2	**36.03**	6 381	**6 392**	**12 594**	24.02
Bidirectional A*	355.98	56 686	61 080	116 108	40.61
HH Dijkstra	84.7	7 057	11 273	45 477	40.64
HH A*	70.06	**5 876**	9 549	36 489	40.99

Table 5.4.1.: The comparison of the overall results of all the algorithms' performance for the train network of the Czech Republic.

The following table compares the *search times* of all the algorithms on the train network of the Czech Republic. The lowest values are highlighted in each column.

Algorithm\Connection length	0-42	43-59	≥ 60	N/A	overall
Unidirectional Dijkstra	55.33	109.6	159.98	159.32	79.52
Bidirectional Dijkstra	290.38	441.13	577.73	1 174.18	372.64
Unidirectional A*	39.26	90.94	137.5	173.69	64.21
A* version 1	135.51	268.18	309.68	634.02	174.39
A* version 2	**28.15**	**42.57**	**38.36**	**36.95**	**36.03**
Bidirectional A*	253.86	428.58	590.37	1 268.37	355.98
HH Dijkstra	69.42	96.41	125.11	60.24	84.7
HH A*	56.55	79.48	105.36	60.91	70.06

Table 5.4.2.: The comparison of all the algorithms' search times for the train network of the Czech Republic.

Comparing the results of the unidirectional Dijkstra and A* algorithms, we can see that A* outperforms Dijkstra in the overall figures as well as in all the 3 first query classes. Dijkstra's algorithm has somewhat lower search time for queries that yielded no connection. The same can be said when comparing the bidirectional versions of these algorithms.

What is very obvious is the big difference between the unidirectional and the bidirectional versions of the algorithms. For Dijkstra's as well as the A* algorithms, the overall search times for the bidirectional version are around five times the search times of their respective unidirectional versions. This is given by the fact that for each query all the arrival nodes corresponding to the destination stations had to be searched from in the backward search. For details, refer to section 3.1.4.

This is also the reason why both the highway hierarchy searches did not have much better search times. In fact the overall search time of the Dijkstra's search for highway hierarchies was even slightly worse than that of the original Dijkstra's search. The same can be said about the original A* and the A* for highway hierarchies, where again the original version outperformed the HH version in the overall figures. Nevertheless, the overall figures may be a little misleading, as the highway hierarchy search actually outperformed all the other algorithms, when only the queries with resulting connection length greater than 42 were considered. The fact that the number of these queries was less than half of all the queries explains why the average search time for highway hierarchies ranked below the “non-HH algorithms”.

The disadvantage of the bidirectional search seen in the bidirectional versions of the algorithms also applies to the HH search. It can be therefore inferred that HH search will be suitable for bigger graphs, where the additional cost caused by the bidirectional search would be outweighed by exploring less nodes in higher levels.

The shortest search times were achieved when the number of changes was limited to 1 (version 2 of A* algorithm). This result is most likely due to the fact that only a few nodes had to be searched before a second change was forced and the search thus finished. Note that the algorithm would return exactly the same connections as the version 1 of A* algorithm if the maximum changes parameter would be set sufficiently high. Nevertheless, the algorithm from version 2 of A* would perform *much worse* if the parameter were higher as various paths with the same tentative distance but different number of changes would have to be considered.

5.4.2 Random graph

The following table compares the *overall results* of all the algorithms on the random graph. The lowest values are highlighted in each column.

Algorithm	*Search time* [ms]	*Settled nodes*	*Reached nodes*	*Relaxed edges*	*Path length*
Unidirectional Dijkstra	1 245.14	121 277	154 091	246 318	18.68
Bidirectional Dijkstra	**26.61**	3 248	6 416	6 593	18.68
Unidirectional A*	891.88	88 305	121 248	179 367	18.68
Bidirectional A*	27.30	**3 223**	**6 369**	**6 542**	18.68

Table 5.4.3.: The comparison of the overall results of all the algorithms' performance for the random graph.

The following table compares the *search times* of all the algorithms on the train network of the random graph. The lowest values are highlighted in each column.

Algorithm\Path length	0-42	43-59	< 60	overall
Unidirectional Dijkstra	866.70	1 569.40	2 006.85	1 245.14
Bidirectional Dijkstra	**17.67**	**33.22**	99.45	**26.61**
Unidirectional A*	560.24	1 188.49	1 726.49	891.88
Bidirectional A*	18.69	33.71	**95.91**	27.30

Table 5.4.4.: The comparison of all the algorithms' search times for the random graph.

The best overall search time for the random graph has the bidirectional version of the Dijkstra's algorithm, with 26.61 ms, followed closely by the bidirectional version of the A* algorithm, with 27.3 ms. This was in spite of the fact that the overall number of settled nodes, reached nodes and relaxed edges was actually slightly lower for the A* algorithm. The slightly worse performance of A* was therefore likely caused by the additional computation time consumed by the continuous evaluation of the heuristic function during the performance of the algorithm. It can also be seen that for longer paths, it was more apparent that A* performed better and for paths of length over 59 actually beat the search time of the Dijkstra's algorithm. Therefore it can be assumed that the A* algorithm would perform better on larger graphs, where an average path length is considerably longer.

What is striking when looking at the results is the performance comparison of the unidirectional against the bidirectional versions of the algorithms. The bidirectional version of the algorithm outperformed the unidirectional one (in terms of search time) by the factor of nearly 48 in the case of the Dijkstra's Algorithm, and by the factor of

almost 33 in the case of A*. This is contrasting with the opposite result for the transportation graph search. This result can be explained by the fact that (as opposed to the search in transportation graph) in the case of the random graph the destination was represented by just one node so the number of nodes settled in the backward search was much lower than in the case of the search in the transportation graph.

Conclusion

We proposed a representation of a transportation network and evaluated the performance of several well known algorithms for the shortest path problem. We also proposed several modifications of the algorithms in order to improve the search times and the quality of the found connections.

From among the tested algorithms, the A* algorithm seems to be the most suitable as it proved to yield the best search times. In addition, it needs no pre-processing in contrast to the highway hierarchy search and therefore could be applied in a dynamic environment.

Our expectation that the highway hierarchy search will outperform dramatically the other algorithms was not fulfilled. Highway hierarchy search even had an overall performance worse than the one of A*. Given that highway hierarchies need additional preprocessing, have extra memory requirements and restricted timetables cannot be handled easily, it did not prove to be a good choice. It still can be expected that highway hierarchy search would give much better search times in comparison with the other algorithms on a considerably bigger transportation networks. This expectation is backed up by the test results where the highway hierarchy search outperformed the other algorithms in queries with comparatively long connections.

The representation of the transportation network presented in chapter 2 allows us to apply several algorithms for the shortest path problem in a straightforward manner. Moreover, some easy modifications to the search algorithms can improve the quality of the connections by returning connections with the smallest number of changes, smallest waiting time at intermediate stops, and pose additional constraints on the connections. Nevertheless, the presented representation of the transportation network has an important flaw, too. A comparatively small train network of the Czech Republic with only 4 138 stations resulted in a graph of 241 263 nodes and 489 734 edges, which was caused by the high number of trains in the network (11 813) and associated great amount of arrival and departure nodes. This expansion of the graph naturally affected the search times.

This calls into question how the search times would improve if a different representation of the transportation network were be used. For instance, we may think about a representation of the network based on a time dependent cost function such as the one used by B. C. Dean in [9].

Bibliography

[1] Uri Zwick (2001): Exact and approximate distances in graphs - a survey. *Lecture Notes In Computer Science; Vol. 2161; Pages 33 - 48*.

[2] J. Demel (2002): Grafy a jejich aplikace. *Academia, Praha.*

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein (2001): Introduction to Algorithms, Second Edition. *MIT Press, USA*.

[4] P. Norvig, S. Russel (2003): Artificial Intelligence: A Modern Approach. *Prentice Hall, USA*.

[5] A. V. Goldberg, C. Harrelson (2004): Computing the Shortest Path: A* Search Meets Graph Theory. *Microsoft Research*.

[6] D. Schultes (2005): Fast and Exact Shortest Path Queries Using Highway Hierarchies. *Universität des Saarlandes, Germany.*

[7] P. Sanders, D. Schultes (2006): Engineering Highway Hierarchies (full paper). *14th European Symposium on Algorithms (ESA); Vol. 4168; Pages 804 - 816.*

[8] S. Knopp, P. Sanders, D. Schultes, F. Schultz, D. Wagner (2007): Computing many-to-many shortest paths using highway hierarchies. *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments.*

[9] B. C. Dean (2004): Algorithms for Minimum-Cost Paths in Time-Dependent Networks with Waiting Policies. *Networks; Vol. 44*; *Pages 41 - 46.*

A Contents of the CD

The attached CD contains the C++ source code of all the implementations of the search algorithms described in this work and the source codes of the underlying data structures. All the source code was written by myself, with the exception of the implementation of the Fibonacci heap, which was taken from the Boost C++ open source library and adjusted for the purposes of this work.

Moreover, the CD contains a simple web-based search engine that allows to search for travel connections from within a web browser. The search engine was written in the Java programming language.

A C++ interface for an integration of the search engine with the search algorithms is included, and a simple C++ program that was used for the performance tests can also be found on the CD.

The contents of the CD is logically separated into directories. A file named dirlist.txt in the root directory of the CD contains a listing of the directories with a brief description of the contents of each of the directories.

B Class Diagrams

It is out of scope of this work to include a detailed design of the software and a detailed description of the algorithms' implementations. Therefore, only high-level class diagrams are presented, along with brief descriptions of the classes and its methods. There are two main entities implemented: a *Graph* and a *Searcher*. A Graph represents a directed graph, and a Searcher represents an implementation of a search algorithm. Class diagrams for these two entities follow.

Graph classes

The following is the class diagram for the Graph entity. Note that only public methods are listed. Short descriptions of each of the classes and its methods are listed in turn.

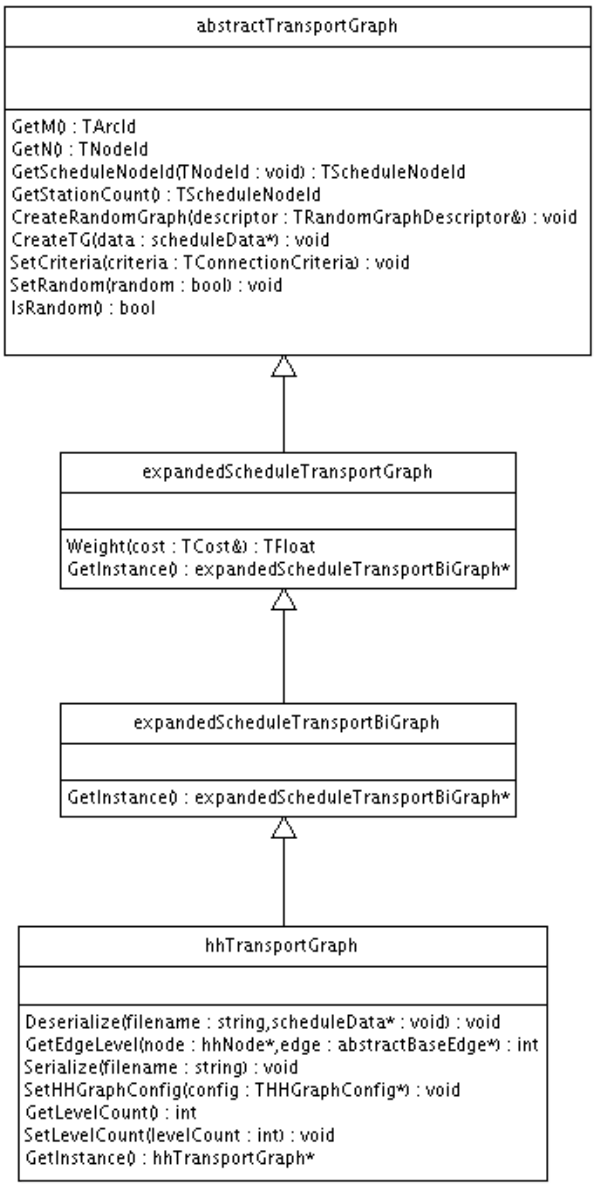


Fig. 1.: Class diagram of the Graph classes.

abstractTransportGraph is a generic abstract class of a directed graph representing a transportation network. All transportation graph implementations inherit from this generic class.

Method name	*Description*
GetM	Returns the number of edges of this graph.
GetN	Returns the number of nodes of this graph.
GetScheduleNodeId	Given a node number, returns the number of station this node represents.
GetStationCount	Returns the number of stations of the underlying transportation network if this instance is a transportation graph.
CreateRandomGraph	Generates a random graph of the size given by the TRandomGraphDescriptor parameter.
CreateTG	Creates a transportation graph based on the given data.
SetCriteria	Sets search criteria.
SetRandom	Sets whether this graph is a random graph (true) or a transportation graph (false).
IsRandom	Returns true if this is a random graph, false otherwise.

Table 1.: Public methods of abstractTransportGraph class.

expandedScheduleTransportGraph is an implementation of the transportation graph.

Method name	*Description*
Weight	An implementation of the weight function of the underlying transportation network.
GetInstance	Returns the only instance of this class (singleton design pattern).

Table 2.: Public methods of expandedScheduleTransportGraph class.

expandedScheduleTransportBiGraph is an implementation of the transportation graph. This class inherits from expandedScheduleTransportGraph and adds the functionality needed by the bidirectional versions of the algorithm (reverse graph).

Method name	*Description*
GetInstance	Returns the only instance of this class (singleton design pattern).

Table 3.: Public methods of expandedScheduleTransportBiGraph class.

hhTransportGraph is an implementation of a highway hierarchy.

Method name	*Description*
Deserialize	Given a file name of a serialized highway hierarchy, loads that highway hierarchy from the file. The file can be created by the Serialize method.
GetEdgeLevel	Given a node u and an edge (u, v), returns the level of the edge in this highway hierarchy.
Serialize	Makes a text stream of this highway hierarchy and stores it into a given file so that the highway hierarchy can be recreated using the Deserialize method.
SetHHGraphConfig	Specifies the configuration of the highway hierarchy. Has to be called before the CreateTG method is called.
GetLevelCount	Returns the number of levels of this highway hierarchy.
SetLevelCount	Sets the number of levels of this highway hierarchy, and allocates any additional memory, if the level count is to be increased or frees up no longer used memory, if the level count is to be decreased.
GetInstance	Returns the only instance of this class (singleton design pattern).

Table 4.: Public methods of hhTransportGraph class.

Searcher classes

The following is the class diagram for the Searcher entity. Note that only public methods are listed. Short descriptions of each of the classes and its public methods are listed in turn.

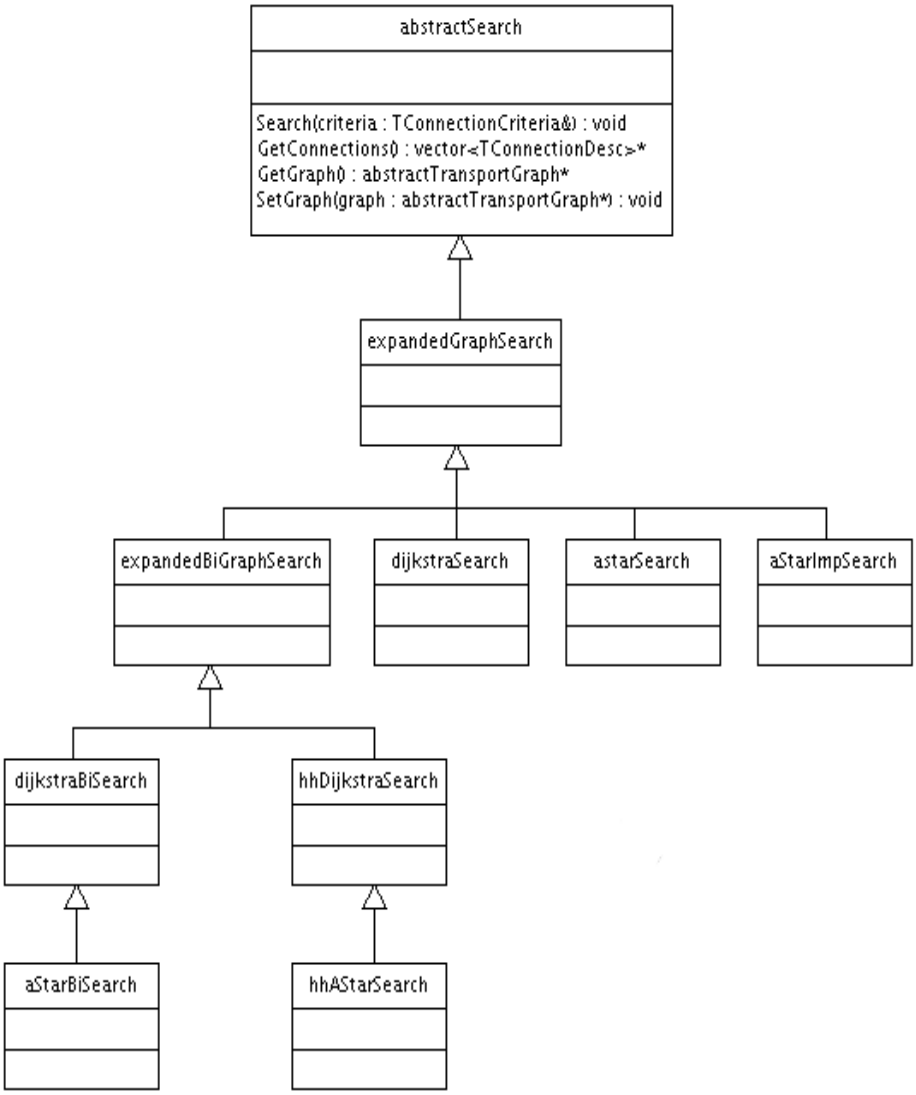


Fig. 2.: Class diagram of the Searcher classes.

abstractSearch is a generic searcher. All searcher classes inherit from abstractSearch.

Method name	*Description*
Search	Subclasses should provide an implementation of the search algorithm.
GetConnections	Returns the connections found by the last Search invocation.
GetGraph	Returns the graph that is searched.
SetGraph	Specifies an abstractTransportGraph instance that will be used during the search.

Table 5.: Public methods of abstractSearch class.

expandedGraphSearch is a convenience abstract class for all the (unidirectional)

searchers operating on a transportation graph.

expandedBiGraphSearch is similar to expandedGraphSearch, but is adjusted for bidirectional searchers.

dijkstraSearch is an implementation of the Dijkstra's search algorithm for transportation graphs.

astarSearch is an implementation of the A* search algorithm for transportation graphs.

aStarImpSearch is an implementation of the A* search algorithm for transportation graphs, which allows to limit the number of changes of found connections.

dijkstraBiSearch is an implementation of the bidirectional version of the Dijkstra's algorithm for transportation graphs.

astarBiSearch is an implementation of the bidirectional version of the A* search algorithm for transportation graphs.

hhDijkstraSearch is an implementation of the Dijkstra's algorithm for highway hierarchies.

hhAStarSearch is an implementation of the A* search algorithm for highway hierarchies.