Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava

Distance oracles for timetable graphs

(Master thesis)

bc. František Hajnovič

Study program: Informatics
Branch of study: 9.2.1 Informatics (2508)
Supervisor: doc. RNDr. Rastislav Královič, PhD.

Bratislava 2013

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. František Hajnovič
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Distance oracles for timetable graphs

Aim: The aim of the thesis is to explore the applicability of results about distance oracles to timetable graphs. It is known that for general graphs no efficient distance oracles exist, however, they can be constructed for many classes of graphs. Graphs defined by timetables of regular transport carriers form a specific class which it is not known to admit efficient distance oracles. The thesis should investigate to which extent the known desirable properties (e.g. small highway dimension) are present int these graphs, and/or identify new ones. Analytical study of graph operations and/or experimental verification on real data form two possible approaches to the topic.

Supervisor:	doc. RNDr. Rastislav Královič, PhD.
Department:	FMFI.KI - Department of Computer Science
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	08.11.2011

Approved: 15.11.2011 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

65787891

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. František Hajnovič
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Efektívny výpočet vzdialeností v grafoch spojení lineik.

Cieľ: Cieľom práce je preštudovať možnosti aplikácie výsledkov o distance oracles v grafoch reprezentujúcich dopravné siete na grafy spojení liniek. Otázka, či a aké dôležité vlastnosti ostávajú zachované sa dá riešiť teoreticky pre rôzne triedy grafov a/alebo experimentálne pre reálne dáta.

Vedúci:	doc. RNDr. Rastislav Královič, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	08.11.2011

Dátum schválenia: 15.11.2011

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

.....

Acknowledgements

I would like to thank very much to my supervisor Rastislav Královič for valuable remarks, useful advices and consultations that helped me stay on the right path during my work on this thesis.

I am also grateful for the support of my family during my studies and the work on this thesis.

František Hajnovič

Abstract

Queries for optimal connection in timetables can be answered by running Dijkstra's algorithm on an appropriate graph. However, in certain scenarios this approach is not fast enough. In this thesis we introduce methods with much better query time than that of the efficiently implemented Dijkstra's algorithm. We analyse these methods from both theoretical and practical point of view, performing experiments on various real-world timetables of country-wide scale.

Our first method called *USP-OR* is based on pre-computing paths, that are worth to follow (the so called underlying shortest paths). This method achieves speed-ups of up to 70 (against Dijkstra's algorithm), although at the cost of high amount of preprocessed data. Our second algorithm computes a small set of important stations and additional information for optimal travelling between these stations. Named *USP-OR-A*, this method is much less space consuming but still more than 8 times faster than the Dijkstra's algorithm on some of the real-world datasets.

Key words: **optimal connection**, **timetable**, **Dijkstra's algorithm**, **distance oracles**, **underlying shortest paths**

Abstrakt

Optimálne spojenia v cestovnom poriadku vieme hľadať pomocou Dijkstrovho algoritmu na vhodnom grafe, avšak v niektorých situáciách tento prístup výkonnostne nepostačuje. V tejto práci uvádzame metódy, ktoré na dotaz na optimálne spojenie odpovedajú podstatne rýchlejšie ako efektívna implementácia Dijkstrovho algoritmu. Tieto metódy analyzujeme ako z teoretického, tak aj z praktického hľadiska pomocou experimentov na viacerých cestovných poriadkoch celonárodnej škály.

Naša prvá metóda nazvaná *USP-OR* je založená na predpočítaní trás, ktoré sa oplatí následovať (tzv. podkladové najkratšie cesty). Táto metóda dosahuje faktor zrýchlenia až 70 (oproti Dijkstrovmu algoritmu), avšak za cenu veľkej pamäťovej náročnosti. Náš druhý algoritmus predrátava malú množinu dôležitých staníc a dodatočné informácie pre optimálne cestovanie medzi nimi. Táto metóda s názvom *USP-OR-A* je už oveľa menej pamäťovo náročná, stále však vyše 8 krát rýchlejšia ako Dijkstrov algoritmus na niektorých reálnych cestovných poriadkoch.

Kľúčové slová: **optimálne spojenie**, **cestovný poriadok**, **Dijkstrov algoritmus**, **dištančné orákulá**, **podkladové najkratšie cesty**

Contents

1 Introduction

World is getting smaller every day as new technologies constantly make communication and travelling faster and more effective then yesterday. Road network, Internet and many other networks are becoming more evolved and denser which also brings along new problems. In order to fully take advantage of such huge networks, we must have efficient algorithms that operate on these networks and give us answers to many questions. Among many others, one that we take particular interest in is the question: "What is the shortest path from place x to place y"?

In different networks this question can make different sense. In the road network, we would like to obtain a sequence of intersections we have to go through in order to reach our destination, driving the shortest possible time (or the smallest possible distance). GPS devices and the likes of Google maps have to deal with this problem. In case of the Internet network, we might be interested in the shortest path to a destination computer in terms of router hops. In a network of social acquaintances, the smallest number of persons connecting us e.g. with guitarist Mark Knopfler or Liona Boyd could be expressed as a shortest path problem. Many problems in artificial intelligence (e.g. planning of actions) can be expressed, or include, looking for shortest paths.

The tremendous amount of work done in this area signifies the importance of quick distance or shortest path retrieval in graphs. A simple Dijkstra's or A* algorithm no longer comply to the requirements of today's applications in which a server often has to answer hundreds of shortest path queries per second in a large-scale networks. To speed up the mentioned algorithms we usually sacrifice generality and concentrate on a particular type of network, or even on one concrete network.

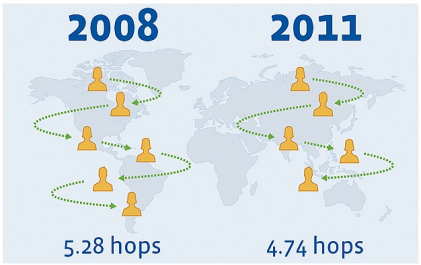

Figure 1.1: In a study carried out by Facebook it was claimed the average distance (in terms of friendship links) between the people on the site dropped from 5.28 in 2008 to 4.74 in 2011 [Bar].

In this thesis, the type of network we deal with is the one representing timetable connections, where nodes are the stations and arcs represent a direct connection between the two stations. We will talk in more details about this in following sections. However, this network has one substantial difference that we would like to point out - it is time-dependent. That means that the shortest path from station x to station y may have different solutions depending on the time when we start at station x. Therefore, we will not talk about shortest paths and distances, but rather about optimal connections and earliest arrivals and each query will now bear a third parameter - the departure time from x.

To informally develop the discussion about optimal connections in timetables, we will now clarify the motivation, approach and the goals of this thesis. We also sketch out the difference between the theory and practice when it comes to timetable search engines.

1.1 Motivation

We have already approached the motivation in the introductory text. We consider that a server (hosting e.g. journey-planning application) has to answer many queries per given time unit. What does it mean many? British National Rails Enquiries website that hosts journey planner supports over 1 million queries per day [1Te]. Even if these queries were distributed evenly throughout the

whole day, there would still be more than 11 queries per second.

11 queries per second is probably not a big issue. There is about 2500 railway stations in Great Britain and a current state of the art computer with basic implementation of a time-dependent Dijkstra's algorithm (to be talked about later) would be able to handle the mentioned load without any problems. However, things get more difficult on a bigger scale, in rush hours and when additional requirements are posed on the search results (transfers, cost of travel or simply outputting more results that the user can choose from).

In shortest path routing on road networks very much has been done to speed-up the query times using pre-processing on the input graph (for a good review of such methods, see [DSSW09]). Some developed methods answer distance queries more than 1 000 000 faster than the Dijkstra's algorithm on large road networks. In timetable scenario, the achieved speed-ups are much more modest. We will talk about the related work and achieved speed-ups in this area in the section 3.

1.2 Approach

We have mentioned that to get more effective algorithms with better query times, we need to focus on a special type of network and take advantage of its properties. In addition to this, what we can do is to pre-compute some information on the particular timetable and to use this information later to speed-up the answering of the queries. This is not a new technique and in the shortest path routing it is commonly referred to as creating a distance oracle [TZ05]. Our approach is different in only that instead of static graphs we deal with graphs representing the timetables [1] and look for optimal connections. We will go more into the details about this approach in the preliminaries section 2.

1.3 Goals

We have set two main goals for this thesis:

- **Analyse real-world timetables** and their properties. More specifically, given the graph representing the timetable, we were interested in its sparsity, connectivity, average and maximal degrees, average optimal connection sizes... We will talk about the various properties mostly in the section 4, but also throughout the rest of this thesis
- **Develop methods with fast query times** for optimal connections, based on pre-computing information

1.4 Theory and practice

This thesis is more theoretically oriented - we consider the optimal connection problem in probably its purest form which does not account for the many requirements posed by travellers using timetable search engines. Those include number of transfers, preferred route, cost of travel and others. These multi-criteria queries are discussed e.g. in [MHSWZ07]. In practice, we also usually want to output multiple connections, so that the user has a chance to choose suitable option. Needless to say, all of this makes the problem much more complicated and challenging than a pure search for an optimal connection.

On the other hand, the real-world timetable search engines concentrate usually on one given dataset, which enables them to exploit its properties [2] and tailor the search engine specifically

[1] Hence the name of this thesis - Distance oracles for timetable graphs, the "distance" being part of the title mostly because the term "distance oracle" is generally recognized.

[2] E.g. the city of Bratislava has only four (functioning) bridges, which could be taken into account when designing a public transportation search engine.

for it. There is also a choice of a suitable timetable model based on the characteristics of the given timetable.

The aim of the theoretical works (like this thesis) is not therefore to develop an algorithm immediately deployable into practice but rather to investigate techniques which might be useful to consider when designing practical timetable search engines.

1.5 Organization & conventions

This thesis is organized as follows:

- **Preliminaries**: We provide the necessary definitions (most notably timetable and its graph representations) and formally define the problem we deal with, as well as the approach we use
- **Related work**: This section summarizes the main related work in distance oracles, static route-planning and time-dependent scenarios
- **Data & analysis**: We introduce real-world timetables we worked with and analyse many of their properties
- **Underlying shortest paths**: In the main part of the thesis, we present the two methods we developed to speed-up optimal connection queries in timetables. These methods are also analysed from both theoretical and practical point of view
- **Neural network approach**: We summarize a little experiment in which we tried to train a neural network to answer optimal connection queries
- **Application TTBlazer**: This section shortly describes the application we used to analyse our datasets and test the methods
- **Conclusion**: Finally, we conclude, pointing out the main results and contribution, drawbacks and possibilities for future work

In this thesis, we also use some conventions:

- With a few exceptions, we will use **bold** font to mark currently defined term (or its notation)
- Names of our algorithms are in *italics*

2 Preliminaries

In this section, we provide most of the definitions and terminology used throughout the thesis.

2.1 Objects

First, we will formalize the notion of a timetable and its derived graph forms, the underlying graph and terms related to these objects.

Definition 2.1. ***Timetable (TT)***
A timetable is a set $\boldsymbol{T} = \{(x, y, p, q) |\ p, q \in \mathcal{N},\ p < q\}$.

- *Elements of* T *(the 4-tuples) are called* ***elementary connections****. For an elementary connection* $e = (x, y, p, q)$*:*
  - $\boldsymbol{from(e)} = x$ *is the* ***departure city (station)***
  - $\boldsymbol{to(e)} = y$ *is the* ***arrival/destination city (station)***
  - $\boldsymbol{dep(e)} = p$ *is the* ***departure time***
  - $\boldsymbol{arr(e)} = q$ *is the* ***arrival time***
- *The set of all* ***cities*** *(stations) will be denoted as* $\boldsymbol{ct_T} = \{x |\ (x, y, p, q) \in T \text{ or } (y, x, p, q) \in T\}$ *and the number of cities as* n_T
- *Pairs* (x, p) *or* (y, q) *such that* $(x, y, p, q) \in T$ *form the set of* ***events*** $\boldsymbol{ev_T}$*. The set of events in a specific city* x *is* $\boldsymbol{ev_T(x)} = \{(x, t) |\ (x, y, t, q) \in T \text{ or } (y, x, p, t) \in T\}$
- *Let* $\boldsymbol{tlow_T} = \min_{e \in T} dep(e)$ *and* $\boldsymbol{thigh_T} = \max_{e \in T} arr(e)$*. The value* $\boldsymbol{tr_T} = thigh_T - tlow_T$ *is called the* ***time range*** *of the timetable*
- ***Height*** *of the timetable is the average number of events in a city:*

$$\boldsymbol{h_T} = \frac{|ev_T|}{n_T}$$

Let us describe some of the defined terms more informally. An elementary connection corresponds to moving from one stop to the next one, e.g. with a bus (thus we disregard the notion of *lines*, i.e. getting on and off from a bus). Note that we express time as an integer - throughout this paper, this integer will represent the minutes elapsed from the time 00:00 of the first day. Thus we may take the liberty of talking about time in integer or *days hh:mm* format, as convenient at the moment. Lastly, an event simply represents an arrival or departure of a e.g. train at some station. The remaining terms should be clear enough.

Place		Time	
From	**To**	**Departure**	**Arrival**
A	B	10:00	10:45
B	C	11:00	11:30
B	C	11:30	12:10
B	A	11:20	12:30
C	A	11:45	12:15

Table 2.1: An example of a timetable - the set of elementary connections (between pairs of cities). An example of an event is a pair (A, 10:00), when some elementary connection departs from A.

Following is a definition of a connection.

Definition 2.2. ***Connection***
A connection from a to b is a sequence of elementary connections $\boldsymbol{c} = (e_1, e_2, ..., e_k), k \geq 1$, *such that* $from(e_1) = a$, $to(e_k) = b$ *and* $\forall i \in \{2, ..., k\} : (from(e_i) = to(e_{i-1}),\ dep(e_i) \geq arr(e_{i-1}))$.

- *We extend* $\boldsymbol{from(c)} = from(e_1)$, $\boldsymbol{to(c)} = to(e_k)$, $\boldsymbol{dep(c)} = dep(e_1)$, $\boldsymbol{arr(c)} = arr(e_k)$
- ***Length*** *of the connection is* $\boldsymbol{len(c)} = arr(c) - dep(c)$
- ***Size*** *of the connection is* $\boldsymbol{size(c)} = k$ [3]
- *We will denote the set of* ***all connections*** *from a to b in a timetable T as* $\boldsymbol{C_T(a, b)}$. *We also define* $\boldsymbol{C_T} = \cup_{a,b} C_T(a, b)$

So we understand connection as a (valid) sequence of elementary connections.

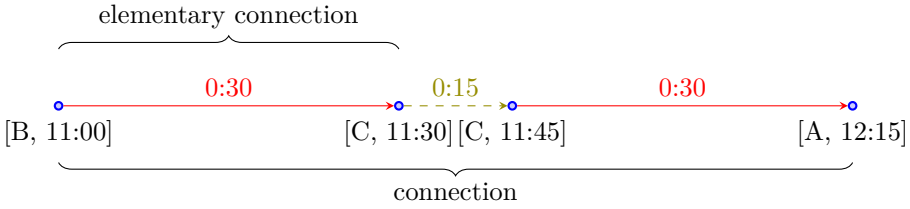

Figure 2.1: A valid connection made out of elementary connections (and waiting, which is implicitly fills out the time between successive elementary connections).

Next, we continue with the underlying graph - a graph representing basically the map on top of which the timetable operates.

Definition 2.3. ***Underlying graph (UG graph)***
The underlying graph of a timetable T, denoted $\boldsymbol{ug_T}$, *is an oriented graph* (V, E), *where V is the set of all timetable cities and* $E = \{(x, y) | \exists (x, y, p, q) \in T\}$

- *By* $\boldsymbol{m_T}$ *we will denote the number of arcs in the UG*

Note, that we do not specify the weights of the edges in the underlying graph - they will be specified based on the current usage of the UG. Most of the time, however, if we work with a weighted UG, the weight of an arc will be the length of the shortest elementary connection on that arc. More specifically, $w(x, y) = \min_{(x,y,p,q) \in T}(q - p)\ \forall (x, y) \in E(ug_T)$. Such weighted UG will be called **optimistic** (denoted $\boldsymbol{ug_T^{opt}}$).

If we want to represent the timetable by a graph, there are two most common options [MHSWZ07] - the time-expanded and time-dependent graph.

Definition 2.4. ***Time-expanded graph (TE graph)***
Let T be a timetable. Time-expanded graph of the timetable T, denoted $\boldsymbol{te_T}$, *is an oriented graph* (V, E) *whose vertices correspond to events of T, that is* $V = \{[x, t] | \ (x, t) \in ev_T\}$. *The edges of G are of two types*

1. $([x, p], [y, q])\ \forall (x, y, p, q) \in T$ *- the so called* ***connection edges***
2. $([x, p], [x, q])\ [x, p], [x, q] \in V,\ p < q$ *and* $\nexists [x, r] \in V : p < r < q$. *- the so called* ***waiting edges***

Weight of the edge $([x, p], [y, q])$ *is* $w([x, p], [y, q]) = q - p$.

Informally, an edge in TE graph represent either the travelling with an elementary connection or waiting for the next event in the same city. Also, the time range and height of a timetable could be easily illustrated on the TE graph (see figure 2.3).

[3] We will use similar terminology when talking about paths - the *size* is the number of vertices (hops) in the path while the *length* refers to the actual distance (sum of weights of the edges in the path).

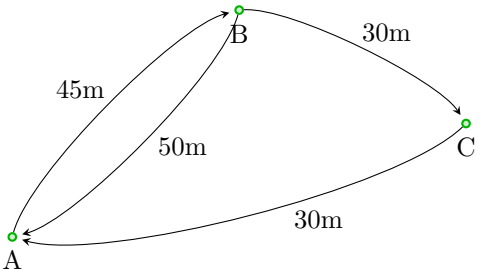

Figure 2.2: An optimistic underlying graph of the timetable 2.1. The nodes are the cities of the timetable.

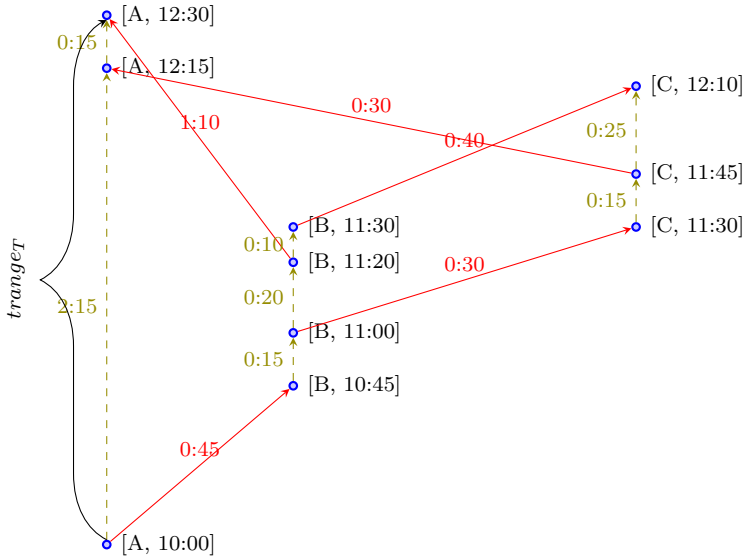

Figure 2.3: Time-expanded graph of the timetable 2.1. Nodes represent the events. There are connection and waiting edges (dashed). The time range is 2h:30m and the height is $\frac{10}{3}$ (since there are as many events in a city on average).

Definition 2.5. ***Time-dependent graph (TD graph)***
Let T be a timetable. Time-dependent graph of the timetable T, denoted $\boldsymbol{td_T}$, is an oriented graph (V, E) whose vertices are the timetable cities and $E = \{(x, y) |\ \exists (x, y, p, q) \in T\}$. Furthermore, the weight of an edge $(x, y) \in E$ is a piece-wise linear function $w(x, y) = f_{x,y}(t) = q - t$ where q is:

- $\min\{arr(e) |\ e \in T,\ dep(e) \geq t\}$
- ∞, *if* $dep(e) < t\ \forall e \in T$

Intuitively, the TD graph is simply the UG graph with each arc carrying a function specifying the traversal time of that arc at any time. For an example, see figure 2.5.

The algorithms in this thesis use almost exclusively the TD graphs, mainly because they are less space consuming. Also, time-dependent Dijkstra searches are a bit faster on TD graphs, because the search space that has to be explored is smaller. On the other hand, TE graphs are more flexible when we need to take additional search parameters into consideration (like transfers, travel

costs). Since we will not talk about these, TD graphs are more suitable.

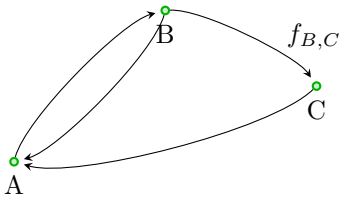

Figure 2.4: Time-dependent graph of the timetable 2.1. The nodes are the cities.

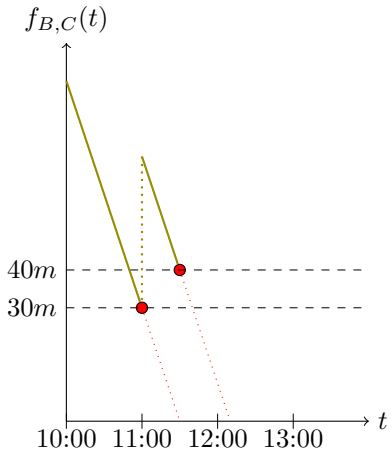

Figure 2.5: Piece-wise linear function - traversal times for the arc (B, C). The highlighted points are called **interpolation points** and each of them corresponds to an elementary connection (its coordinates are $dep(e), len(e)$ for a corresponding elementary connection e). Note that a list of all interpolation points fully defines the piece-wise linear function.

To sum up, there are four main types of objects we will be working with:

- Timetable (TT)
- Underlying graph (UG)
- Time-expanded graph (TE)
- Time-dependent graph (TD)

For future reference, we will call TT, TE and TD as **timetable objects** and UG, TE and TD as **graph-like objects**.

Note: Throughout this paper, we will relax a bit the notation and leave out subscripts (e.g. $ug_T \rightarrow ug$, $n_T \rightarrow n$, etc.) in situations, where the context is clear enough.

2.2 Earliest arrival and optimal connection

Now we formulate the main problems this thesis deals with.

Definition 2.6. ***Earliest arrival problem (EAP)***
Given a timetable T, departure city x, destination city y and a departure time t, the task is to determine $\boldsymbol{t^*_{(x,t,y)}} = \min_{c \in C_T(x,y)}\{arr(c) | \, dep(c) \geq t\}$.

- *We will refer to the tuple* $\boldsymbol{(x, t, y)}$ *as an* ***EAP instance****, or an* ***EAP query*** *(or just* ***query****)*
- *The time* $\boldsymbol{t^*_{(x,t,y)}}$ *is called the* ***earliest arrival (EA)*** *for the given EAP instance*

A bit more difficult version of this problem is one where we require to actually output the connection arriving at time given by EA.

Definition 2.7. ***Optimal connection problem (OCP)***
Given a timetable T, departure city x, destination city y and a departure time t, the task is to find the ***optimal connection (OC)*** $\boldsymbol{c^*_{(x,t,y)}} = argmin_{c \in C_T(x,y)}\{arr(c) |\ dep(c) \geq t\}$.

The instance/query in case of the optimal connection problem has the same form as EAP query. Also, note that the OCP is at least as hard to solve as EAP since having the optimal connection implies the optimal (earliest) arrival time. In order to avoid technical issues (e.g. in the definition), we may assume that the optimal connection is unique (i.e., there is not a different connection with the same end time). However, we consider any connection which arrives at time $t^*_{(x,t,y)}$ to be optimal for the given query.

Example 2.1. *Consider our timetable from table 2.1. For the EAP instance (B, 10:45, A), the earliest arrival (EA) is 12:15 and the optimal connection (OC) is ((B, C, 11:00, 11:30), (C, A, 11:45, 12:15)), as could be easily seen from the figure 2.6.*

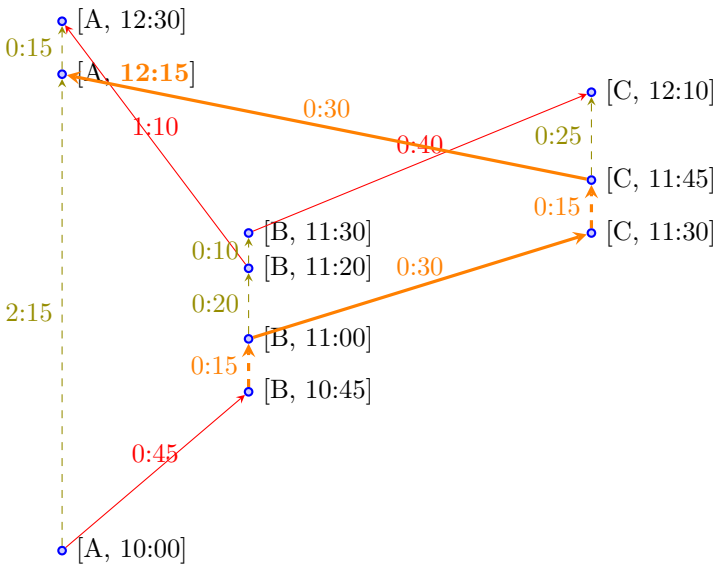

Figure 2.6: Depicting the situation from example 2.1 on TE graph. The optimal connection and earliest arrival time are marked in **bold**.

2.3 (Distance) Oracles

The term *distance oracle* was first coined in 2001 by Thorup and Zwick [TZ05], when talking about quick shortest path (or distance) computations on graphs. One approach to this problem is to pre-compute some information on the graph to speed-up answering of the queries. The paper of Thorup and Zwick was dealing with trade-offs among the time complexity of the pre-computation, the amount of preprocessed information, the speed-up in query times and the accuracy of the answers. Since the pre-computed data structure is something that helps us answer the queries more efficiently, it resembles an oracle, thus the term distance oracle.

In this thesis, we will discuss methods that behave the same way, but deal with the optimal connection problem (or earliest arrival problem) - there is some preprocessing of the timetable with a resulting data structure that speeds up answering subsequent queries. To formalize this a little

more, we will refer to this kind of methods as **oracle based methods**. For such a method m, we are interested mainly in its four parameters:

- **Preprocessing time** (***prep*(*m*)**) - the time complexity of the pre-computation
- **Preprocessed space** (***size*(*m*)**) - the space complexity of the pre-computed data structure (the so called **oracle**)
- **Query time** (***qtime*(*m*)**) - the time complexity of answering a single query
- **Stretch** (***stretch*(*m*)**) - the worst-case ratio against the optimal value of the earliest arrival (the lower, the better)

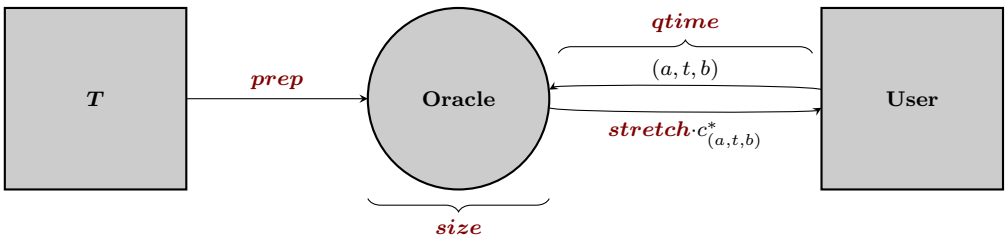

Figure 2.7: Principle of oracle based methods - we preprocess the timetable, creating a structure that helps us speed-up the answers to queries for optimal connection.

The preprocessing time is probably the least critical resource. A reasonable polynomial should bind its time complexity, depending on the computational power of the user and the scale of the timetable. The size of the preprocessed oracle is much more important - in the optimal case, it should be bound by the space complexity of the timetable itself. Optimality of the query time depends on which problem we are solving. If we query for the whole optimal connection, we have to count with a time complexity at least proportional to the average optimal connection size. If we require only the EA value as an output, much better speed-ups could be expected. The stretch should be of course as low as possible.

2.4 Dijkstra's algorithm

Throughout this thesis, we will often use Dijkstra's algorithm and its modifications both as a part of our algorithms and as a reference point against which we will compare the performance of our methods. This is a common practice. Researchers working on methods answering distance or shortest path queries in road networks commonly use the term *speed-up*, i.e. *how many times faster* is their algorithm against the Dijkstra's algorithm.

Dijkstra's algorithm is originally an algorithm that looks for shortest paths in weighted oriented graphs. It was published by E. W. Dijkstra in 1959 [Dij59] and we will not explain it at this place, as the algorithm is very well explained at many other places (e.g. [KP]). For a good summary of Dijkstra's algorithm related implementations and publications see [Som10].

As our task is to compute earliest arrivals or optimal connections instead of distances and shortest paths, our "reference point" will be a slightly modified Dijkstra's algorithm called **time-dependent Dijkstra's algorithm** [DW09] (or **TD Dijkstra** for short). The algorithm is run on a time-dependent graph and works just like the ordinary Dijkstra's algorithm, except that the weight of each arc (x, y) is determined for the time t at which we had settled vertex x.

If we assume that the evaluation of an arc by the cost function of the TD graph is imple-

mented in constant time, the running time of the *TD Dijkstra* is $\mathcal{O}(n^2)$, just like the normal Dijkstra's algorithm. On sparse graphs, this bound can be improved using a quick data structure to determine the next node we settle. A good option is a priority queue implemented as a *Fibonacci heap*, which implements deletion in $\mathcal{O}(\log n)$ and all other operations in constant amortized time [Som10]. This yields the running time of *TD Dijkstra* $\mathcal{O}(n \log n + m)$.

We may therefore introduce a fifth parameter of our oracle based methods, the speed-up:

Definition 2.8. ***Speed-up (spd(m))***
A speed-up of an oracle based method m is the ratio $\frac{qtime_{avg}(TD\ Dijkstra)}{qtime_{avg}(m)}$ *where* $qtime_{avg}(m')$ *is the average query time of the respective oracle based method* m' [4].

The definition is rather loose in the sense that we may refer to a concrete speed-up of the method on a concrete dataset, or a general theoretical speed-up expressed as a function of the size of input.

[4] Note that we may also consider the *TD Dijkstra* algorithm to be an oracle based method - it just happens that it does not require any preprocessing.

3 Related work

In this section, we summarize the work related to the subject of this thesis. Apart from the papers discussing searching for optimal connections and earliest arrivals in time-dependent scenarios, we also briefly summarize the research done on route planning in road networks and on distance oracles in general.

3.1 Distance oracles and route-planning

We have already mentioned in section 2 the paper of **Thorup and Zwick** [TZ05] where the term "distance oracle" originated. The authors have shown that given an undirected weighted graph of n vertices and m edges and a chosen integer $k \geq 1$, we can build a distance oracle such that:

- preprocessing takes $O(kmn^{1/k})$ expected time
- resulting distance oracle is of size $O(kn^{1+1/k})$
- answering queries takes $O(k)$ time
- stretch is at most $2k - 1$

Moreover, the authors have reasoned that their construction is essentially optimal with respect to space - i.e., if we want to have exact and constant-time answers, we will in general be forced to pre-compute $\Omega(n^2)$ information. The parameter k however provides a nice option to make trade-offs between the four parameters, as depicted on figure 3.1.

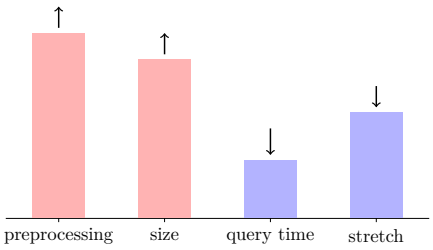

Figure 3.1: By moving k (decreasing on the picture), we can achieve compromises between the four parameters of the distance oracle.

Another work by **Gavoille** et al. [GPPR04] concerned distance labelling - a somewhat restricted version of a distance oracle where we assign each node in the graph its distance label. This is again only some pre-computed information and upon a query from x to y, we should be able to figure out their distance only using the corresponding distance labels. In the paper it is shown that for all n, there exist infinitely many graphs of n vertices for which we have an exact distance labelling scheme of a small overall size ($\mathcal{O}(n \log n)$), but for which the process of figuring out the distance from the labels takes too long from practical point of view.

Even though these results imply that we cannot create a sufficiently small efficient distance oracle in general, it may still be possible for sub-classes of general graphs, or even better, for a single particular graph. In that respect, the road network is the point of interest and fortunately it has a few "nice" properties (it is sparse, almost planar, the maximum node degree is small...) which made

it possible to design exact and efficient algorithms with extremely fast query times. To name a few of these:

- **Highway hierarchies** (2005, [SS05]). The preprocessing of the algorithm works in iterations - in each of them the edges of little importance are pruned, the remaining graph is contracted (long chains of edges are replaced with shortcuts) and the result forms the new layer, connected to the previous one, and used as an input for the next iteration. On such hierarchy of layers, bidirectional Dikstra's algorithm is run, climbing up the hierarchy in each direction.
  - Speed-up: about 2500
  - Techniques: hierarchy, shortcuts, bidirectional Dijkstra
- **Transit node routing** (2006, [BFM06]). The algorithm completely replaces searching with table look-ups. There is a small set of transit nodes between which the exact distance is stored in a table. Also each node remembers its nearest transit nodes (called access nodes [5]) and their distance. A search is necessary only in case of a local query. A disadvantage is a bigger space consumption.
  - Speed-up: more than 1 000 000
  - Techniques: landmarks

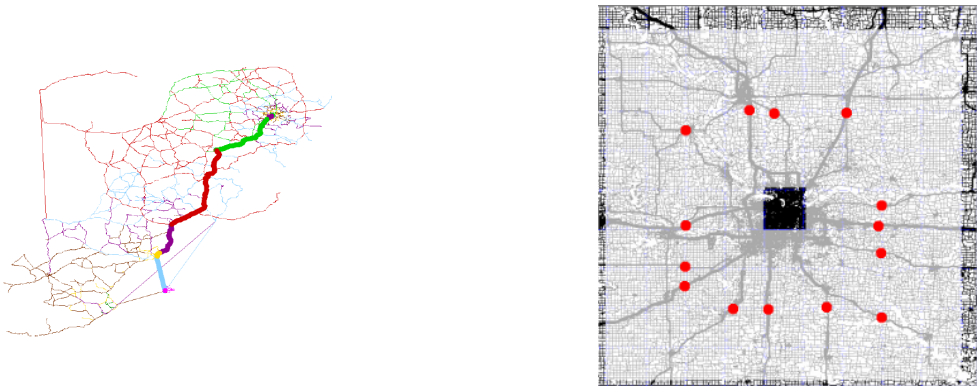

Figure 3.2: Highway hierarchies (left) - the bidirectional Dijkstra search climbs up the hierarchy to reach the most sparse level. Transit node routing (right) - access nodes (in red) that cannot be avoided when going "out of town".

- **Contraction hierarchies** (2008, [GSSD08]). The preprocessing creates additional shortcut edges in the graph. This is done by deleting one by one the vertices of the original graph and adding shortcuts where necessary - to preserve original distances. The quality of the preprocessing depends mostly on the order in which we delete the vertices. Upon a query, a bidirectional Dijkstra search is run on the original graph enriched with the added shortcuts. The algorithm is less memory demanding than Transit node routing.
  - Speed-up: more than 30 000
  - Techniques: shortcuts (contractions), bidirectional Dijkstra

One thing these methods have in common is that their query time is very low in practice, but it is not guaranteed theoretically. This was the point of interest in the work [AFGW10], which introduces a parameter called *highway dimension*. The authors show that a low highway dimension guarantees good query times of many route-planning algorithms, including the three we have mentioned.

[5] This served partly as an inspiration for our algorithm *USP-OR-A* discussed in section 5.

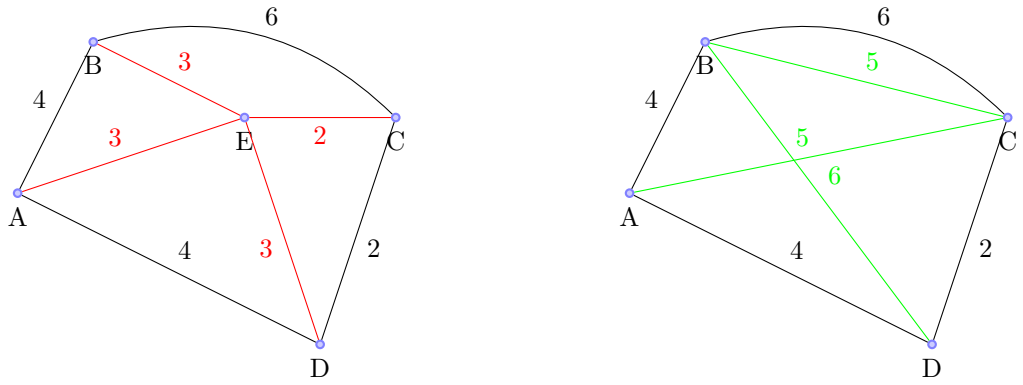

Figure 3.3: Deleting vertex E in Contraction hierarchies. Before (left) and after (right).

A very good summary of the techniques devised for road network route-planning up to the year 2009 can be found in [DSSW09]. Efficient distance oracles are also known for graphs with small recursive separators [GPPR04]. The work [Som10] suggests efficient distance oracle for power-law graphs and another distance oracle method for general graphs, offering trade-offs between stretch and query times. It also gives an exhaustive and comprehensive discussion regarding shortest path queries in general, which we point out to interested readers.

3.2 Time-dependent scenario

The time-dependent scenario has so far seen much smaller speed-ups than static routing in road networks, one reason for this being that the adaptation of the many techniques used for road networks to the time-dependent scenario is not so straightforward. This is mainly due to the fact that running bidirectional Dijkstra's algorithm (commonly used with static route-planning techniques) in time-dependent networks requires the knowledge of the destination time [DPW09]. All the same, for some methods this adaptation was carried out with good results:

- **Time-dependent contraction hierarchies** (2009, [BDSV09]). The focus in this case was on road networks having time-dependent edge weights (e.g. the traversal time varies due to congestions) and on computing earliest arrival value for a given query. The main difference between the static Contraction hierarchies is that the backward search is run from more arrival times of the destination node. Each such run may then contribute a new lower or upper bound for the actual earliest arrival value, based on if the forward and backward search met (a solution was found) or not.
  - Speed-up (TD road-network, 18 million nodes): up to 2000
- **Time-dependent SHARC** (2008, [Del08]). Static SHARC is an algorithm using unidirectional Dijkstra, it was therefore a good candidate to be adjusted for time-dependent scenario. It combines several techniques, perhaps the most important being pre-computing arc flags (see e.g. [KMS06]) for a multi-partition of the input graph, which is basically an information stating if the given arc should/should not be considered when travelling to the destination cell.
  - Speed-up (TD road-network, 5 million nodes): up to 800
  - Speed-up (timetable, 30 000 stations): up to 27
- **Engineering time-expanded graphs...** (2009, [DPW09]). While the previous papers used the time-dependent model of the timetable, this work concentrates on the time-expanded model. On a high level, this model is further refined by bypassing some low degree nodes,

remodelling unimportant stations and introducing time-dependent shortcuts - still in the phase of preprocessing. During the query, additional speed-up techniques are deployed to reduce the search space explored by Dijkstra's algorithm, as this can get quite huge in time-expanded graphs.

- Speed-up (timetable, 30 000 stations): up to 57

A summary of some time-dependent route planning techniques (up to year 2009) can be found in [DW09]. The paper [MHSWZ07] discusses also multi-criteria queries and gives an overview on comparisons between the time-dependent and time-expanded timetable model.

4 Data & analysis

In this section we would like to introduce the timetable datasets we were working with and provide the analysis of their properties. The main reason for this analysis is that it gives some insight into the characteristics of the timetables and so may contribute to develop an oracle based method with better qualities.

4.1 Data

We have obtained timetable datasets from numerous sources, in varying formats and of different types. Some of them were freely available on the Internet while others were provided by companies upon demand. Let us provide their brief description.

The dataset *air01* contains schedules of **domestic flights in United States** for the January of 2008. It is not comprehensive in the sense that it contains entries only for flights of some of the major airports in US. However it is large enough for our purposes (almost 300 airports). This dataset is just a fraction of the data that are freely available at the pages of American Statistical Association [6] in CSV format.

Timetable *cpsk* represent the **regional bus** schedules from the areas of **Ružomberok and Žilina, Slovakia**. The data were provided by the company in charge of the *cp.sk* portal - Inprop s.r.o. . The timetable contains about 1900 bus stops and came in a JDF 1.9 format [7]. Apart from the actual schedules, the data in JDF contain numerous other information which were not relevant for our purposes. From both timetables we have extracted subsets with a time range of one day.

The *gb-coach* and *gb-train* timetables are freely available from National Public Transport Data Repository (NPTDR) [8] in an ATCO-CIF format. These are not actually timetables but rather weekly snapshots of national public transport journeys made by **coach and train in Great Britain** (during certain week in year 2011). The datasets contain about 2500 stations each.

The *montr* dataset is part of a public feed for **Greater Montreal public transportation**, available at Google Transit Feeds [9]. The data are in a GTFS format (defines relations between CSV files listing stations, routes, stop-times...) and were made available by Montreal's Agence métropolitaine de transport. Our timetable *montr* corresponds to daily schedules of the Chambly-Richelieu-Carignan bus services (more than 200 bus stops).

Also in GTFS format come the data of **French railways** operated by company SNCF, publicly available at their website [10]. The schedules are weekly and there were two of them: one for intercity trains and one for TER trains (regional trains). Thus the three timetables *sncf-inter* (366 stations), *sncf-ter* (2637 stations) and their union *sncf* (2646 stations).

Finally, one more country-wide railway timetable was provided by ŽSR, the company in charge of the **Slovak national railways**. This timetable was exported in a MERITS format and its time range is for one year. The number of stations in *zsr* dataset is 233.

With the help of Python and Bash scripts, we converted each of these datasets to our timetable format (described in appendix A). This timetables were then loaded by our application TTBlazer,

[6] `http://stat-computing.org/dataexpo/2009/the-data.html`
[7] Jednotný dátový formát (JDF).
[8] `http://data.gov.uk`
[9] `http://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds`
[10] `http://test.data-sncf.com/index.php/ter.html`

which can further generate sub-timetables (with less stations or smaller time range), underlying graphs and TE and TD graphs.

For a summary of the used timetables' descriptions, see table 4.1 and for their main properties, refer to table 4.2.

Name	Description	Format	Provided by	Publicly available
air01	domestic flights (US)	CSV	American Stat. Assoc.	✔
cpsk	regional bus (Ružomberok & Žilina, SVK)	JDF 1.9	Inprop s.r.o.	✗
gb-coach	country-wide buses (GB)	ATCO-CIF	NPTDR	✔
gb-train	country-wide rails (GB)	ATCO-CIF	NPTDR	✔
montr	public transport (Montreal, CA)	GTFS	Montreal AMT	✔
sncf	country-wide rails (FRA)	GTFS	SNCF	✔
zsr	country-wide rails (SVK)	MERITS	ŽSR	✗

Table 4.1: Datasets descriptions.

Name	El. conns.	Cities	UG arcs	Time range	Height
air01	601489	287	4668	1 month	2512.3
cpsk	97916	1905	5093	1 day	50.2
gb-coach	260710	2448	5793	1 week	106.3
gb-train	1714535	2555	8335	1 week	800.4
montr	7153	217	349	1 day	33.1
sncf	416302	2646	7994	1 week	288.1
sncf-inter	22750	366	901	1 week	111.7
sncf-ter	393587	2637	7647	1 week	274.2
zsr	932052	233	588	1 year	7322.2

Table 4.2: Main properties of the timetables. The value of time range is approximate.

To see better the differences in the properties of different timetable types (train, flight, bus...), we made sub-timetables with 200 cities and with the upper bound on time range being 1 day and 6 hours [11] ($thigh_T <$ 1 day and 6 hours) from each of our dataset. We name these datasets by appending to the original name "*-200d*" [12]. See table 4.3 for details.

Name	El. conns.	Cities	UG arcs	Height
air01-200d	19010	200	3973	112.7
cpsk-200d	14747	200	592	50.7
gb-coach-200d	2760	200	564	48.0
gb-train-200d	24323	200	792	129.6
montr-200d	6841	200	320	35.0
sncf-200d	4192	200	611	42.4
sncf-inter-200d	2172	200	493	20.8
sncf-ter-200d	8469	200	600	34.0
zsr-200d	2031	200	454	21.6

Table 4.3: 200-station sub-timetables with the time range of one day.

Also, to further justify our choice of using TD graphs instead of TE graphs in this thesis, we provide

[11] We took all elementary connections that were within our time range. From this timetable, we made an UG and its (random) sub-graph of 200 cities. Finally we selected only those elementary connections, that were on top of this sub-graph to form a timetable with 200 cities and the desired (maximal) time range.

[12] Similarly, throughout this thesis, suffix "*-d*" would mean "with daily time range", "*-w*" "weekly time range" and suffix "*-#*" would mean sub-timetable with # stations.

their space consumption comparison in table 4.4.

	TD graph			TE graph		
Name	**Nodes**	**Arcs**	**Size (MB)**	**Nodes**	**Arcs**	**Size (MB)**
air01	287	4668	27	715211	1307432	72
cpsk	1905	5093	5	95601	189205	11
gb-coach	2448	5793	12	259589	512862	32
gb-train	2555	8335	79	2042316	3745751	263
montr	217	349	0.4	7182	13992	0.9
sncf	2646	7994	19	758867	1166646	85
sncf-inter	366	901	1.1	39765	60602	4.6
sncf-ter	2637	7647	18	720651	1107301	81
zsr	233	588	42	1706077	2637896	173

Table 4.4: Space consumption of time-dependent vs. time-expanded model. The number of nodes and arcs for TD graph is the same as for the corresponding underlying graph.

4.2 Analysis of properties

First we will take a look at the optimal connection *sizes* (size is the number of elementary connections in a connection) in the timetables. For a given timetable T, we will denote the average optimal connection size as $\boldsymbol{\gamma_T}$ and will call it the **optimal connection diameter** (OC diameter). We computed an approximate OC diameter for each of our datasets by measuring an average connection size of sufficiently many OCs. The results in table 4.5 indicate that the average OC size generally falls under $\sqrt{n}$.

Next we would like to get an idea of the sparsity of the underlying graphs. We see from the table 4.2 that the graphs are pretty sparse (with the exception of *air01*), but we would like to make sure that the sparsity is uniform. More specifically, we will be interested in the δ-density:

Definition 4.1. ***δ-density***
A graph G of n vertices and m arcs is δ-dense $\iff \forall G' \subseteq G, n' \geq \sqrt[4]{n} : \frac{m'}{n'} \leq \delta$

- *For a timetable T, we will denote its **density** parameter* [13] *as* $\boldsymbol{\delta_T} = min\{\delta |\ ug_T \text{ is } \delta\text{-dense}\}$

To find out at least approximate δ_T values for our timetables, we have randomly sampled their UGs for (connected) sub-graphs of various sizes (starting from $\sqrt[4]{n}$ [14]). In table 4.6 you can see the maximal density found during the sampling. With exception of *air01*, it is less than $\log n$.

The density is related to the **average degree** deg_{avg} in the UG, since in oriented graphs:

$$deg_{avg} = \frac{m}{n}$$

So the average degree is a lower bound on the graph's density. Table 4.7 lists the average and maximal degrees in the underlying graphs.

We would also assume, that the underlying graphs of each timetable will be **connected** (and even strongly connected), or at least that the largest connected component spans almost the whole graph.

[13] Note that this has nothing to do with the frequency of elementary connections, only with the density of the underlying graph.

[14] The choice of $\sqrt[4]{n}$ will be justified later, during the analysis of the algorithms.

Name	γ_T	Max. OC size found	$\sqrt{n}$
air01	2.4	8	16.9
cpsk	40.8	162	43.6
gb-coach	25.2	128	49.5
gb-train	25.6	111	50.5
montr	21.1	63	14.7
sncf	36.8	111	51.4
sncf-inter	17.1	58	19.1
sncf-ter	48.0	167	51.3
zsr	15.0	57	15.3

Table 4.5: With one exception, OC diameter is less then $\sqrt{n}$ (this was expected, as *montr* is the only timetable with "geographically one dimension long" - all other timetables span areas with more uniform shape). Note extremely low value for airline timetable - this is due to the fact that UGs of airline timetables have small-world characteristics [Som10]. Another thing we may notice is that regional timetables (*cpsk*, *sncf-ter*) have higher OC diameter then country-wide and inter-city timetables. We also point out that the inter-city trains in French railways decrease the average optimal connection size by about one third.

Name	Maximal δ_T found
air01	34.5
cpsk	4.1
gb-coach	5.0
gb-train	5.8
montr	1.9
sncf	5.0
sncf-inter	3.0
sncf-ter	4.8
zsr	3.2

Table 4.6: Approximate density of the underlying graphs.

Name	Avg. degree	Max. degree
air01	16.3	166
cpsk	2.7	27
gb-coach	2.4	103
gb-train	3.3	30
montr	1.6	5
sncf	3.0	27
sncf-inter	2.5	12
sncf-ter	2.9	27
zsr	2.5	12

Table 4.7: Average and maximal degree in the underlying graphs.

From the table 4.8 we may see that this assumption holds.

		Connectivity		Strong connectivity	
Name	n	Connected	Largest comp.	Connected	Largest comp.
air01	287	✔	287	✗	286
cpsk	1905	✔	1905	✗	1903
gb-coach	2448	✗	2374	✗	2332
gb-train	2555	✔	2555	✔	2555
montr	217	✗	211	✗	209
sncf	2646	✔	2646	✗	2594
sncf-inter	366	✗	328	✗	316
sncf-ter	2637	✔	2637	✗	2583
zsr	233	✔	233	✗	225

Table 4.8: Connectivity of underlying graphs.

In the previous section (3) we have mentioned the highway dimension [AFGW10] as a parameter which, when being low, guarantees low query times for certain route-planning methods. Here we were interested in the highway dimension of our underlying graphs.

Definition 4.2. ***Highway dimension***
Highway dimension $\boldsymbol{HD(G)}$ *for a directed, edge-weighted graph* $G = (V, E)$ *is the smallest integer* h*, such that:*

$$\forall r \in R^+,\ \forall u \in V,\ \exists S \subseteq B_{u,2r},\ |S| \leq h,\ \forall v, w \in B_{u,2r}:$$
$$\textit{if } r < |P(v,w)| \leq 2r \textit{ and } P(v,w) \subseteq B_{u,2r} \textit{ then } P(v,w) \cap S \neq \emptyset$$

where:

- $\boldsymbol{P(v,w)}$ *is the* ***shortest path*** *between* v *and* w
- $\boldsymbol{B_{u,r}} = \{v \in V |\ |P(u,v)| \leq r$ *or* $|P(v,u)| \leq r\}$ *and is called* ***ball*** *of radius* r *centred at* u.

Intuitively, a graph has a low HD, if for any r we have a *sparse* set of vertices S_r, such that every shortest path longer then r includes a vertex from S_r. By the set being sparse, we mean that every ball of radius $\mathcal{O}(r)$ contains just a few elements of S_r.

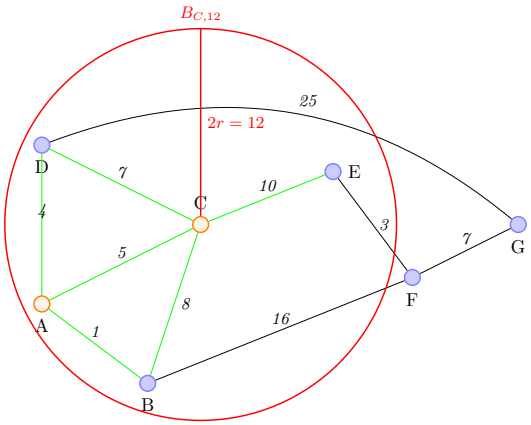

Figure 4.1: Demonstration of a definition of HD. We chose some r ($r = 6$) and some vertex v ($v = C$) to root the ball $B_{v,2r}$. All the shortest paths *longer* than r *inside* the ball have to contain a vertex from S (orange vertices C and A in our case). The upper bound on $|S|$, considering any ball with any radius, is the required highway dimension. Note: in our case, we had to choose also A to be part of S, since a shortest path from B to D does not include C.

Highway dimension is difficult to measure due to the way it is defined. We compute at least approximate value by randomly sampling balls of various radii and using greedy algorithm to compute shortest path covers. Results could be seen in table 4.9, where we also show the approximate highway dimension for a sub-graph with 2500 nodes of the Slovakia's road network. The results indicate that the highway dimension of the underlying graphs is comparable to that of the road network.

Name	$\boldsymbol{n}$	**apx.** $\boldsymbol{HD}$	**normalized** $\boldsymbol{HD}$
svk	2500	**53**	**0.021**
air01	287	49	0.171
cpsk	1905	30	0.016
gb-coach	2448	91	0.037
gb-train	2555	56	0.022
montr	217	18	0.083
sncf	2646	31	0.010
sncf-inter	366	13	0.036
sncf-ter	2637	36	0.014
zsr	233	9	0.039

Table 4.9: Highway dimension of 200 vertex sub-graphs of the underlying graphs. Normalized HD is the highway dimension divided by n.

We conclude this section with following observations about our timetables:

- The average daily number of events in a city (i.e. height) ranges from 20 to 130
- Time-dependent graphs are much less (more than 4 times in some cases) space-consuming then time-expanded graphs
- With exception of *air01*, the underlying graphs of our timetables are uniformly sparse ($\delta \leq \log n$)
- The average size of an optimal connection in all of our timetables is generally up to $\sqrt{n}$
- The average degree of underlying graphs is very small (less than 4), with the exception of *air01* which is much higher (more then 16)
- All the underlying graphs contain one large strongly connected component spanning almost the whole graph
- Highway dimensions of the underlying graphs are fairly low (again with exception of *air01*)

We will continue studying other properties of the timetables throughout the remaining of this thesis, especially in the section 5.

5 Underlying shortest paths

In section 2 we have defined a timetable as a set of elementary connections. While we do not pose any other restrictions on this set or on the elementary connections themselves, the real world timetables usually have a specific nature. Quite often are the connections repetitive, that is, the same sequence of elementary connections is repeated in several different moments throughout the day.

Another thing we may notice is that if we talk about *optimal* connections between a pair of distant cities u and v, we are often left with a few possibilities as to *which way should we go*. This is not only because the underlying graph is usually quite sparse [15], but also because for longer distances we generally need to make use of some express connection that stops only in (small number of) bigger cities.

Thus the main idea common to the methods presented in this section: *when carrying out an optimal connection between a pair of cities, one often goes along the same path regardless of the departure time* [16].

To formalize this idea, we will introduce the definition of an *underlying shortest path* - a path in the underlying graph that corresponds to some optimal connection in the timetable. To do this, we will first define a function *path* that extracts the **underlying path** (trajectory in the UG) from a given connection. Let c be a connection $c = (e_1, e_2, ..., e_k)$.

$$\boldsymbol{path(c)} = shrink(from(e_1), from(e_2), ..., from(e_k), to(e_k))$$

Note, that if the connection involves waiting in a city (as e.g. in figure 5.1), $e_x^i = e_x^{i+1}$ for some i. That is why we apply the *shrink* function, which replaces any sub-sequences of the type $(z, z, ..., z)$ by (z) in a sequence. This is a rather technical way of expressing a simple intuition - for a given connection, the *path* function outputs a sequence of visited cities. Now we can formalize the notion of underlying shortest path.

Definition 5.1. ***Underlying shortest path (USP)***
*A path $p = (v_1, v_2, ..., v_k)$ in UG_T is an **underlying shortest path** if, and only if $\exists t \in \mathcal{N} : p = path(c^*_{(v_1,t,v_k)}), c^*_{(v_1,t,v_k)} \in C_T$*

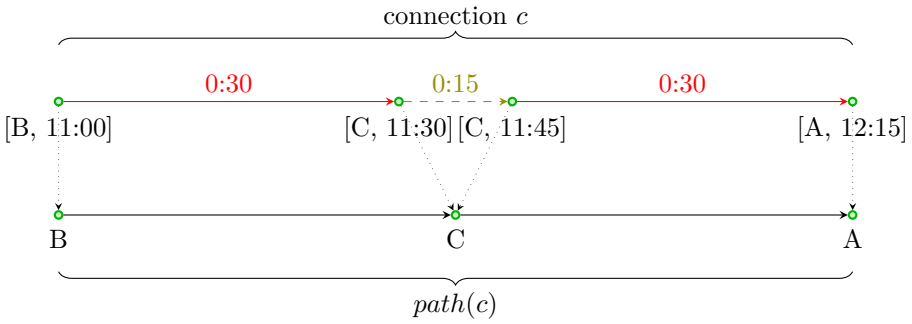

Figure 5.1: The *path* function applied on a connection to get the underlying path.

Please note that the terminology might be a bit misleading - an USP is not necessarily a shortest

[15] Maybe with exception of the airline timetables, which tend to be more dense.
[16] Or similarly, there are only few paths that are worth to follow.

path in the given UG. For example, the connections on a shortest path may require too much waiting and thus it might be that travelling along the paths with greater distance proofs to be faster.

5.1 *USP-OR*

We can easily extract the underlying path from a given connection. Now let us look at this from the other way - if, for a given EA query, we know the underlying shortest path, can we reconstruct the optimal connection? One thing we could do is to blindly follow the USP and in each city take the first elementary connection to the next one on the USP. This simple method called *expand* is described in algorithm 5.1.

Algorithm 5.1 *expand*

Input

- timetable T
- path $p = (v_1, v_2, ..., v_k)$, $v_i \in ct_T$
- departure time t

Algorithm

1: c = empty connection
2: $t' = t$
3: **for all** $i \in \{1, ..., k-1\}$ **do**
4: $\quad e = argmin_{e' \in C_T(v_i, v_{i+1})}\{dep(e') |\ dep(e') \geq t'\}$ *# take first available el. conn.*
5: $\quad t' = arr(e)$
6: $\quad c$.= e *# add the el.connection to the resulting connection*
7: **end for**

Output

- connection c

A question is - will we get an optimal connection if we expanded all possible USPs between a pair of cities? We show that we will, provided the timetable has no *overtaking* [DW09] of elementary connections.

Definition 5.2. ***Overtaking***

An elementary connection e_1 ***overtakes*** *e_2 if, and only if $dep(e_1) > dep(e_2)$ and $arr(e_1) < arr(e_2)$.*

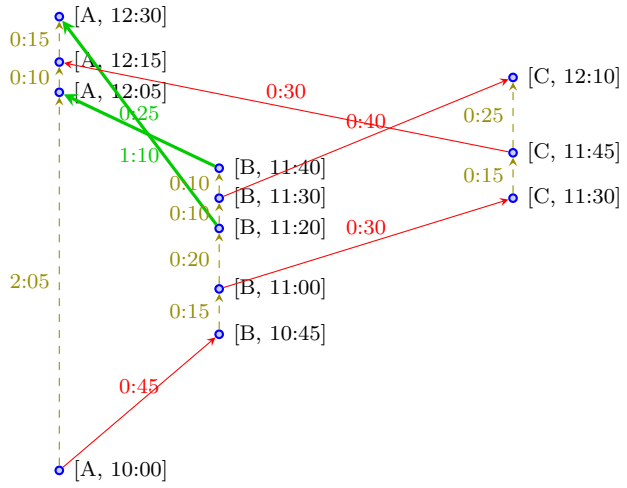

Figure 5.2: An example of **overtaking** (in thick), depicted in a TE graph.

Name	Overtaken
air01	1%
cpsk	2%
gb-coach	1%
gb-train	0%
montr	1%
sncf	1%
sncf-inter	6%
sncf-ter	1%
zsr	0%

Figure 5.3: Percentage of overtaken elementary connections in the timetables.

Lemma 5.1. *Let T be a timetable without overtaking, (x, t, y) an query in this timetable and $\boldsymbol{usps(x, y)} = \{p_1, p_2, ..., p_k\}$ a set of all USPs from x to y. Define c_i =expand(T, p_i, t) to be the connection returned by the algorithm expand (5.1). Then $\exists j : \ c_j = c^*_{x,t,y}$.*

Proof. The optimal connection $c^*_{x,t,y}$ has an USP p which must be present in the set $usps(x, y)$, as it is the set of all USPs from x to y. So $p = p_j = (v_1, v_2, ..., v_l)$ from some j. We want to show that c_j is the optimal connection. This may be shown inductively:

1. *Base: expand* reaches city $v_1 = x$ as soon as possible (since the connection just starts there).
2. *Induction: expand* reached city v_i as soon as possible, it then takes the first available elementary connection to the next city v_{i+1}. Since the elementary connections do not overtake, *expand* reached the city v_{i+1} as soon as possible.

□

We would like to stress that overtaking is understood as a situation when e.g. one train overtakes another between *two subsequent stations*. This situation is not that common, however it is still present in the real world timetables [17], as shown in table 5.3. All the same, we can simply remove the overtaken elementary connections from the timetables, as they can be substituted by the quicker connection plus some waiting, thus we will not change the earliest arrival time for any query.

The basic idea of the algorithm *USP-OR* (**USP or**acle) is therefore simply to pre-compute all the USPs for each pair of cities. Upon a query, the algorithm expands all the USPs for a given pair of cities, reconstructs respective connections and chooses the best one.

Algorithm 5.2 *USP-OR* query

Input
- timetable T
- query (x, t, y)

Pre-computed
- $\forall x, y : \ usps(x, y)$

Algorithm

```
1: c* = null
2: for all p ∈ usps(x, y) do
3:     c = expand(T, p, t)
4:     c* = better out of c* and c
5: end for
```

Output
- connection c

5.1.1 Analysis of *USP-OR*

We will now have a look at the four parameters of this oracle based method. As for the preprocessing time, we need to find the optimal connections from each *event* in the timetable to each *city* (or in other words - solve all possible OC queries). On these connections we apply the *path* function to obtain the USPs. There is hn events and one search from a single event to all cities can be done in time $\mathcal{O}(n \log n + m)$ with *TD Dijkstra* (in this section we use exclusively the time-dependent graphs).

[17] In Slovak rails, no overtaking has been detected. This is not surprising as (to my knowledge) there are no inter-station tracks with multiple rails going in one direction. French railways, on the other hand have designated high-speed tracks and thus overtaking is not impossible.

In worst case, m could be as much as n^2 but we may bound it as $m \leq \delta_T n$ (where δ_T is the density of the timetable, defined in section 4). We therefore get the **preprocessing time $\mathcal{O}(hn^2(\log n + \delta))$**.

As for the preprocessed space, we need to store USPs for each pair of the cities (n^2 pairs) and each USP might be long at most $\mathcal{O}(n)$ hops. What is more, there might be multiple USPs for a single pair of cities. Therefore we have two questions with respect to the space complexity of the preprocessing:

1. What is the average size of the USPs?
2. How many are there USPs between pairs of cities on average?

As for the first question, we will call the average size of USPs in a timetable T the **USP diameter** and denote it $\boldsymbol{\omega_T}$. This value is generally higher then the OC diameter [18], but can still be very well approximated by $\sqrt{n}$ (see table 5.1 and plot 5.4).

To answer the second question, we will introduce the following definition:

Definition 5.3. ***USP coefficient***
Given a timetable T and a pair of cities x, y, we define the USP coefficient $\boldsymbol{\tau_T(x,y)} = |usps_T(x,y)|$. By $\boldsymbol{\tau_T}$ we will denote the average USP coefficient in timetable T.

From the table 5.1 we may see that τ is quite small (≈ 10). Important thing however is whether or not it is constant with respect to:

- n - we found τ to be slightly increasing, sometimes almost constant (see plot 5.5)
- time range - again the value of τ was slightly increasing (plot 5.6)

Generally, we may bound the **size of the preprocessed oracle** as $\mathcal{O}(\tau n^2 \omega)$.

Name	$\boldsymbol{\tau}$	**max** $\boldsymbol{\tau(x,y)}$	$\boldsymbol{\omega}$
air01-200d	5.6	29	3.7
cpsk-200d	7.7	37	19.4
gb-coach-200d	3.5	29	7.2
gb-train-200d	6.8	40	10.3
montr-200d	2.7	18	26.1
sncf-200d	3.8	16	10.5
sncf-inter-200d	1.8	13	14.8
sncf-ter-200d	4.1	14	15.1
zsr-200d	2.3	13	16.2

Table 5.1: Average and maximal USP coefficients and USP diameter for daily timetables with 200 stations ($\sqrt{200} \approx 14$).

[18] If, for example, we have 8 optimal connections (on the same underlying path) with size 1 and 1 optimal connection with size 10, the OC diameter will be 2 but the average USP size will be 5.5.

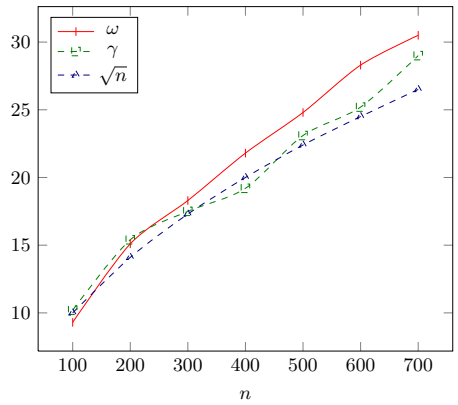

Figure 5.4: Changing of ω with increased number of stations in *cpsk* dataset. Compared to the OC diameter and $\sqrt{n}$.

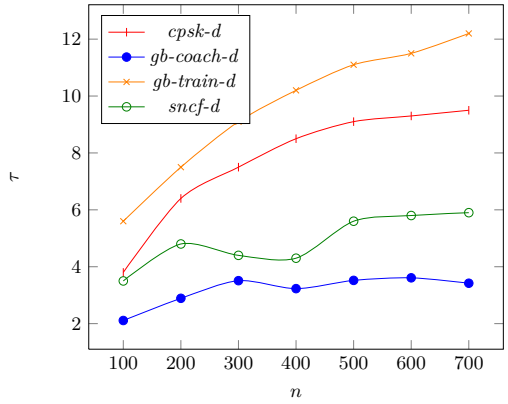

Figure 5.5: Changing of τ with increased number of stations.

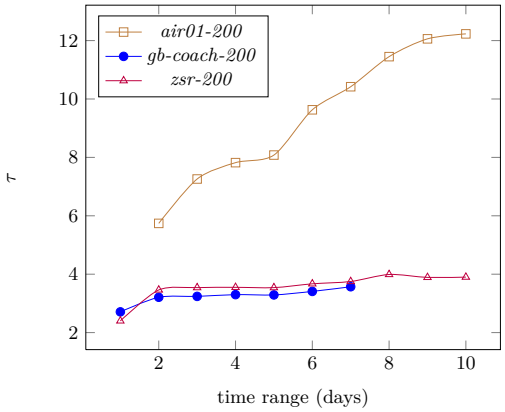

Figure 5.6: Changing of τ with increased time range.

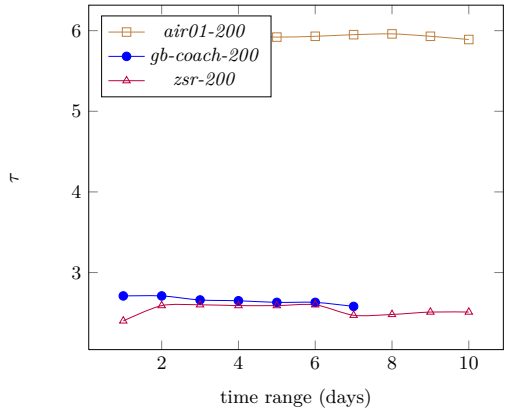

Figure 5.7: Changing of τ with increased time range when using segmentation.

Also the query time depends on the USP coefficient of a given pair of cities x, y, as we have to try out all USPs in $usps(x, y)$. The expansion of a USP by *expand* function takes time linear in the size of the USP [19], leading to **query time $\mathcal{O}(\tau\omega)$** on average. Note, that if τ was a constant this would be pretty much optimal, as we need to output the connection itself which takes linear time in its size.

To alleviate the problem of increased τ in timetables with e.g. weekly time range, we did a simple trick called **segmentation**. First, we normally computed the USPs. Then we segmented the timetable to individual days and for each of them we stored the pointers to necessary USPs. This does not require additional memory but it makes the value of τ constant, or even decreasing (see plot 5.7) with increasing time range. Note that this would be reflected only in an improved query time of *USP-OR*, the size of the preprocessed data will be left unaffected. From this point on we assume the use of segmentation for multi-day timetables (also in *USP-OR-A* algorithm, explained in the next section).

[19] In time-dependent graphs, this requires a constant-time retrieval of the correct interpolation point of the cost function (the piece-wise linear function that tells us the traversal time of an arc at a given time) for some time t. More specifically, we need to obtain an interpolation point $argmin_{(t',l)}\{t' | \ t' > t\}$. If we assume uniform distribution of departures throughout the time range of the timetable, this can be implemented in constant time. Otherwise, binary search lookup is possible in time $\mathcal{O}(\log h)$.

Finally, the **stretch** of *USP-OR* is **1**, since it returns exact answers.

USP-OR	***prep***	***size***	***qtime***	***stretch***
guaranteed	$\mathcal{O}(hn^2(\log n + \delta))$	$\mathcal{O}(\tau n^2 \omega)$	avg. $\mathcal{O}(\tau\omega)$	1
$\boldsymbol{\omega \leq \sqrt{n}}$, $\boldsymbol{\delta \leq \log n}$	$\mathcal{O}(hn^2 \log n)$	$\mathcal{O}(\tau n^{2.5})$	avg. $\mathcal{O}(\tau\sqrt{n})$	1

Table 5.2: The summary of the *USP-OR* algorithm parameters.

5.2 *USP-OR-A*

With *USP-OR* the main disadvantage is its space consumption. We may decrease this space complexity by pre-computing USPs only between *some* cities. The nodes that we select for this purpose will be called **access nodes** (AN for short), as for each city they would be the crucial nodes we need to pass in order to access most of the cities of T. It would be suitable for this access node set to have several desirable properties. In order to formulate them, we need to define a few terms first.

Definition 5.4. ***Front neighbourhood***
Given a timetable T and access node set $\mathcal{A}$, a front neighbourhood of city x is the set of all cities (including x) that are reachable from x without the need to pass a city from $\mathcal{A}$. Formally $\boldsymbol{neigh_{\mathcal{A}}(x)} = \{y \in ct_T | \exists \text{ path } p = (p_1, p_2, ..., p_k) \text{ from } x \text{ to } y \text{ in } ug_T : p_i \neq a \; \forall a \in \mathcal{A}, \; i \in \{2, ..., k-1\}\}$ [20]

We define analogically **back neighbourhood** (denoted $\boldsymbol{bneigh_{\mathcal{A}}(x)}$), as nodes that could be reached in UG with reversed orientation ($\overleftarrow{ug_T}$). Note that the access nodes that are on the boundary of x's neighbourhoods are also part of these neighbourhoods. These access nodes form some sort of separator between the x's neighbourhood and the rest of the graph and we will call them **local access nodes (LAN)** ($\boldsymbol{lan_{\mathcal{A}}(x)} = \mathcal{A} \cap neigh_{\mathcal{A}}(x)$), or analogically **back local access nodes** ($\boldsymbol{blan_{\mathcal{A}}(x)}$).

Now we may formulate the three desirable properties of the access node set. Given a timetable T, we would like to find access node set $\mathcal{A}$ with parameters r_1, r_2 and r_3 such that:

1. The access node set is sufficiently small

$$|\mathcal{A}| \leq r_1 \cdot \sqrt{n} \tag{5.1}$$

2. The average square of neighbourhood [21] size for cities not in $\mathcal{A}$ is at most $r_2 \cdot n$

$$\frac{\sum_{x \in ct_T \setminus \mathcal{A}} |neigh_{\mathcal{A}}(x)|^2}{|ct_T \setminus \mathcal{A}|} \leq r_2 \cdot n \tag{5.2}$$

3. The average square of the number of local access nodes [22] for cities not in $\mathcal{A}$ is at most r_3

$$\frac{\sum_{x \in ct_T \setminus \mathcal{A}} |lan_{\mathcal{A}}(x)|^2}{|ct_T \setminus \mathcal{A}|} \leq r_3 \tag{5.3}$$

[20] In $neigh_{\mathcal{A}}(x)$ we leave out subscript identifying the timetable T. In situation with clear context, we may also leave out the $\mathcal{A}$ subscript.
[21] We required the same for back neighbourhoods.
[22] We required the same for back LANs.

An access node set $\mathcal{A}$ with the above mentioned properties will be called (r_1, r_2, r_3) **access node set** (AN set). We will now explain how the *USP-OR-A* (**USP or**acle with **a**ccess nodes) algorithm works and return to its analysis later.

During preprocessing, we need to find a good AN set and compute the USPs between every pair of access nodes. For every city $x \notin \mathcal{A}$, we also store its $neigh_{\mathcal{A}}(x)$, $bneigh_{\mathcal{A}}(x)$, $lan_{\mathcal{A}}(x)$ and $blan_{\mathcal{A}}(x)$. On a query from x to y at time t, we will first make a local search in the neighbourhood of x up to x's local access nodes. Subsequently, we want to find out the earliest arrival times to each of y's *back* local access nodes. To do this, we take advantage of the pre-computed USPs between access nodes - try out all the pairs $u \in lan(x)$ and $v \in blan(y)$ and expand the stored USPs. Finally, we make a local search from each of y's back LANs to y, but we run the search *restricted* to y's back neighbourhood. For more details, see algorithms 5.3 and 5.4 and figure 5.8, where we have split the algorithm answering queries to 3 distinct phases.

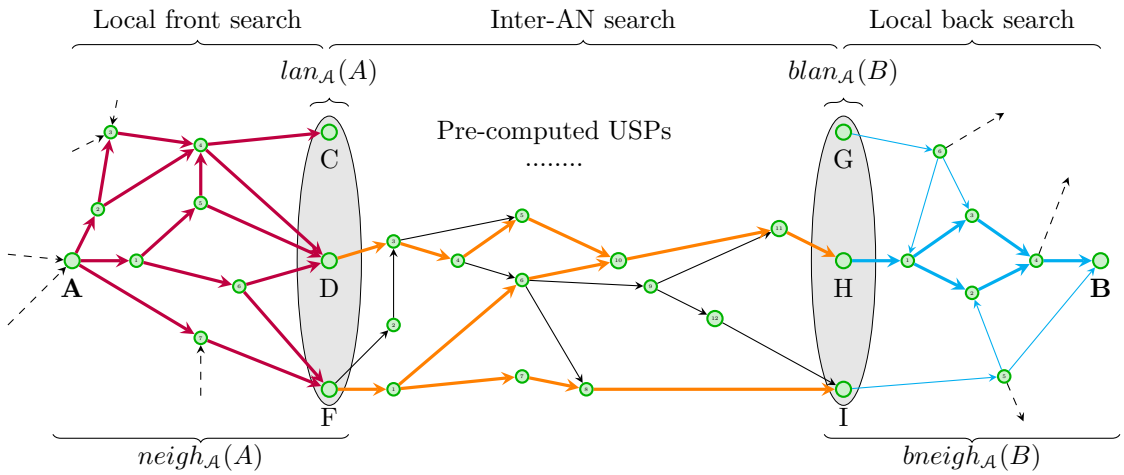

Figure 5.8: Principle of *USP-OR-A* algorithm. The arcs in **bold** mark areas that will be explored: all nodes in $neigh_{\mathcal{A}}(x)$, USPs between LANs of x and back LANs of y and the back neighbourhood of y (possibly only part of it will be explored, since the local back search goes against the direction of the back neighbourhood).

Algorithm 5.3 *USP-OR-A* preprocessing

Input

- timetable T

Algorithm

1: find a good AN set $\mathcal{A}$
2: $\forall x, y \in \mathcal{A}$ compute $usps(x, y)$
3: $\forall x \in ct_T \setminus \mathcal{A}$ compute $neigh_{\mathcal{A}}(x)$, $bneigh_{\mathcal{A}}(x)$, $lan_{\mathcal{A}}(x)$ and $blan_{\mathcal{A}}(x)$

Algorithm 5.4 *USP-OR-A* query

Input

- timetable T
- query (x, t, y)

Algorithm

1: let $lan(x) = x$ if $x \in \mathcal{A}$
2: let $blan(y) = y$ if $y \in \mathcal{A}$

Local front search
perform *TD Dijkstra* from x at time t up to $lan(x)$
if $y \in neigh(x)$ **then**
  let c^*_{loc} be the connection to y obtained by *TD Dijkstra* # *the optimal connection may still go via ANs (though it is unlikely)*
end if
$\forall u \in lan(x)$ let $ea(u)$ be the arrival time and $oc(u)$ the connection to u obtained by *TD Dijkstra*
Inter-AN search
for all $v \in blan(y)$ **do**
  $oc(v) = null$
  for all $u \in lan(x)$ **do**
    for all $p \in usps(u, v)$ **do**
      $c = expand(T, p, ea(u))$
      $oc(v) =$ better out of $oc(v)$ and c
    end for
  end for
end for
$\forall v \in blan(y)$ let $ea(v) = arr(oc(v))$
Local back search
for all $v \in blan(y)$ **do**
  perform *TD Dijkstra* from v at time $ea(v)$ to y restricted to $bneigh(y)$
  let $fin(v)$ be the connection returned by *TD Dijkstra*
end for
$v^* = argmin_{v \in blan(y)}\{arr(fin(v))\}$
$u^* = from(oc(v^*))$
let $c^* = oc(u^*).oc(v^*).fin(v^*)$ # *the dot (.) symbol is concatenation of connections*
output better out of c^*_{loc} and c^*

Output

- optimal connection $c^*_{(x,t,y)}$

5.2.1 Analysis of *USP-OR-A*

Let us now analyse the properties of this oracle-based method. Clearly, much depends on the way we look for the access node set. We will address this issue in next subsections but for now, we will assume we can find (r_1, r_2, r_3) AN set $\mathcal{A}$ in time $f(n)$. Then, in the preprocessing, we have to find USPs among the access nodes, which requires running Dijkstra's algorithm from each event in an access node (city from $\mathcal{A}$). There is $\mathcal{O}(r_1 h\sqrt{n})$ such events which leads to the time complexity $\mathcal{O}(r_1 h n^{1.5}(\log n + \delta))$. We also have to find local access nodes and neighbourhoods for each city, which can be accomplished with e.g. depth first search exploring the neighbourhood. This search algorithm (run from non-access city) has complexity linear in the number of arcs and so we could bound the total complexity as:

$$\sum_{x \in ct_T \setminus \mathcal{A}} |E(neigh_{\mathcal{A}}(x))| \leq \sum_{x \in ct_T \setminus \mathcal{A}} |neigh_{\mathcal{A}}(x)|^2 \leq r_2 n^2$$

where $E(V)$ is the set of arcs among vertices of V. However this is very loose upper bound, as our UGs are actually very sparse. Therefore we can improve it. We know from the equation 5.2 that the average square of neighbourhood size is at most $r_2 \cdot n$. As a consequence of the Cauchy-Schwarz Inequality [ops] the following holds for positive real numbers x_i:

$$\sqrt{\frac{x_1^2 + x_2^2 + ... + x_n^2}{n}} \geq \frac{x_1 + x_2 + ... + x_n}{n}$$

Applying this to our neighbourhood sizes, we get that the average size of the neighbourhood is at most $\sqrt{r_2 n}$. We now split the vertices of $ct_T \setminus \mathcal{A}$ to two categories: those with neighbourhoods of size at most $\sqrt[4]{n}$ will be part of the set $S_{\leq}$ and those with neighbourhoods of size bigger then $\sqrt[4]{n}$ will be in $S_{>}$. A neighbourhood in the first category cannot possibly contain more than $\sqrt{n}$ arcs while those in the second category can have at most $\delta_T |neigh_{\mathcal{A}}(x)|$ arcs (thus depending on the timetable's density).

$$\sum_{x \in ct_T \setminus \mathcal{A}} |E(neigh_{\mathcal{A}}(x))| \leq$$
$$\sum_{x \in S_{\leq}} \overbrace{|E(neigh_{\mathcal{A}}(x))|}^{\leq \sqrt{n}} + \sum_{x \in S_{>}} \overbrace{|E(neigh_{\mathcal{A}}(x))|}^{\leq \delta |neigh_{\mathcal{A}}(x)|} \leq$$
$$n\sqrt{n} + \delta n \sqrt{r_2 n} \leq$$
$$\delta r_2 n^{1.5}$$

Therefore, the total **time complexity of the preprocessing** is $\mathcal{O}(f(n) + r_1 h n^{1.5}(\log n + \delta)) + \mathcal{O}(\delta r_2 n^{1.5}) = \boldsymbol{\mathcal{O}(f(n) + (r_1 + r_2)(\delta + \log n) h n^{1.5})}$.

As for the size of the preprocessed data - we need to store all the neighbourhoods, LANs and USPs between access nodes. We already know that the average size of the neighbourhood is at most $\sqrt{r_2 n}$, thus the total size of the (front and back) neighbourhoods is $\mathcal{O}(r_2 n^{1.5})$ [23]. This term bounds also the size of the pre-computed local access nodes for each node.

Finally we have the preprocessed USPs. There is at most $r_1^2 n$ pairs of access nodes and for each of them we have possibly several USPs. We will denote by $\boldsymbol{\tau_{\mathcal{A}}}$ the average USP coefficient between pairs of cities from $\mathcal{A}$ and by $\boldsymbol{\omega_{\mathcal{A}}}$ the average USP size between cities in $\mathcal{A}$. This amounts to $\mathcal{O}(r_1^2 \tau_{\mathcal{A}} \omega_{\mathcal{A}} n)$ for storage of USPs and to a total **preprocessing size $\boldsymbol{\mathcal{O}(r_2 n^{1.5} + r_1^2 \tau_{\mathcal{A}} \omega_{\mathcal{A}} n)}$**.

Name	n	$\tau_{\mathcal{A}}$	$\omega_{\mathcal{A}}$	$\sqrt{n}$
air01-d	284	10.4	3.4	16.9
cpsk-d	1905	15.9	42.6	43.6
gb-coach-d	2427	6.3	20.2	49.3
gb-train-d	2550	21.8	23.0	50.5
montr-d	217	4.0	24.3	14.7
sncf-d	2608	9.8	34.0	51.1
sncf-inter-d	344	2.8	12.2	18.5
sncf-ter-d	2600	8.9	45.5	51.1
zsr-d	225	4.7	16.5	15

Table 5.3: USP coefficient and diameter for access node sets, daily timetables.

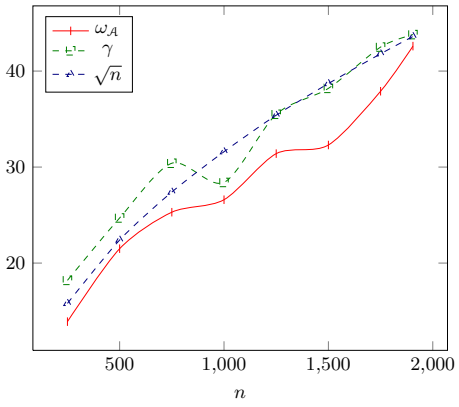

Figure 5.9: Changing of $\omega_{\mathcal{A}}$ with increased number of stations in *cpsk* dataset. Compared to the OC diameter and $\sqrt{n}$.

[23] As r_2 should be a very small constant, we may disregard the square root.

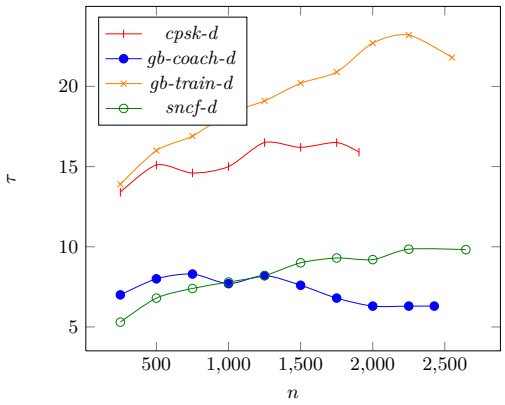

Figure 5.10: Changing of $\tau_{\mathcal{A}}$ with increased number of stations.

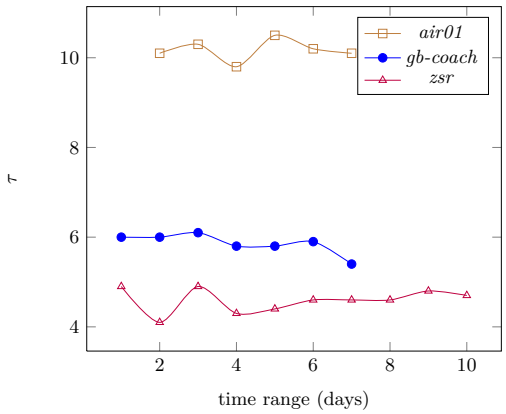

Figure 5.11: Changing of $\tau_{\mathcal{A}}$ with increased time range (using segmentation).

On a query from x at time t to y, we first perform the *local front search* (see algorithm 5.4). In this step we explore the neighbourhood of x with a time-dependent Dijkstra's algorithm, which takes on average time $\mathcal{O}(\sqrt{r_2 n}(\log(\sqrt{r_2 n}) + \delta))$. We then expand all the USPs between u and v such that $u \in lan(x)$ and $v \in blan(y)$, which takes on average $\mathcal{O}(r_3 \tau_{\mathcal{A}} \omega_{\mathcal{A}})$. Finally, from each $v \in blan(y)$ we do a *TD Dijkstra*, restricted to $bneigh(y)$, leading to time complexity $\mathcal{O}(\sqrt{r_3 r_2 n}(\log(\sqrt{r_2 n}) + \delta))$.

Summing up the three terms we obtain the **query time** of $\boldsymbol{\mathcal{O}(r_2\sqrt{r_3}\sqrt{n}(\log(r_2 n) + \delta) + r_3 \tau_{\mathcal{A}} \omega_{\mathcal{A}})}$.

Stretch of the *USP-OR-A* algorithm is **1**, as it is exact algorithm.

The resulting bounds do not look very appealing. This is because we wanted to preserve the generality - the concrete bounds will depend on what kind of properties the timetables have and what algorithm for finding the AN set is plugged in. In table 5.4, we summarize the parameters of *USP-OR-A* method and provide the bounds for a case when the properties of the timetables correspond to those we have measured in our datasets and when we have an algorithm that finds good AN set.

USP-OR-A	**guaranteed**	$\boldsymbol{r_1, r_2, r_3 = \mathcal{O}(1)}$, $\boldsymbol{\omega \leq \sqrt{n}}$, $\boldsymbol{\delta \leq \log n}$
prep	$\mathcal{O}(f(n) + (r_1 + r_2)(\delta + \log n) h n^{1.5})$	$\mathcal{O}(f(n) + h n^{1.5} \log n)$
size	$\mathcal{O}(r_2 n^{1.5} + r_1^2 \tau \omega n)$	$\mathcal{O}(\tau n^{1.5})$
qtime	avg. $\mathcal{O}(r_2\sqrt{r_3}\sqrt{n}(\log(r_2 n) + \delta) + r_3 \tau \omega)$	avg. $\mathcal{O}(\tau \sqrt{n} \log n)$
stretch	1	1

Table 5.4: The summary of the *USP-OR-A* algorithm parameters. Here we left out subscripts identifying AN set for τ and ω.

5.2.2 Correctness of *USP-OR-A*

Finally, we will proof the correctness of the algorithm, i.e. that it always returns the optimal connection.

Theorem 5.1. *The algorithm USP-OR-A (5.3, 5.4) always returns the optimal connection.*

Proof. Let $\mathcal{A}$ be the set of access nodes and consider a query from city x to city y at any time t. If $x \in \mathcal{A}$ and $y \in \mathcal{A}$, an optimum is returned due to lemma 5.1 (in such a case, we basically run *USP-OR* algorithm).

In the following we will assume that $\boldsymbol{y \notin neigh(x)}$, which means that the optimal connection goes through some access node $u \in lan(x)$ and $v \in blan(y)$. Note that it may be that $u = v$.

What we would like to prove as a next step is that we reach the back LANs of y (or y itself if it is an access node) at the earliest arrival time. After the *local front search*, we have reached the x's local ANs at times $ea(u)$ $\forall u \in lan(x)$. For some local access node this value is the true earliest arrival. Let us denote the set of such local ANs as $lan^*(x)$. The crucial thing to realize is that the optimal connection to any city out of the x's neighbourhood will lead via some $u \in lan^*(x)$ (see figure 5.12). And because the *inter-AN search* phase finds *optimal* connections between pairs $u \in lan(x)$ and $v \in blan(y)$, it follows that for each $v \in blan(y)$ the $ea(v)$ is the earliest arrival to this city after the *inter-AN search* phase.

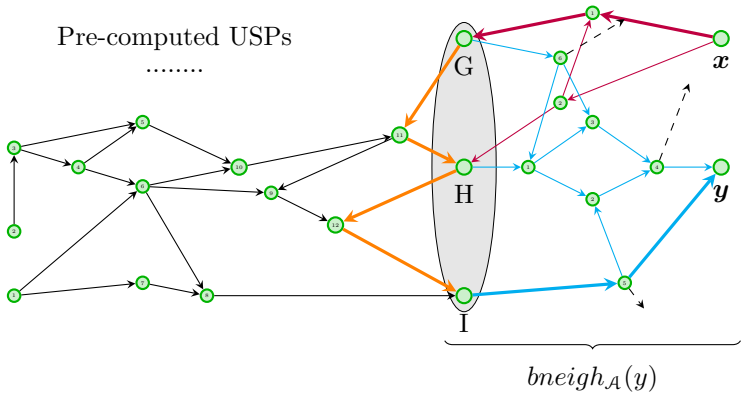

Figure 5.12: On the picture $lan(x) = \{G, H\}$ and $blan(y) = \{G, H, I\}$. In **thick** we have highlighted the optimal connection. The connection to H is sub-optimal after the *local front search* phase, however the optimal connection to y (and to H and I as well) leads through $lan^*(x)$ (some of x's local access nodes to which we have an optimal connection after the *local front search*. Particularly, in this case it goes through G).

In the *local back search* we run a *TD Dijkstra* search from all back LANs of y. And since this algorithm is exact and starts from each back LAN as early as possible, we get the optimal connection to y.

It remains to show that if $\boldsymbol{y \in neigh(x)}$, we also get the optimal connection. In such case, we simply compare the connection that goes via access nodes and the one that was obtained solely within the neighbourhood and output the shorter one. As there are no other options, the proof is complete. □

5.2.3 Modifications of *USP-OR-A*

Our implementation of the *USP-OR-A* algorithm uses one slight improvement, which we did not mention in its description, since it is more of an optimization technique without any theoretical guarantees on actual improvement of the running time. However, we consider it an interesting idea so we mention it at this place.

Definition 5.5. ***USP tree***
Given a pair of cities x and y in a timetable T, we will call a USP tree the graph made out of edges of all USPs in $usp_T(x, y)$: $\boldsymbol{usp^3_T(x, y)} = (V^3, E^3)$ *where* $V^3 = \{v |\ v$ *lays on some* $p \in usp_T(x, y)\}$ *and* $E^3 = \{(a, b) |\ (a, b)$ *is part of some* $p \in usp_T(x, y)\}$.

We could take advantage of these USP trees to speed up the *local front search* phase of the algorithm, where we unnecessarily explore the whole neighbourhood when we could just go along the arcs of the USP trees. The figure 5.13 depicts this.

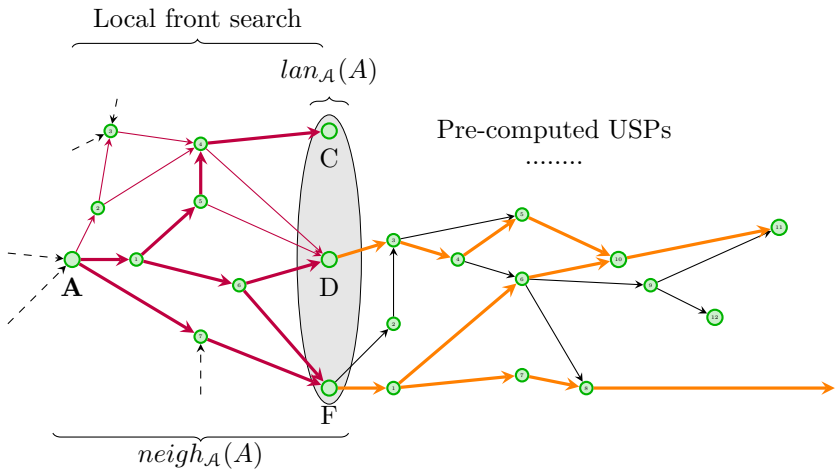

Figure 5.13: Using USP trees (**thick** non-dashed arcs in $\boldsymbol{neigh_{\mathcal{A}}(x)}$) to decrease the explored area in *local front search*. A full neighbourhood search is done only when $y \in neigh(x)$.

The interesting thing about this is the exploitation of both - timetable and its underlying graph. While the neighbourhood of a node is something static, related only to the structure of the UG and generally time-independent, the USP trees reflect to some extent the properties of the timetable (e.g. which ways are frequently serviced and thus provide optimal connections). By intersecting these two things, we get the area that is *worth* to be explored and that is *small* at the same time (provided, of course, that the neighbourhoods are small).

5.3 Selection of access node set

The challenge in the *USP-OR-A* algorithm comes down to the selection of a good access node set - a (r_1, r_2, r_3) AN set with both three parameters as low as possible. However, intuitively (and experimentally verified), decreasing e.g. r_1 (the AN set size) increases r_2 (the size of the neighbourhoods). We therefore have to do some compromises.

In the following we first show the problem of choosing an optimal access node set to be NP-hard. We then present our methods for heuristic selection of access nodes and show their performance on real data.

5.3.1 Choosing the optimal access node set

A question stands - what is an optimal access node set? To keep the query time as low as possible, we need to avoid large neighbourhood sizes, because that would mean spending too much time doing local searches. A pretty good upper bound for neighbourhood sizes seems to be $\sqrt{n}$ (i.e. $r_1 = 1$) - the idea is that in such case the local searches cannot possibly last longer then $\mathcal{O}(n)$ while the *inter-AN search* is linear in the size of the connection and can also be at most $\mathcal{O}(n)$. In practice, both of these steps will be faster because the neighbourhoods are sparse and because the connections

are on average much shorter then n. However, it gives an idea of why $\sqrt{n}$ should be considered for a target neighbourhood size.

Therefore, the question stands: What is the smallest set of ANs, such that the neighbourhood sizes are all under $\sqrt{n}$? More formally, for a timetable T, the task is to minimize $|\mathcal{A}|$ where $\mathcal{A} \subseteq ct_T$ and $\forall x \in ct_T \setminus \mathcal{A} : |neigh_{\mathcal{A}}(x)| \leq \sqrt{n}$. We will call this the **problem of the optimal access node set** and in what follows we will show that it is NP-complete.

Theorem 5.2. *The problem of the optimal access node set is NP-complete*

Proof. We will make a reduction of the *min-set cover* problem (a NP-complete problem) to the problem of optimal AN set.

Consider an instance of the min-set cover problem:

- A universe $U = \{1, 2, ..., m\}$
- k subsets of U: $S_i \subseteq U$ $i = \{1, 2, ..., k\}$ whose union is U: $\bigcup_{1 \leq i \leq k} S_i = U$

Denote $\mathcal{S} = \{S_i | \ 1 \leq i \leq k\}$. The task is to choose the smallest subset $\mathcal{S}^*$ of $\mathcal{S}$ that still covers the universe ($\bigcup_{S_i \in \mathcal{S}^*} S_i = U$). We will now do a simple conversion (in polynomial time) of the instance of min-set cover to the instance of the optimal AN set problem (which is represented by the underlying graph of T).

For each $j \in U$, we will make a complete graph of β_j vertices (the value of β_j will be discussed later) named m_j and for each set S_i we make a vertex s_i and vertex s'_i. We now connect all vertices of m_j to s_i for each $j \in S_i$. Finally, for we connect s_i to s'_i, $1 \leq i \leq k$.

Example. Let $m = 10$ (thus $U = \{1, 2, ..., 10\}$) and $k = 13$:

- $S_1 = \{1, 3, 10\}$
- $S_2 = \{1, 2\}$
- ...
- $S_{13} = \{2, 3, 10\}$

For this instance of min set-cover, we construct the graph as depicted on figure 5.14.

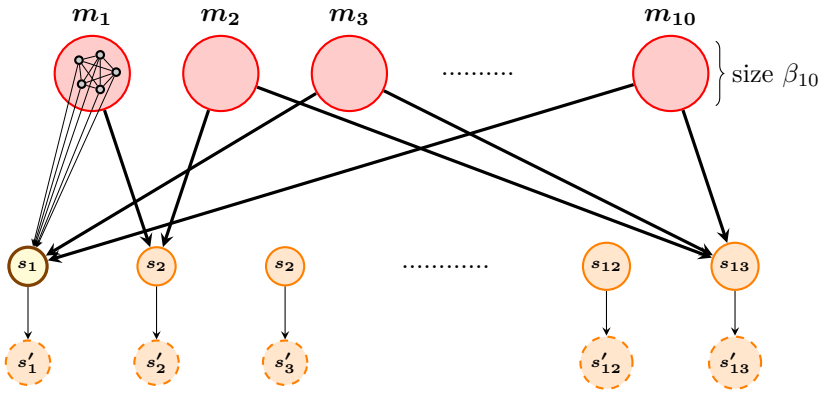

Figure 5.14: The principle of the reduction. In m_i, there are actually complete graphs of β_i vertices (as shown for m_1). **Thick** arcs represent arcs from all the vertices of respective m_i. The s_i vertices are connected to their s'_i versions. If e.g. s_1 is selected as an access node, s'_1 is no longer part of any neighbourhood (except for its own).

Now we would like to clarify the sizes of m_i. Define α_i to be the number of sets S_j that contain i: $\alpha_i = |\{S_j \in \mathcal{S} |\ i \in S_j\}|$ and assume the constructed graph has n vertices. We want the β_i to satisfy $\beta_i \geq 2$ and $\beta_i + 2\alpha_i - 1 \leq \sqrt{n}$ but $\beta_i + 2\alpha_i > \sqrt{n}$. The last two inequalities would mean that if at least one s_j connected to m_i is chosen as an access node, the neighbourhood for nodes in m_i will be still large at most $\sqrt{n}$, but if none of them is chosen, the neighbourhood size will be just over $\sqrt{n}$. We leave out the details of the construction at this place.

Now consider an optimal AN set which contains a vertex from within some m_i. If this is the case, **either** some s_j to which m_i is connected is selected as AN, **or** *all* vertices from m_i are access nodes **or** the neighbourhood is too large. Keep in mind that the local access nodes are also part of neighbourhoods, so unless we select for AN some of the s_j that m_i is connected to, the neighbourhood of any non-access node in m_i will be too large. As there are at least two nodes in every m_i, it is more efficient to select some s_j rather then select all nodes in m_i. Thus when it comes to selecting ANs *it is worth to consider only vertices* s_j.

From this point on, it is easy to see that it is optimal to select those s_j that correspond to the optimal solution of min-set cover. The reason is that each of the m_i will be connected to at least one access node s_j and will thus have neighbourhood size at most $\sqrt{n}$, while the number of selected access nodes will be optimal.

It remains to show how to choose values β_i. Due to the condition $\beta_i \leq \sqrt{n} - 2\alpha_i + 1$ we need to have sufficiently big n to fulfil $\beta_i \geq 1$. We will accomplish this by adding dummy isolated vertices to the graph. Define function *nextSquare(x)* to output the smallest $y^2 > x$ where y is a natural number. We then compute $w = (max\{2\alpha_i\} + 2)^2$ and select the starting value of n to be $n' =$ *nextSquare(*$max\{w - 1,\ \sqrt{2k + m}\}$*)*. We create the s_j and s'_j vertices and the complete graphs m_i containing so far only one vertex each. We connect everything according to the rules stated earlier in this proof and we create dummy vertices up to the capacity defined by n. Now we repeat the following:

- We compute $\sqrt{n}$ which is a natural number
- For i from 1 to m we add vertices to m_i till it does not contain $\sqrt{n} - 2\alpha_i + 1$ vertices. For each added vertex we delete one dummy vertex
- If we run out of dummy vertices, $n =$ *nextSquare(n)*
- Break out of the loop if $|m_i| = \sqrt{n} - 2\alpha_i + 1\ \forall i$

With each iteration of this little algorithm we will be forced to add one more vertex to all m_i (since $\sqrt{n}$ increased by one), a so called *inefficient increase*. At the beginning, we need to make at most $m\sqrt{n'}$ efficient increases to meet the breaking condition. And since m is constant and the capacity of new dummy vertices increases linearly, after t steps we create $\mathcal{O}(t^2)$ dummy vertices that may be used for efficient increases. Therefore, the algorithm will stop after $\mathcal{O}(\sqrt{mn'})$ steps. □

5.3.2 Choosing ANs based on node properties

In the previous sub-subsection, we have shown the problem of choosing the optimal AN set to be NP-hard. In this sub-subsection we perform a simple experiment of choosing for the access nodes the cities that seem to be the most important. More specifically, in the optimistic underlying graph (see section 2) ug_T^{opt} we were looking for cities with:

1. High **degree**. We consider the sum of in-degree and out-degree [24] of the respective node x: $\boldsymbol{deg(x)} = deg_{in}(x) + deg_{out}(x)$.

[24] In-degree is the number of arcs going into the node and out-degree the number of outgoing arcs.

2. High **betweenness centrality** (BC). Betweenness centrality for a node v is defined as

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where $\sigma_{st}(v)$ is the number of shortest paths from s to t passing through v and σ_{st} is the total number of shortest paths from s to t [Bra01]. We then scale the values to the range $< 0, 1 >$ to obtain for each city x its scaled betweenness centrality $\boldsymbol{bc(x)}$.

We will denote by $\mathcal{A}^{deg}(k)$ the set of k cities with highest $deg(x)$ value. We were interested in the smallest k such that $\mathcal{A}^{deg}(k)$ is (r_1, r_2, r_3) AN set with $r_2 \leq 1$ (the average square of neighbourhoods is at most n). Denote such set as $\boldsymbol{\mathcal{A}^{deg}}$ and the triplet (r_1, r_2, r_3) as $\boldsymbol{(r_1^{deg}, r_2^{deg}, r_3^{deg})}$ [25].

We define similarly $\boldsymbol{\mathcal{A}^{bc}}$ and $\boldsymbol{r_i^{bc}}$. Plots 5.15 summarize the properties of access node sets obtained this way on daily datasets *sncf* and *cpsk*.

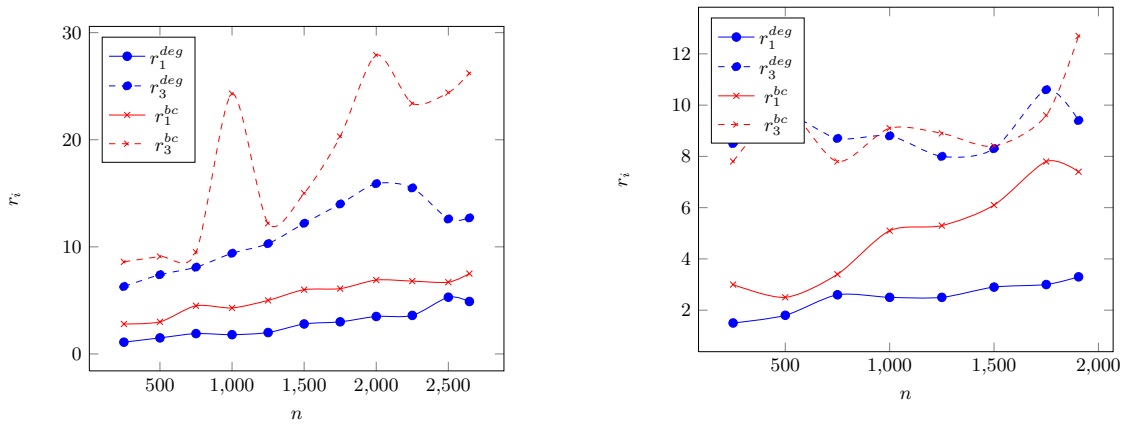

Figure 5.15: Parameters of the access node sets $\mathcal{A}^{deg}$ and $\mathcal{A}^{bc}$ with increasing n. Datasets *sncf* (left) and *cpsk* (right). $r_2 \leq 1$. The occasional "roller coaster" bumps (for value of r_3) are due to our stopping criterion which does not consider r_3 at all.

5.3.3 Choosing ANs heuristically - the *locsep* algorithm

Clearly, selecting the cities for access nodes solely by high degree or BC value is not the best way. Probably the few nodes with highest degrees and BC will indeed be part of the AN set, as they are intuitively some sort of central hubs without which the network would not work. However, after we select these most important nodes to the AN set, we need some better measure of node's importance, or suitability to be an access node. In the following we present a simple heuristic approach run on the underlying graph ug_T of a given timetable T that evaluates its vertices based on how good local separators they are.

The algorithm that we call *locsep* (as it looks for good local separators) will work in iterations, each of them resulting in a selection of the city with the highest score to the access node set $\mathcal{A}$ [26]. We continue to select access nodes until we meet the following stopping criterion: $\mathcal{A}$ is

[25] Intuitively - r_1^{deg} is the smallest r_1 such that $r_1\sqrt{n}$ highest-degree cities selected as ANs are enough to satisfy that the average square of neighbourhoods is at most n.

[26] Actually, in our implementation, we allow an occasional deselection of an already selected node with the *lowest* score, to avoid having in the resulting set cities that had high score when selected but were not very useful access nodes at the end.

(r_1, r_2, r_3) AN set with $r_2 \leq 1$ (the average square of neighbourhoods is at most n) [27]. We will denote the resulting set $\boldsymbol{\mathcal{A}^{loc}}$ and its parameters as $\boldsymbol{r_i^{loc}}$.

The important thing that remains to be shown is how we compute the score for a particular city. The following text explains this.

In each iteration, we first compute the neighbourhoods and back neighbourhoods (given the current access node set $\mathcal{A}$) for each city. We need this to evaluate the stopping criteria, but the information is also used in the computation of the **potential** (the score) of the cities.

For a city x, we compute its potential $\boldsymbol{p_x}$ in the following way: we explore an area $\boldsymbol{A_x}$ of $\sqrt{n}$ nearest cities around x, ignoring branches of the search that start with an access node (x is an exception to this, since we start the search from it, although $x \notin A_x$ holds). We do this exploration in an underlying graph with no orientation and no weights. Next we get the front and back neighbourhoods of x within A_x ($\boldsymbol{fn(x)} = neigh(x) \cap A_x$, $\boldsymbol{bn(x)} = bneigh(x) \cap A_x$).

For a set of access nodes $\mathcal{A}$, let us call a path p in ug_T **access-free** if it does not contain a node from $\mathcal{A}$. Now as long as x is not in $\mathcal{A}$, we have a guarantee that for every pair $u \in bn(x)$ and $v \in fn(x)$ there is an access-free path from u to v within A_x. Our interest is how this will change after the selection of x.

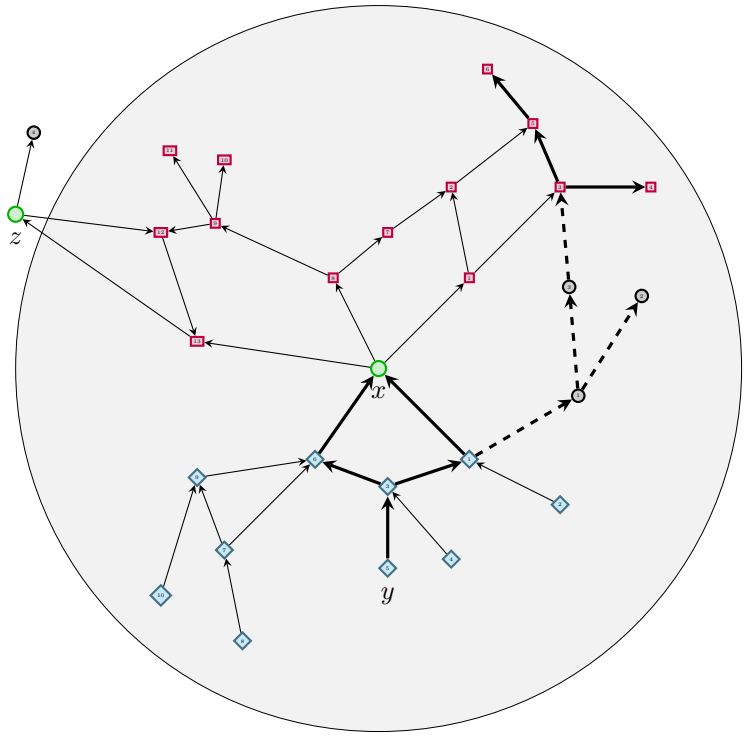

Figure 5.16: The principle of computing potentials in *locsep* algorithm. We explored an area of $\sqrt{n}$ nearest cities (in terms of hops) around x. Access nodes (like z) and cities behind them are ignored. Little squares are nodes from $fn(x)$ and diamonds are part of $bn(x)$. From y we run a forward search (the **thick** arcs). Nodes from the $fn(x)$ that were not explored in this search can only be reached via x itself. Such nodes contribute to x's potential assuming y has large neighbourhood size that needs to be decreased.

[27] In our implementation, we perform some further adjustments of the resulting set, such as removing unnecessary access nodes and optimising for the r_3 value.

Consider now a node $y \in bn(x)$. We will call $\boldsymbol{sur(y)} = \max\{0, |neigh(y)| - \sqrt{n}\}$ the **surplus** of y's neighbourhood, i.e., by how much we wish to reduce it so that it is at most $\sqrt{n}$. If the surplus is zero, y will not add anything to the x's potential. Otherwise, we run a restricted (to A_x) search from y during which we explore j vertices in $fn(x)$. We increase the potential of x by $\min\{sur(y), |fn(x) - j|\}$ - i.e. by how much we can decrease the surplus of y's neighbourhood if we select x. We do the same for all $y \in bn(x)$ and a similar thing for all $y \in fn(x)$ (we use $\overleftarrow{ug_T}$ instead of ug_T, $bneigh(y)$ instead of $neigh(y)$ etc...). For an illustration of potential computing, see figure 5.16.

Finally, we simply get the city $x \notin \mathcal{A}$ with the highest potential and select it as an access node. We check the stopping criterion and in case it is not satisfied yet, we move on to next iteration. However, note that when a new node x' is selected to $\mathcal{A}$, we do not have to re-compute neighbourhoods and potentials of all cities - it is only necessary for those cities that could reach/be reached access-free from x' (i.e. nodes from $neigh_{\mathcal{A}}(x') \cup bneigh_{\mathcal{A}}(x')$). Algorithm 5.5 provides a high-level overview of the locsep method.

Algorithm 5.5 *locsep*

Input

- ug_T

Algorithm

1: $\mathcal{A} = \emptyset$
2: $ct' = ct_T$
3: **while** $r_2 > 1$ **do**
4: $\quad \forall x \in ct'$: compute $neigh_{\mathcal{A}}(x)$, $bneigh_{\mathcal{A}}(x)$
5: $\quad \forall x \in ct'$: compute p_x
6: $\quad x' = argmax_{x \notin \mathcal{A}}\{p_x\}$
7: $\quad \mathcal{A} = \mathcal{A} \cup \{x'\}$
8: $\quad ct' = neigh_{\mathcal{A}}(x') \cup bneigh_{\mathcal{A}}(x')$
9: **end while**
10: Remove unnecessary ANs
11: Optimise r_3

Output

- AN set $\mathcal{A}^{loc}$ ($|\mathcal{A}^{loc}| = \sqrt{n} r_1^{loc}$)

Now we would like to estimate the **time complexity** of *locsep* algorithm. As mentioned, one iteration consists of three parts:

1. Computing neighbourhoods. Unfortunately, at the beginning when $\mathcal{A} = \emptyset$, the neighbourhood sizes may be as large as $\mathcal{O}(n)$. Therefore, we may bound the complexity of this phase only as $\mathcal{O}(nm) = \mathcal{O}(\delta n^2)$
2. Computing potentials. For a city x we explore area of the size $\sqrt{n}$ and from each node in that area we do a restricted search. Therefore the total complexity of this step is $\mathcal{O}(n \cdot \sqrt{n} \cdot \delta\sqrt{n}) = \mathcal{O}(\delta n^2)$
3. Selecting the node with the highest potential. This can be done in $\mathcal{O}(n)$

Adding up the individual terms, we get the complexity of one iteration to be at most $\mathcal{O}(\delta n^2)$. As we aim for the resulting access node set of size $\mathcal{O}(\sqrt{n})$, we would get the total running time of $\boldsymbol{\mathcal{O}(\delta n^{2.5})}$. However, we remind that the algorithm is only a heuristics with no guarantees on the

resulting access node set size [28].

The resulting running time is still quite impractical for bigger timetables. For example, the computation on the dataset *sncf* took more than an hour. This is due to the initial iterations, during which average neighbourhood is still very large (spanning almost the whole graph) and thus we have to do a lot of re-computations (potentials, neighbourhoods). We therefore embrace a simple trick: we do not start with $\mathcal{A} = \emptyset$ but with some access nodes already selected based on high degree. We chose to start with $\frac{2\sqrt{n}}{3}$ nodes with the highest degree (i.e. with the set $\mathcal{A}_{deg}(\frac{2\sqrt{n}}{3})$) - enough to speed-up the computation but not influencing the resulting AN set too much.

The access node sets chosen with the *locsep* algorithm had much better properties compared to those selected by the previous approaches. The summary of their properties for each of our datasets can be seen in table 5.5, while plots 5.17 illustrate the evolution of r_1 and r_3 with increasing n.

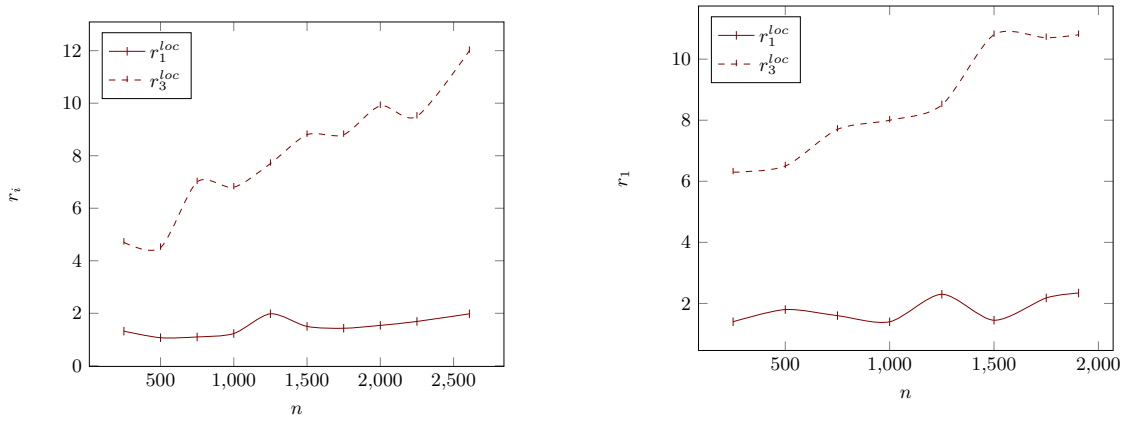

Figure 5.17: Parameters of the access node set $\mathcal{A}^{loc}$ with increasing n. Datasets *sncf* (left) and *cpsk* (right). $r_2 \leq 1$. An ideal situation would be constant (or non-increasing) functions.

Name	r_1^{loc}	r_3^{loc}
air01-d	3.9	57.4
cpsk-d	2.3	10.8
gb-coach-d	3.4	20.7
gb-train-d	3.2	22.7
montr-d	2.1	3.3
sncf-d	2.0	12.0
sncf-ter-d	1.8	9.7
sncf-inter-d	1.3	3.9
zsr-d	1.7	4.5

Table 5.5: Parameters of $\mathcal{A}^{loc}$ for all of our datasets in their maximum size. $r_2 \leq 1$. The timetable of airlines is the most challenging to find a good AN set on, since it forms a highly interconnected network and thus makes it difficult to separate the nodes' neighbourhoods with few local access nodes.

To sum up, in all of our datasets (except for *air01*), we were always able to find (r_1, r_2, r_3) access node set with the *locsep* algorithm, such that:

[28] The algorithm basically selects access nodes on a greedy basis. However, even that is done only heuristically, using local scope to reduce the time complexity.

- $r_1 \leq 3.5$
- $r_2 \leq 1$
- $r_3 \leq 25$ (i.e. up to 5 local access nodes on average)

In what follows we try to analyse the parameters of *USP-OR-A* combined with *locsep*. We however ask the reader to consider the following more like an informal discussion providing a better insight on performance of the mentioned combination, rather than a rigorous analysis.

Suppose the following conditions to be true:

1. We have a timetable with $\delta \leq \log n$ on which we found $\mathcal{A}^{loc}$ (thus $r_2^{loc} \leq 1$)
2. $\omega_{\mathcal{A}^{loc}} \leq \sqrt{n}$
3. r_1^{loc} is bound by a constant
4. $r_3^{loc} \leq \log n$
5. $\sqrt{r_3^{loc}} \leq \tau_{\mathcal{A}^{loc}}$

The first two conditions were generally satisfied by our timetables (again, not including *air01*). As for the third one, we found the value of r_1^{loc}, very small, almost constant and only very slightly increasing with n [29]. Finally, if the last two conditions hold, we may estimate the average query time of *USP-OR-A* with *locsep* as:

$$\begin{aligned}
\mathcal{O}(r_2\sqrt{r_3}\sqrt{n}(\log(r_2 n) + \delta) + r_3\tau\omega) = \\
\mathcal{O}(\sqrt{r_3}\sqrt{n}(\log n + \delta) + r_3\tau\omega) = \\
\mathcal{O}(\sqrt{r_3}\sqrt{n}\log n + r_3\tau\sqrt{n}) = \\
\mathcal{O}(\tau\sqrt{n}\log n + \log n\ \tau\sqrt{n}) = \\
\mathcal{O}(\tau\sqrt{n}\log n)
\end{aligned}$$

Most of our datasets came very close to satisfy the mentioned conditions, under which *USP-OR-A* with *locsep* reaches parameters as described in the following table.

USP-OR-A + locsep	***prep***	***size***	***qtime***	***stretch***
Under certain conditions	$\mathcal{O}(n^{2.5}\log n)$	$\mathcal{O}(\tau n^{1.5})$	avg. $\mathcal{O}(\tau\sqrt{n}\log n)$	1

Table 5.6: Parameters for *USP-OR-A* with *locsep* under certain conditions listed above.

Using *USP-OR-A* with *locsep* therefore seems to be a solution which (theoretically) should work pretty well. To justify this statement we performed experiments, the results of which we provide in the next subsection.

5.4 Performance and comparisons

In this subsection we show the performance of our algorithms on our datasets. We focus on query time and space complexity of the preprocessed oracles [30]. We have already introduced the speed-up as the ratio of average query time for the *TD Dijkstra* and the average query time for the given

[29] Although this is nowhere near the *proof* that r_1^{loc} could be bound by a constant.

[30] The methods are exact (i.e. *stretch* = 1) and we consider the preprocessing time as the least important parameter out of the four.

algorithm. We will have a similar measure for the size of the preprocessed data, which we compare against the amount of memory needed to store the actual timetable.

Definition 5.6. ***Size-up (szp(m))***
A size-up of an oracle based method m is the ratio $\frac{size(TD)}{size(m)}$ where $size(TD)$ is the size of the memory necessary to store the time-dependent graph.

Furthermore, all of our timetables have very sparse underlying graphs [31](i.e. $m \leq n \log n$). Therefore we can bound the time complexity of the *TD Dijkstra* as $\mathcal{O}(n \log n)$. Thus if we look at *TD Dijkstra* as an oracle based method (with the oracle size being simply the size of the time-dependent graph representing the timetable), we can summarize its parameters as in the table 5.7.

TD Dijkstra	***prep***	***size***	***qtime***	***stretch***
$\boldsymbol{m \leq n \log n}$	$\mathcal{O}(1)$	$\mathcal{O}(mh)$	$\mathcal{O}(n \log n)$	1

Table 5.7: The summary of the *TD Dijkstra* algorithm parameters.

5.4.1 Performance of *USP-OR*

Query time-wise, *USP-OR* clearly outperforms time-dependent Dijkstra's algorithm, however at the cost of high space consumption. The best results were achieved for the timetable of domestic US flights (with both daily and weekly time range), with speed-up of more than 110 against *TD Dijkstra* (see table 5.8). Interestingly, the size-up for this timetable was also the best one - the size of preprocessed data was about 10 times the size of the timetable itself. The reasons for such good values is that the USP diameter ω is very low in *air01* dataset (much lower than $\sqrt{n}$), while the USP coefficient τ is also a reasonable value (see table 5.1).

For similar reasons, the *gb-coach* dataset also achieved fairly good results. However, with this dataset (and few others marked with asterisk) we may notice the main disadvantage of *USP-OR-A* - we only could carry out the preprocessing for sub-timetable of up to 700 stations due to high space requirements and our space limitations.

The preprocessing time ranged according to the size of the timetable. For weekly *gb-train* dataset, it came close to 4 hours, which was the maximal time.

Name	***n***	***spd***	***szp***
air01-d	284	**110.4**	11.4
*cpsk-d**	700	14.4	301.8
*gb-coach-d**	700	**64.7**	79.1
*gb-train-d**	700	24.4	134.6
montr-d	217	8.1	40.7
*sncf-d**	700	28.7	263.7
sncf-inter-d	344	26.1	58.1
*sncf-ter-d**	700	22.2	390.6
zsr-d	225	15.1	66.4

Name	***n***	***spd***	***szp***
air01-w	287	**113.0**	9.6
*gb-coach-w**	700	**69.5**	50.6
*gb-train-w**	400	16.0	40.5
*sncf-w**	600	30.3	161.1
sncf-inter-w	366	24.5	69.8
*sncf-ter-w**	500	23.5	168.2
zsr-d	233	15.0	42.8

Table 5.8: Speed-ups and size-ups of the *USP-OR* algorithm for the whole timetables (for those marked with asterisk we took only a subset of n stations, as we were limited by the space). Daily time range on the left, weekly on the right.

[31] Maybe with exception of *air01*.

The query times were almost constant with increasing n, especially in *sncf* and *gb-coach* datasets, where τ is very low and only slightly increasing (see table 5.5) and similarly for ω. With *cpsk* the ω increases as $\sqrt{n}$ (plot 5.4), thus the increase in query time is more noticeable.

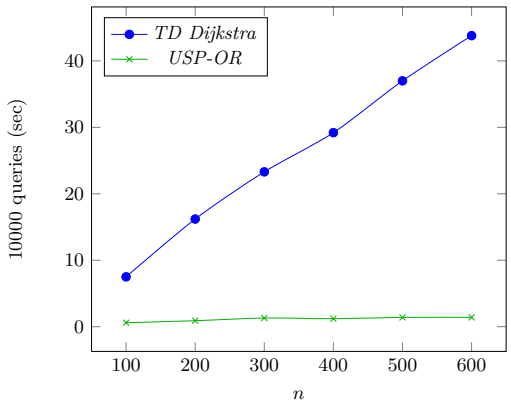

Figure 5.18: **Query time** of *USP-OR* algorithm compared to *TD Dijkstra* on the ***sncf*** dataset. **Changing *n***.

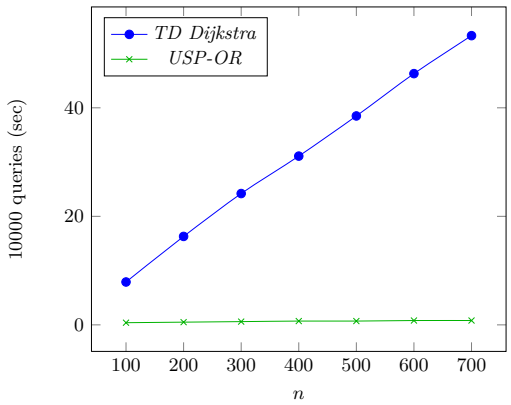

Figure 5.19: **Query time** of *USP-OR* algorithm compared to *TD Dijkstra* on the ***gb-coach*** dataset. **Changing *n***.

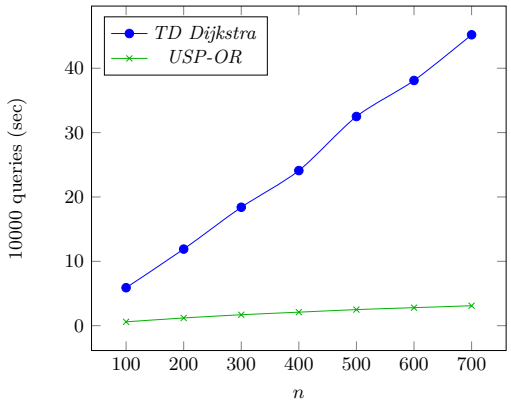

Figure 5.20: **Query time** of *USP-OR* algorithm compared to *TD Dijkstra* on the ***cpsk-d*** dataset. **Changing *n***.

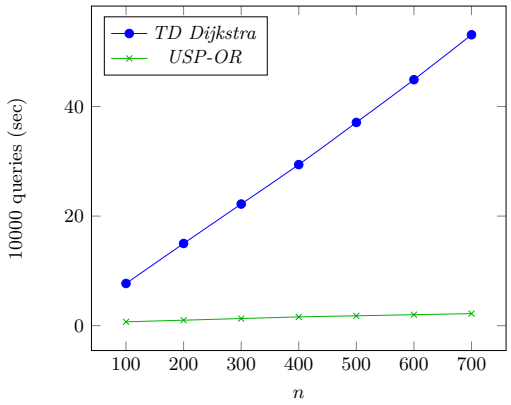

Figure 5.21: **Query time** of *USP-OR* algorithm compared to *TD Dijkstra* on the ***gb-train-d*** dataset. **Changing *n***.

From plots 5.22 and 5.23 we see that the time range does not play a role in the query time of neither *TD Dijkstra*, nor *USP-OR-A*.

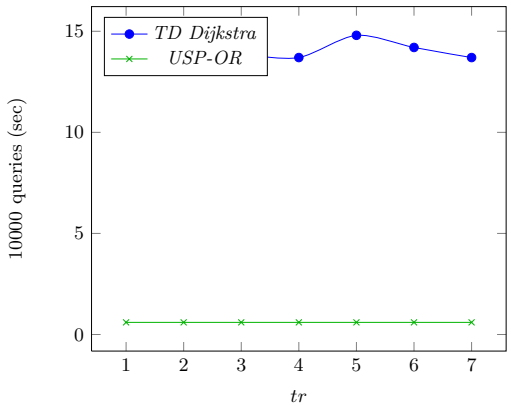

Figure 5.22: **Query time** of *USP-OR* algorithm compared to *TD Dijkstra* on the ***gb-coach-200*** dataset. **Changing *tr*** (days).

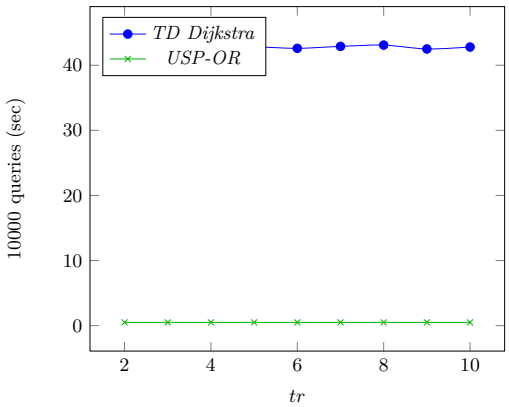

Figure 5.23: **Query time** of *USP-OR* algorithm compared to *TD Dijkstra* on the ***air01-200*** dataset. **Changing *tr*** (days).

The size of *USP-OR* oracle increases as $\mathcal{O}(\tau n^{2.5})$ in our datasets. This kind of (super-quadratic) increase can be observed from plots 5.25 and 5.24.

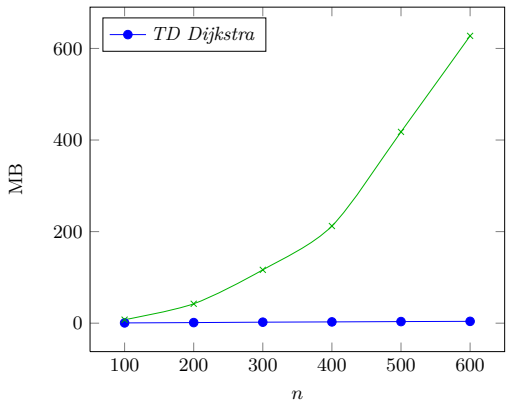

Figure 5.24: **Size** (in MB) of the oracle for *USP-OR* vs. size of TD graph on ***sncf*** dataset. **Changing *n***.

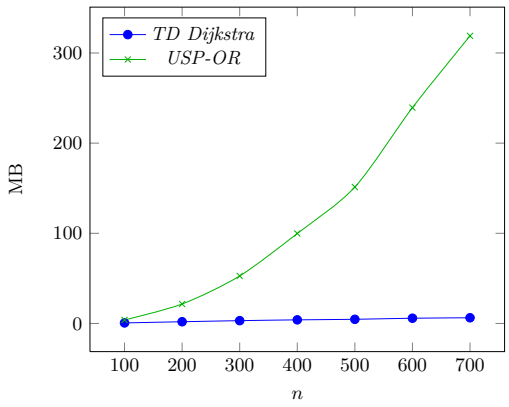

Figure 5.25: **Size** (in MB) of the oracle for *USP-OR* vs. size of TD graph on ***gb-coach*** dataset. **Changing *n***.

5.4.2 *USP-OR-A*

In these tests, we coupled *USP-OR-A* exclusively with *locsep*. The speed-ups were no longer so high as in case of *USP-OR*, but neither were the size-ups, so this time we were able to try out our datasets in their full size. The maximum speed-up was achieved in weekly *gb-coach* timetable, where *USP-OR-A* outperformed *TD Dijkstra* more than 8 times, requiring memory 4 times the size of the timetable itself.

The *air01* timetable was not so successful this time for the reasons stated in table 5.5 - it is difficult to find an access node set with desired properties. However, fairly good speed-ups were achieved also for *sncf* datasets. Again, the main factors for the query time were $\tau_{\mathcal{A}}$, $\omega_{\mathcal{A}}$ (table 5.3) but also r_3 (table 5.5).

The preprocessing times were similar as in case of *USP-OR*. E.g. the weekly dataset *sncf* was preprocessed in about one hour, while *gb-train* required almost 5 hours of preprocessing time.

Name	n	*spd*	*szp*
air01-d	284	2.4	1.7
cpsk-d	1905	2.9	6.6
gb-coach-d	2427	**8.2**	7.4
gb-train-d	2550	2.9	4.3
montr-d	217	2.8	1.7
sncf-d	2608	**5.4**	5.1
sncf-inter-d	344	**5.2**	1.9
sncf-ter-d	2600	**5.5**	5.2
zsr-d	225	3.0	2.7

Name	n	*spd*	*szp*
air01-w	287	2.4	1.1
gb-coach-w	2448	**8.5**	3.8
gb-train-w	2555	2.9	2.3
sncf-w	2646	**6.3**	4.2
sncf-inter-w	366	**4.9**	1
sncf-ter-w	2637	**5.6**	3.0
zsr-d	233	2.6	1

Table 5.9: Speed-ups and size-ups of the *USP-OR-A* for the whole timetables. Daily time range on the left, weekly on the right.

Comparisons of query times with increasing n are on plots 5.26 to 5.29. We have also observed (sometimes quite irregular) increase of speed-ups with increasing n. [32]

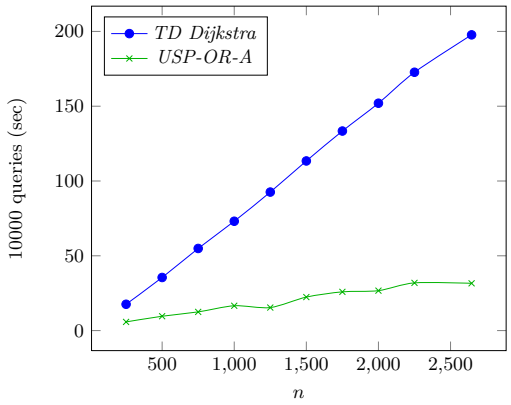

Figure 5.26: **Query time** of *USP-OR-A* algorithm compared to *TD Dijkstra* on the ***sncf*** dataset. **Changing n.**

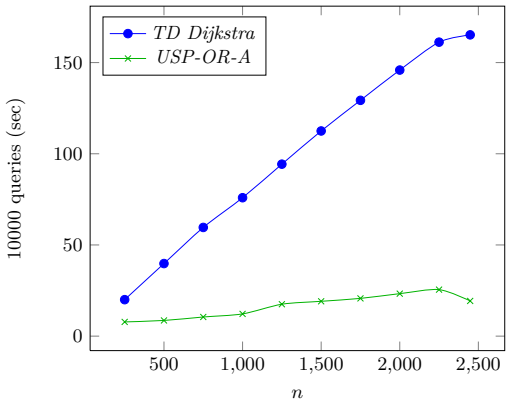

Figure 5.27: **Query time** of *USP-OR-A* algorithm compared to *TD Dijkstra* on the ***gb-coach*** dataset. **Changing n.**

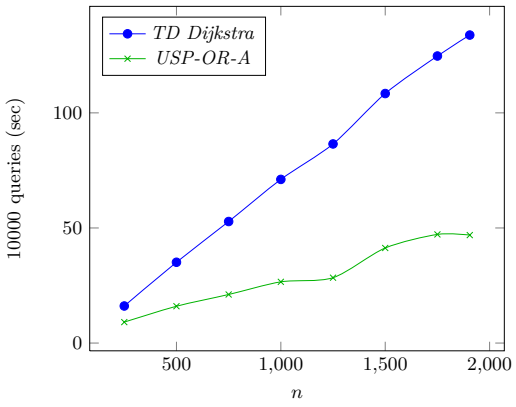

Figure 5.28: **Query time** of *USP-OR-A* algorithm compared to *TD Dijkstra* on the ***cpsk-d*** dataset. **Changing n.**

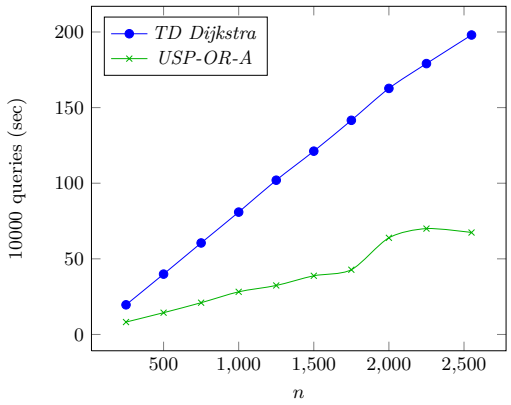

Figure 5.29: **Query time** of *USP-OR-A* algorithm compared to *TD Dijkstra* on the ***gb-train-d*** dataset. **Changing n.**

[32] Basically, we would expect the speed-up to increase as $\mathcal{O}(\frac{\sqrt{n}}{\tau_{\mathcal{A}}})$.

Again it seems that increased time range has no effect on the query time of neither method.

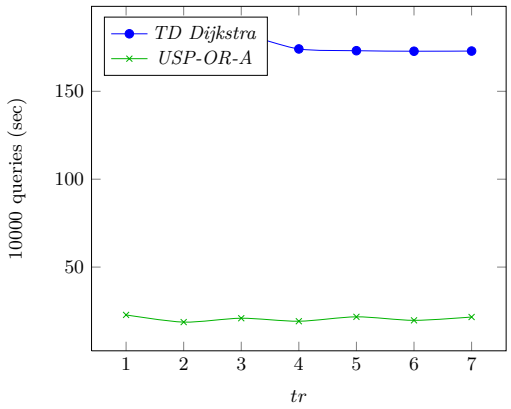

Figure 5.30: **Query time** of *USP-OR-A* algorithm compared to *TD Dijkstra* on the ***gb-coach*** dataset. **Changing *tr*** (days).

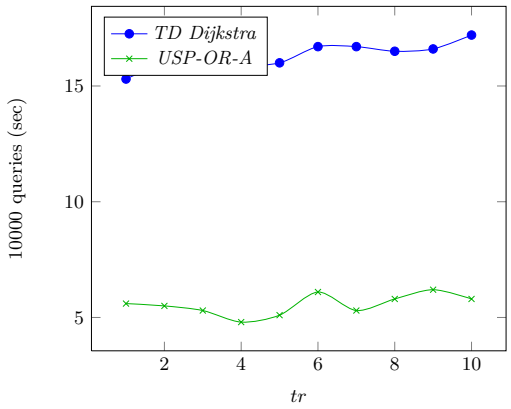

Figure 5.31: **Query time** of *USP-OR-A* algorithm compared to *TD Dijkstra* on the ***zsr*** dataset. **Changing *tr*** (days).

Finally, the size of the preprocessed oracle in *USP-OR-A* increases less steeply than with *USP-OR*. Plot 5.33 is a good example of this and suggests that the theoretical space complexity $\mathcal{O}(\tau n^{1.5})$ is plausible. In plot 5.32 the sudden increase of the size at the end is caused by our optimisation in *locsep* which may add up to $\sqrt{n}$ more cities to the AN set at the end of the algorithm described in the previous subsections.

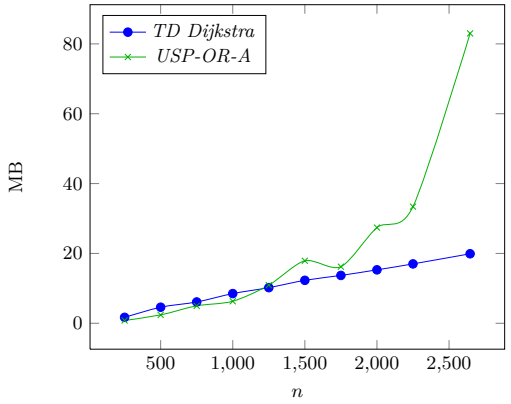

Figure 5.32: **Size** (in MB) of the oracle for *USP-OR-A* vs. size of TD graph on ***sncf*** dataset. **Changing *n***.

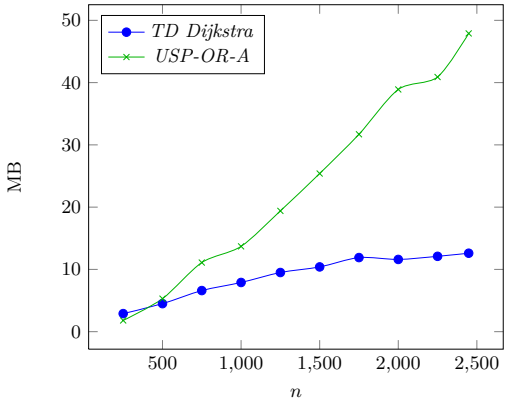

Figure 5.33: **Size** (in MB) of the oracle for *USP-OR-A* vs. size of TD graph on ***gb-coach*** dataset. **Changing *n***.

6 Neural network approach

In this section we describe our experiment approaching the optimal connection problem in timetables with the help of an artificial neural network, which is the "oracle" in this case. More specifically, we consider multi-layer perceptron with back-propagation training algorithm. The training time corresponds to the preprocessing time and the size can be parametrized by the number of hidden layers (and the size of each layer). Our main interest was if the network was able to answer with reasonable connections for given queries.

The input layer of the perceptron has one neuron for each event of the timetable and one neuron for each city of the timetable. Thus each instance of the earliest arrival query can be represented by exactly two neurons on the input layer. The output layer has one neuron for each arc of the underlying graph. The idea is that the network will activate those output neurons (arcs), that correspond to the underlying path of the connection. As we have pruned the overtaking elementary connections, we can easily reconstruct the actual connection by expanding the underlying path (algorithm 5.1). See figure 6.1 for an illustration.

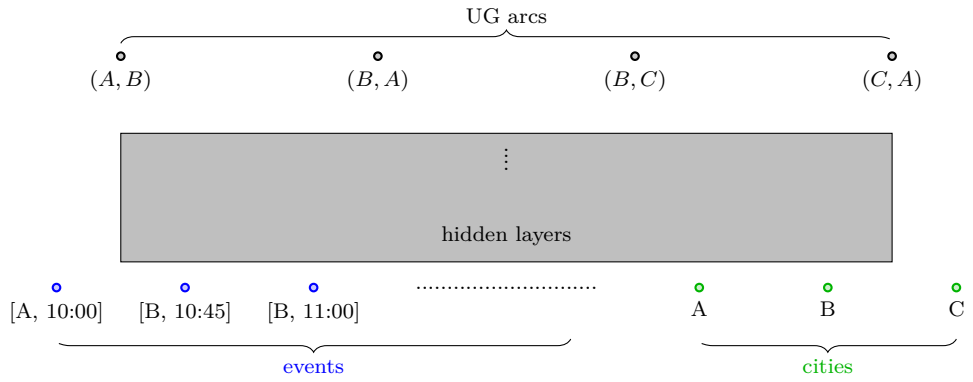

Figure 6.1: The model of our neural network - on the input layer (down) we activate the neurons corresponding to the departure event and destination city. On the output, we expect the activation of neurons corresponding to the underlying path of the connection.

For example, consider the following query in the timetable 2.1: "From A at *10:00* to C". The optimal connection is *[A, 10:00]*, *[B, 10:45]*, *[B, 11:00]*, *[C, 11:30]*. The query will activate two input neurons - the one corresponding to *[A, 10:00]* and the one corresponding to destination C. On the output we expect two activated neurons, since we really moved only two times - from A to B and from B to C.

After the training, we feed the network random inputs (queries for optimal connection). It might be that even though the network was trained, the desired neurons at the output will not be sufficiently activated and rounding them would yield all-zero output. That is why we employed the following method to recover the sequence of traversed cities:

- Start at departing city
- Consider all leaving arcs from this city
- Choose the one with strongest activation (leading to yet unvisited city)
- Terminate when the target city is reached or we cannot continue

From the method mentioned above we can see that the network may fail to find the answer in some cases (sometimes there is simply no solution, e.g. when "the last train already left"). In such situation, it outputs at least the partial answer, so to speak, "how to begin the journey".

The perceptron was trained with training data divided into two groups: **validation** and **estimation** data with 80% being part of the latter group. To eliminate overfitting, early stopping was used as a stopping criterion in some cases (that is, we stopped once the validation error [33] started to increase). To disregard noise on the validation error curve (and thus too early halt of the training when validation error accidentally increased) we compare the validation error against its value from 10 iterations back. Sigmoid activation function is used in all layers.

For the testing, we used several subsets of our datasets, described in table 6.1.

Name	n	m
tt	4	5
air30-10	30	187
air50-20	50	185
cpsk30-10	30	97
cpsk50-20	50	120
montr30	30	40
montr50	50	71
zsr30-10	30	71
zsr50-20	50	137

Table 6.1: Datasets used for testing.

6.1 Results

For each dataset, we have trained 7 types of networks:

- 1 hidden layer with 30 neurons
  - 1.) 100 training examples (t = 100), α [34] = 0.1, early stopping
  - 2.) 300 training examples (t = 300), $\alpha = 0.1$, early stopping
  - 3.) 100 training examples (t = 100), $\alpha = 0.1$, minimum of 300 iterations
- 5 hidden layers, each with 30 neurons
  - 4.) 100 training examples (t = 100), $\alpha = 1$, early stopping
  - 5.) 300 training examples (t = 300), $\alpha = 1$, early stopping
  - 6.) 100 training examples (t = 100), $\alpha = 1$, minimum of 300 iterations
- 1 hidden layer with 100 neurons
  - 7.) 100 training examples (t = 100), $\alpha = 0.1$, minimum of 500 iterations

For the first six types of network, the *maximum* number of iterations was set to 300, for the last one it was 500 (thus it was trained with exactly 500 iterations). Also, each network was trained on a different randomly generated training set.

Once the networks were trained, we have compared the first triple (table 6.2) on 1000 randomly generated earliest arrival queries (on a given dataset) and the second triple (table 6.3) on

[33] Validation error was computed as $\sum_{v \in \mathcal{V}} \frac{\sum_i^m (d_i^v - y_i^v)^2}{2}$ where $\mathcal{V}$ is the validation set, m the number of output neurons, d_i^v the i-th neuron's target value for training example v and y_i^v the actual value of i-th neuron for that training example.

[34] α is the parameter used in adjusting weights in the back-propagation algorithm.

another 1000 generated queries. In each case, Dijkstra's algorithm was part of the comparison so that we obtained the optimal values and could evaluate the accuracy of the networks (the seventh network was compared simply against the Dijkstra's algorithm, table 6.4). Even though the generated queries were different in each case, the comparison among them all can still be made based on the percentage of the times when the network was able to answer with the optimum (table 6.5).

		1.) t = 100, α = 0.1, E.S.				2.) t = 300, α = 0.1, E.S.				3.) t = 100, α = 0.1, 300 it.			
Name	**Conn ∃**	E.E.	V.E.	Found	Opt	E.E.	V.E.	Found	Opt	E.E.	V.E.	Found	Opt
tt	462	0.442	0.77	462	435	0.31	0.91	462	462	0.67	1.02	450	450
air30-10	945	105.29	28.93	184	21	331.61	88.84	187	51	46.20	26.21	464	82
air50-20	931	133.90	36.57	112	33	385.32	94.89	166	104	48.30	31.52	426	204
cpsk30-10	504	63.83	12.42	146	57	124.55	32.96	287	160	16.74	13.96	270	147
cpsk50-20	620	140.59	47.67	214	182	429.81	119.36	197	181	20.63	43.59	221	189
montr30	545	69.28	35.02	268	234	52.24	60.05	418	376	4.89	18.51	366	310
montr50	572	81.19	54.28	290	253	161.01	124.99	351	314	6.61	33.43	341	300
zsr30-10	643	57.08	29.24	278	204	298.64	69.62	179	144	13.59	23.02	306	229
zsr50-20	892	130.74	46.05	152	53	448.37	134.80	185	60	34.02	41.85	347	122

Table 6.2: The first triple of trained neural network. *Conn* ∃ - number of test cases (out of 1000) when there existed a connection (found by Dijkstra's algorithm) for the query. *E.S.* - early stopping. *E.E. and V.E.* - estimation and validation error at the end of training. *Found* - found a connection for the query. *Opt* - found optimal connection.

		4.) t = 100, α = 1, E.S.				5.) t = 300, α = 1, E.S.				6.) t = 100, α = 1, 300 it.			
Name	**Conn ∃**	E.E.	V.E.	Found	Opt	E.E.	V.E.	Found	Opt	E.E.	V.E.	Found	Opt
tt	449	12.16	3.13	292	292	2.79	1.02	403	403	0.00	1.35	449	449
air30-10	939	119.56	30.77	178	21	339.35	79.04	121	26	34.21	37.8	505	94
air50-20	922	132.05	28.36	30	16	371.12	106.18	50	36	33.86	54.17	387	171
cpsk30-10	518	61.83	13.91	133	94	195.12	35.84	156	101	30.25	20.52	174	80
cpsk50-20	601	133.25	46.01	164	119	363	115.19	179	159	64.85	53.07	162	143
montr30	495	103.71	33.63	349	248	191.92	107.10	304	273	8.67	36.86	250	214
montr50	565	124.11	44.92	240	226	411.93	111.62	249	243	68.37	50.77	238	229
zsr30-10	658	115.50	26.28	140	117	331.12	82.83	147	124	16.42	37.08	211	165
zsr50-20	900	156.06	32	110	57	475.49	104.63	135	45	43.78	50.12	321	134

Table 6.3: The second triple of trained neural network. *Conn* ∃ - number of test cases (out of 1000) when there existed a connection (found by Dijkstra's algorithm) for the query. *E.S.* - early stopping. *E.E. and V.E.* - estimation and validation error at the end of training. *Found* - found a connection for the query. *Opt* - found optimal connection.

		7.) t = 100, α = 0.1, E.S.			
Name	**Conn ∃**	E.E.	V.E.	Found	Opt
tt	471	0.1	2.24	471	471
air30-10	931	23.47	30.76	573	107
air50-20	938	37.66	38.44	405	203
cpsk30-10	481	11.67	17.22	281	135
cpsk50-20	605	8.51	46.87	252	195
montr30	527	2.85	43.44	346	300
montr50	615	4.13	24.97	386	346
zsr30-10	672	7.7	26.92	307	234
zsr50-20	887	22.68	42.68	426	178

Table 6.4: The 7th trained neural network. *Conn* ∃ - number of test cases (out of 1000) when there existed a connection (found by Dijkstra's algorithm) for the query. *E.S.* - early stopping. *E.E. and V.E.* - estimation and validation error at the end of training. *Found* - found a connection for the query. *Opt* - found optimal connection.

There are several things to notice about these results:

- Interestingly, validation error is smaller (in most cases) when we did not use the early stopping criterion than in the case when we used it for networks with one hidden layer (first triple). For the second triple, this is no longer the case. Upon closer look, we have found out that this phenomenon is simply due to different training sets used with each network (e.g. on the

montr50 dataset, type *3.)*, the network has already started with validation error being lower than validation error on a trained network of type *1.)* on this dataset).

- Unfortunately, already we can see that the neural networks performed very poorly, especially on the datasets *zsr* and *air*, finding optimal connections in barely 5% of the queries. Better summary will be visible in table 6.5.

We have taken a closer look at the evolution of validation errors in the training of type *7.)* networks and have found that in general, the validation error went steeply down at the beginning, followed by an interval of fluctuations (which probably caused early stopping when this criterion was in use) and finally found a pretty stable value. The estimation error was decreasing all this time, which is why the networks of type *7.)* obtained most of the time the best results in the final comparison.

	1.)		2.)		3.)		4.)		5.)		6.)		7.)	
Name	O.f.F.	F.O.	O.f.F.	F.O.	O.f.F.	F.O.	O.f.F.	F.O.	O.f.F.	F.O.	O.f.F.	F.O.	O.f.F.	F.O.
tt	94.2	94.2	100.0	100.0	100.0	97.4	100.0	65.0	100.0	89.8	100.0	100.0	100.0	**100.0**
air30-10	11.4	2.2	27.3	5.4	17.7	8.7	11.8	2.2	21.5	2.8	18.6	10.0	18.7	**11.5**
air50-20	29.5	3.5	62.7	11.2	47.9	**21.9**	53.3	1.7	72.0	3.9	44.2	18.5	50.1	21.6
cpsk30-10	39.0	11.3	55.7	**31.7**	54.4	29.2	70.7	18.1	64.7	19.5	46.0	15.4	48.0	28.1
cpsk50-20	85.0	29.4	91.9	29.2	85.5	30.5	72.6	19.8	88.8	26.5	88.3	23.8	77.4	**32.2**
montr30	87.3	42.9	90.0	**69.0**	84.7	56.9	71.1	50.1	89.8	55.2	85.6	43.2	86.7	56.9
montr50	87.2	44.2	89.5	54.9	88.0	52.4	94.2	40.0	97.6	43.0	96.2	40.5	89.6	**56.3**
zsr30-10	73.4	31.7	80.4	22.4	74.8	**35.6**	83.6	17.8	84.4	18.8	78.2	25.1	76.2	34.8
zsr50-20	34.9	5.9	32.4	6.7	35.2	13.7	51.8	6.3	33.3	5.0	41.7	14.9	41.8	**20.1**

Table 6.5: Table summarizes in how many cases (in percents) the network found the optimum value (*F.O. - found optimum*) and in how many cases (percents) when the network found some connection sequence, it was the optimal sequence (*O.f.F - optimum from found*).

In table 6.5 we note down two measures of the network's quality:

- **F.O.**: The network's capability to output the optimum connection (when it, in fact, exists)
- **O.f.F**: The network's reliability - that once the network has found a connection, it is really the best one

Network with one big hidden layer, trained without the early stopping criterion was the most successful one, when it comes to the F.O. measure, though the best value (69 %) was achieved by a network of type *2.)* trained on a bigger training set. Perceptrons trained on these bigger training sets (*2.)* and *5.)*) were also more reliable.

6.2 Conclusion of the experiment

First of all, we would like to point out some positive aspects. The networks were able to learn optimal answers to queries that were *not* part of the training. Also, it looks like enhanced training of the network could produce much better results.

On the other hand, there are other parameters that could possibly help more then just increasing the number of iterations: different activation functions, error functions, weights adjustments (using momentum, or weight decay) or even using different type of network (some publications have used Hopfield network to search for shortest paths in graphs).

However, we conclude that we failed to train neural network to answer optimal connection queries in timetables. We suppose the main reason for this is that the problem is simply too challenging for a neural network.

7 Application TTBlazer

For the purposes of analysing our timetables and testing our methods we developed a C++ command line application named *TTBlazer* [35], which we will now shortly describe. The application is *not* meant for common users but rather for those who would like to try out algorithms by themselves, review their code or continue with our work.

The application works with 4 main types of **objects** that we have introduced in the section 2:

- Underlying graph
- Time-expanded graph
- Time-dependent graph
- Timetable

A user can save/load each of this object from a file (in appendix A we describe the file formats). Once loaded, there are four types of **actions** to be performed on the objects:

- **Analysing** properties of the objects (e.g. analysis of degree distribution of the underlying graph)
- **Generating** other objects from those that are loaded. This includes also conversions e.g. from a timetable to the time-expanded graph. Another example may be extracting the largest strongly connected component of a graph
- **Modifying** the object, e.g. removing overtaken connections from a timetable
- **Creating an oracle** on the object. We distinguish two types of oracles:
  - **Distance oracle**. Answers queries for *shortest-path* or a *distance* between two nodes
  - **Timetable oracle**. Answers queries for *optimal connection* or the *earliest arrival* between two cities, given the departure time

  Once the oracle is created (preprocessing is finished), the user can query it for the optimal shortest-paths/connections

There is one more auxiliary type of *action* that can be carried out - we call it **posting**. A postman (posting method) generates something (a mail) and stores it (in a postbox) to be later retrieved by another action. For example, we compute cities in the underlying graph with high betweenness centrality value, store the computed set and then use it to pre-compute *USP-OR-A* using the set as an access node set.

The main purpose of the application is therefore to serve as an environment for algorithms that manipulate timetables and other objects. The list of implemented actions can be found in the table 7.1.

7.1 Features

Besides the implemented actions, the application has the following features:

- The program may be fed commands either through standard command line input, or through UDP port on which it is listening. The user may switch between these two options to specify the so called *command source*, however listening on a UDP port is a default option

[35] The latest and full version with all the datasets can be found on the accompanying CD or at `http://fish.studenthosting.sk/?page_id=183`.

Action type	Action	Objects	Description
Analysing	usp	TE, TD	analyses the underlying shortest paths *(created by posting action)*
	conns	TE, TD	analyses the optimal connections (avg. size, length...)
	hd	UG, TE, TD	analyses the highway dimension
	var	TE, TD, TT	analyses various properties of timetable objects (e.g. height)
	conn	UG, TE	analyses connectivity
	strconn	UG	analyses strong connectivity of the underlying graph
	degs	UG	analyses degrees of the underlying graph
	paths	UG	analyses shortest paths (avg. size, length...)
	betw	UG, TE	analyses betweenness centralities
	accn	UG	analyses access node set *(created by posting action)*
	density	UG	analyses the density of the underlying graph
	overtake	TT	analyses overtaking in a timetable
Generating	subcon	UG	generates connected subgraph of the given UG
	strcomp	UG	generates the UG with nodes from the largest strongly connected component
	2ug	TE, TD, TT	generates the underlying graph from given timetable object
	2te	TT	generates the time-expanded graph from the given timetable
	2td	TT	generates the time-dependent graph from the given timetable
	subtt	TT	generates a sub-timetable from the given timetable
Modifying	rmover	TT	removes overtaking from given timetable
Timetable oracle	neural	TE, TD	creates the oracle based on a neural network
	uspor	TD	creates the oracle based on *USP-OR*
	usporseg	TD	creates the oracle based on *USP-OR*, uses segmentation
	uspora	TD	creates the oracle based on *USP-OR-A (requires access node set computed by posting action)*
	usporaseg	TD	creates the oracle based on *USP-OR-A*, uses segmentation *(requires access node set computed by posting action)*
	dijkstra	TE, TD	creates the oracle based on time-dependent Dijkstra's algorithm
Distance oracle	dijkstra	UG	creates the oracle based on Dijkstra's algorithm
Posting	anhbc	UG	creates the access node set $\mathcal{A}^{bc}$
	andeg	UG	creates the access node set $\mathcal{A}^{deg}$
	usp	TE, TD	computes the USPs (between all pairs or just on a given subset)
	locsep	UG	creates the access node set $\mathcal{A}^{loc}$

Table 7.1: A list of actions implemented in *TTBlazer*. In the actual implementation there may be more actions that we created for other experiments not included in this thesis.

- The output is done through logging of the program. The logs could be output to screen, written to files or sent on ports. There are 3 types of logs:
  - *Info*: information for the user (is always printed)
  - *Error*: is always printed
  - *Debug message*: information for the developer, printed only if debugging is turned on. The debug message is further parametrized by a number (*debug level*), usually a different one for each module of the program
- The application can run scripts with commands stored in a file
- A user can see the duration of a an action that was carried out. Also the size of the oracle is output (or at least the lower bound)

7.2 Design

In the whole solution, there are basically 2 applications that share some common parts:

- *TTBlazer* - the main application, that carries all the logic and does all the computation
- *Commander* - small program that sends commands to *TTBlazer*. The advantage of this is,

that even when the main application is busy computing, it still listens on a port for commands to be executed later
- *Common* - these are just common files shared by both applications (like logging, working with network connections etc...)

The source files of *TTBlazer* are further structured. There is a folder for each of the mentioned 4 + 1 types of actions:
- *Analysers* - contains analysing actions
- *Generators* - contains generating actions
- *Modifiers* - contains modifying actions
- *Oracles* - contains oracle-creating actions
- *Postbox* - contains posting actions (postmen)

7.3 Compilation and usage

In the directory of the solution, following folders can be found:
- *common/*
- *commander/*
- *data/*
- *ttblazer/*
- *inscripts/*

The folders *ttblazer*, *commander* and *common* contain source files for respective applications (or shared files). Both applications have a file *buildinfo.sh* specifying (relative) paths to source files folder, binary files folder, name of the final binary, compilation flags and then the individual modules (one per source file). This file is used by *depmake.sh* - a bash script that creates a makefile from the information in *buildinfo.sh*.

In order to compile the programs, boost library must be installed in the parent folder ("solution" in this case) in a directory called "boostlib". It should look like this:

```
1 solution/
2 |-- boostlib
3 |   |-- boost
4 |       |-- accumulators
5 |       |   |-- accumulators_fwd.hpp
6 |       |   |-- accumulators.hpp
7 |       |   |-- framework
8 ...
```

Listing 1: Location of the Boost library.

Boost library is freely available at `http://www.boost.org/`. In case you have the library installed, you may also just adjust the path to the library in *buildinfo.sh* files (located in directories *ttblazer* and *commander*).

For both programs there are following 3 simple shell script:
- *Xcomp.sh* [36] - recreates the makefile, compiles the sources and builds the binary. Its arguments are forwarded to *make*

[36] "X" is *t* for *TTBlazer* or *c* for *Commander*.

- *Xrun.sh* - runs the binary. Arguments are forwarded to the started binary
- *Xboth.sh* - combination of *Xcomp.sh* and *Xrun.sh*. Arguments are forwarded to the started binary

In the folder *data/example*, there are 4 files - one for each type of object mentioned at the beginning of this section. In the *data/real* folder we provide some of the real-world timetables, converted to our format (described in appendix A).

Finally, the folder *inscripts* contains 3 scripts which could be run with *TTBlazer* to demonstrate some functionality. In order to do so, proceed as follows:

- Install the Boost library according to the steps above
- Compile the main application: `./tcomp.sh`
- Run *TTBlazer* scripts like this: `./trun.sh -script inscripts/uspora-sncf.sc`. All three scripts contain a short description of what functionality they demonstrate.

In case you want to send commands to *TTBlazer* manually, open another terminal window and proceed e.g. like this:

```
1 ./ccomp.sh                                    //compile the Commander program
2 ./crun.sh load tt data/example/tt.tt          //load the timetable from file
3 ./crun.sh gen tt 2td last                     //generate the time-expanded graph
      from the loaded timetable
4 ./crun.sh or td dijkstra last                 //create oracle (based on Dijkstra's
      algorithm) on the time-expanded graph
5 ./crun.sh conn td last dijkstra A 0 10:00 D   //use the oracle to find out earliest
      arrival for some query
```

Listing 2: Sending commands to the main application through Commander.

8 Conclusion

In this thesis we studied the optimal connection problem in timetables on which we are allowed to carry out preprocessing. We formally approached the topic by clearly defining the terminology, model of the timetable and its graph representations, as well as the approach that is based on the preprocessing of the timetable. On a more practical note, we have gathered numerous real-world timetables of various type and scale and analysed their main properties.

In the hearth of this thesis, we have developed exact methods to considerably speed-up the query time for the optimal connections compared to the time-dependent Dijkstra's algorithm (running in $\mathcal{O}(m + n \log n)$). Our first algorithm - *USP-OR* - is based on pre-computing paths, that are worth to follow (the so called underlying shortest paths). The method achieves speed-ups of up to 113 on the weekly timetable of US domestic flights, requiring about 10 times the memory needed to represent the timetable itself. A noticeable speed-up (up to 70) was also reached for the sub-timetable of country-wide coaches in Great Britain. However, here the space consumption was much higher (a factor of more than 50), which was also the case with other timetables of bigger scale. We conclude that the algorithm is suitable for smaller-size timetables with very short connections, such as airline timetables. With most bus/train timetables this method has a space complexity $\mathcal{O}(\tau n^{2.5})$ and an average query time $\mathcal{O}(\tau\sqrt{n})$.

Our second algorithm called *USP-OR-A* computes a small set of important access stations and an additional information for optimal travelling between these stations. It is much less space demanding, however still more than 8 times faster then the time-dependent Dijkstra's algorithm on the dataset of British country-wide coaches (about 2500 stations) and about 6 times faster on the dataset of French railways (also about 2500 stations). The size of the preprocessed data was no more than 7.5 times the size of the timetable for any of our datasets. Under certain conditions, this algorithm has an average query time $\mathcal{O}(\tau\sqrt{n}\log n)$ and the space complexity $\mathcal{O}(\tau n^{1.5})$. This method works well on timetables with sparse underlying graphs that contain small sets of important transit-like stations. We showed that finding these sets is NP hard problem in general, however, the heuristics we developed for this purpose seems to behave reasonably well.

With respect to *USP-OR-A*'s query times, we would like to note that we measured the query times using a completely uniform distribution of queries, which is not very realistic scenario. A real-world distribution is much different in that it strongly favours queries concerning the most important cities. We observed that such cities were generally part of the access node sets in *USP-OR-A* [37]. As computing optimal connections between these cities is very fast (just like in *USP-OR*), we could expect much better speed-ups in real-world situations.

As a possibility for the future work, it would be interesting to see how our techniques could be combined with other, already developed. Extension to a model that includes lines and transfers might also be worth considering. Finally, we also believe that relaxing the requirement for exactness (e.g. pre-computing only most important underlying shortest paths) might lead to better query times.

From other contribution of this thesis we mention the application *TTBlazer* for timetable analysis and performance tests that we developed for the purpose of this thesis, as well as the experiment in which we tried to train a neural network to answer optimal connection queries. This approach turned out as not working good enough, leading to our belief that the problem in question is too

[37] E.g. stations like Gare de Lyon were *always* part of the access node set in *sncf* dataset.

demanding for a neural network to solve.

To conclude, we feel this thesis provides useful techniques, results and information in general that might be of interest when designing a large-scale timetable information system.

Appendices

A File formats

Timetable is simply a set of elementary connections, thus the format is:

- number of el. connections
- the list of all el. connections (one per line, format "*FROM TO DEP-DAY DEP-TIME ARR-DAY ARR-TIME*")

```
7                       //number of elementary connections
A B 0 10:00 0 10:45     //el. connection
A B 0 11:00 0 11:45
A B 0 12:00 0 12:45
A C 0 09:30 0 10:00
A C 0 10:15 0 10:45
C D 0 11:00 0 11:30
C D 0 13:00 0 13:30
```

Listing 3: TT file format.

Underlying graph is basically an oriented graph, with some optional parameters. The format is the following:

- number of cities
- number of arcs
- the list of all cities (one per line)
  - optional coordinates (otherwise null)
- the list of all arcs (one per line, format "*FROM TO*")
  - optional length (otherwise null)
  - optional list of lines operating on that arc (otherwise null)

```
4                                       //number of cities
5                                       //number of arcs
A 45 32                                 //name of the city, optional coordinates
B null
C 56 34
D null
A B 57 Northern                         //arc, optional length and list of lines
A C null Picadilly Victoria
C B 45 Circle Jubilee Picadilly
C D 32 null
D A null null
```

Listing 4: UG file format.

Time-expanded graph is simply an oriented weighted graph, with nodes being the events and arcs being the elementary connections or waiting edges:

- number of nodes (i.e. events)
- number of arcs (el. connections + waiting)
- the list of all events (in the format "CITY DAY TIME")
- the list of all arcs (in the format "FROM-EVENT TO-EVENT")

```
5                       //number of events
15                      //number of arcs
A 0 13:30               //event
A 0 14:00
B 0 13:45
B 0 15:00
C 0 14:15
A 0 13:30 A 0 14:00     //waiting arc
A 0 13:30 B 0 13:45     //el. connection arc
A 0 14:00 B 0 15:00
A 0 13:30 B 0 15:00
C 0 14:15 B 0 15:00
...
```

Listing 5: TE file format.

Time-dependent graph is an oriented graph with a function on the arc specifying the arc's traversal time at any moment. In timetable networks this function is piece-wise linear and it is fully represented by the list of its interpolation points. Thus the TD file format:

- number of cities
- number of arcs
- the list of all cities (one per line)
  - optional coordinates (otherwise null)
- the list of all arcs (one per line). Arc has the format "*FROM TO INT-POINTS*" where *INT-POINTS* is a list of interpolation points [38], see the listing 6 for an example

```
4                                           //number of stations
5                                           //number of arcs
A 0 0                                       //name of the city, optional coordinates
B 4 4
C null
D 12 0
A B (0 13:30 45) (0 14:00 40)               //arc and the list of interpolation
    points
A C (1 14:15 10)
C B (0 15:00 20)
C D (2 10:00 70)
D A (1 17:20 35) (1 18:00 40) (1 18:50 35)
...
```

Listing 6: TD file format.

[38] An interpolation point is described by a triple "*DAY TIME MINUTES*", where *MINUTES* are the traversal time.

References

[1Te] 1Tech. National rail case study.

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In Moses Charikar, editor, *SODA*, pages 782–793. SIAM, 2010.

[Bar] Emma Barnett. Facebook cuts six degrees of separation to four.

[BDSV09] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-dependent contraction hierarchies. In Irene Finocchi and John Hershberger, editors, *ALENEX*, pages 97–105. SIAM, 2009.

[BFM06] Holger Bast, Stefan Funke, and Domagoj Matijevic. Transit ultrafast shortest-path queries with linear-time preprocessing, 2006.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[Del08] Daniel Delling. Time-dependent sharc-routing. In Dan Halperin and Kurt Mehlhorn, editors, *ESA*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, 2008. ISBN 978-3-540-87743-1.

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

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering time-expanded graphs for faster timetable information. In Ravindra Ahuja, Rolf Mohring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 182–206. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-05464-8.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *ALGORITHMICS OF LARGE AND COMPLEX NETWORKS. LECTURE NOTES IN COMPUTER SCIENCE*. Springer, 2009.

[DW09] Daniel Delling and Dorothea Wagner. Time-dependent route planning. In *Robust and Online Large-Scale Optimization, LNCS*. Springer, 2009.

[GPPR04] Cyril Gavoille, David Peleg, Stephane Perennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85 – 112, 2004. ISSN 0196-6774. URL `http://www.sciencedirect.com/science/article/pii/S0196677404000884`.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008. ISBN 978-3-540-68548-7.

[KMS06] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *IN: 9TH DIMACS IMPLEMENTATION CHALLENGE [29*, 2006.

[KP] Kelly Kinahan and Jennifer Pryor. Dijkstra's algorithm for shortest route problems.

[MHSWZ07] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, chapter Timetable Information: Models and Algorithms, pages 67 – 90. Springer, 2007.

[ops] Art of problem solving. Root-mean square-arithmetic mean-geometric mean-harmonic mean inequality.

[Som10] Christian Sommer. *Approximate Shortest Path and Distance Queries in Networks*. PhD thesis, Graduate School of Information Science and Technology, The University of Tokyo, 2010.

[SS05] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In Gerth Stølting Brodal and Stefano Leonardi, editors, *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005. ISBN 3-540-29118-0.

[TZ05] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.