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: Computer science
Branch of study: 2508 Informatics
Supervisor: doc. RNDr. Rastislav Královič, PhD.

Bratislava 2013

65787891

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

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 ...

Abstract

This thesis...

Key words: **oracles**, **timetable**

Abstrakt

V tejto práci...

Kľúčové slová: **oracles**, **timetable**

Contents

1 Introduction

World is getting smaller every day...

1.1 Motivation

1.2 Approach

1.3 Goals

1.4 Organization

2 Preliminaries

In this section, we would 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 notions related to these objects.

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

- *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***
  - $\boldsymbol{to(e)} = y$ *is the* ***arrival/destination city***
  - $\boldsymbol{dep(e)} = p$ *is the* ***departure time***
  - $\boldsymbol{arr(e)} = q$ *is the* ***arrival time***
- *The set of all* ***cities*** *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* $\boldsymbol{r_T} = thigh_T - tlow_T$ *is called the* ***time range*** *of the timetable.*
- ***Height*** *of the timetable is the maximum number of events in a city:* $\boldsymbol{h_T} = \max_{x \in cities_T} \{|ev_T(x)|\}$

Let us describe some 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* in our timetables). 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 represent 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)

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* $\forall i \in \{2, ..., k\} : (to(e_i) = from(e_{i-1}),\ arr(e_i) \geq dep(e_{i-1})$.

- *Connection* ***starts*** *at the departure time* $\boldsymbol{start(c)} = dep(e_1)$ *and* ***ends*** *at the arrival time* $\boldsymbol{end(c)} = arr(e_k)$.
- *We also extend* $\boldsymbol{from(c)} = from(e_1)$ *and* $\boldsymbol{to(c)} = to(e_k)$
- ***Length*** *of the connection is* $\boldsymbol{len(c)} = end(c) - start(c)$
- ***Size*** *of the connection is* $\boldsymbol{size(c)} = k$ [1]
- *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.

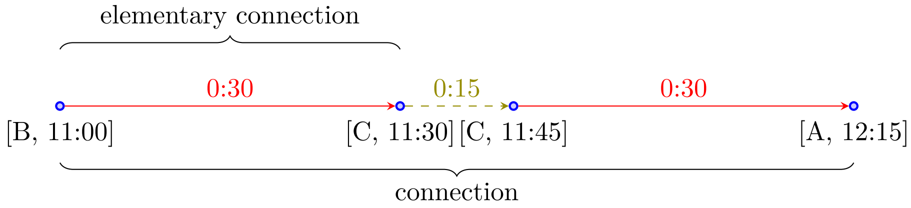

Figure 2.1: A valid connection made out of elementary connections (and waiting, which is implicit)

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* $G = (V, E)$, *where* V *is the set of all timetable cities and* $E = \{(x, y) | \exists(x, y, p, q) \in T\}$

- *By* 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, we will work with UG where the weight of each arc is the length of the shortest elementary connection on that arc. More specifically, $w(x, y) = \min_{(x,y,p,q)\in T}(q - p)\ \forall(x, y) \in E(ug_T)$.

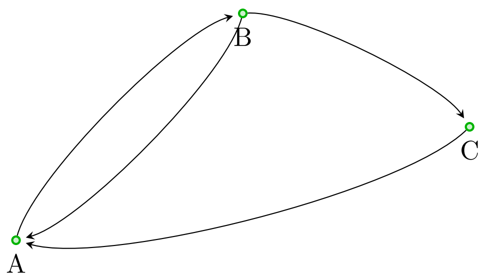

Figure 2.2: Underlying graph of the timetable in picture 2.1. The nodes are the cities

If we want to represent the timetable itself by a graph, there are two most common options [MHSWZ07].

Definition 2.4. ***Time-expanded graph (TE graph)***
Let T *be a timetable. Time-expanded graph from* T, *denoted* $\boldsymbol{te_T}$, *is an oriented graph* $G = (V, E)$

[1] 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)

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 picture 2.3).

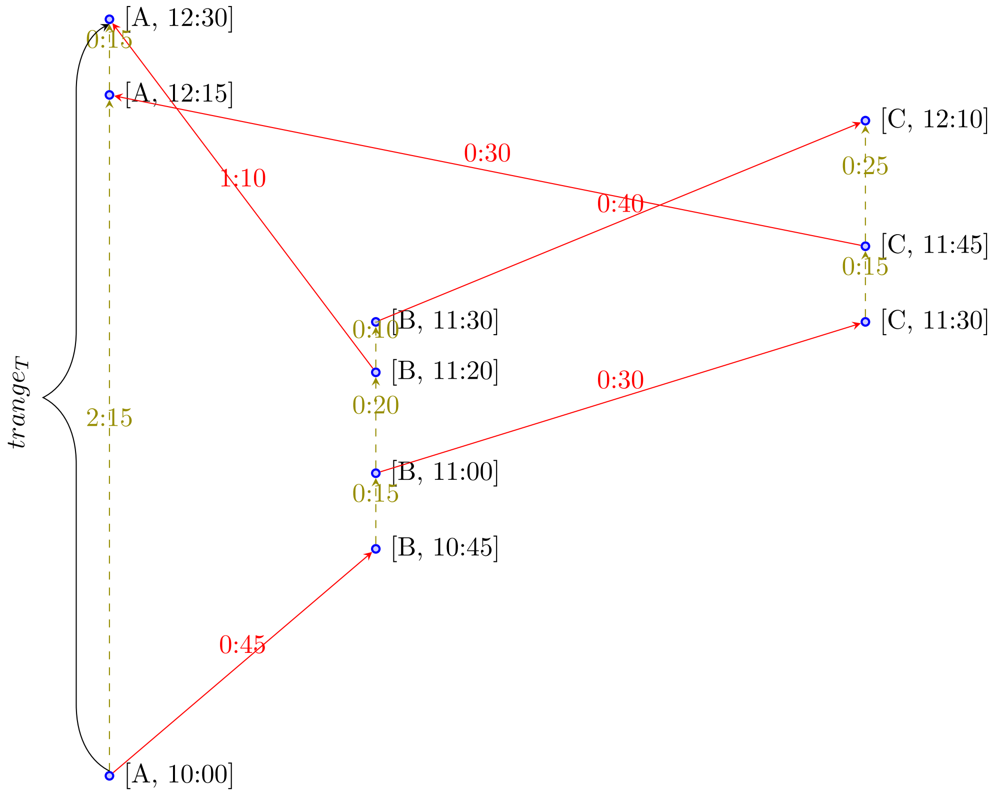

Figure 2.3: Time-expanded graph of the timetable in picture 2.1. Nodes represent the events. There are connection and waiting edges (dashed). The time range is 2h:30m and the height is 4 (as there are 4 events in city B)

Definition 2.5. ***Time-dependent graph (TD graph)***
Let T be a timetable. Time-dependent graph from T, denoted $\boldsymbol{td_T}$, *is an oriented graph $G = (V,E)$ 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 where each arc carries a function specifying the traversal time of that arc at any time. For an example, see picture 2.5: The latest point of every linear segment is called the **interpolation point** and it corresponds to an elementary connection (its coordinates are $dep(e), len(e)$ for corresponding el. connection e). Note that a list of all interpolation points fully defines the piece-wise linear function.

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.

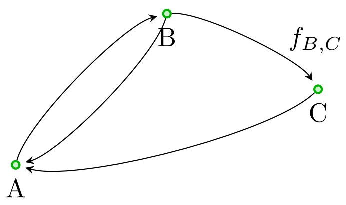

Figure 2.4: Time-dependent graph of the timetable in picture 2.1. The nodes are the cities

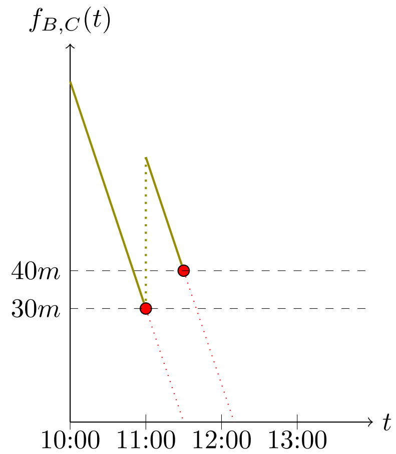

Figure 2.5: Piece-wise linear function - traversal times for the arc (B, C). The highlighted points are the interpolation points

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 further reference, we will call **timetable objects** those, that fully represent a timetable (TT, TE, TD) and **graph objects** those, that can be viewed as a graph (UG, TE, TD).

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

Now we would like to 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 $\boldsymbol{t^*_{(x,t,y)}} = \min_{c \in C_T(x,y)}\{t + len(c) | start(c) \geq t\}$.

- *We will refer to the tuple $\boldsymbol{(x, t, y)}$ as an **EAP instance**, or an **EAP 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 ending 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 determine the ***optimal connection (OC)*** $\boldsymbol{c^*_{(a,t,b)}} = argmin_{c \in C_T(a,b)}\{t + len(c) | start(c) \geq t\}$.

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 in later parts of the thesis, we will assume the optimal connection is unique (i.e., there is not a different connection with the same end time) or that ties are won by a lexicographically first connection.

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 picture 2.6 of the TE graph.*

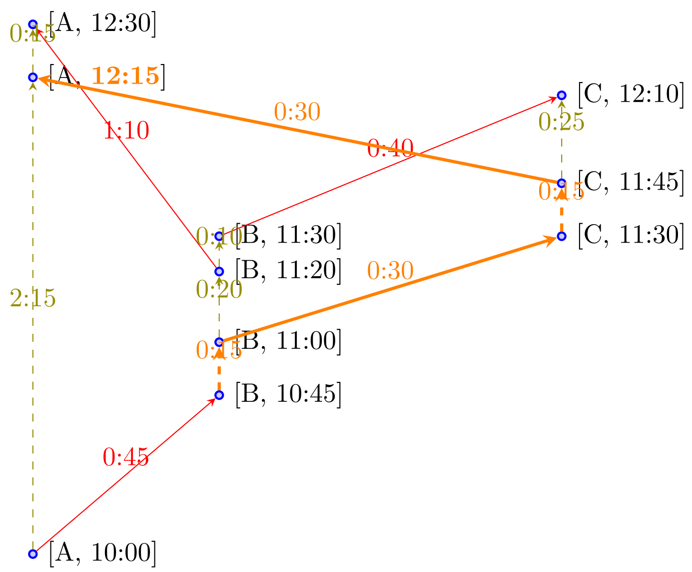

Figure 2.6: Optimal connection and earliest arrival time are marked in **bold**

2.3 (Distance) Oracles

The term *distance oracle* was first coined in 2001 by Thorup and Zwick [?], 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 pre-computed 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 earliest arrival problem (or optimal connection problem) - there is some pre-processing 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** ($\boldsymbol{prep(m)}$) - the time complexity of the pre-computation
- **Preprocessed space** ($\boldsymbol{size(m)}$) - the space complexity of the pre-computed data structure (the so called **oracle**)
- **Query time** ($\boldsymbol{qtime(m)}$) - the time complexity of answering a single query
- **Stretch** ($\boldsymbol{stretch(m)}$) - the worst-case ratio against the optimal value of earliest arrival (the lower, the better)

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 diameter of the underlying graph (as connections could be that long, or even longer). 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.

3 Related work

4 Data & analysis

In this section we would like to introduce the timetable datasets we were working with and provide the results of the analysis which we carried out on the data. The main reason for this analysis is that it gives some insight into the properties of the timetables, and thus may contribute to the make 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 briefly describe each of these timetables.

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 [2] in CSV format.

Timetables *cpru* and *cpza* 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. . Both of the timetables concern about 1000 bus stops and came in a JDF 1.9 format [3]. 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 *montr* dataset is part of a public feed for Greater **Montreal public transportation**, available at Google Transit Feeds [4]. 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 [5]. The schedules are weekly, but we have extracted just a subrange corresponding to Monday. Also, there were two types of schedules: 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 and sub-timetables (with less stations, smaller time-range or smaller height) were generated. Also the UG, TE and TD were generated from each timetable.

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

[2] `http://stat-computing.org/dataexpo/2009/the-data.html`
[3] Jednotný dátový formát (JDF)
[4] `http://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds`
[5] `http://test.data-sncf.com/index.php/ter.html`

Name	Description	Format	Provided by	Publicly available
air01	domestic flights (US)	CSV	American Stat. Assoc.	✔
cpru	regional bus (Ružomberok, SVK)	JDF 1.9	Inprop s.r.o.	✗
cpza	regional bus (SVK, Žilina)	JDF 1.9	Inprop s.r.o.	✗
montr	public transport (Montreal, CA)	GTFS	Montreal AMT	✔
sncf	country-wide intercity rails (FRA)	GTFS	SNCF	✔
zsr	country-wide rails (SVK)	MERITS	ŽSR	✗

Table 4.1: Timetable descriptions

Name	El. conns.	Cities	UG arcs	Time range	Height
air01	601489	287	4668	1 month	24374
cpru	37148	871	2415	1 day	239
cpza	60769	1108	2778	1 day	370
montr	7153	217	349	1 day	363
sncf	90676	2646	7994	1 day	488
sncf-inter	4796	366	901	1 day	209
sncf-ter	85932	2637	7647	1 day	488
zsr	932052	233	588	1 year	60308

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

Some of the timetables have time range greater then 1 day. Furthermore, even for those marked as a 1 day timetable the exact time range is different (e.g., part of the Monday timetable might be some overnight trains with arrival on Tuesday morning). 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 1 day, 6 hours [6] ($thigh_T <$ 1 day, 6h $\forall T$) from each of our dataset. See table 4.3 for details.

Name	El. conns.	Cities	UG arcs	Exact time range	Height
air01-200d	19546	200	3986	1 day, 05h:00m	766
cpru-200d	8721	200	647	0 days, 18h:45m	239
cpza-200d	13225	200	583	0 days, 19h:01m	370
montr-200d	6985	200	320	0 days, 20h:33m	363
sncf-200d	8599	200	601	1 day, 05h:29m	456
sncf-inter-200d	2283	200	466	1 day, 01h:10m	186
sncf-ter-200d	7617	200	585	1 days, 00h:02m	450
zsr-200d	2289	200	464	1 day, 03h:26m	142

Table 4.3: 200-station sub-timetables with the maximal time range of little more than one day

Also, to provide idea as to how big the time-expanded graphs can get consult table **??**.

4.2 Basic properties

First we will take a look at the optimal connection *sizes* (size is the number of el. connections) in the timetables. For a given timetable T, we will denote the average optimal connection size as $\boldsymbol{\gamma_T}$

[6] 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 time range

and will call it the **optimal connection radius** (OC radius). We computed an approximate OC radius for each of our datasets by measuring an average connection size of sufficiently many OCs. The results in table 4.4 indicate that the average OC sizes move around value $\sqrt{n}$ (with exception of the *air01* timetable).

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 (again, with 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** as $\boldsymbol{\delta_T} = min\{\delta |\ ug_T$ is δ-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 $\sqrt[4]{n}$). In table 4.5 you can see the maximal density found during the sampling.

Name	$\boldsymbol{\gamma_T}$
air01	2.4
cpru	32.3
cpza	33.2
montr	20.6
sncf	25.8
sncf-inter	8.3
sncf-ter	27.5
zsr	15.1

Table 4.4: OC radius

Name	Maximal $\boldsymbol{\delta_T}$ found
air01	34.5
cpru	4.0
cpza	3.4
montr	1.9
sncf	4.3
sncf-inter	3.1
sncf-ter	4.7
zsr	3.2

Table 4.5: Approximate density of the underlying graphs

5 Underlying shortest paths

5.1 USP

In section 2 we have defined a timetable as a set of elementary connections. While 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 [7], 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 which will repeat often throughout this section: *when carrying out an optimal connection between a pair of cities, one often goes along the same path regardless of the starting time.*

To formalize this idea, we will introduce the definition of an *underlying shortest path* - a path in UG 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 picture 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 was rather technical way of expressing a simple intuition - for a given connection, the *path* function simply outputs a sequence of visited cities. Now we can formalize the 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* $\exists t \in \mathbb{N} : p = path(c^*_{(v_1,t,v_k)}), c^*_{(v_1,t,v_k)} \in C_T$

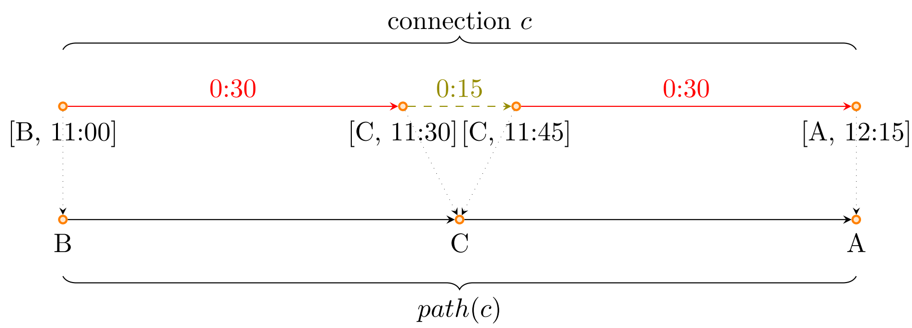

Figure 5.1: The *path* function applied on a connection to get the underlying path

[7]Maybe with exception of the airline timetables, which tend to be more dense

Please note that the terminology might be a bit misleading - an USP is not necessarily a shortest path in the given UG. Connections on a shortest path may simple require too much waiting (the el. connections simply do not follow well enough one another) and thus it might be that travelling along the paths with greater distance proof to be faster options.

5.2 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 at each stop take the first elementary connection to the next stop on the USP. This simple algorithm called *ExpandUsp* is described in algorithm 1.

Algorithm 1 ExpandUsp

Input
- timetable T
- USP $p = (v_1, v_2, ..., v_k)$
- departure time t

Algorithm

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

Output
- connection c

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* 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 $arr(e_1) < arr(e_2)$.*

Lemma 5.1. *Let T be a timetable without overtaking, (x, t, y) an EA query in this timetable and $\mathcal{P} = \{p_1, p_2, ..., p_k\}$ a set of all USPs from x to y. Define $c_i = ExpandUsp(T, p_i, t)$ to be the connection returned by the algorithm ExpandUsp 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 $\mathcal{P}$, 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:* ExpandUsp reaches city $v_1 = x$ as soon as possible (since the connection just starts there)
2. *Induction:* ExpandUsp reached city v_i as soon as possible, it then takes the first available el. connection to the next city v_{i+1}. Since the el. connections do not overtake, ExpandUsp reached the city v_{i+1} as soon as possible.

□

We would like to stress that overtaking is understood as a situation when one carrier overtakes another between *two subsequent stations.* This situation is not that common, however it is still

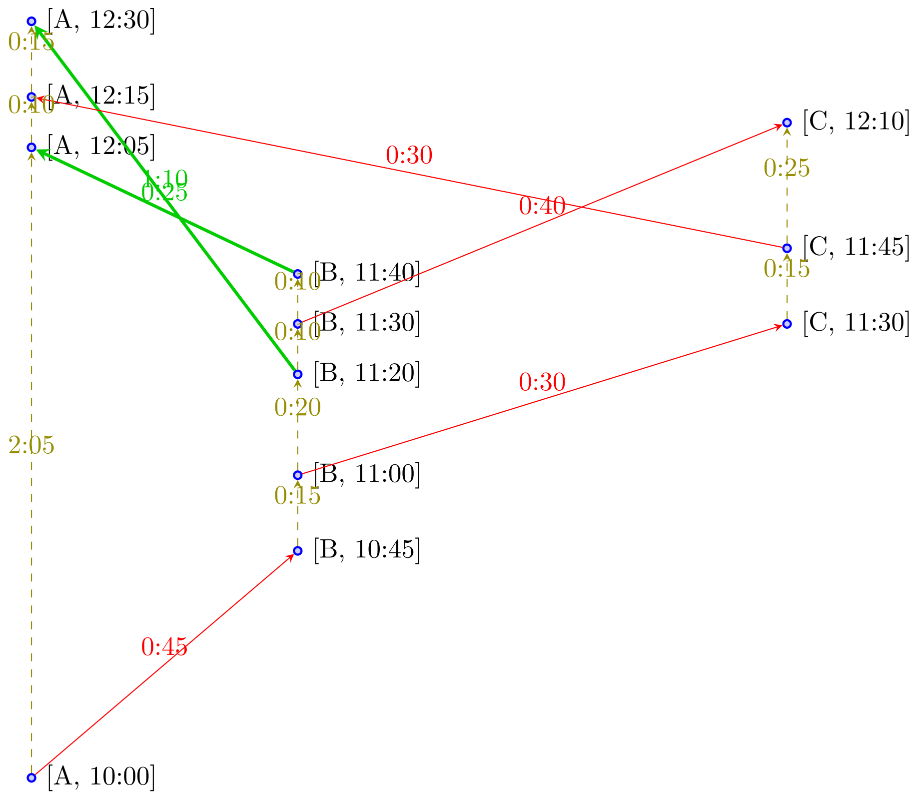

Figure 5.2: An example of **overtaking** (in thick), depicted in a TE graph

present in the real world timetables [8], as shown in table 5.1. All the same, we can simply remove the overtaken el. connections from the timetables, as they can be substituted by the quicker connection plus some waiting.

Name	**Overtaken edges (%)**
air01	1%
cpru	2%
cpza	2%
montr	1%
sncf	2%
sncf-ter	2%
sncf-inter	8%
zsr	0%

Table 5.1: Presence of overtaking in the timetables

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

[8] 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.

Algorithm 2 USP-OR query

Input

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

Pre-computed

- $\forall x, y$: set of USPs between x and y ($usps(x, y)$)

Algorithm

$c^* = null$
for all $p \in usps_{x,y}$ **do**
  $c = ExpandUsp(T, p, t)$
  $c^* =$ better out of c^* and c
end for

Output

- connection c

We will now have a look at the four parameters of this oracle based method. As for the preprocessing time, we need to find 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. The maximum number of events in one city is the height h and there is n cities, thus hn is the upper bound on the number of events. One search from a single event to all cities can be done in time $\mathcal{O}(n \log n + m)$ with a time-dependent Dijkstra's algorithm run on the time-dependent graph of our timetable (TD_T). 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 sparsity of the timetable, defined in section 4). We therefore get the **preprocessing time $\boldsymbol{\mathcal{O}(hn^2(\log n + \delta))}$**.

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 many 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 a single pair of cities?

The answer for the first question is that the average USP size is equal to the OC radius of the timetable (γ_T) defined in section 4. An upper bound for this value is $\mathcal{O}(n)$ but generally γ_T varies around $\sqrt{n}$ (see table 5.3).
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, the USP coefficient $\boldsymbol{\tau_T(x, y)} = |usps_T(x, y)|$, where $\boldsymbol{usps_T(x, y)}$ is the set of USPs between x and y. By $\boldsymbol{\tau_T}$ we will denote the average USP coefficient in timetable T.

From the table 5.2 we can see, that there are not many USPs on average, meaning that τ is usually some small number. Also, we see that it slightly increases with increasing time range (plot 5.3), but not with increasing n, the size of the timetable (plot 5.4). Thus we can consider τ to be bound by a small constant when it comes to daily timetables.

From the answers to our two questions we see that the **size of the preprocessed oracle** is $\boldsymbol{\mathcal{O}(\tau n^2 \gamma)}$.

The query time also 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 *ExpandUsp* function takes time

Name	τ	max $\tau(x, y)$
air01-200d	5.8	30
cpru-200d	7.0	64
cpza-200d	5.1	42
montr-200d	4.3	30
sncf-200d	4.3	24
sncf-inter-200d	0.6	19
sncf-ter-200d	6.1	33
zsr-200d	2.5	19

Table 5.2: Average and maximal USP coefficients

Name	avg USP size
air01-200d	3.0
cpru-200d	13.8
cpza-200d	11.1
montr-200d	20.3
sncf-200d	10.5
sncf-inter-200d	7.9
sncf-ter-200d	10.8
zsr-200d	13.7

Table 5.3: Average USP sizes vary around $\sqrt{n} \approx 14$. Note extremely low value for airline timetable - this is due to the fact that UGs of airline timetables have small-world characteristics [?]

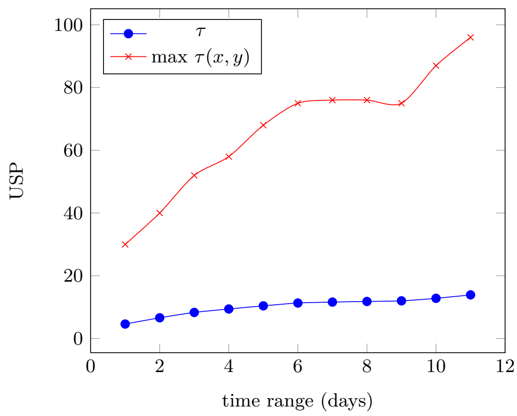

Figure 5.3: Changing of τ with increased time range in *air01* dataset. 1 day = about 800 in height

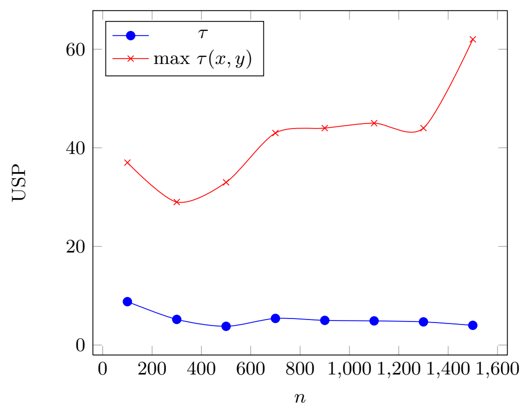

Figure 5.4: Changing of τ with increased number of stations in *sncf* dataset

linear in the size of the USP [9], leading to **query time $\mathcal{O}(\tau\gamma)$** on average. Note, that this is pretty much optimal, as τ is basically constant and we need to output the connection itself, which takes linear time in its size.

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

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

Table 5.4: The summary of the *USP-OR* algorithm parameters. The second row corresponds e.g. to the *sncf* dataset

[9] 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 $\mathcal{O}(\log h)$

5.3 USP-OR-A

With *USP-OR* the main disadvantage is its space consumption. We may decrease this space complexity by pre-computing USPs only among *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 are all cities (including x) that are reachable from x not via $\mathcal{A}$. Formally $\boldsymbol{neigh_{\mathcal{A}}(x)} = \{y | \exists$ path $p = (p_1, p_2, ..., p_k)$ from x to y in $ug_T : p_i \neq a\ \forall a \in \mathcal{A},\ i \in \{2, ..., k-1\}\}$ [10]

We define analogically **back neighbourhood** (denoted $\boldsymbol{bneigh_{\mathcal{A}}(x)}$), as nodes that could be reached in reversed UG ($\overleftarrow{ug_T}$). 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 desired properties of the access node set $\mathcal{A}$. Given a timetable T and small constants r_1, r_2 and r_3, we would like to find access node set $\mathcal{A}$ 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 [11] 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 number of local access nodes for each node is bounded by r_3:

$$|lan_{\mathcal{A}}(x)| \leq r_3, \forall x \in ct_T \tag{5.3}$$

An access node set $\mathcal{A}$ with the above mentioned properties will be called $\boldsymbol{(r_1, r_2, r_3)}$ **access node set** (AN set). We will now explain how the *USP-OR-A* (USP-OR with access 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 to find out optimal connections 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 3 and 4 and picture 5.5, where we have split the algorithms to 3 distinct phases.

[10] We leave out subscript identifying the timetable T. In situation with clear context, we may also leave out the $\mathcal{A}$ subscript

[11] We required the same for back neighbourhoods

Algorithm 3 USP-OR-A preprocessing

Input

- timetable T

Algorithm

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

Output

- output everything we have computed

Algorithm 4 USP-OR-A query

Input

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

Algorithm

```
Local front search
perform time-dependent Dijkstra from x at time t up to lan(x)
if y ∈ neigh(x) then
  output optimal connection to y obtained by Dijkstra's algorithm
end if
∀u ∈ lan(x) let ea(u) be the EA to this node (obtained by Dijkstra's algorithm)
∀u ∈ lan(x) let oc(u) be the OC to this node (obtained by Dijkstra's algorithm)
Inter-AN search
for all v ∈ blan(y) do
  oc(v) = null
  for all u ∈ lan(x) do
    for all p ∈ usps(u, v) do
      c = ExpandUsp(T, p, ea(u))
      oc(v) = better out of oc(v) and c
    end for
  end for
end for
∀v ∈ blan(y) let ea(v) = end(oc(v)) (the EA to this node)
Local back search
for all v ∈ blan(y) do
  perform time-dependent Dijkstra from v at time ea(v) to y restricted to bneigh(y)
  let fin(v) be the connection returned by Dijkstra's algorithm
end for
v* = argmin_{v∈blan(y)}{end(fin(v))}
u* = from(oc(v))
output c* = oc(u*).oc(v*).fin(v*)          # the dot (.) symbol is concatenation of connections)
```

Output

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

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 a 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

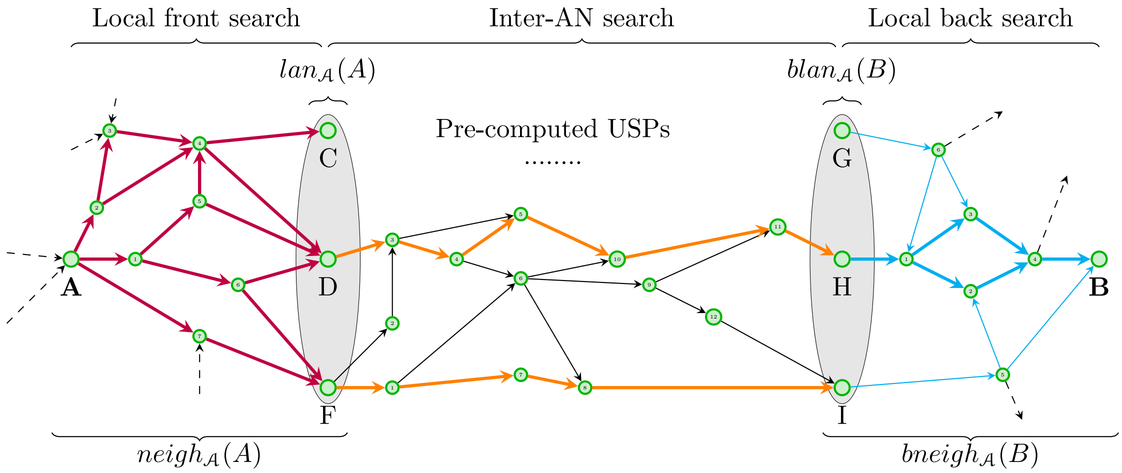

Figure 5.5: Principle of *USP-OR-A* algorithm. The arcs in **bold** mark areas that will be explored: all nodes in $neigh_{\mathcal{A}}(x)$, USPs between LANs of x and back LANs of y and the back neighbourhood of y (possibly only part of it will be explored, since the local back search goes against the direction in which the back neighbourhood was created)

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 $\leq r_2 \cdot n$. As a consequence of the Cauchy-Schwarz Inequality [**?**] the following holds for positive real numbers x_i:

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

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 $\leq \sqrt[4]{n}$ 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, depending on the timetable's density.

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

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 $\leq \sqrt{r_2 n}$, thus the total size of the (front and back) neighbourhoods is $\mathcal{O}(r_2 n^{1.5})$ [12]. 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 $\boldsymbol{\gamma_{\mathcal{A}}}$ the average optimal connection size (or equivalently, USP size) between cities in $\mathcal{A}$. This amounts to $\mathcal{O}(r_1^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}} n)$ for storage of USPs and to a total **preprocessing size** $\boldsymbol{\mathcal{O}(r_2 n^{1.5} + r_1^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}} n)}$.

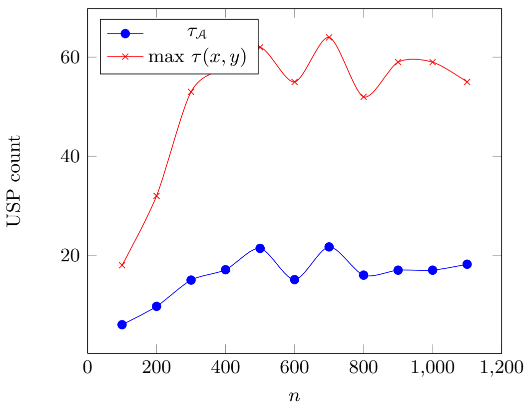

Figure 5.6: Changing of $\tau_{\mathcal{A}}$ with increased number of stations in *cpza* dataset. $\mathcal{A}$ was obtained using algorithm we will talk about later

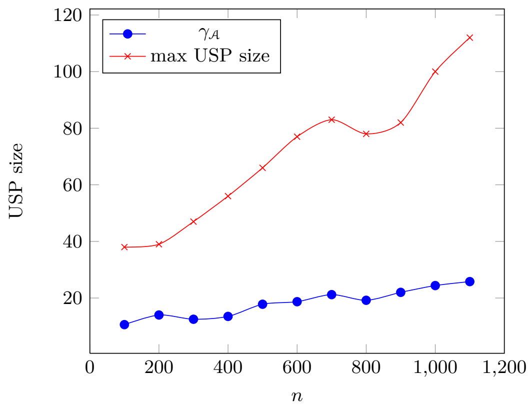

Figure 5.7: Changing of $\gamma_{\mathcal{A}}$ with increased number of stations in *cpza* dataset. $\mathcal{A}$ was obtained using algorithm we will talk about later

On a query from x at time t to y, we first perform the *local front search* (see algorithm 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 $\mathcal{O}(r_3^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}})$. Finally, from each $v \in blan(y)$ we do a time-dependent Dijkstra's algorithm, restricted to $bneigh(y)$, leading to time complexity $\mathcal{O}(r_3 \sqrt{r_2 n}(\log(\sqrt{r_2 n}) + \delta))$.

Summing up the three terms we obtain the **query time** of $\boldsymbol{\mathcal{O}(r_2 r_3 \sqrt{n}(\log(r_2 n) + \delta) + r_3^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}})}$.

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

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 5.5, 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.

[12] As r_2 will be a very small constant, we may disregard the square root

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

Table 5.5: The summary of the *USP-OR-A* algorithm parameters

5.4 Choosing the optimal 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.

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 usually 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 a **problem of the optimal access node set** and in what follows we will show that it is NP-complete.

Theorem 5.1. *The problem of the optimal access node set is NP-complete*

Proof. We will make a reduction of the *min-set cover* 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: $\bigcup\limits_{1 \leq i \leq k} S_i = U$
- 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 ($\bigcup\limits_{S_i \in \mathcal{S}^*} S_i = U$). We will 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_i vertices (the value of β_i will be discussed later) named m_j and from each set S_i we will make a vertex s_i and vertex s'_i. We will connect all vertices of m_j to $s_i \iff j \in S_i$. Finally, for we will connect s_i to s'_i for $1 \leq i \leq k$.

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

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

- $S_{14} = \emptyset$ (a dummy set)

For this instance of min set-cover, we construct the graph depicted on picture 5.8.

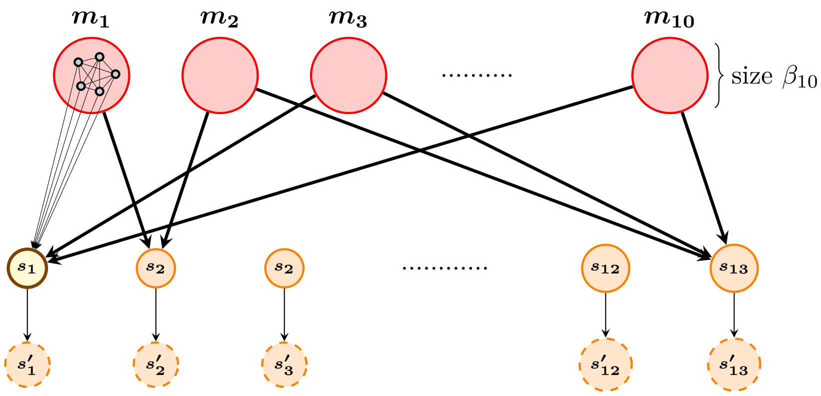

Figure 5.8: The principle of the reduction. In m_i, there are actually complete graphs of β_i vertices (as shown for m_1). **Thick** arcs represent arcs from all the vertices of respective m_i. The s_i vertices are connected to their s'_i versions. If e.g. s_1 is selected as an access node, s'_1 is no longer part of any neighbourhood

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 $\beta_i + 2\alpha_i - 1 \leq \sqrt{n}$ but $\beta_i + 2\alpha_i > \sqrt{n}$. This would mean that if at least one s_j that is connected to m_i is chosen as an access node, the neighbourhood for nodes in m_i will be still $\leq \sqrt{n}$, but if none of them is chosen, the neighbourhood will be just too large. We will return to values β_i later.

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 some of the s_j that m_i is connected to as AN, the neighbourhood of any non-access node in m_i will be too large. Therefore, 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 $\leq \sqrt{n}$.

It remains to show how to choose values β_i. As by the first condition $\beta_i \leq \sqrt{n} - 2\alpha_i + 1$, we need to have sufficiently big n. 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 $n' = nextSquare(max\{w-1, 2k+m\})$. We create the s_j and s'_j vertices and 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 $\sqrt{n} - 2\alpha_i$ 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 $|m_i| = \beta_i \ \forall i$

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. $\square$

6 Neural network approach

7 Application TTBlazer

8 Conclusion

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 1: 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 2: 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 3: 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 [13], see the listing 4 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 4: TD file format

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

References

[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.

[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.

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