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

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

In this thesis we deal with queries for optimal connections in timetables on which we have carried out some preprocessing. Based on the analysis of the properties of real-world timetables, we developed exact methods that answer the queries considerably faster then the time-dependent Dijkstra's algorithm implemented with Fibonacci heap priority queue. More specifically, our method *USP-OR-A* with space complexity $\mathcal{O}(n^{1.5})$ achieves average query time $\mathcal{O}(\sqrt{n}\log n)$, outperforming the time-dependent Dijkstra's algorithm up to 7 times in our largest datasets.

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

Abstrakt

V tejto práci sa zaoberáme hľadaním optimálnych spojení v cestovných poriadkoch, na ktorých sme si predpočítali určité informácie. Na základe analýzy reálnych cestovných poriadkov sme vyvinuli exaktné metódy, ktoré na dotaz na optimálne spojenie odpovedajú podstatne rýchlejšie ako časovo závislá implementácia Dijkstrovho algoritmu využívajúca prioritnú frontu na základe Fibonacciho haldy. Presnejšie, náš algoritmus *USP-OR-A* s priestorovou zložitosťou $\mathcal{O}(n^{1.5})$ dosahuje časovú zložitosť odpovede na dotaz $\mathcal{O}(\sqrt{n}\log n)$, prekonávajúc časovo závislý Dijkstrov algoritmus takmer 7 krát v našom najväčšom cestovnom poriadku.

Klúčové slová: **optimálne spojenie**, **cestovný poriadok**, **Dijkstrov algoritmus**, **Dištančné orákulá**, **podkladové najkratšie cesty**

Contents

1 Introduction

2 Preliminaries

3 Related work

4 Data & analysis

5 Underlying shortest paths

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 [1], 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* $\exists t \in \mathcal{N} : p = path(c^*_{(v_1,t,v_k)}), c^*_{(v_1,t,v_k)} \in C_T$

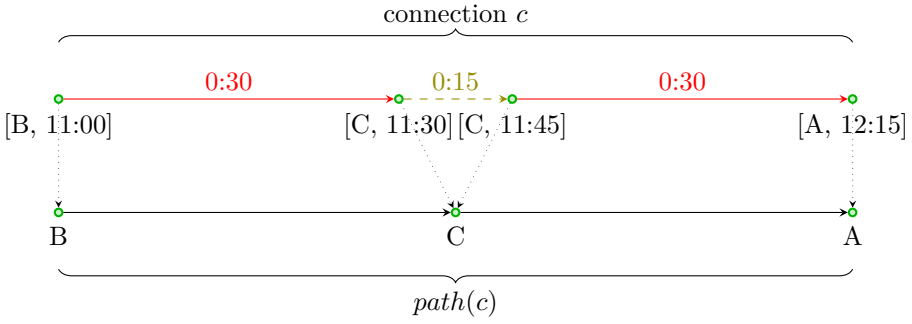

Figure 5.1: The *path* function applied on a connection to get the underlying path.

Please note that the terminology might be a bit misleading - an USP is not necessarily a shortest path in the given UG. Connections on a shortest path may simple require too much waiting (the

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

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.

5.1 *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 at each stop take the first elementary connection to the next stop on the USP. This simple algorithm 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

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 $c_i = Expand(T, p_i, t)$ 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 $\mathcal{P}$, 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:* 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 el. connection to the next city v_{i+1}. Since the el. 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 one carrier overtakes another between *two subsequent stations*. This situation is not that common, however it is still present in the real world timetables [2], as shown in table 5.1. All the same, we can simply remove the

[2] 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.

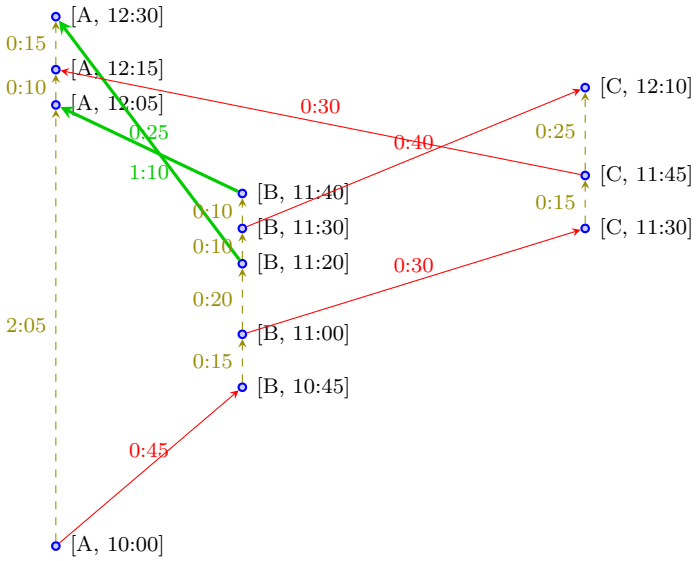

Figure 5.2: An example of **overtaking** (in thick), depicted in a TE graph.

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 precompute 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.

Algorithm 5.2 *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**
  $c = Expand(T, p, t)$
  c^* = better out of c^* and c
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 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 TD 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 $\boldsymbol{\mathcal{O}(\tau n^2 \gamma)}$.

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 [**?**].

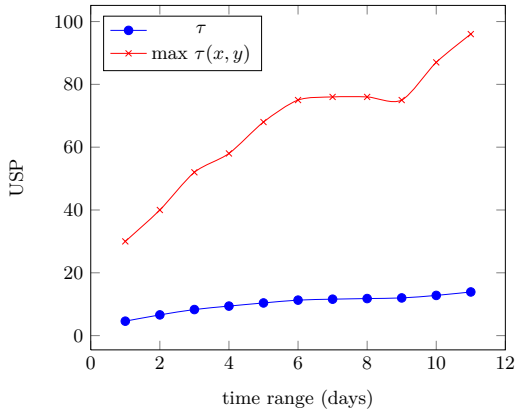

Figure 5.3: Changing of τ with increased time range in *air01* dataset. 1 day = about 800 in height.

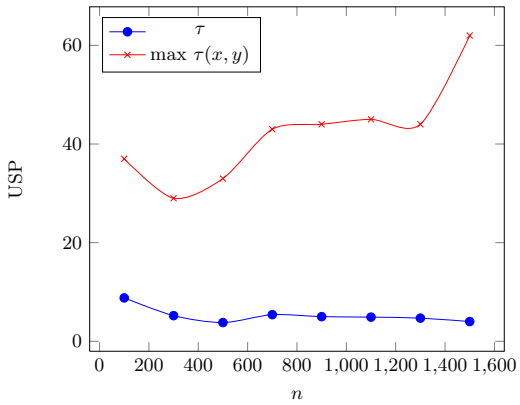

Figure 5.4: Changing of τ with increased number of stations in *sncf* dataset.

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 *Expand* function takes time linear in the size of the USP [3], 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	$\mathcal{O}(hn^2(\log n + \delta))$	$\mathcal{O}(\tau n^2 \gamma)$	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.

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

5.2 *USP-OR-A*

With *USP-OR* the main disadvantage is its space consumption. We may decrease this space complexity by pre-computing USPs only 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* $ug_T : p_i \neq a\ \forall a \in \mathcal{A},\ i \in \{2, ..., k-1\}\}$ [4]

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 [5] 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 for cities not in $\mathcal{A}$ is at most r_3

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

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 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

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

[5] We required the same for back neighbourhoods

neighbourhood. For more details, see algorithms 5.3 and 5.4 and picture 5.5, where we have split the algorithms to 3 distinct phases.

Algorithm 5.3 *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 5.4 *USP-OR-A* query

Input

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

Algorithm

let $lan(x) = x$ if $x \in \mathcal{A}$
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 conn. 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 $ea(v) = end(oc(v))$
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
$v^* = argmin_{v \in blan(y)}\{end(fin(v))\}$
$u^* = from(oc(v^*))$
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 $c^*_{(x,t,y)}$

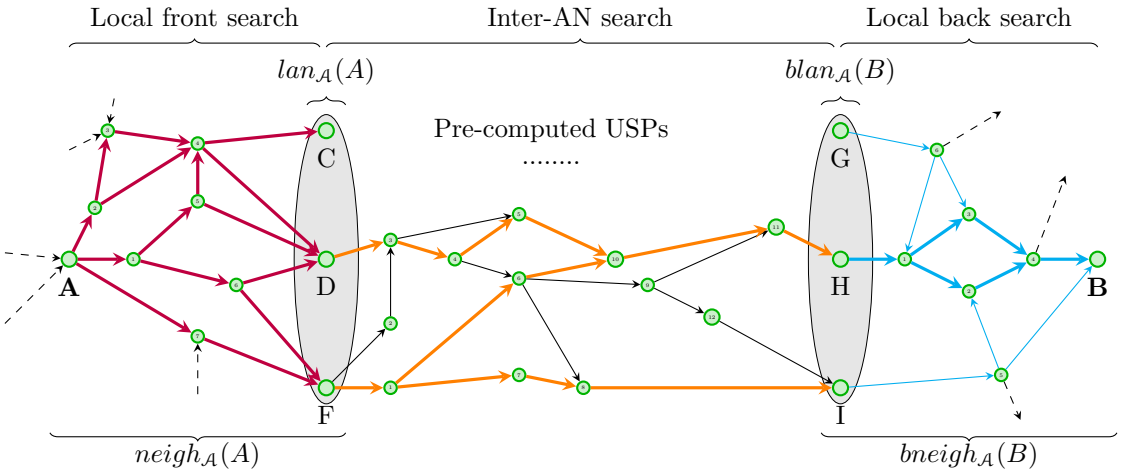

Figure 5.5: Principle of *USP-OR-A* algorithm. The arcs in **bold** mark areas that will be explored: all nodes in $neigh_{\mathcal{A}}(x)$, USPs between LANs of x and back LANs of y and the back neighbourhood of y (possibly only part of it will be explored, since the local back search goes against the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

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 a city from $\mathcal{A}$. There is $\mathcal{O}(r_1 h\sqrt{n})$ such events which leads to the time complexity $\mathcal{O}(r_1 hn^{1.5}(\log n+\delta))$. 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 $\leq r_2 \cdot n$. As a consequence of the Cauchy-Schwarz Inequality [**?**] the following holds for positive real numbers x_i:

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

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 $\leq \sqrt[4]{n}$ 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.

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

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

$$n\sqrt{n} + \delta n \sqrt{r_2 n} \leq$$

$$\delta r_2 n^{1.5}$$

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 $\leq \sqrt{r_2 n}$, thus the total size of the (front and back) neighbourhoods is $\mathcal{O}(r_2 n^{1.5})$ [6]. 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 $\boldsymbol{\mathcal{O}(r_2 n^{1.5} + r_1^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}} n)}$**.

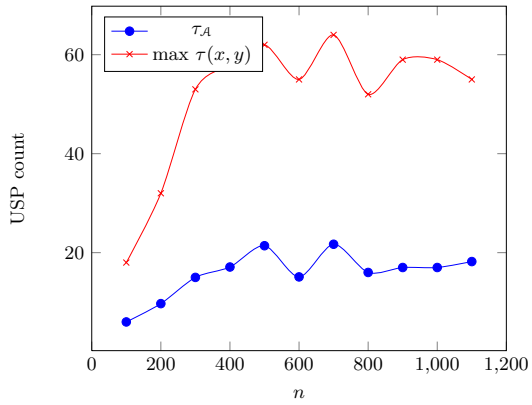

Figure 5.6: Changing of $\tau_{\mathcal{A}}$ with increased number of stations in *cpza* dataset. $\mathcal{A}$ was obtained using algorithm *locsep* we will talk about later

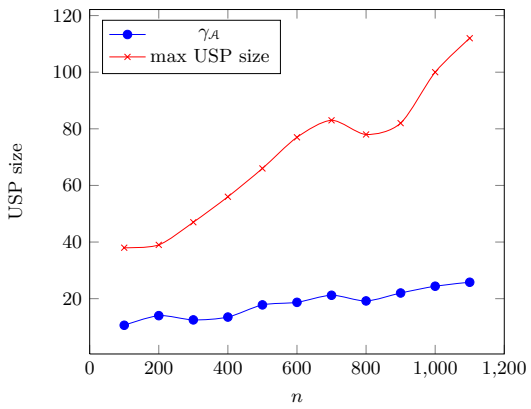

Figure 5.7: Changing of $\gamma_{\mathcal{A}}$ with increased number of stations in *cpza* dataset. $\mathcal{A}$ was obtained using algorithm *locsep* we will talk about later

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 $\mathcal{O}(r_3^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}})$. Finally, from each $v \in blan(y)$ we do a TD Dijkstra, 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 $\boldsymbol{\mathcal{O}(r_2 r_3 \sqrt{n}(\log(r_2 n) + \delta) + r_3^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}})}$.

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

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 5.5, 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.

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.

5.2.2 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 picture 5.8). 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.

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. □

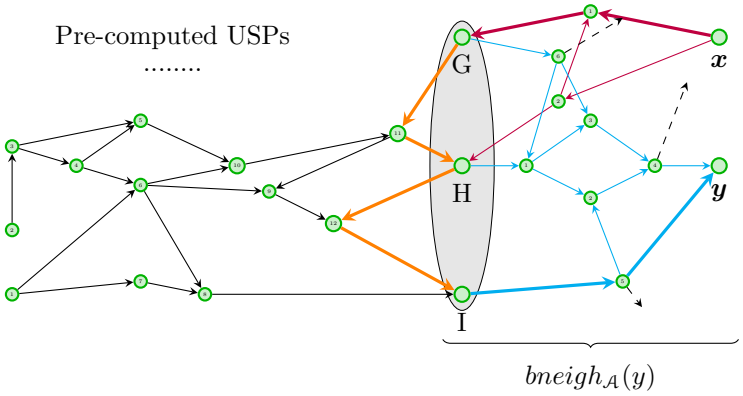

Figure 5.8: On the picture $lan(x) = \{G, H\}$ and $blan(y) = \{G, H, I\}$. In **thick** we have highlighted the optimal connection. The connection to H is sub-optimal after the *local front search* phase, however the optimal connection to y (and to H and I as well) leads through $lan^*(x)$ (some of x's local access nodes to which we have an optimal connection after the *local front search*. Particularly, it goes through G).

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 a 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 picture 5.9 depicts this.
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.

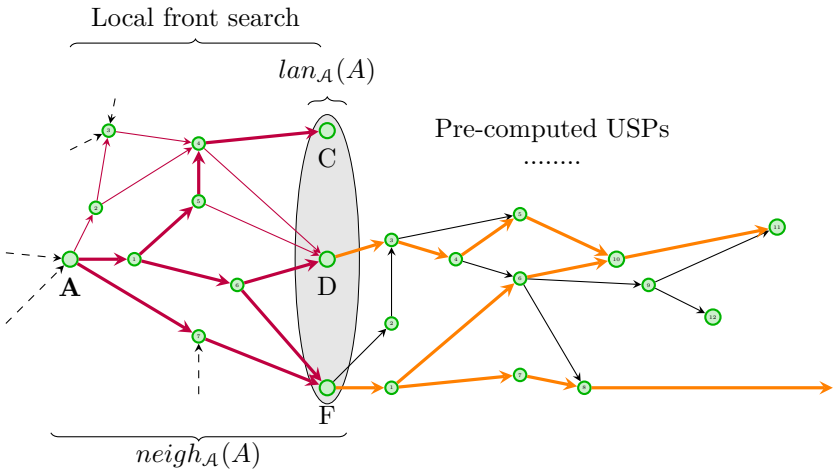

Figure 5.9: Using USP trees (**thick** non-dashed arcs in $\boldsymbol{neigh_{\mathcal{A}}(x)}$) to decrease the explored area in *local front search*. A full neighbourhood search is done only when $y \in neigh(x)$.

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 $s_i \iff j \in S_i$. 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 depicted on picture 5.10.

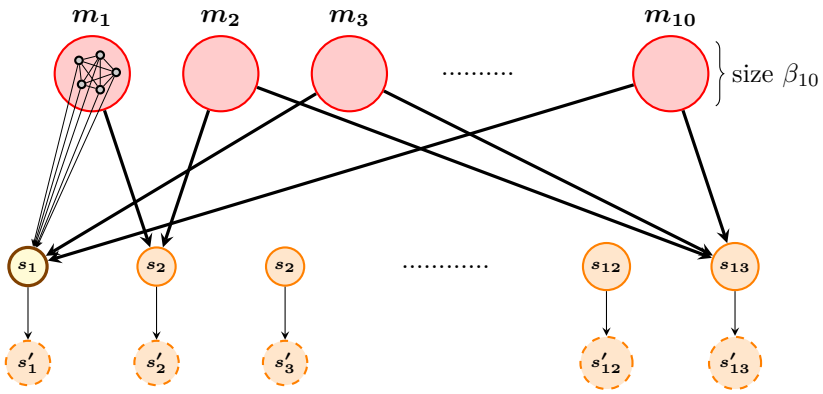

Figure 5.10: The principle of the reduction. In m_i, there are actually complete graphs of β_i vertices (as shown for m_1). **Thick** arcs represent arcs from all the vertices of respective m_i. The s_i vertices are connected to their s'_i versions. If e.g. s_1 is selected as an access node, s'_1 is 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 $\beta_i \geq 2$ 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 $\leq \sqrt{n}$, but if none of them is chosen, the neighbourhood will be just over $\sqrt{n}$. For now we will assume that we have constructed the graph in such a way that all β_i satisfy the mentioned inequalities. We will return to construction of the graph at the end.

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 $\leq \sqrt{n}$, 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 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 [7] of the respective node x: $\boldsymbol{deg(x)} = deg_{in}(x) + deg_{out}(x)$.
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 [**?**]. 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:

1. $\mathcal{A}_{deg}(k)$ is (r_1, r_2, r_3) AN set with $r_2 \leq 1$ (the average square of neighbourhoods is $\leq \sqrt{n}$). Denote such r_1 as $\boldsymbol{r_{deg}^{avg}}$ [8].
2. $\mathcal{A}_{deg}(k)$ is (r_1, r_2, r_3) AN set where $\forall x \in ct_T$: $|neigh_{\mathcal{A}_{deg}(k)}(x)| \leq \frac{3\sqrt{n}}{2}$ and $|bneigh_{\mathcal{A}_{deg}(k)}(x)| \leq \frac{3\sqrt{n}}{2}$ (all neighbourhoods are large at most $\frac{3\sqrt{n}}{2}$). Denote such r_1 as $\boldsymbol{r_{deg}^{max}}$.

We define similarly $\boldsymbol{r_{bc}^{avg}}$ and $\boldsymbol{r_{bc}^{max}}$. The resulting values for datasets *sncf* and *cpru* could be seen on plots 5.11.

[7] In-degree is the number of arcs going towards to node and out-degree the number of outgoing arcs.

[8] Intuitively - r_{deg}^{avg} is the smallest r_1 such that $r_1\sqrt{n}$ highest-degree nodes selected as ANs are enough to satisfy that the average square of neighbourhoods is $\leq \sqrt{n}$

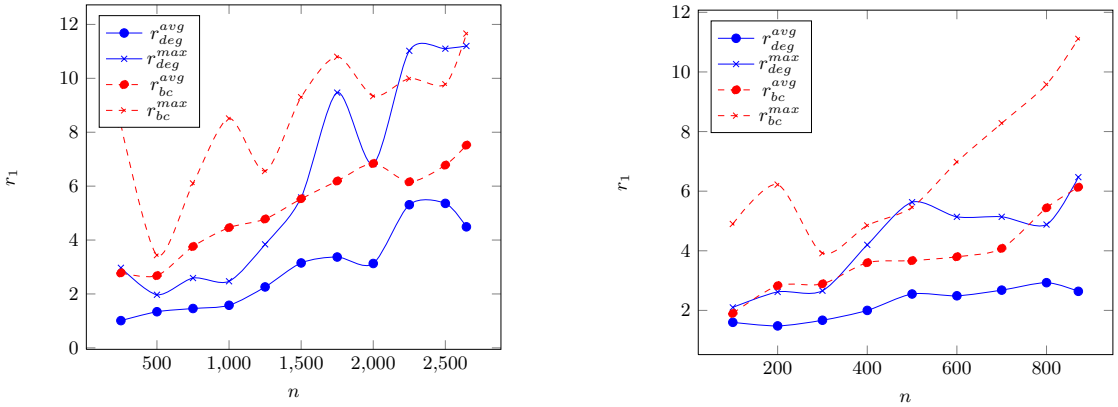

Figure 5.11: Necessary sizes of access node sets ($|\mathcal{A}| = r_1\sqrt{n}$) based on high-degree/high-BC cities. Datasets *sncf* (left) and *cpru* (right). Notice the occasional "roller coaster" bumps (especially on the left plot) - an explanation of this phenomena is that in the immediately smaller sub-timetable we have erased just the high-degree node that proved to be a good access node, and which now must be substituted by many other access nodes.

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 the 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 underlying graph ug_T of given timetable T that evaluates its vertices based on how good local separators they are.

The algorithm will work in iterations, each of them resulting in a selection [9] of a city with highest score to the access node set $\mathcal{A}$. Similarly to the previous approach that used degree/BC, we consider two stopping criterion:

1. $\mathcal{A}$ is (r_1, r_2, r_3) AN set with $r_2 \leq 1$ (the average square of neighbourhoods is $\leq \sqrt{n}$). Denote $\boldsymbol{r_{locsep}^{avg}} = r_1$ when this stopping criterion is met.
2. $\forall x \in ct_T$: $|neigh_{\mathcal{A}}(x)| \leq \frac{3\sqrt{n}}{2}$ and $|bneigh_{\mathcal{A}}(x)| \leq \frac{3\sqrt{n}}{2}$ (all neighbourhoods are large at most $\frac{3\sqrt{n}}{2}$). Denote $\boldsymbol{r_{locsep}^{max}} = r_1$ when this stopping criterion is met.

The algorithm using the first stopping criterion will be referred to as *Locsep* and the one using the second criterion as *Locsep Max*. The algorithms differ only in the stopping criterion, therefore what follows is common to both of them.

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 will be used also in the computation of the **potential** (a score) of the cities, which is the second phase of the algorithm.

For a city x, we will compute its potential $\boldsymbol{p_x}$ in the following way: we explore an area $\boldsymbol{A_x}$ of

[9] Actually, in our implementation, we allow an occasional de-selection 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.

$\sqrt{n}$ 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. However $x \notin A_x$ 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.

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., how much we wish to reduce it so that it is $\leq \sqrt{n}$. 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. We do the similar thing for $y \in fn(x)$ (we use $\overleftarrow{ug_T}$ instead of ug_T, $bneigh(y)$ instead of $neigh(y)$ etc...). For an illustration of potential computing, see picture 5.12.

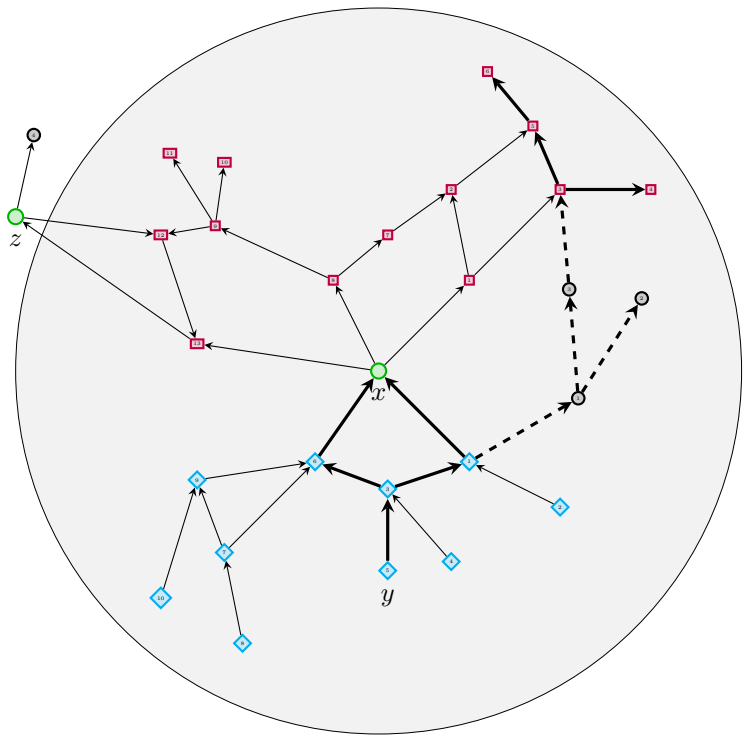

Figure 5.12: The principle of computing potentials in Locsep algorithm. We explored an area of $\sqrt{n}$ nearest cities (in terms of hops) around x. Access nodes (like z) and cities behind them are ignored. Little squares are nodes from $fn(x)$ and diamonds are part of $bn(x)$. From y we run a forward 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.

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

$\mathcal{A} = \emptyset$
$ct' = ct_T$
while $r_2 > 1$ **do**
  $\forall x \in ct'$: compute $neigh_{\mathcal{A}}(x)$, $bneigh_{\mathcal{A}}(x)$
  $\forall x \in ct'$: compute p_x
  $x' = argmin_x\{p_x\}$
  $\mathcal{A} = \mathcal{A} \cup \{x'\}$
  $ct' = neigh_{\mathcal{A}}(x') \cup bneigh_{\mathcal{A}}(x')$
end while

Output

- AN set $\mathcal{A}$, such that $|\mathcal{A}| = k^{avg}_{locsep}$

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 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 [10].

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 in the first two phases we have to do a lot of re-computations. 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 $\frac{\sqrt{n}}{2}$ nodes with the highest degree ($\mathcal{A}_{deg}(\frac{\sqrt{n}}{2})$) - enough to speed-up the computation but not influencing the resulting AN set too much.

The access node sets chosen with the *Locsep* algorithm were much smaller then those selected by the previous approaches and with the increasing number of stations, the r^{avg}_{locsep} value was found to increaser only slightly, as could be seen from plots 5.13. The average number of local access nodes for each city was also found to be very small and increasing only very little with increasing n.

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

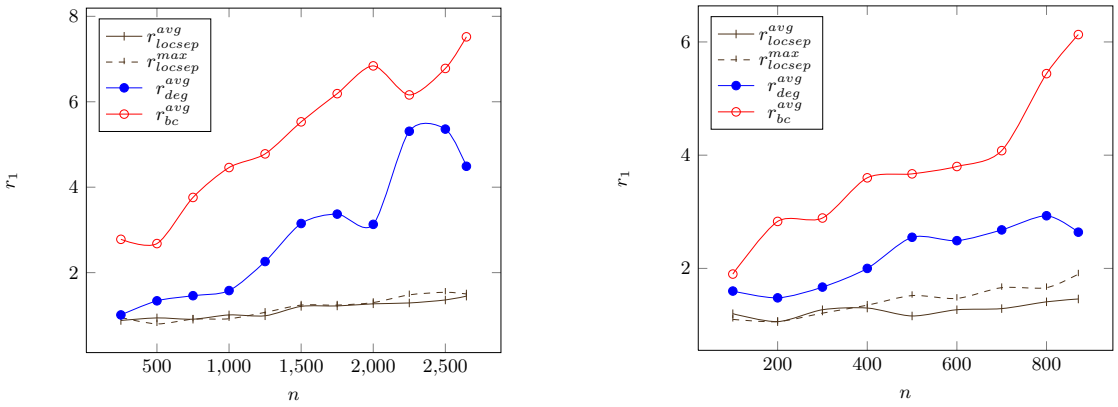

Figure 5.13: The necessary size of $\mathcal{A}$ when selecting ANs based on degree, BC or *Locsep* potential. Datasets *sncf* (left) and *cpru* (right). An ideal situation would be a constant or non-increasing function, to which *Locsep* comes closest.

To sum up, in *all* of our datasets [11], each scaled to *various* sizes, we were always able to find (r_1, r_2, r_3) access node set $\mathcal{A}$ with the *Locsep* algorithm, such that:

- $r_1 \leq 2$
- $r_2 \leq 1$
- $r_3 \leq 4$
- $\mathcal{A}$ can be found in $\mathcal{O}(\delta n^{2.5})$

Therefore, when we use *USP-OR-A* together with *Locsep* on our timetables, we achieve parameters as described in table 5.6.

USP-OR-A + Locsep	***prep***	***size***	***qtime***	***stretch***
Our timetables	$\mathcal{O}(\delta n^{2.5})$	$\mathcal{O}(n^{1.5})$	avg. $\mathcal{O}(\sqrt{n}\log n)$	1

Table 5.6: Parameters for *USP-OR-A* with *Locsep*.

5.4 Performance and comparisons

In this subsection we give the results of the performance of our algorithms on our datasets. We focus on query time and space complexity of the preprocessed oracles. 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 algorithm. We will have a similar measure for the size of the preprocessed data, which we compare against the amount of data needed to represent the actual timetable itself.

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.*

5.4.1 Performance of *USP-OR*

Query time-wise, *USP-OR* outperforms time-dependent Dijkstra's algorithm almost 80 times (on the subset of *sncf-ter* dataset). However, this was at the cost of high space consumption of the

[11] Except *air01*, which is a special type of timetable

method, in some cases requiring almost 400 times more memory then necessary for storage of the time-dependent graph. Therefore, we were not even able to preprocess the method for some of our bigger datasets. Table 5.7 gives a good overview of achieved speed-ups and size-ups.

Name	***n***	***spd***	***szp***
*cpru**	700	14.5	396.7
*cpza**	700	14.3	265.1
montr	217	8.8	61.1
*sncf**	1000	64.8	106.2
sncf-inter	366	27.0	30.3
*sncf-ter**	1000	78.3	87.4
zsr (daily)	233	19.3	60.8

Table 5.7: 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).

In both timetables from *cp.sk* we obtained quite similar results. The query time of *USP-OR* mildly rises with increasing n. This is due to two (not surprising) facts:

- The USP coefficient gets slightly bigger with increasing n
- The OC radius increases too with increasing n

The speed-up was up to 15, however, at the cost of very high space complexity, which made it possible to try out only timetables of the size up to 700. With the *montr* dataset we got similar results, but on a smaller scale.

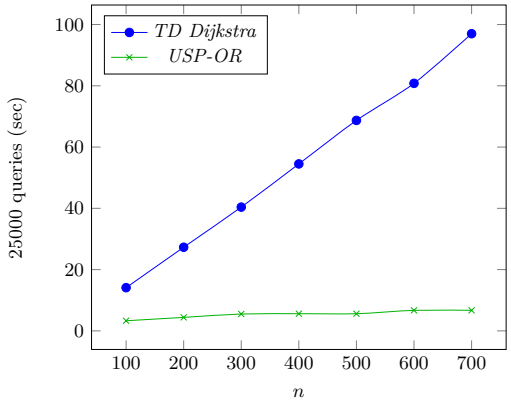

Figure 5.14: **Query time** of *USP-OR* algorithm compared to TD Dijkstra on the ***cpru*** dataset. **Changing *n***.

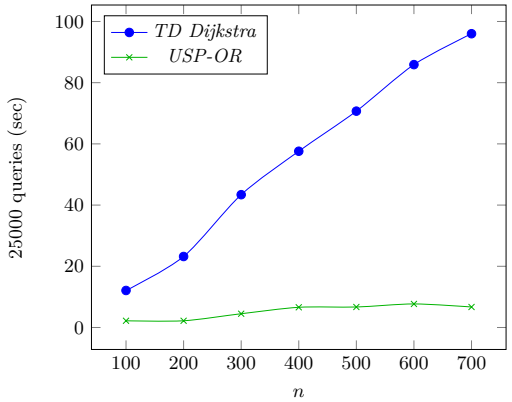

Figure 5.15: **Query time** of *USP-OR* algorithm compared to TD Dijkstra on the ***cpza*** dataset. **Changing *n***.

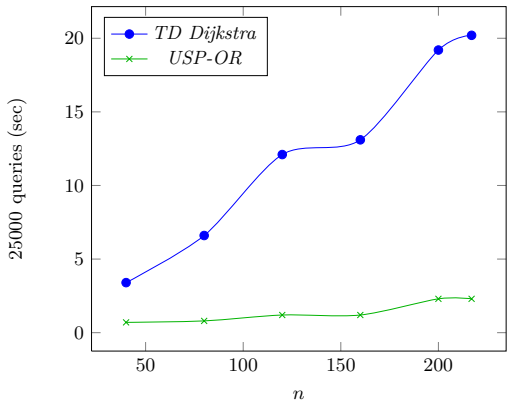

Figure 5.16: **Query time** of *USP-OR* algorithm compared to TD Dijkstra on the ***montr*** dataset. **Changing n.**

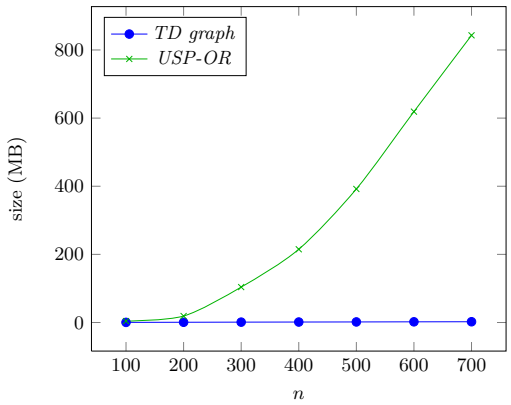

Figure 5.17: **Size** (in MB) of the oracle for *USP-OR* vs. size of TD graph on ***cpza*** dataset. **Changing n.**

In the datasets from SNCF, an interesting thing was that the the query-time actually decreased with increased size. This was due to the average USP coefficient getting smaller in bigger datasets while OC radius not increasing too much. We measured here the speed-ups of up to 80 for *sncf-ter*, with smaller size-ups then in case of *cp.sk* timetables.

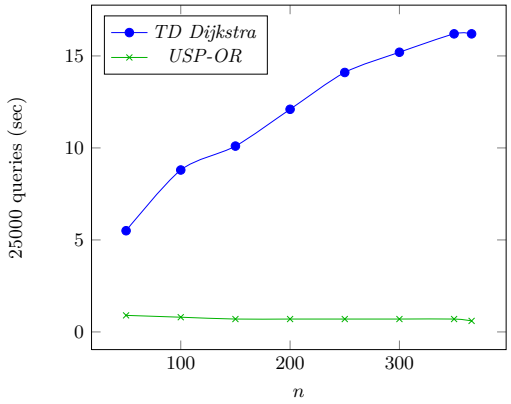

Figure 5.18: **Query time** of *USP-OR* algorithm compared to TD Dijkstra on the ***sncf-inter*** dataset. **Changing n.**

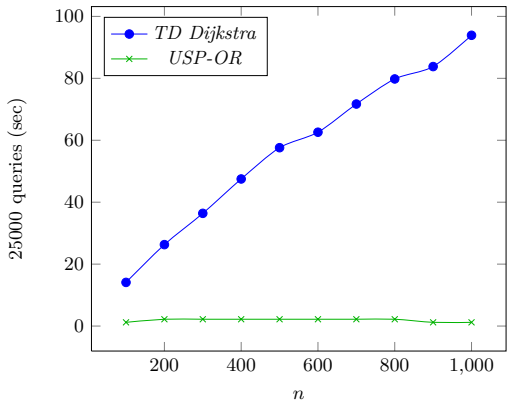

Figure 5.19: **Query time** of *USP-OR* algorithm compared to TD Dijkstra on the ***sncf-ter*** dataset. **Changing n.**

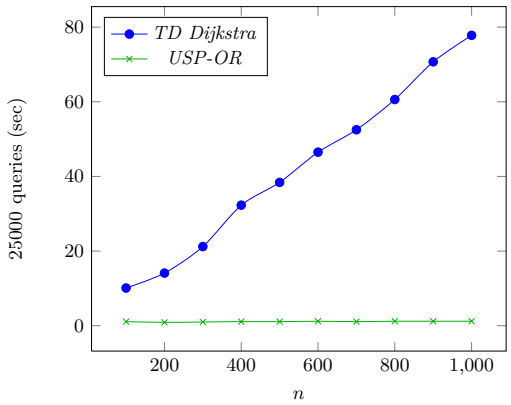

Figure 5.20: **Query time** of *USP-OR* algorithm compared to TD Dijkstra on the ***sncf*** dataset. **Changing n.**

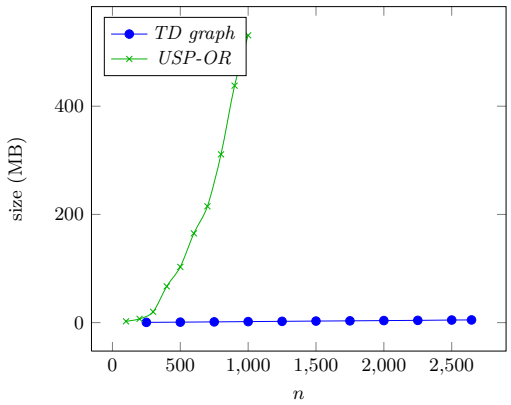

Figure 5.21: **Size** (in MB) of the oracle for *USP-OR* vs. size of TD graph on ***sncf*** dataset. **Changing n.**

On the *zsr* dataset we measured how increased time range influences the query time. You may see that for both algorithms the query time almost stops increasing at some point - this is because (informally) adding time range no longer brings along new optimal connections (or underlying shortest paths in case of *USP-OR*).

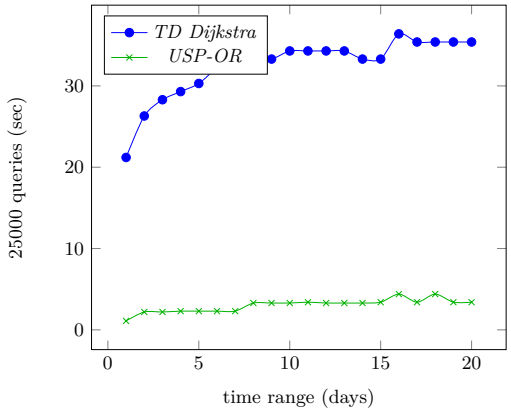

Figure 5.22: **Query time** of *USP-OR* algorithm compared to TD Dijkstra on the ***zsr*** dataset. **Changing r.**

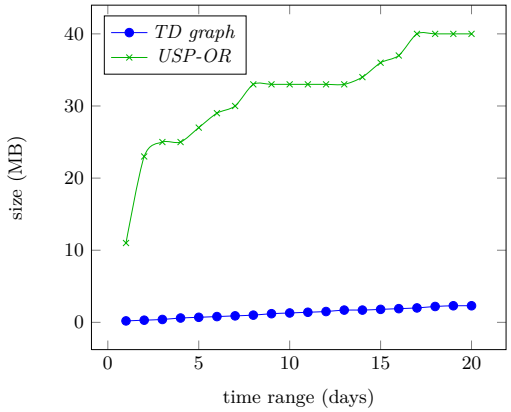

Figure 5.23: **Size** (in MB) of the oracle for *USP-OR* vs. size of TD graph on ***zsr*** dataset. **Changing r.**

5.4.2 *USP-OR-A* with *Locsep*

With *USP-OR-A*, the speed-ups were no longer so high as in case of *USP-OR-A*, but neither were the size-ups, so we could fully try out all of our datasets. The maximum speed-up was achieved in *sncf-ter*, where the *USP-OR-A* outperformed *TD Dijkstra* almost 7 times. In our datasets, the preprocessed oracle of *USP-OR-A* did not need more than 3 times the size of the TD graph. Table 5.8 provides an overview of achieved speed-ups and size-ups.

Name	n	*spd*	*szp*
cpru	871	1.8	2.97
cpza	1108	1.9	2.52
montr	217	1.6	1.14
sncf	2646	6.3	3.0
sncf-inter	366	3.9	1.59
sncf-ter	2637	6.9	2.43
zsr (daily)	233	2.5	1.99

Table 5.8: Speed-ups and size-ups of the *USP-OR-A* with *Locsep* for the whole timetables (for those marked with asterisk we took only a subset of n stations, as we were limited by the space).

The bus timetables proved to be a bigger challenge for *USP-OR-A*, achieving milder speed-ups and requiring more memory then railways timetables. However, bigger timetables would be necessary to obtain more relevant results.

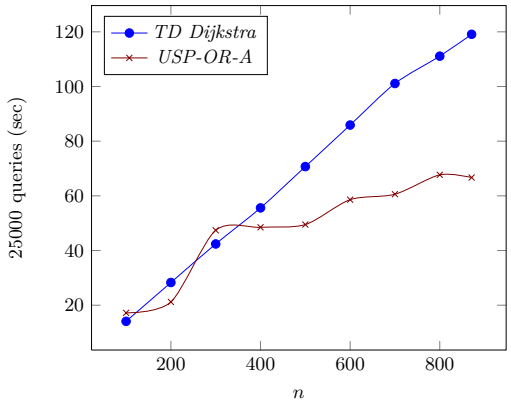

Figure 5.24: **Query time** of *USP-OR-A* with *Locsep* compared to TD Dijkstra on the ***cpru*** dataset. **Changing n.**

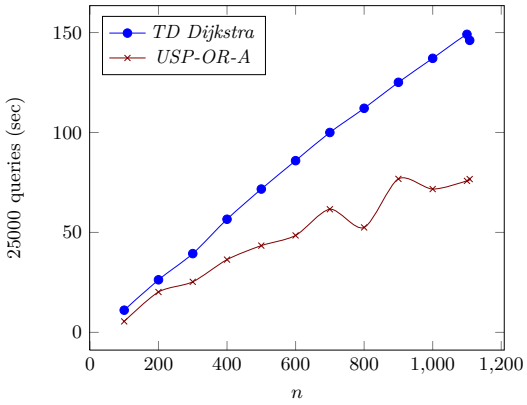

Figure 5.25: **Query time** of *USP-OR-A* with *Locsep* compared to TD Dijkstra on the ***cpza*** dataset. **Changing n.**

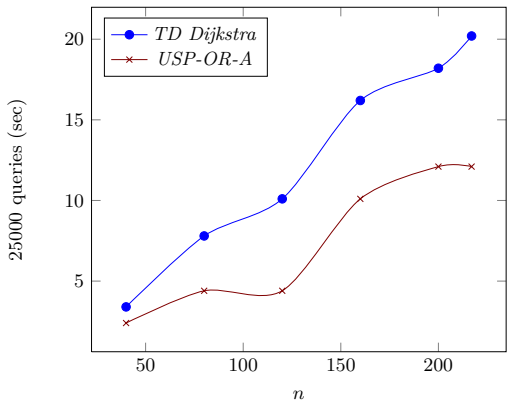

Figure 5.26: **Query time** of *USP-OR-A* with *Locsep* compared to TD Dijkstra on the ***montr*** dataset. **Changing n.**

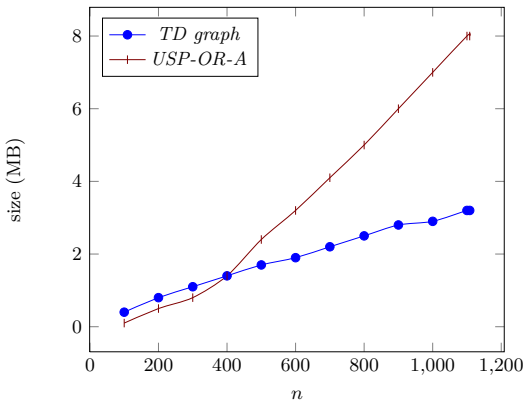

Figure 5.27: **Size** (in MB) of the oracle for *USP-OR-A* with *Locsep* vs. size of TD graph on ***cpza*** dataset. **Changing n.**

In our biggest datasets, we achieved the best speed-ups while the size-up still stayed relatively small (though here we better see its tendency to increase as $n^{1.5}$). It would be interesting to try out even bigger datasets as the speed-up was gradually increasing with increasing n.

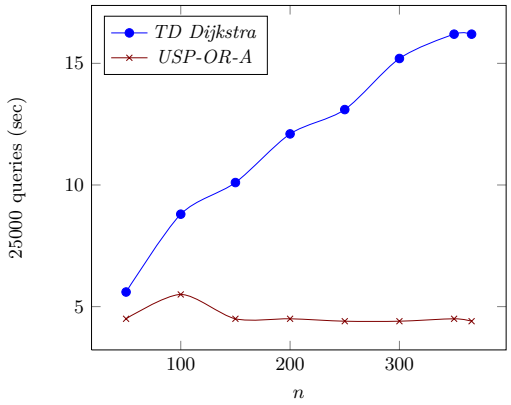

Figure 5.28: **Query time** of *USP-OR-A* with *Locsep* compared to TD Dijkstra on the ***sncf-inter*** dataset. **Changing n.**

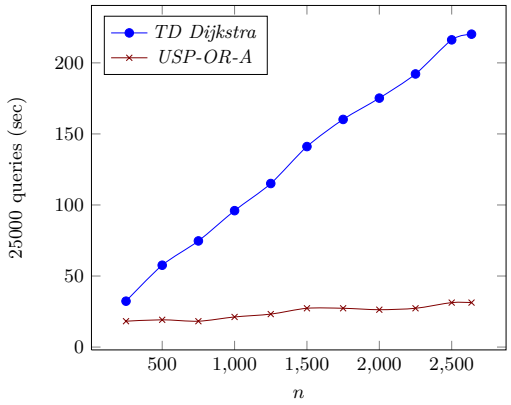

Figure 5.29: **Query time** of *USP-OR-A* with *Locsep* compared to TD Dijkstra on the ***sncf-ter*** dataset. **Changing n.**

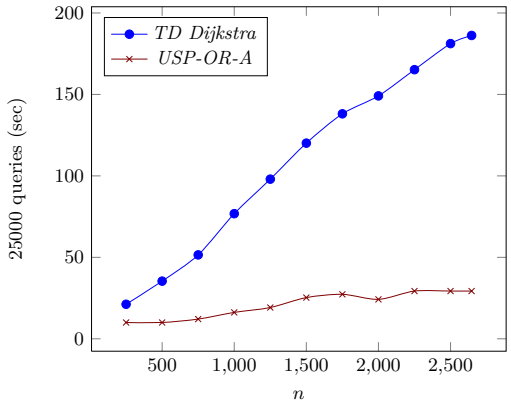

Figure 5.30: **Query time** of *USP-OR-A* with *Locsep* compared to TD Dijkstra on the ***sncf*** dataset. **Changing n.**

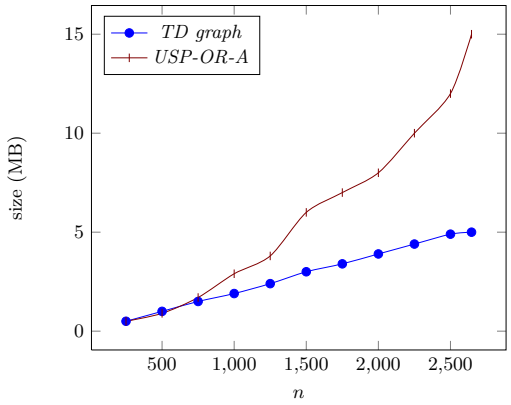

Figure 5.31: **Size** (in MB) of the oracle for *USP-OR-A* with *Locsep* vs. size of TD graph on ***sncf*** dataset. **Changing n.**

Finally, on the *zsr* timetable, we see two things:

- The space-complexity of *USP-OR-A* is left pretty much unaffected with increased time range.
- The speed-up decreases (since with increased time range, there are generally more USPs between pairs of cities which we have to try out during the query)

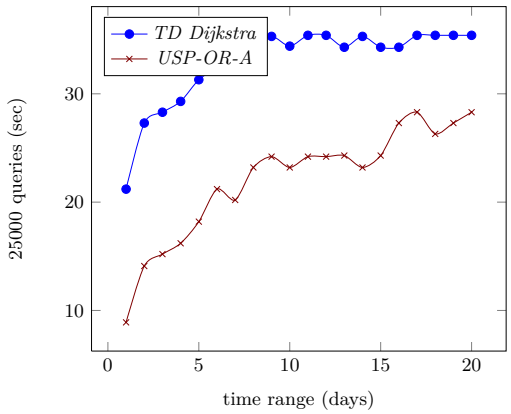

Figure 5.32: **Query time** of *USP-OR-A* with *Locsep* compared to TD Dijkstra on the ***zsr*** dataset. **Changing *r***.

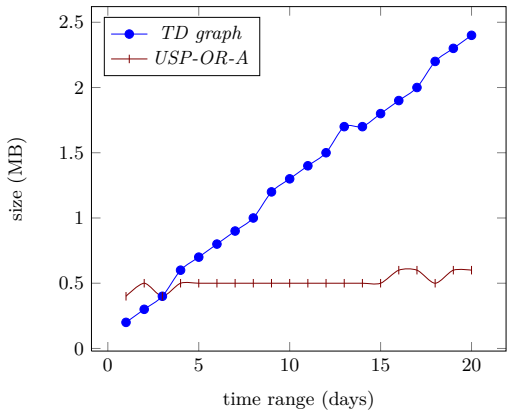

Figure 5.33: **Size** (in MB) of the oracle for *USP-OR-A* with *Locsep* vs. size of TD graph on ***zsr*** dataset. **Changing *r***.

5.4.3 *USP-OR-A* with *Locsep Max*

We also tried out *USP-OR-A* with *Locsep Max* to see if the difference in the stopping criterion of *Locsep* would influence the query times. It did help, but the difference in the performance is minimal, therefore we list only the table summarizing the speed-ups and size-ups (5.9) and the details for datasets *cpza* and *sncf*.

Name	***n***	***spd***	***szp***
cpru	871	2.1	4.5
cpza	1108	2.1	3.1
montr	217	2.1	1.9
sncf	2646	6.6	3.0
sncf-inter	366	4.3	1.6
sncf-ter	2637	7.1	2.43
zsr (daily)	233	2.3	1.94

Table 5.9: Speed-ups and size-ups of the *USP-OR-A* with *Locsep Max* for the whole timetables (for those marked with asterisk we took only a subset of n stations, as we were limited by the space).

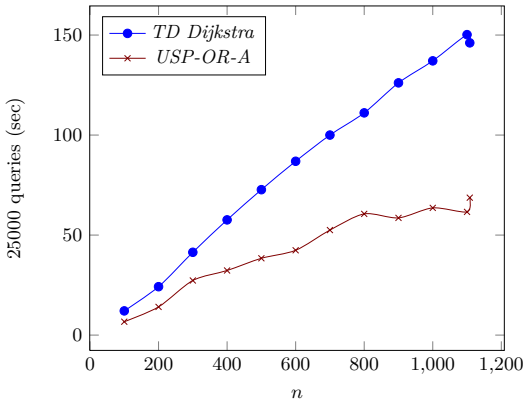

Figure 5.34: **Query time** of *USP-OR-A* with *Locsep Max* compared to TD Dijkstra on the ***cpza*** dataset. **Changing *n***.

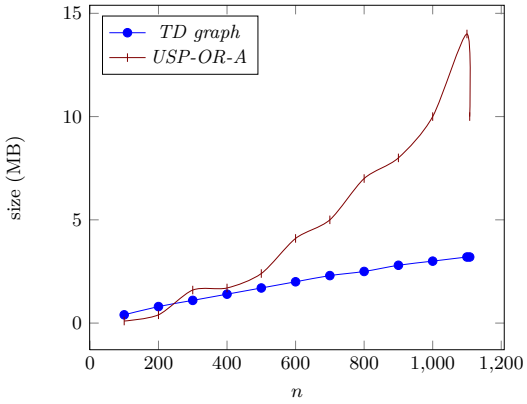

Figure 5.35: **Size** (in MB) of the oracle for *USP-OR-A* with *Locsep Max* vs. size of TD graph on ***cpza*** dataset. **Changing *n***.

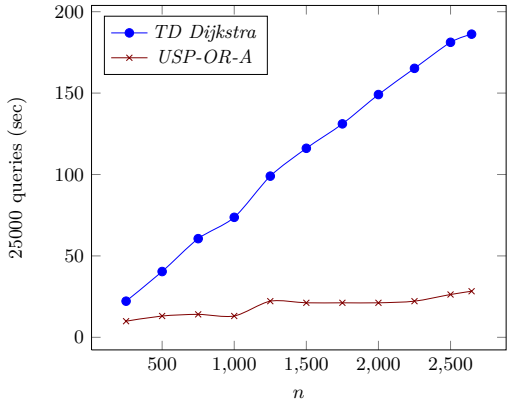

Figure 5.36: **Query time** of *USP-OR-A* with *Locsep Max* compared to TD Dijkstra on the ***sncf*** dataset. **Changing *n***.

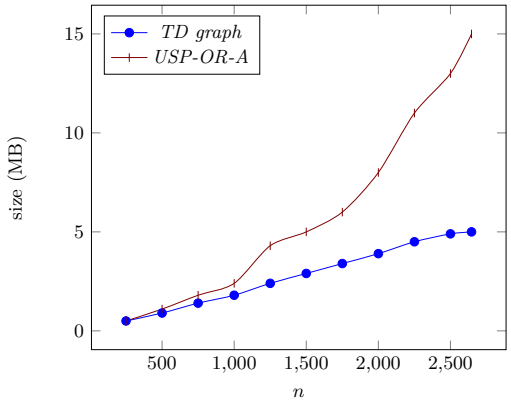

Figure 5.37: **Size** (in MB) of the oracle for *USP-OR-A* with *Locsep Max* vs. size of TD graph on ***sncf*** dataset. **Changing *n***.

6 Neural network approach

7 Application TTBlazer

8 Conclusion

Appendices

A File formats