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

Underlying shortest paths in timetables

(SVOČ 2013)

František Hajnovič

Supervisor: doc. RNDr. Rastislav Královič, PhD.

Bratislava 2013

Underlying shortest paths in timetables

František Hajnovič[1*]

Supervisor: Rastislav Královič[1†]

Katedra informatiky, FMFI UK, Mlynská Dolina 842 48 Bratislava

Abstract: Queries for optimal connection in timetables can be answered by running Dijkstra's algorithm on an appropriate graph. However, in certain scenarios this approach is not fast enough. We introduce methods with much better query time than that of the efficiently implemented Dijkstra's algorithm.

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

Keywords: optimal connection, timetable, Dijkstra's algorithm, underlying shortest paths

1 Introduction

We consider a problem of answering queries (in the form "from a at time t to b", denoted $\boldsymbol{(a,t,b)}$) for an optimal connection ($\boldsymbol{c^*_{(a,t,b)}}$) in a timetable on which we carried out some preprocessing. We define **timetable** simply as a set of **elementary connections**, which are quadruples (x, y, p, q) meaning that a train departs from **city** x at time p and arrives to city y at time q. A **connection** is a valid sequence of elementary connections which may include also some waiting in visited cities. We also define an **underlying graph** ($\boldsymbol{ug_T}$) of the timetable T whose nodes are all the cities of T and there is an arc (x, y) if T contains an elementary connection (x, y, p, q) for some p and q.

Finally, we define the **underlying shortest path** (USP) to be every path p in ug_T such that for some optimal connection $c^*_{(a,t,b)}$: $path(c^*_{(a,t,b)}) = p$, where function $path$ simply extracts the sequence of cities visited by the connection (see figure 2).

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 1: An example of a timetable.

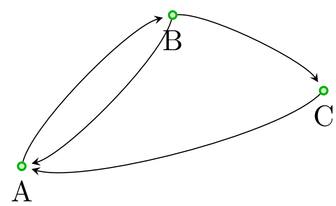

Figure 1: An underlying graph of the timetable 1.

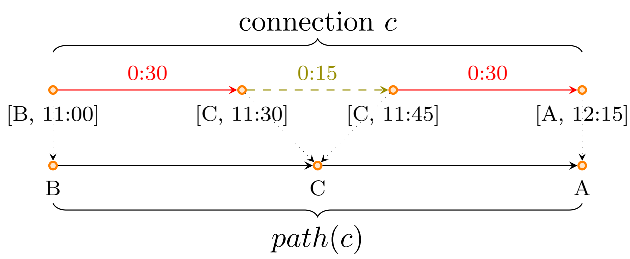

Figure 2: The *path* function applied on a connection to get the underlying path.

2 Methods

2.1 *USP-OR*

Our first method, called *USP-OR* (**USP oracle**), is based on pre-computing USPs between every

*ferohajnovic@gmail.sk

†kralovic@dcs.fmph.uniba.sk

pair of cities. Then, upon a query from x to y at time t we consider one by one the computed USPs between x and y and perform a reverse operation to the *path* function - $expand(p, t)$ where p is an USP and t is the departure time. The *expand* function simply follows the sequence of cities in p and from each of them it takes the first available elementary connection at the given time to the next one, thus constructing one by one a connection from x to y.

Algorithm 1 *USP-OR* query

Input
- timetable T
- 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**
  $c = expand(p, t)$
  c^* = better out of c^* and c
end for

Output
- connection c

We define an elementary connection (x, y, p_1, q_1) to **overtake** (x, y, p_2, q_2) if $p_1 > p_2$ but $q_1 < q_2$ [Delling and Wagner, 2009]. If the timetable T has no overtaking elementary connections, the *USP-OR* algorithm returns exact answers, which can be easily proved. In the real-world timetables that we used for testing we found a small percentage of overtaking elementary connections. Furthermore, we may simply remove the overtaken elementary connections from the timetable, as this will not influence the optimal connection for any query.

In this paper, we will denote the number of cities in T as n and the number of arcs in ug_T as m. The table 2 summarizes the parameters of *USP-OR* based on the following parameters of the timetable:

- $\boldsymbol{\tau}$ - the average number of different USPs between pairs of cities - the **USP coefficient**
- $\boldsymbol{\gamma}$ - the average size (i.e. number of elementary connections) of optimal connections - the **OC radius**
- $\boldsymbol{\delta}$ - the **density** of T describing, intuitively, how uniformly dense is the underlying graph of T [1]
- $\boldsymbol{h}$ - the **height** of the timetable, i.e. the average number of **events** in a city (where event is any arrival or departure of an elementary connection)

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

Table 2: The summary of the *USP-OR* algorithm parameters.

In our datasets (consisting of regional/country wide coach/train timetables) the OC radius and the density were generally found to be less than $\sqrt{n}$ or $\log n$, respectively. The hight of the timetables depends on **time range** of the timetable, which we define to be the time difference between the earliest and the latest event in the timetable. Generally, for one day of time range the height h ranged from 40 to 130.

Name	$\boldsymbol{\tau}$	$\boldsymbol{\gamma}$	$\boldsymbol{\delta}$	$\boldsymbol{h}$
cpsk	12.1	15.9	3.9	50.7
gb-coach	5.3	5.3	5.6	48
gb-train	10.3	7.6	5.7	129.6
sncf	5.6	8.6	3.4	42.4

Table 3: Daily, 200 station timetable subsets.

As for the value of τ, we may see from the table 3 that it is quite small (≈ 10). Important thing however is whether or not it is constant with respect to:
- n - we found τ to be constant (or only very slightly increasing) in this case (see plot 3)
- time range - again the value of τ was almost constant, or slightly increasing (plot 4)

[1] The exact definition is more technical: it considers all subsets of ug_T with n' cities and m' arcs, where $n' \geq \sqrt[4]{n}$. The density is the maximal $\frac{m'}{n'}$ found this way.

To alleviate the problem of increased τ in timetables with e.g. weekly time range, we did a simple trick. First, we normally computed the USPs. Then we **segmented** the timetable to individual days and for each of them we stored the pointers to necessary USPs. This does not require additional memory but it makes the value of τ constant (or even decreasing, as could be seen from plot 5) with increasing time range [2]. From this point on we assume the use of segmentation for multi-day timetables, also in *USP-OR-A* algorithm (explained in the next section).

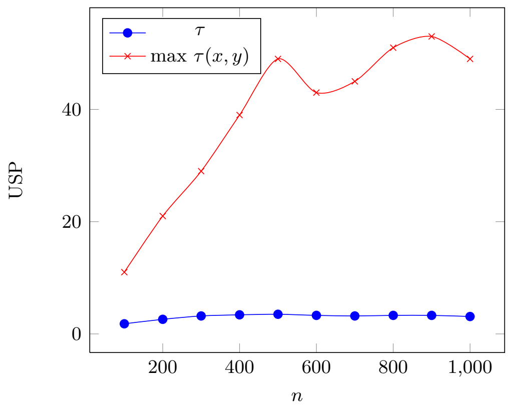

Figure 3: Changing of τ with increasing n. Dataset *gb-coach*.

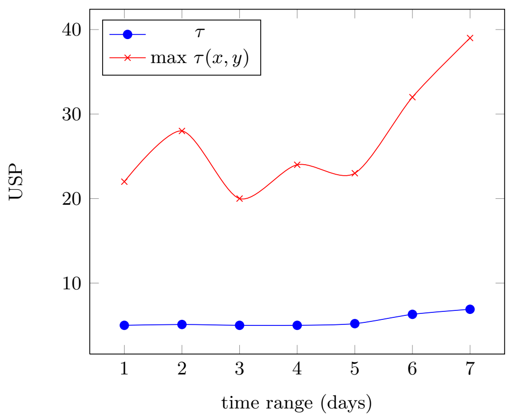

Figure 4: Changing of τ with increasing time range. Subset of *sncf* dataset with 200 stations.

[2]Note, that this would be reflected only in an improved query time of *USP-OR*, the size of preprocessed data will be left unaffected by segmentation.

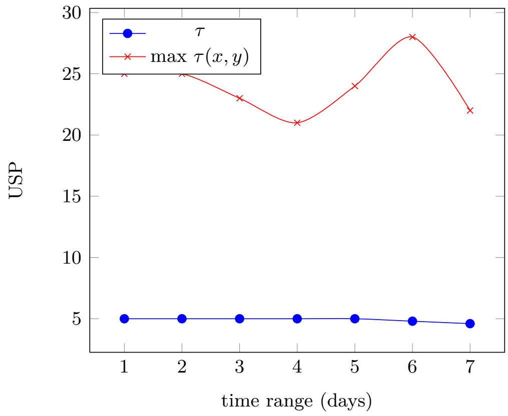

Figure 5: Changing of τ with increasing time range when using segmentation. Subset of *sncf* dataset with 200 stations.

2.2 *USP-OR-A*

The main drawback of *USP-OR* is its high amount of preprocessed data. To decrease the space complexity, in *USP-OR-A* (**USP oracle** with **access** nodes) we compute USPs only between cities from a smaller set - called **access node set** (AN set, denoted $\boldsymbol{\mathcal{A}}$). Given timetable T and a set of access nodes $\mathcal{A}$, we define for a city x its **neighbourhood** $\boldsymbol{neigh_{\mathcal{A}}(x)}$ as all the cities that could be reached (in ug_T) from x without going through any access node. The access nodes within this neighbourhood are called **local access nodes** (LANs, $\boldsymbol{lan_{\mathcal{A}}(x)}$). We do the same in $\overleftarrow{ug_T}$ (ug_T with reversed orientation) to get **back neighbourhood** and **back local access nodes** ($\boldsymbol{bneigh_{\mathcal{A}}(x)}$, $\boldsymbol{blan_{\mathcal{A}}(x)}$).

During the preprocessing we:

- find $\mathcal{A}$ (discussed later)
- compute USPs between x and y, where $x, y \in \mathcal{A}$
- compute $neigh_{\mathcal{A}}(x)$, $bneigh_{\mathcal{A}}(x)$, $lan_{\mathcal{A}}(x)$ and $blan_{\mathcal{A}}(x)$ for all $x \notin \mathcal{A}$

Upon a query from x to y at time t, we will first make a local search [3] in the neighbourhood of x up to x's local access nodes (*local front search* phase). Subsequently, we want to find out the

[3]We use the time-dependent Dijkstra's algorithm for this purpose (TD Dijkstra for short), which is a simple time-dependent modification of the Dijkstra's algorithm and is described e.g. in [Delling and Wagner, 2009].

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 (*inter-AN search* phase). 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 (*local back search* phase). See algorithm 2 and figure 6 for more clarification.

Algorithm 2 *USP-OR-A* query

Input

- timetable T
- 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
do TD Dijkstra from x at time t up to $lan(x)$
if $y \in neigh(x)$ **then**
  c^*_{loc} = conn. to y obtained by TD Dijkstra
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**
      $c = expand(p, ea(u))$
      $oc(v)$ = better out of $oc(v)$ and c
    end for
  end for
end for
$\forall v \in blan(y)$ let $ea(v)$ be the arrival time of $(oc(v))$
Local back search
for all $v \in blan(y)$ **do**
  perform TD Dijkstra from v at time $ea(v)$ to y restricted to $bneigh(y)$
  $fin(v)$ = the conn. returned by TD Dijkstra
end for
$v^* = argmin_{v \in blan(y)}\{\text{arrival time of } fin(v)\}$
u^* = departure city of $(oc(v^*))$
$c^* = oc(u^*).oc(v^*).fin(v^*)$ *# concat.*
output better out of c^*_{loc} and c^*

Output

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

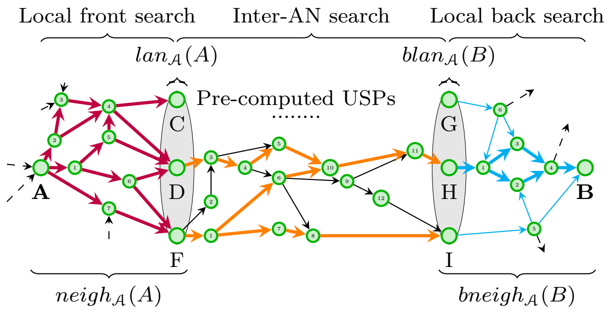

Figure 6: Principle of *USP-OR-A* algorithm. The arcs in **bold** mark areas that will be explored: all nodes in $neigh_{\mathcal{A}}(x)$, USPs between LANs of x and back LANs of y and the back neighbourhood of y (possibly only part of it).

We will call $\mathcal{A}$ a (r_1, r_2, r_3) **AN set** if:

- $|\mathcal{A}| \leq r_1 \cdot \sqrt{n}$
- $avg\ (|neigh_{\mathcal{A}}(x)|^2) \leq r_2 \cdot n$
- $avg\ (|lan_{\mathcal{A}}(x)|^2) \leq r_3$

If we can manage to find a (r_1, r_2, r_3) AN set in time $f(n)$, the parameters of the *USP-OR-A* algorithm are as summarized in table 4. Table 5 lists the parameters of *USP-OR-A* for timetables with some desirable properties (that our datasets had) and on which we can find (r_1, r_2, r_3) AN set with each r_i being a constant (with respect to n).

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

Table 4: Guaranteed parameters of the *USP-OR-A* algorithm. $\tau_{\mathcal{A}}$ and $\gamma_{\mathcal{A}}$ are defined just like τ and γ, but on the set of cities from $\mathcal{A}$.

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

Table 5: Parameters of the *USP-OR-A* algorithm under certain conditions.

2.3 Selecting access nodes

The challenge in *USP-OR-A* algorithm therefore comes down to the selection of a good access node set. However, consider the following problem: *minimize* $|\mathcal{A}|$ *such that* $\forall x \notin \mathcal{A} : |neigh_{\mathcal{A}}(x)| \leq \sqrt{n}$. We call this the problem of the optimal AN set.

Theorem 1. *The problem of the optimal AN set is NP-complete*

Proof. We will provide a sketch of the proof, which in full extend would be available in [Hajnovic, 2013]. 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 ($\bigcup\limits_{S_i \in \mathcal{S}^*} S_i = U$). For each $j \in U$, we will make a complete graph of β_j vertices (the value of β_j will be discussed later) named m_j and for each set S_i we make a vertex s_i and vertex s'_i. We now connect all vertices of m_j to s_i for each $j \in S_i$. Finally, for we connect s_i to s'_i, $1 \leq i \leq k$.

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

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

For this instance of min set-cover, we construct the graph depicted on figure 7.

Define α_i to be the number of sets S_j that contain i: $\alpha_i = |\{S_j \in \mathcal{S}| \ i \in S_j\}|$ and assume the constructed graph has n vertices. We want the β_i to satisfy $\beta_i \geq 2$ and $\beta_i + 2\alpha_i - 1 \leq \sqrt{n}$ but $\beta_i + 2\alpha_i > \sqrt{n}$. The last two inequalities would mean that if at least one s_j connected to m_i is chosen as an access node, the neighbourhood for nodes in m_i will be still large at most $\sqrt{n}$, but if

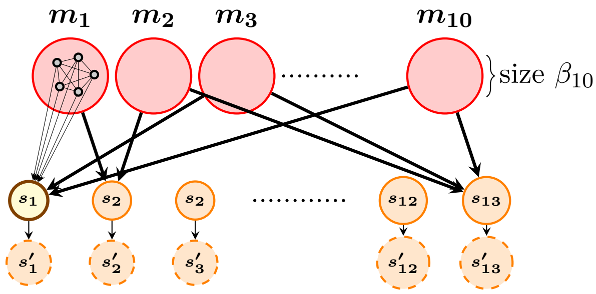

Figure 7: In m_i, there are actually complete graphs of β_i vertices (as shown for m_1). **Thick** arcs represent arcs from all the vertices of respective m_i. The s_i vertices are connected to their s'_i versions. If e.g. s_1 is selected as an access node, s'_1 is no longer part of any neighbourhood (except for its own).

none of them is chosen, the neighbourhood size will be just over $\sqrt{n}$. We leave out the details of the construction at this place.

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

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

□

We have therefore approached selection a good AN set heuristically. We iteratively selected ANs by their importance until the average square of the neighbourhood size became $\sqrt{n}$ or less (i.e. until $r_1 \leq 1$). To estimate the city's importance, we tried three values:

- Degree of the node in ug_T
- Betweenness centrality [Brandes, 2001] of the node in ug_T
- Our own value called **potential**, high for nodes that are good local separators in ug_T

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

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

Consider now a node $y \in bn(x)$. We will call $\boldsymbol{sur(y)} = \max\{0, |neigh(y)| - \sqrt{n}\}$ the **surplus** of y's neighbourhood, i.e., by how much we wish to reduce it so that it is at most $\sqrt{n}$. If the surplus is zero, y will not add anything to the x's potential. Otherwise, we run a restricted (to A_x) search from y during which we explore j vertices in $fn(x)$. We increase the potential of x by $\min\{sur(y), |fn(x) - j|\}$ - i.e. by how much we can decrease the surplus of y's neighbourhood. We do the same for all $y \in bn(x)$ and a similar thing for all $y \in fn(x)$ (we use $\overleftarrow{ug_T}$ instead of ug_T, $bneigh(y)$ instead of $neigh(y)$ etc...). For an illustration of potential computing, see figure 8.

The time complexity of *Locsep* can be estimated by $\mathcal{O}(\delta n^2)$, thus being the dominant part of the *USP-OR-A*'s preprocessing time complexity. In [?], we mention further optimisations to speed-up and improve this heuristic's performance. During the tests of our algorithms that follow in the next section we coupled *USP-OR-A* exclusively with *Locsep*.

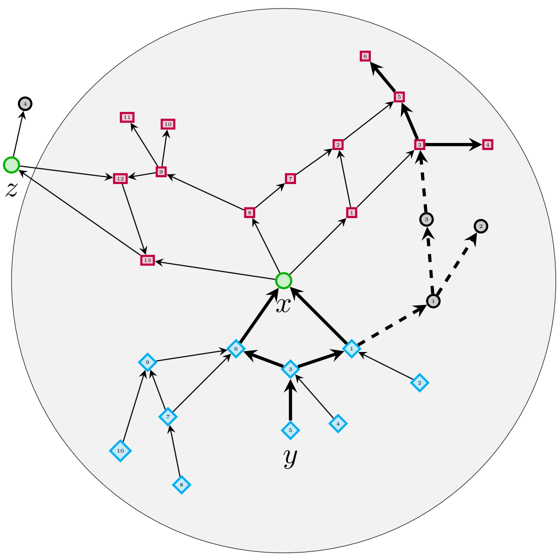

Figure 8: The principle of computing potentials in *Locsep* algorithm. We explored an area of $\sqrt{n}$ nearest cities (in terms of hops) around x. Access nodes (like z) and cities behind them are ignored. Little squares are nodes from $fn(x)$ and diamonds are part of $bn(x)$. From y we run a forward 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.

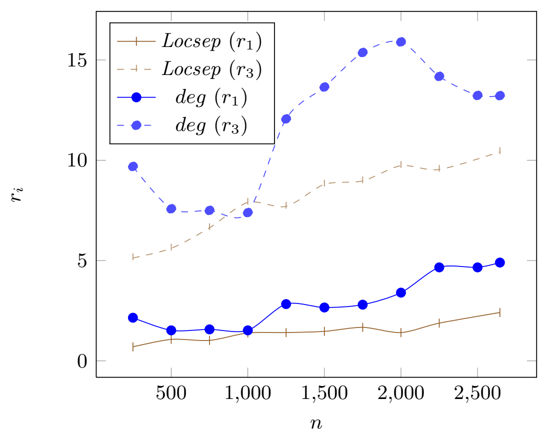

Figure 9: The parameters of the access node set when choosing access nodes based on degree and *Locsep* potential in *sncf* dataset. Value $r_1 \leq 1$. An ideal situation would be constant or non-increasing functions. *Locsep* does visibly better then choosing access nodes based on high degrees (or high betweenness centrality values). Dataset *sncf*.

3 Performance and comparisons

We have run tests on the datasets described in table 6. We selected 10000 random queries (random departure and arrival city and departure time). We used the timetables in a repetitive mode, i.e. even if we queried for an optimal connection starting e.g. on Sunday night in a 1-week timetable, we simply continued searching in Monday's schedule. This way, close to 100 % of the queries had a solution [4].

Name	Cities	UG arcs	Time range
cpsk	1905	5093	1 day
gb-coach	2448	5793	1 week
gb-train	2555	8335	1 week
sncf	2646	7994	1 week

Table 6: Our biggest datasets used for testing: regional buses from Žilina and Ružomberok in Slovakia (*cpsk*), country wide coaches (*gb-coach*) and trains (*gb-train*) in Great Britain and French railways (*sncf*)).

We compared the query time of *USP-OR* and *USP-OR-A* (with *Locsep*) with that of the time-dependent Dijkstra's algorithm using priority queues based on Fibonacci heaps (*TD Dijkstra* for short). Under certain conditions (see table 2 and table 4), the average query times for these algorithms are theoretically determined as:

- $\mathcal{O}(\sqrt{n})$ for *USP-OR*
- $\mathcal{O}(\sqrt{n}\log n)$ for *USP-OR-A*
- $\mathcal{O}(n \log n)$ for *TD Dijkstra* [5]

We wanted to see how many times faster are our algorithms in practice than the TD Dijkstra - a so called **speed-up** of the algorithm. To measure speed-up is a common practice to demonstrate efficiency of methods in route planning for road networks (see e.g. [Delling et al., 2009b]), but also in time-dependent scenarios.

As for the size of the preprocessed data, we need:

- $\mathcal{O}(n^{2.5})$ for *USP-OR*
- $\mathcal{O}(n^{1.5})$ for *USP-OR-A*
- $\mathcal{O}(hn)$ to store the timetable itself

In this case, we measured how many times more memory is necessary for the preprocessed data than the amount of memory occupied by the timetable itself. We call this value the **size-up** of the given method.

On plots 10 and 11 we show the evolution of query times with increasing n for *USP-OR*, *USP-OR-A* and *TD Dijkstra*. Plots 12 and 13 demonstrate the space complexity of *USP-OR-A*, again with respect to n. Finally, tables 7 and 8 summarize the speed-ups and size-ups for all datasets.

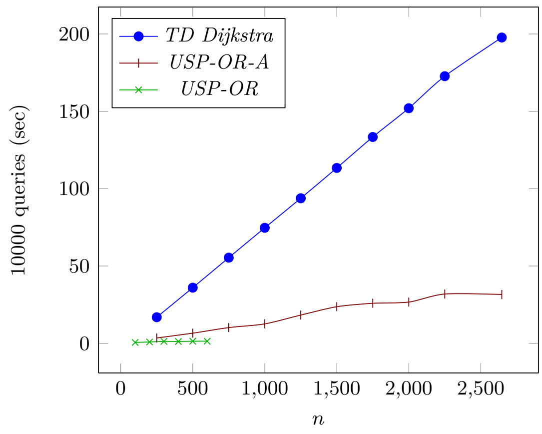

Figure 10: Query time, Dataset *sncf*.

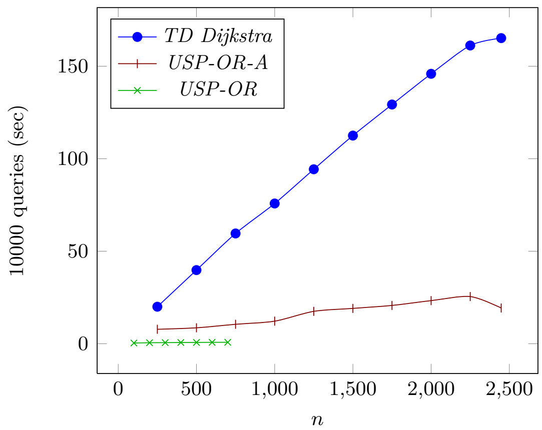

Figure 11: Query time, Dataset *gb-coach*.

[4] Some of the underlying graphs were not strongly connected, allowing for a query without solution to exist.

[5] Actually, the complexity of time-dependent Dijkstra's algorithm with Fibonacci heap priority queues is $\mathcal{O}(m + n \log n)$ [Sommer, 2010], but in our timetables $m \leq n \log n$

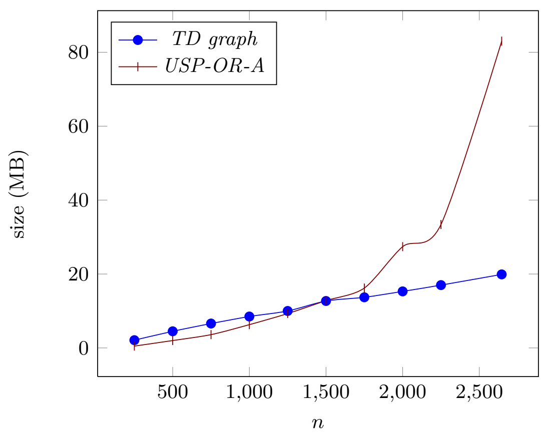

Figure 12: Size of preprocessed data, Dataset *sncf*.

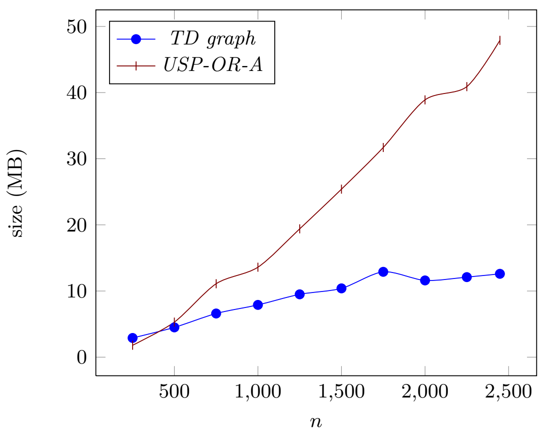

Figure 13: Size of preprocessed data, Dataset *gb-coach*.

Name	*n*	*spd*	*szp*
cpsk	600	16.3	242.7
gb-coach	700	**69.5**	50.6
gb-train	600	22.2	61.0
sncf	600	30.3	161.1

Table 7: Speed-ups and size-ups of the *USP-OR* algorithm. Due to memory limitations and high space complexity of *USP-OR*, we tried out only timetables with up to 700 stations.

Name	*n*	*spd*	*szp*
cpsk	1905	2.8	6.6
gb-coach	2448	**8.5**	3.8
gb-train	2555	2.9	2.3
sncf	2646	6.3	4.2

Table 8: Speed-ups and size-ups of the *USP-OR-A* algorithm, using *Locsep* to find access nodes.

4 Related work

In shortest path routing on road networks very much has been done to speed-up the query times using preprocessing on the input graph (for a good review of such methods, see [Delling et al., 2009b]). Some developed methods answer distance queries about million faster than the Dijkstra's algorithm. The timetable scenario has so far seen much smaller speed-ups, one reason for this being that the adaptation of the many techniques used for road networks to the time-dependent scenario is not so straightforward [Delling et al., 2009a].

The work [Batz et al., 2009] adapted one such technique (contraction hierarchies) to a time-dependent scenario. More specifically, the work dealt with road networks having time-dependent edge weights and authors have achieved speed-ups of up to 2000, outputting earliest arrival values. In [Delling, 2008] another route-planning technique is adapted to its time-dependent version. In this case, the timetable scenario is considered as well, with speed-ups of up to 27 on Europe-wide railway timetable with 30000 stations.

Both of the mentioned papers used the time-dependent model of timetable representation (as we did too). In [Delling et al., 2009a] the authors have considered the optimal connection problem in time-*expanded* graphs, and achieved speed-ups of up to 56 against Dijkstra's algorithm (also in the mentioned Europe-wide railway timetable).

The main disadvantage of *USP-OR-A* remains the relatively high space complexity, which is due to pre-computing underlying shortest paths between access nodes. This was partly inspired by the TRANSIT node routing [Bast et al., 2006] algorithm, which used a similar technique to achieve extremely fast distance query times in road networks, however also at the cost of high space consumption.

5 Conclusion

We have developed exact methods to considerably speed-up the query time for optimal connections in timetables compared to the time-dependent Dijkstra's algorithm (running in $\mathcal{O}(m + n \log n)$). Our first algorithm - *USP-OR* - achieves speed-ups of up to 70 in the sub-timetable of country-wide coaches in Great Britain. However, it does so at the cost of high space consumption, requiring more that 50 times the space that is needed to represent the timetable itself. Theoretically, for real-world timetables with certain properties, this algorithm has the space complexity $\mathcal{O}(n^{2.5})$ and the average query time $\mathcal{O}(\sqrt{n})$.

Our second algorithm called *USP-OR-A* is still 8.5 times faster then the time-dependent Dijkstra's algorithm on the dataset of British coaches (2500 stations) and at the same time, it requires about 4 times the space needed to represent the timetable. We believe the speed-up of *USP-OR-A* against Dijkstra's algorithm can be even higher for bigger timetables, since its query time is under certain conditions theoretically determined as $\mathcal{O}(\sqrt{n} \log n)$, while the algorithm can handle much bigger datasets for its space complexity is essentially $\mathcal{O}(n^{1.5})$.

Finally, it would be interesting to measure the query times of *USP-OR-A* if we used random sampling of queries with a distribution according to the reality. Such distribution strongly favours queries concerning the most important cities which are generally part of the access node set in *USP-OR-A*. As computing optimal connections between these cities is very fast (just like in *USP-OR*), we could expect much better speed-ups in real-world situations.

Acknowledgments

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 project.

References

[Bast et al., 2006] Bast, H., Funke, S., and Matijevic, D. (2006). Transit ultrafast shortest-path queries with linear-time preprocessing.

[Batz et al., 2009] Batz, G. V., Delling, D., Sanders, P., and Vetter, C. (2009). Time-dependent contraction hierarchies. In [Finocchi and Hershberger, 2009], pages 97–105.

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

[Delling, 2008] Delling, D. (2008). Time-dependent sharc-routing. In [Halperin and Mehlhorn, 2008], pages 332–343.

[Delling et al., 2009a] Delling, D., Pajor, T., and Wagner, D. (2009a). Engineering time-expanded graphs for faster timetable information. In Ahuja, R., Mohring, R., and Zaroliagis, C., editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 182–206. Springer Berlin / Heidelberg.

[Delling et al., 2009b] Delling, D., Sanders, P., Schultes, D., and Wagner, D. (2009b). Engineering route planning algorithms. In *ALGORITHMICS OF LARGE AND COMPLEX NETWORKS. LECTURE NOTES IN COMPUTER SCIENCE*. Springer.

[Delling and Wagner, 2009] Delling, D. and Wagner, D. (2009). Time-dependent route planning. In *Robust and Online Large-Scale Optimization, LNCS*. Springer.

[Finocchi and Hershberger, 2009] Finocchi, I. and Hershberger, J., editors (2009). *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2009, New York, New York, USA, January 3, 2009*. SIAM.

[Hajnovic, 2013] Hajnovic, F. (2013). Distance oracles for timetable graphs. Master's thesis, Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava.

[Halperin and Mehlhorn, 2008] Halperin, D. and Mehlhorn, K., editors (2008). *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*. Springer.

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