Underlying shortest paths in timetables

František Hajnovič[1*]

Supervisor: Rastislav Královič[1†]

Katedra informatiky, FMFI UK, Mlynská Dolina 842 48 Bratislava

Abstract: We introduce methods to speed-up optimal connection queries in timetables based on pre-computing paths that are worth to follow. We present a very fast but space consuming method *USP-OR* which we enhance to considerably decrease the size of the preprocessed data while still achieving speed-ups up to 9 against time-dependent Dijkstra's algorithm implemented with Fibonacci heap priority queue.

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 an arrives to city y at time q. A **connection** is simply 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.

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.

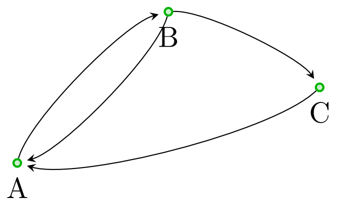

Figure 1: An underlying graph of the timetable 1.

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 picture 2).

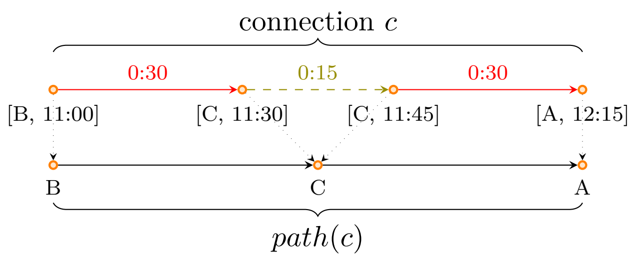

Figure 2: The *path* function applied on a connection to get the underlying path.

2 Approaches

2.1 *USP-OR*

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

*ferohajnovic@gmail.sk

†kralovic@dcs.fmph.uniba.sk

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 $q_1 < q_2$ [**?**]. 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 we have used for testing our methods we found 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 - the average number of **events** in a city (where event is any arrival or departure of an elementary connection)

[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.

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 $\sqrt{n}$ and $\log n$, respectively. The hight of the timetables varies with **time range** of the timetable, which we define to be the time difference between the earliest and the latest event in the timetable. For one day of time range the height h ranged from 50 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 ($\approx$ 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)

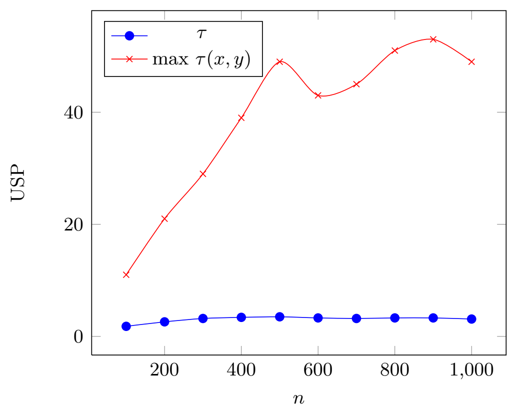

Figure 3: Changing of τ with increasing n. Dataset *gb-coach*.

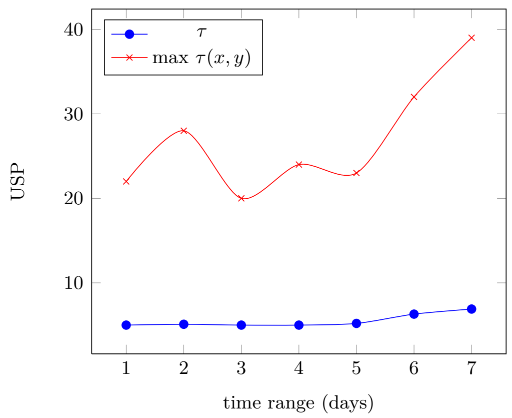

Figure 4: Changing of τ with increasing time range. Subset of *sncf* dataset with 200 stations.

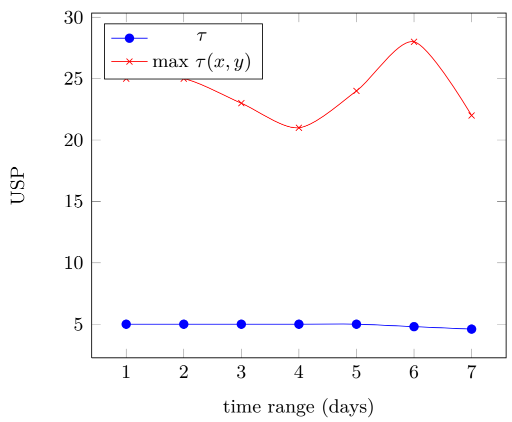

Figure 5: Changing of τ with increasing time range when using segmentation. Subset of *sncf* dataset with 200 stations.

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].

Still, the size of the data preprocessed by *USP-OR* is too large for practical use, hence the extension of this algorithm, called *USP-OR-A*.

2.2 *USP-OR-A*

To decrease the space complexity, in *USP-OR-A* (**USP** **or**acle with **a**ccess 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 in the neighbourhood of x up to x's local access nodes (*local front search* phase). 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 (*inter-AN search*

[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.

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 picture 6 for more clarification [3].

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 $ea(v) = end(oc(v))$
Local back search
for all $v \in blan(y)$ **do**
  perform TD Dijkstra from v at time $ea(v)$ to y restricted to $bneigh(y)$
  $fin(v)$ = the conn. returned by TD Dijkstra
end for
$v^* = argmin_{v \in blan(y)}\{end(fin(v))\}$
$u^* = from(oc(v^*))$
$c^* = oc(u^*).oc(v^*).fin(v^*)$ # *concat.*
output better out of c^*_{loc} and c^*

Output

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

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

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

[3] In the algorithm 2 we use TD Dijkstra as a shortcut for time-dependent Dijkstra's algorithm, which is a simple time-dependent modification of the Dijkstra's algorithm. The time-dependent version is described e.g. in [?].

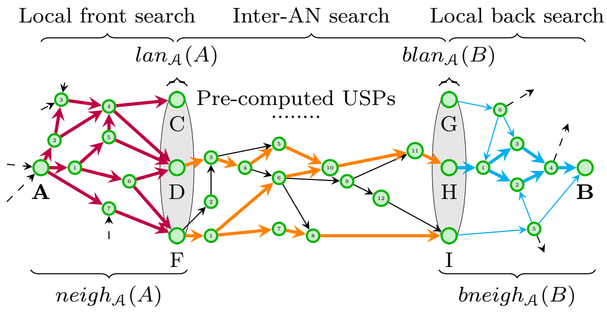

Figure 6: Principle of *USP-OR-A* algorithm. The arcs in **bold** mark areas that will be explored: all nodes in $neigh_{\mathcal{A}}(x)$, USPs between LANs of x and back LANs of y and the back neighbourhood of y (possibly only part of it will be explored, since the local back search goes against the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

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 specific properties (as had our datasets) and on which we can find (r_1, r_2, r_3) AN set with each r_i being a constant (with regard 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. $\mathcal{O}(r_2 r_3 \sqrt{n}(\log(r_2 n) + \delta) + r_3^2 \tau_{\mathcal{A}} \gamma_{\mathcal{A}})$
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.,** $\boldsymbol{\gamma \le \sqrt{n}, \delta \le \log n}$
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, generally fulfilled by our timetables.

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 [**?**]. 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 $s_i \iff j \in S_i$. 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 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 $\leq \sqrt{n}$, but if none

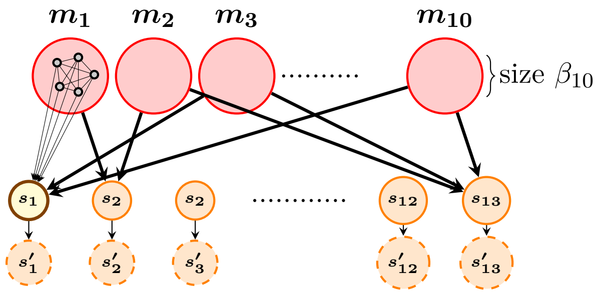

Figure 7: In m_i, there are actually complete graphs of β_i vertices (as shown for m_1). **Thick** arcs represent arcs from all the vertices of respective m_i. The s_i vertices are connected to their s'_i versions. If e.g. s_1 is selected as an access node, s'_1 is no longer part of any neighbourhood.

of them is chosen, the neighbourhood 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 $\leq \sqrt{n}$, 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 was not $\leq \sqrt{n}$ (i.e. the r_1 parameter of the set was ≤ 1). To estimate the city's importance, we tried three values:

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

Our algorithm, called *Locsep*, computes the city's potential in the following way: we explore an area $\boldsymbol{A_x}$ of $\sqrt{n}$ nearest cities around x. We do this 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 $\mathcal{A}$, there is 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.

Let $bneigh_i = bneigh(b_i) \cap A_x$ for each arc $(b_i, x) \in ug_T$. We run a restricted (to $A_x \setminus \{x\}$) search from each $bneigh_i$ during which we explore e_i vertices in $fn(x)$. This is going to contribute up to $e_i|bneigh_i|$ to x's potential, depending on if the cities in $bneigh_i$ actually have large neighbourhoods (and thus selecting x would help). More details regarding the *Locsep* algorithm will be in [?]. For illustration of the principle of computing the potential, see picture 8. Also, see plot 9 for comparison of *Locsep* algorithm to the approach of choosing ANs based on high degree/BC values.

3 Performance and comparisons

We have run tests on the datasets described in table 6. We compared the query time of *USP-OR*, *USP-OR-A* with *Locsep* and time-dependent Dijkstra's algorithm using priority queues based on Fibonacci heaps (*TD Dijkstra* for short). The average query times for these algorithms (at our timetables) are theoretically determined as:

- $\mathcal{O}(\sqrt{n})$ for *USP-OR*
- $\mathcal{O}(\sqrt{n}\log n)$ for *USP-OR-A* with *Locsep*
- $\mathcal{O}(n \log n)$ for *TD Dijkstra* [4]

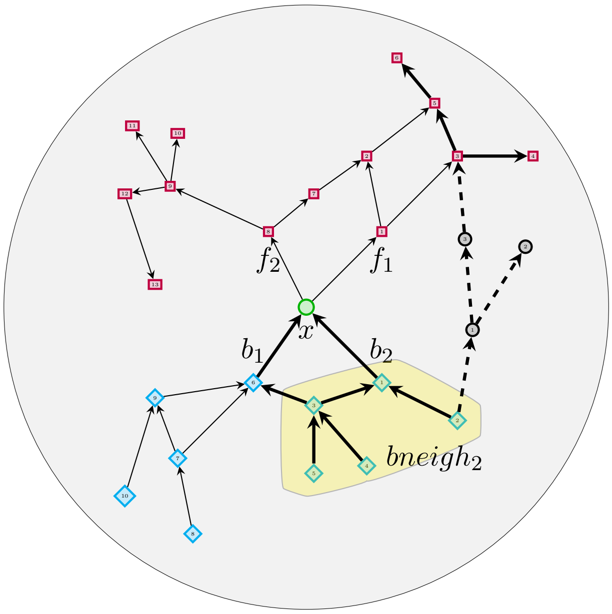

Figure 8: The principle of computing potentials in Locsep algorithm. We explored an area of $\sqrt{n}$ nearest cities (in terms of hops) around x. Little squares are nodes from $neigh(x)$ and diamonds are part of $bneigh(x)$. The highlighted area represents the back neighbourhood for node b_2. From its nodes we run a forward search (the **thick** arcs). Nodes from the $neigh(x)$ that were not explored in this search can only be reached via x itself and contribute to x's potential.

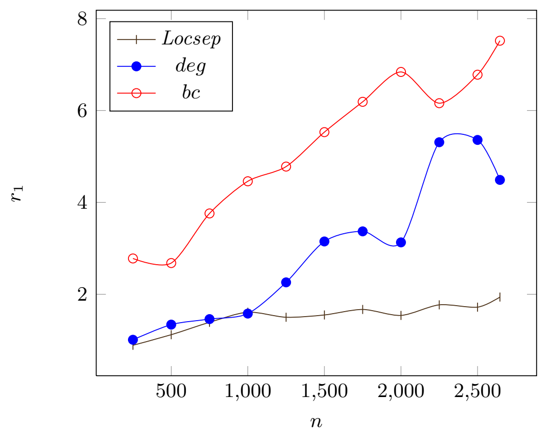

Figure 9: The necessary size of $\mathcal{A}$ when selecting ANs based on degree, BC or *Locsep* potential in *sncf* dataset ($|\mathcal{A}| = r_1\sqrt{n} \approx r_1 51$). Ideal situation would be a constant or non-increasing value, to which *Locsep* comes closest.

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

For the dataset of French railways (*sncf*) we measured how the query times evolve with increased n (see plot 10) and for the dataset of Slovak railways (*zsr*) we measured the evolution of query times with respect to increased time range (plot 11). Finally, for all of our timetables, we measured the speed-up against TD Dijkstra, i.e. how many times faster were our algorithms then TD Dijkstra. For the speed-ups, refer to table 7.

Name	Description	Cities	UG arcs
cpru	Reg. bus (SVK - RK)	871	2415
cpza	Reg. bus (SVK - ZA)	1108	2778
montr	Public tr. (Montreal)	217	349
sncf	Railways (FRA)	2646	7994
sncf-inter	Inter-city rail. (FRA)	366	901
sncf-ter	Regional rail. (FRA)	2637	7647
zsr	Railways (SVK)	233	588

Table 6: Datasets used for testing.

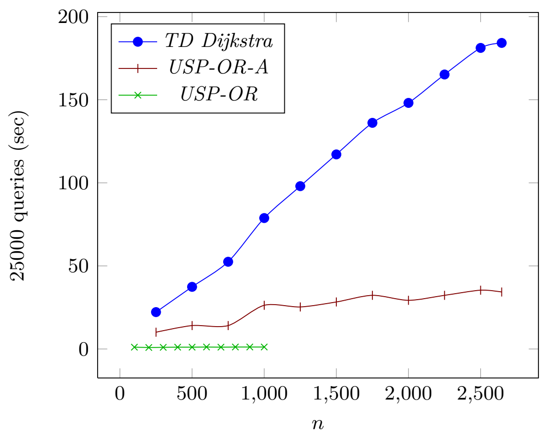

Figure 10: Query times (in sec., 25000 queries). *USP-OR-A* + *Locsep* vs. *USP-OR* vs. *TD Dijkstra* on *sncf* dataset. Changing n.

As for the size of the preprocessed data, we compared *USP-OR* to *USP-OR-A* with *Locsep* in a similar way we did for the query times: in plots 12, 13 there are the actual sizes (in MB) of the data preprocessed by our algorithms and the amount of memory needed to actually represent the timetable in memory (as a time-dependent graph [?]). In table 8 we provided the ratios specifying how many times more MB was pre-

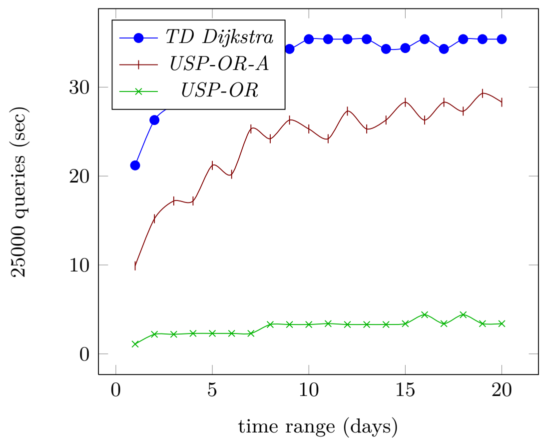

Figure 11: Query times (in sec., 25000 queries). *USP-OR-A* + *Locsep* vs. *USP-OR* vs. *TD Dijkstra* on *zsr* dataset. Changing time range.

Name	*USP-OR*	*USP-OR-A*
cpru	14.5	1.7
cpza	14.3	1.7
montr	8.8	1.5
sncf	64.8	5.4
sncf-inter	27.0	3.6
sncf-ter	78.3	6.3
zsr (daily)	19.3	2.14

Table 7: Speed-up of *USP-OR* and *USP-OR-A* with *Locsep*.

computed then the memory needed for the time-dependent graph. Again we remind the theoretically determined sizes of preprocessed data for our algorithms (and on our timetables):

- $\mathcal{O}(n^{2.5})$ for *USP-OR*
- $\mathcal{O}(n^{1.5})$ for *USP-OR-A* with *Locsep*

Name	*USP-OR*	*USP-OR-A*
cpru	396.7	3.0
cpza	265.1	3.5
montr	61.1	1.3
sncf	106.2	4.0
sncf-inter	30.3	1.5
sncf-ter	87.4	2.6
zsr (daily)	60.8	2.2

Table 8: The ratio $\frac{\text{size of TD graph}}{\text{preprocessed size}}$ for *USP-OR* and *USP-OR-A* with *Locsep*.

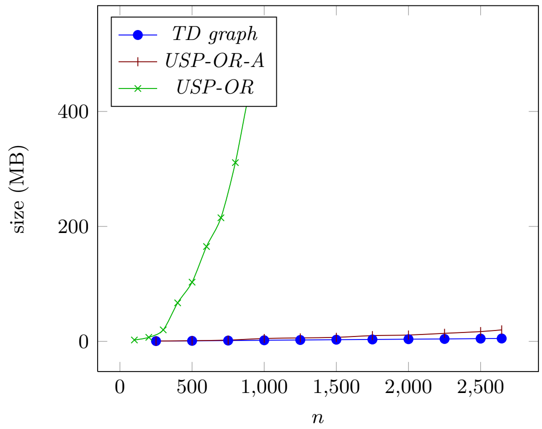

Figure 12: Size (in MB) of preprocessed data. *USP-OR-A + Locsep* vs. *USP-OR* vs. *TD Dijkstra* on *sncf* dataset. Changing n.

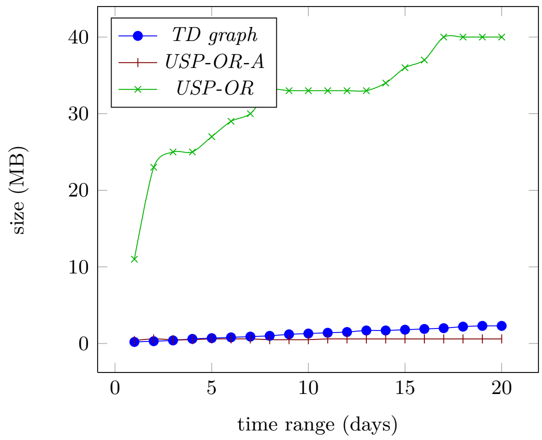

Figure 13: Size (in MB) of preprocessed data. *USP-OR-A + Locsep* vs. *USP-OR* vs. *TD Dijkstra* on *zsr* dataset. Changing time range.

4 Conclusion

In [**?**] the authors have considered the optimal connection problem in time-expanded graphs, and achieved speed-ups of up to 56 against Dijkstra's algorithm in railway timetables with about 30000 stations.

We approached the problem through the time-*dependent* model and have developed exact methods to considerably speed-up the query time for optimal connections in timetables compared to the time-dependent Dijkstra's algorithm using Fibonacci heaps as priority queues (running in $\mathcal{O}(m + n \log n)$). Our first algorithm - *USP-OR* - achieves speed-ups of up to 65 on our largest dataset representing French railways (2500+ stations). However, it does so at the cost of high space consumption, requiring more that 100 times the space that is needed to represent the timetable itself. Theoretically, for real-world timetables (that usually have 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 6 times faster then time-dependent Dijkstra's algorithm and at the same time, for all of our datasets it requires space up to 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 theoretically determined as $\mathcal{O}(\sqrt{n} \log n)$, while the algorithm can handle much bigger datasets for its space complexity is essentially $\mathcal{O}(n^{1.5})$.

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.