International Journal of
Geo-Information

Article

The ε-Approximation of the Time-Dependent Shortest Path Problem Solution for All Departure Times

František Kolovský [1,*], Jan Ježek [1] and Ivana Kolingerová [2]

1 Department of Geomatics, University of West Bohemia, Univerzitní 2732/8, 301 00 Plzeň, Czech Republic; jezekjan@kgm.zcu.cz
2 Department of Computer Science, University of West Bohemia, Univerzitní 2732/8, 301 00 Plzeň, Czech Republic; kolinger@kiv.zcu.cz
* Correspondence: kolovsky@kgm.zcu.cz

Received: 30 September 2019; Accepted: 24 November 2019; Published: 27 November 2019

Abstract: In this paper, the shortest paths search for all departure times (profile search) are discussed. This problem is called a *time-dependent shortest path problem* (TDSP) and is suitable for time-dependent travel-time analysis. Particularly, this paper deals with the ε-approximation of profile search computation. The proposed algorithms are based on a label correcting modification of Dijkstra's algorithm (LCA). The main idea of the algorithm is to simplify the arrival function after every relaxation step so that the maximum relative error is maintained. When the maximum relative error is 0.001, the proposed solution saves more than 97% of breakpoints and 80% of time compared to the exact version of LCA. Furthermore, the runtime can be improved by other 15% to 40% using heuristic splitting of the original departure time interval to several subintervals. The algorithms we developed can be used as a precomputation step in other routing algorithms or for some travel time analysis.

Keywords: time-dependent shortest path problem; approximation; travel time function; road network

1. Introduction

Computing the arrival function from a source node to all other nodes is important for a lot of transportation applications. More formally, given a directed graph $G = (V, E)$, a source node $s \in V$, we want to know the travel time between the source node s and all other nodes for every departure time (in some literature called a *travel time profile*). This problem is generally called the *time-dependent shortest path problem* (TDSP). The main principle is that the arrival time t_u at the node u is used as the argument of the arrival time function f corresponding with the edge that origins at u.

The common approach is to use a piecewise linear function as a realization of the arrival function. Let us have two consecutive edges (e.g, the edges (s, u) and (u, d) in Figure 1a); then, the arrival time at the node d is the value of the arrival function f_{ud} in the arrival time $f_{su}(t_d)$ at the node u, where t_d is the departure time at the node s. In the exact case, the combination $f_2(f_1(t))$ of two piecewise linear functions f_1, f_2 with $|f_1|$, $|f_2|$ linear pieces is also a piecewise linear function with up to $|f_1| + |f_2|$ linear pieces [1]. It means that the arrival function at the end of the path with n edges can have up to $\sum_{i=1}^{n} |f_i|$ linear pieces. For example, the path across the Pilsen city has around 100 edges. If every arrival function on the path has 24 linear pieces, the resulting arrival function has 2400 liner pieces. It can be seen that the computational time and memory requirements strongly increase with the length of the paths.

The problem with an increase in the number of linear pieces can be solved using the ε-approximation of the resulting arrival function. This approach reduces the number of linear pieces, and thus reduces memory requirements as well as computation time. Our proposed approach is based on the so-called label correcting modification of Dijkstra's algorithm [2]. The main idea is to perform a simplification of the arrival functions during the computation with a suitable maximum absolute error so that the relative error ε is maintained. Further, this technique can be accelerated by the heuristic. The idea is that the original departure time interval is split into subintervals, and thus the number of edge relaxations is reduced. These subintervals are independent too, so the parallelization or distribution of computation is possible and effective.

2. Definitions and Preliminaries

2.1. Road Network

Let $G = (V, E)$ be a directed graph that represents a road network, where V is a set of nodes and E is a set of edges. Each edge $(u, v) \in E$ has an *arrival function* (AF) $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ that for the given departure time at u returns the arrival time at v. Alternatively, we can define a *travel time function* (TTF) that returns the time needed to cross the edge. A relationship between the TTF g and the corresponding AF f is defined as $g(t) = f(t) - t$.

It is assumed that every AF f fulfills the first-in-first-out (FIFO) property: $\forall t_1 < t_2 : f(t_1) \leq f(t_2)$ and the departure time t_d must be smaller then the arrival time t_a (the travel time must be positive). AFs are implemented as piecewise linear functions.

In Figure 1a you can see AFs (f_{su}, f_{sv}, f_{ud} and f_{vd}) for every edge in a small example graph with four nodes $V = \{s, u, v, d\}$ and four edges $E = \{(s, u), (s, v), (u, d), (v, d)\}$.

The points of AFs are called *breakpoints*. The number of breakpoints of AF f can be written as $|f|$. The following operation must be defined for two AFs:

- There are two consecutive edges (s, u) and (u, d) with AFs f_{su}, f_{ud}. The operation *combination* $f_{ud} * f_{su} : t \mapsto f_{ud}(f_{su}(t))$ represents AF from s to d. In Figure 1b there are AFs as results of the combination along the paths (s, u, d) (solid red line) and (s, v, d) (solid blue line).
- There are two parallel paths p_1, p_2 from s to d with AFs f^1_{sd}, f^2_{sd}. The operation *minimum* $\min(f^1_{sd}, f^2_{sd}) : t \mapsto \min\{f^1_{sd}, f^2_{sd}\}$ represents the earliest AF from s to d. In Figure 1a $p_1 = (s, u, d)$ and $p_2 = (s, v, d)$. In Figure 1c you can see this earliest AF as a result of the operation minimum (green line).

2.2. Problem Definition

More precisely, TDSP can be defined as minimizing the travel time over the set $P_{s,d}$ of all paths in G from the source node s to the destination node d:

$$f_d = \min\{f_p(t) | p \in P_{s,d}\}, \tag{1}$$

where f_d is the function of the earliest arrival time (minimal AF) from s to d and f_p is AF of the path $p \in P_{s,d}$.

This paper deals with *one-to-all* problem. The input data are the graph G, AF f_{uv} for every edge $(u, v) \in G$ and the source node s. The output is the set F of the earliest AFs from the source node s to all other nodes u: $F = \{f_u | u \in V \setminus \{s\}\}$.

2.3. Approximation

In this paper the ε-approximation of AF $f^{\updownarrow}$ is understood as the ε-approximation of TTF $f(t) - \varepsilon g(t) \leq f^{\updownarrow}(t) \leq f(t) + \varepsilon g(t)$. So the ε-approximation of the set F is $F^{\updownarrow} = \{f^{\updownarrow}_u | u \in V \setminus \{s\}\}$.

Let us present some useful theorems about the approximation that were derived for a use in the proposed algorithm.

Theorem 1. *Let $g^{\updownarrow}$ be an ε-approximation of TTF g. Then it holds that*

$$g^{\downarrow} = \frac{g^{\updownarrow}}{1+\varepsilon} \leq g \leq \frac{g^{\updownarrow}}{1-\varepsilon} = g^{\uparrow}$$

.

Proof. If the function $g^{\updownarrow}$ is substituted by its extreme values $(1-\varepsilon)g$, $(1+\varepsilon)g$, the expression is still valid. □

Theorem 2. *Let $f_u^{\updownarrow}$ be an ε-approximation of AF f_u and $\overline{f}_v^{\updownarrow} = f_{uv} * f_u^{\updownarrow}$. Let $\alpha \in [0, \frac{\overline{g}_v^{\downarrow}}{g_u^{\downarrow}}]$ be the maximum slope of AF f_{uv}, $approx(f, \delta)$ be a function that simplifies the AF f with the maximum absolute error $\delta \geq 0$. Then $f_v^{\updownarrow} = approx(\overline{f}_v^{\updownarrow}, \varepsilon\overline{g}_v^{\downarrow} - \alpha\varepsilon g_u^{\downarrow})$ is the ε-approximation of AF f_v.*

Proof. The maximum absolute error of f_v is εg_v and the maximum absolute error of the operation $f_{uv} * f_u^{\updownarrow}$ is $\alpha\varepsilon g_u$. Then, the result of the combination can be simplified with the maximum absolute error:

$$\delta = \varepsilon g_v - \alpha\varepsilon g_u \geq \varepsilon\overline{g}_v^{\downarrow} - \alpha\varepsilon g_u^{\downarrow}$$

Then δ must be ≥ 0

$$\begin{aligned} \varepsilon\overline{g}_v^{\downarrow} - \alpha\varepsilon g_u^{\downarrow} &\geq 0 \\ \alpha &\geq \frac{\overline{g}_v^{\downarrow}}{g_u^{\downarrow}}. \end{aligned}$$

□

Theorem 2 can be also formulated in a local form for a given departure time.

In Figure 1b there are dotted lines that represent the approximation of AFs. In Figure 1d you can see the ε-approximation $f_d^{\updownarrow}$ of the earliest AF f_d from s to d (solid green line) and its upper bound $f_d^{\uparrow}$ and lower bound $f_d^{\downarrow}$ (green dashed lines).

2.4. Related Work

There are two groups of methods that compute an approximation of the AF. The first methods use *forward* and *backward probes*. The forward probe computes the arrival time at the node d with the given departure time at the node s. The backward probe solves the inverse problem. The arrival time at d is given and we want to know the departure time at s. These probes can be computed using the well-known Dijkstra's algorithm [3].

These methods recognize two types of breakpoints. The **V** points represent points that are created as images of the breakpoints that lie on the edge arrival functions $\{f_{uv} | (u,v) \in E\}$. The **X** points are created as an intersection of two AF in the *minimum* operation. It can be proven that the AF between two consecutive **V** points is concave or a line segment [1] (see example in Figure 1c). The algorithms described in [1,4] use this concavity. First, the **V** points are computed using one backward probe and two forward probes (more in [3]), and then the approximation of AF between the **V** points is determined. The main problem of this approach is that the computation of **V** points requires $3\sum_{(u,v)\in E} |f_{uv}|$ probes [3].

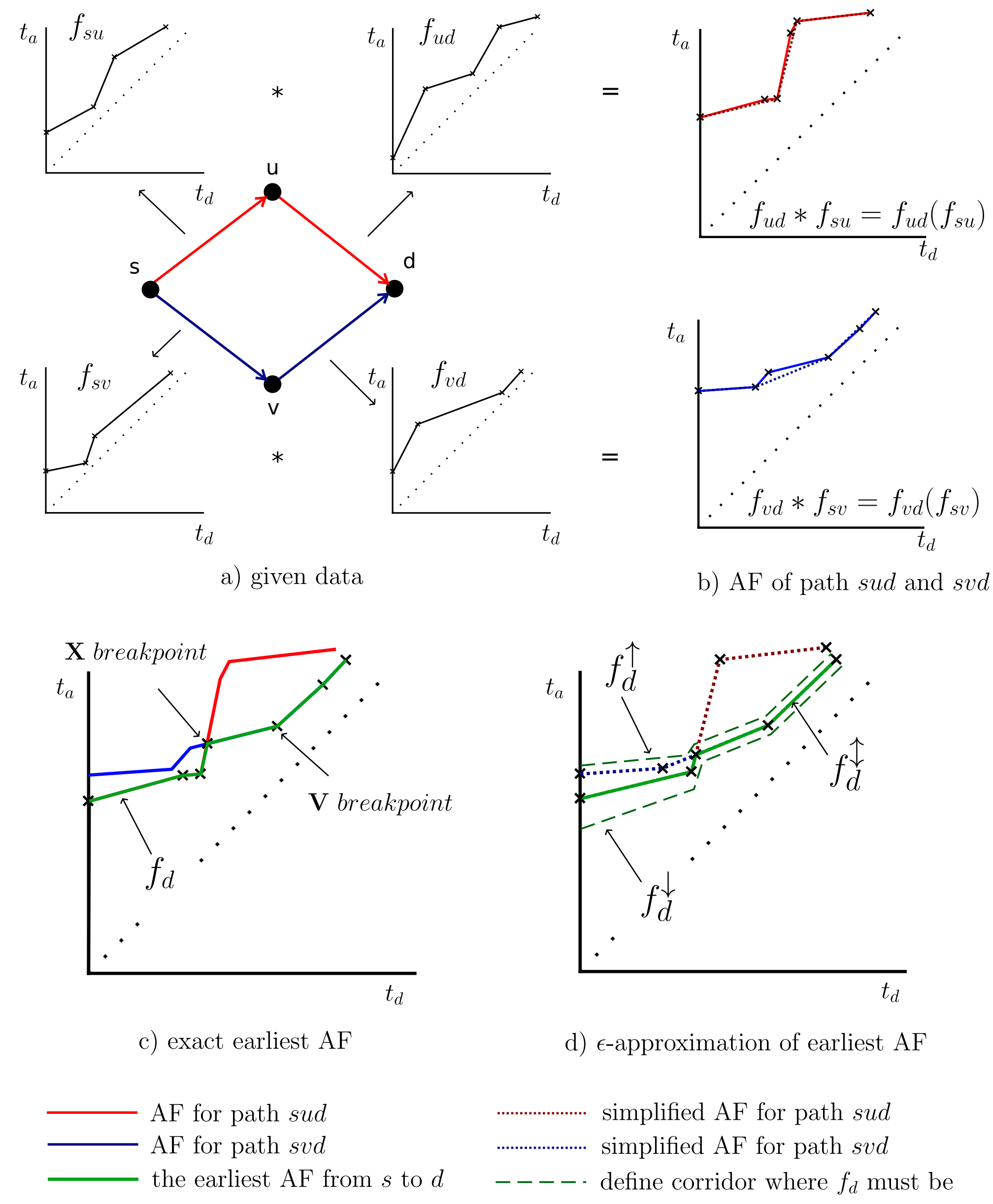
t_a
f_{su}
$*$
f_{ud}
$=$
$f_{ud} * f_{su} = f_{ud}(f_{su})$
t_d
u
s
d
v
f_{sv}
f_{vd}
$f_{vd} * f_{sv} = f_{vd}(f_{sv})$
a) given data
b) AF of path sud and svd
X *breakpoint*
V *breakpoint*
f_d
$f_d^{\uparrow}$
$f_d^{\updownarrow}$
$f_d^{\downarrow}$
c) exact earliest AF
d) ϵ-approximation of earliest AF
AF for path sud
AF for path svd
the earliest AF from s to d
simplified AF for path sud
simplified AF for path svd
define corridor where f_d must be

Figure 1. Example of calculation of the arrival function from node s to d (t_d—departure time, t_a—arrival time).

The second group of methods uses a label correcting modification of the Dijkstra's algorithm (LCA) (Algorithm 1). The modifications of the Dijkstra's algorithm are:

- The node labels are AFs from s.
- The key of the priority queue is the minimum of AF ($\min f$).
- The relaxation of the edge (u, v) is performed using $f_v = \min(f_v, f_{uv} * f_u)$.

LCA has time complexity $O(|V||E|)$ [2]. However, a real road network is far from the worst case, because it is close to planar graph and the cost of edges is the travel time; i.e., is bounded by physics rules, and thus the shortest path is near to the direct line.This technique is widely used; see e.g, [5–7]. First LCA is performed in the exact form and after that the resulting arrival functions F are simplified and used for further computation (e.g, some query algorithm). Some guarantees about the error of AF are presented in [5], but these guarantees give only a maximum error dependent on the degree of approximation.

Algorithm 1: LCA in the exact form.

```
1  PQ = minimum priority queue where key is min $f$
2  $\forall u \in V : f_u = \infty$
3  $g_s = 0$
4  PQ.put($f_s$)
5  while queue is not empty do
6      $f_u =$ PQ.get()
7      foreach $v : (u, v) \in E$ do
8          $\overline{f}_v = f_{uv} * f_u$                         // combination
9          if $\exists t : \overline{f}_v(t) < f_v(t)$ then                // compare
10             $f_v = \min(\overline{f}_v, f_v)$                       // min
11             PQ.put($f_v$)
```

In Algorithm 1 the initialization is performed in the lines 1–4. All node labels (AFs) are set to infinity in the line 2. The travel time at s is set to zero (line 3) and the node label at s (f_s) is added to the priority queue (PQ) (line 4). In the line 6 the algorithm takes the node on the top of PQ and relaxes all edges that lead from this node. The relaxation is represented by the lines 8–11. The line 8 performs the combination of the node label at u (f_u) and the edge AF (f_{uv}). The condition in the line 9 checks for updates. Updates of the label at the node v are performed in the line 10. Line 11 puts the node v to the PQ.

There are a lot of other variants of LCA. Those variants differ in the type of data structure—a queue (FIFO, first-in-first-out) [8], a priority queue (as in our case) [5,7,9], a stack (LIFO, last-in-first-out) [10]. The algorithms also differ in the insertion strategy into the data structure. Some innovative insertion strategies are presented in [11,12]. The choice of LCA variant depends on the type of the target graph.

The main task is to develop an algorithm which solves TDSP with the given maximum relative error and is effective for a real road network. It follows that we focused on the ε-approximation of the LCA.

3. Proposed Algorithms

This section describes two algorithms solving the ε-approximation of TDSP based on LCA (Algorithm 1).

3.1. ε-LCA Algorithm

The basic idea of the first proposed algorithm (ε-LCA) is that the simplification of AF is performed after every edge relaxation (the operation combination). The degree of the simplification is directed by Theorem 2.

The ε-LCA computes AF with the maximum relative error ε assuming that

$$\forall (u, v) \in E : \alpha_{uv} \in \left[0, \frac{g_v^{\downarrow}}{g_u^{\downarrow}}\right], \tag{2}$$

where α_{uv} is the maximum slope of AF f_{uv}. The slope α_{uv} must be bounded because Theorem 2 is used in the ε-LCA and the theorem needs this assumption.

The ε-LCA differs from the exact LCA only in the computation of $\overline{f_v}$. The AF $\overline{f_v}$ in the line 8 in Algorithm 1 is simplified with the maximum absolute error δ (according to Theorem 2), so the line 8 is replaced by two lines

$$\begin{aligned} \delta(t) &= \varepsilon((f_{uv} * f_u^{\updownarrow})^{\downarrow}(t) - t) - \alpha(t)\varepsilon g_u^{\downarrow}(t) \\ \overline{f_v} &= approx((f_{uv} * f_u^{\updownarrow}), \delta), \end{aligned} \tag{3}$$

where $\alpha(t)$ is the maximum slope of f_{uv} in the interval $[f_u^{\downarrow}(t), f_u^{\uparrow}(t)]$. The simplification was performed using Douglas–Peucker algorithm (DP) or Imai and Iri algorithm (II) [13].

The main problem of ε-LCA is that if the assumption (2) is not complied, $\delta < 0$ and the algorithm cannot ensure the given relative error ε. This occurs when the maximum slope α_{uv} is too large. This issue is resolved using the second algorithm that is described in the upcoming section.

3.2. ε-LCA-BS Algorithm

The second proposed algorithm (ε-LCA-BS) is based on backsearch. It has no limitations for the slope α. The pseudo-code of the ε-LCA-BS is in Algorithm 2. The basic idea is that if the algorithm finds an edge (u, v) where α is too big in some departure time interval $[t_m, t_n]$ ($\delta < 0$), it determines f_v in $[t_m, t_n]$ again with a higher accuracy (lines 11–15 of the Algorithm 2). The algorithm returns back to the point such that the edge (u, v) can be reached from this point with sufficient precision using the exact LCA.

When the label at the node v (AF f_v) is updated (the condition at the line 16 is fulfilled), the edge (u, v) is added to the predecessor list $pred(v)$ of the node v (lines 17–20). The set of predecessors form a graph $R = (V_R, E_R)$ (red color in Figure 2). We assume that the graph R is acyclic. In general, the graph R may not be acyclic, but in real case it is very unlikely.

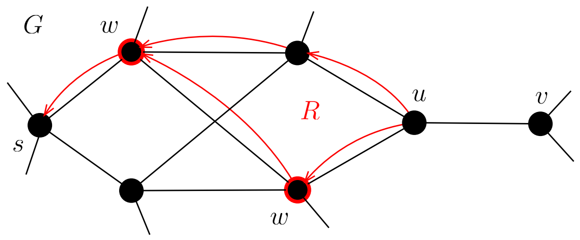

Figure 2. Example of a backsearch procedure.

We want to find nodes w such that if the exact LCA is performed from these nodes w, the AF f_v is an ε-approximation. The nodes w have to satisfy the following inequalities in the interval $[t_m, t_n]$:

$$\max\left\{\prod_{e\in p}\alpha_e \middle| p \in P_{vw}\right\} \le \min\left(\frac{\overline{g}_v^{\downarrow}}{g_w^{\downarrow}}\right), \tag{4}$$

where P_{vw} is the set of all paths from v to w in the graph R (the paths must contain the edge (u, v)) and α_e represents the maximum slope of AF f_e corresponding with the edge e. The set W is the set of all nodes w that meet the condition (4) and there is a path $p \in P_{vw}$ that does not contain any other node from the set W (W is the smallest possible).

Algorithm 2: ε-LCA-BS.

```
1  PQ = minimum priority queue where key is min f
2  $\forall u \in V : f_u^{\updownarrow} = \infty$
3  $g_s^{\updownarrow} = 0$
4  PQ.put($f_s^{\updownarrow}$)
5  pred($s$) = null
6  while queue is not empty do
7      $f_u^{\updownarrow} = \text{PQ.get()}$
8      foreach $v : (u,v) \in E$ do
9          $\delta(t) = \varepsilon((f_{uv} * f_u^{\updownarrow})^{\downarrow}(t) - t) - \alpha(t)\varepsilon g_u^{\downarrow}(t)$        // according Equation (3)
10         $\overline{f}_v = approx(f_{uv} * f_u^{\updownarrow}, \delta^+)$        // $\delta^+(t) = \max(\delta(t), 0)$
11         if $\exists t : \delta(t) < 0$ then
12             find all intervals $[t_m, t_n]$ where $\delta$ is negative
13             foreach $[t_m, t_n]$ do
14                 $h = backSearch((u,v), [t_m, t_n])$
15                 substitute $\overline{f}_v$ by $h$ in interval $[t_m, t_n]$
16         if $\exists t : \overline{f}_v(t) < f_v^{\updownarrow}(t)$ then
17             if $\overline{f}_v < f_v^{\updownarrow}$ then
18                 pred($v$) = $(u,v)$
19             else
20                 pred($v$).add($(u,v)$)
21             $f_v^{\updownarrow} = \min(\overline{f}_v, f_v^{\updownarrow})$
22             PQ.put($f_v^{\updownarrow}$)
```

These nodes w can be found using a topological ordering of V_R (Algorithm 3). The node labels α_b correspond to the left side of the inequalities (4), so the algorithm finds maximal paths in R. First, the labels are set to negative infinity (the line 2) and the label at u is set to α_{uv} (the line 3). The lines 4–5 ensure a topological ordering. If the condition (4) in line 6 is fulfilled, b is added to the W. The lines 9–12 ensure updating of the node labels. The part of $\overline{f}_v$ in the interval $[t_m, t_n]$ is substituted by a more accurate result of the exact LCA with the initial priority queue PQ that is created by adding all $f_w \in \{f_w | w \in W\}$ (the lines 13–16).

In Figure 2 there is an example of backsearch. The black color represents the original graph G and the red color represents the acyclic graph R. The edge (u,v) violates the condition (2). Then the algorithm starts *backSearch* procedure and finds the set W (red nodes) using the graph R.

Algorithm 3: backSearch.

```
$W = \{\}$                                                    // set of all $w$
$\forall b \in V_R : \alpha_b = -\infty$
$\alpha_u = \alpha_{uv}$
while $\exists b : b \in V_R \setminus W \wedge deg^-(b) = 0$ do     // topological ordering
    $b =$ some node that meets the conditions above
    if $\alpha_b \leq \min\left(\frac{\overline{g}_v^{\downarrow}}{g_b^{\downarrow}}\right)$ then
        $W = W \cup \{b\}$
    else
        foreach $a : (b, a) \in R$ do
            if $\alpha_b \alpha_{ba} > \alpha_a$ then
                $\alpha_a = \alpha_b \alpha_{ba}$
        $V_R = V_R \setminus \{b\}$
foreach $w \in W$ do
    PQ.put($f_w$)
run LCA on G with initial priority queue PQ on interval $[t_m, t_n]$    // algorithm 1
return $f_v$
```

When the graph R is not acyclic, it is necessary to modify the algorithm for searching the set W.

If the condition (2) is fulfilled, the ε-LCA-BS is reduced to ε-LCA, because the algorithm then does not perform any *backSearch* procedure.

The ε-LCA-BS has one main disadvantage. If there are a lot of edges with the arrival function where the maximum slope is too big, the algorithm will perform a lot of *backSearch* procedures (Algorithm 3). The *backSearch* procedure is time-consuming so the whole algorithm will be slow in such a case. But, in a real road network, the maximum slopes of AFs are not too steep [9,14]. Therefore, the main disadvantage is not a too big problem.

3.3. Heuristic Improvement

The proposed solution can be further accelerated on the basis of departure time interval decomposition as follows. The profile search problem on the departure time interval $[a, b]$ can be decomposed in time to two subproblems on intervals $[a, c]$ and $[c, b]$ and these subproblems are independent [15]. It follows that the problem can be split into N independent computation parts. This property enables effective parallelization and distribution of the computation.

A surprising property is that the splitting of the problem often saves total computation time in the serial case, too. The ε-LCA-BS runtime on the interval $[a, b]$ is often higher than the sum of runtimes on the intervals $[a, c]$ and $[c, b]$. In the following text, the reasons why the decomposition is faster than the original problem are presented. Some splitting strategies are also discussed.

The number of relaxations K (the number of iterations of the main loop) of ε-LCA-BS is greater than or equal to the number of relaxations of the Dijkstra's algorithm [2]. The aim is to approach as close to the Dijkstra's algorithm as possible. The condition at line 16 in Algorithm 2 controls adding to the priority queue (PQ) and thereby controls the number of iterations. The condition compares two AFs. The small piece of the $\overline{f}_v$ under $f_v^{\updownarrow}$ is enough for adding to the PQ. It follows that a more fluctuating TTF indicates a higher K.

The fluctuation of the TTFs can be reduced by splitting the function to intervals. The function on a shorter interval has a smaller or equal difference between its maximum and minimum. It means that in more cases the condition $\exists t : \overline{f}_v(t) < f_v^{\updownarrow}(t)$ is false. In Figure 3 there is an example of splitting. The original function $\overline{f}_v(t)$ on the interval $[a, d]$ is divided into three intervals $[a, b]$, $[b, c]$ and $[c, d]$.

As you can see, for the intervals $[a, b]$ and $[c, d]$ the condition at line 16 in Algorithm 2 is false. Thus, the node will not be added to the PQ and K is reduced.

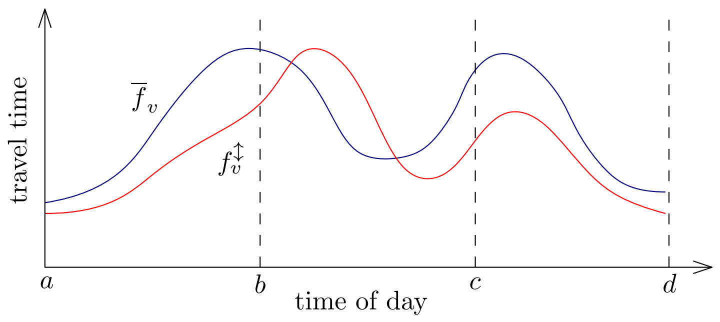

Figure 3. Splitting effect.

The question is how to split the departure time interval so that the computation is the most effective. In Figure 4 there is a visualization of speed profiles for the testing dataset. The x-axis represents a time of day and the y-axis displays the average speed in time on edges. In the visualization, all speed profiles from the tested dataset are rendered in various colors. As you can see, the fluctuation in the night time (the first part and the last part) is small. There are two main concepts for splitting:

- Split the origin interval into equal subintervals.
- Split the origin interval into inhomogeneous subintervals (e.g., longer at night and shorter by day).

Which of these concepts is better strongly depends on the TTFs shape, as further presented in experiments.

The splitting into intervals enables a very simple parallelization. Due to the independence in the departure time intervals, we can set one interval equal to one thread.

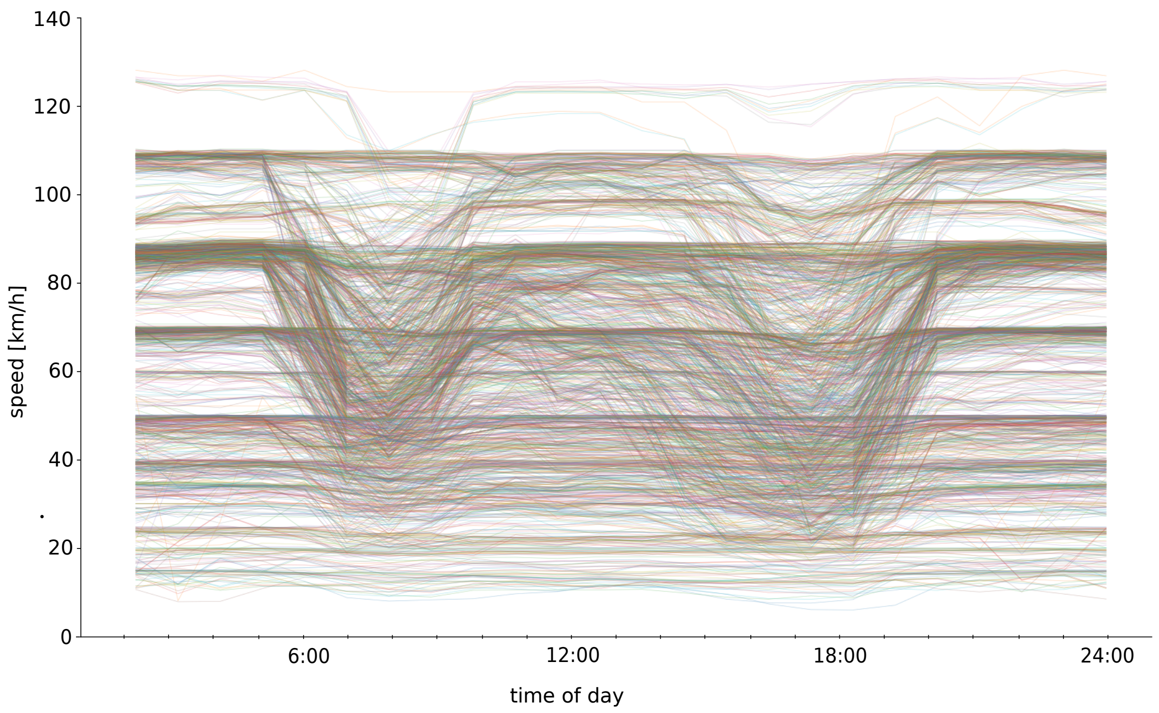

Figure 4. Speed profiles visualization. The colors represent the individual speed profiles.

4. Experiments

The real road network with real speed profiles that were computed from GPS tracks was used for testing. These data represent part of Paris (Figure 5). Compared to [9], the speed profiles' preprocessing was improved so the measured results are a little bit different.

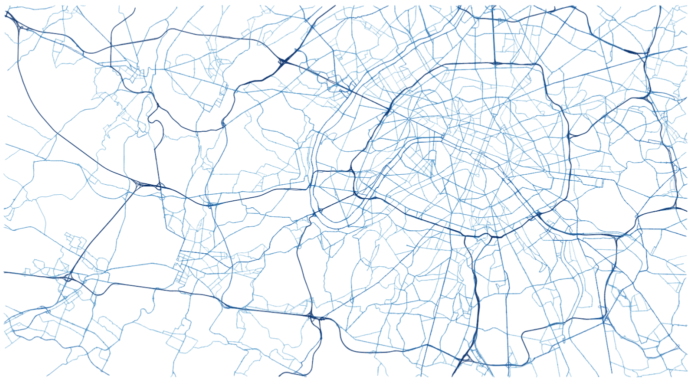

Figure 5. Route network for testing.

The algorithms were implemented using Scala programming language (OpenJDK 1.9, Debian 11). The testing was performed on a computer with Intel$^{(R)}$ Core$^{(TM)}$ i5-8250U CPU with 1.60GHz and 16 GB RAM.

4.1. The ϵ-LCA-BS Testing

Only the ϵ-LCA-BS was tested because ϵ-LCA is only a special case of ϵ-LCA-BS. The maximum allowed relative error ε was set to $10^{-2}, 10^{-3}, 10^{-4}$ and 10^{-5}.

Four graphs (G1, G2, G3 and G4) were created to show the performance of the developed algorithms. Every edge in the graphs has AF with 24 linear pieces. Every graph represents different classes of roads. In Table 1 there are numbers of edges for each graph and performance results of ϵ-LCA-BS: the relative time t_r and the relative number of breakpoints bps related to the exact version of LCA. The same results can be seen in Figure 6.

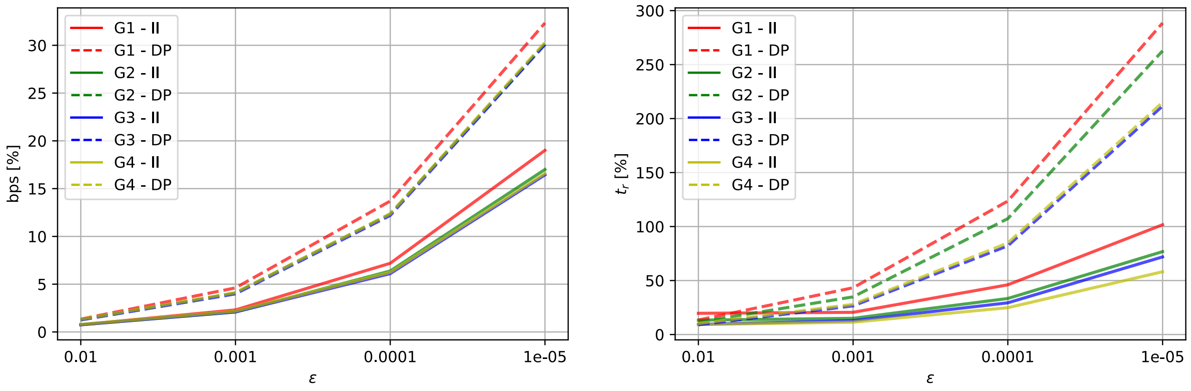

Figure 6. The relative number of breakpoints and the relative time related to exact LCA (II—Imai and Iri, DP—Douglas–Peucker).

Table 1. Results of testing ϵ-LCA-BS (t_r—the relative time related to the exact version of LCA, bps—the relative number of breakpoints, ε—the maximum allowed relative error).

Dataset	# Edges	ε	Imai and Iri		Douglas Peucker	
			t_r [%]	bps [%]	t_r [%]	bps [%]
G 1	10,798	10^{-2}	19.7	0.8	13.4	1.3
		10^{-3}	20.6	2.3	43.3	4.6
		10^{-4}	46.0	7.2	123.6	13.7
		10^{-5}	101.7	19.0	288.5	32.3
G2	33,354	10^{-2}	13.3	0.8	11.6	1.3
		10^{-3}	15.0	2.1	34.9	4.1
		10^{-4}	33.4	6.4	107.3	12.4
		10^{-5}	76.8	17.0	262.7	30.1
G3	107,476	10^{-2}	9.4	0.8	9.0	1.3
		10^{-3}	12.8	2.1	26.5	4.0
		10^{-4}	29.4	6.1	82.1	12.2
		10^{-5}	71.9	16.4	211.6	30.1
G4	160,092	10^{-2}	9.2	0.8	10.2	1.3
		10^{-3}	11.4	2.2	27.9	4.1
		10^{-4}	24.8	6.3	84.8	12.4
		10^{-5}	58.1	16.6	215.0	30.3

The results in Table 1 show that the maximum relative error 10^{-4} brings only a small improvement and the maximum relative error 10^{-5} is slower than the exact version of the LCA, but this accuracy is too big for a real use. The maximum relative error from 10^{-2} to 10^{-3} seems to be a good compromise between accuracy and performance.

In Table 2 there are absolute values of parameters measured for the maximal relative error 10^{-3}. The column bps represents the number of breakpoints in the resulting AFs.

Table 2. Absolute values of parameters measured for $\varepsilon = 0.001$.

	Imai and Iri		Douglas Peucker	
	Time [s]	# bps	Time [s]	# bps
G1	0.9	561,853	1.8	1,122,458
G2	3.0	1,429,080	5.6	2,779,762
G3	9.3	3 855,536	18.8	7,352,745
G4	14.1	5,298,280	28.8	10,057,055

The results show that the breakpoints savings are significant. That means the ε-LCA-BS saves a lot of memory. Let us assume a path that takes 1 h; then, the relative error 0.1 % implicates the absolute error 3.6 s. In this case the epsilon approximation saves more than 97% of memory and 80% of time. In case that ε is too small, the algorithm can run slower than the exact version, because the simplification takes too much time.

The disadvantage of the ε-LCA-BS is that it is sensitive to values of the maximum slope of AFs. If AFs have too big slope then the algorithm performs too many calls of the *backSearch* procedure and thereby makes the computation too slow. In practice, the functions usually have small slopes. When it is certain that input data do not violate the condition (2), the algorithm is more suitable.

Furthermore, the computation of **V** breakpoints was implemented. This computation is the first step of the all algorithms presented in [1,3,4,16]. As mentioned above, the step needs $3\sum_{(u,v)\in E}|f_{uv}|$ static shortest path computation. The **V** breakpoints determination was performed on G1 and takes 21 min.

4.2. Splitting Tests

The first test is focused on equal subintervals. The original departure time interval was divided into 2–12 subintervals. The two datasets (G1 and G2) were used and the maximum relative error was set to 10^{-2}, 10^{-3}, and 10^{-4}. The more effective simplification method by Imai and Iri was set for all tests in this section. The test was performed only in serial (one thread was used only). In Figure 7 there are results of the test. The chart shows the dependence between the speed-up and the number of subintervals. The speed-up is defined as:

$$\text{speed-up}(N) = \frac{\text{runtime for 1 interval}}{\text{runtime for N subintervals}}. \tag{5}$$

As you can see, the speed-up is between 1.1 and 1.8 and is very variable, but never under 1. For our testing data, 4–10 equal subintervals are a good choice. In case when the number of subintervals is too big (in our case more than 10), the overhead costs override the benefits of splitting.

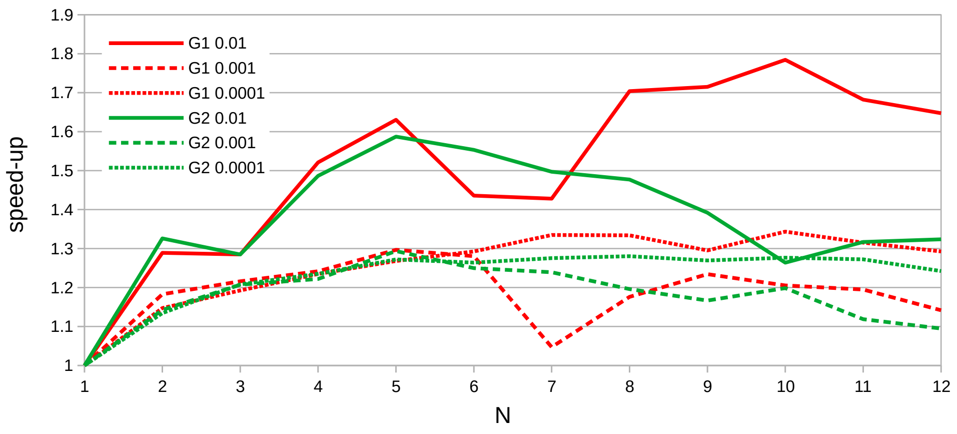

Figure 7. Speed-up dependence on the number of subintervals N in serial case (1 thread).

The inhomogeneous subintervals were also tested. Several methods for the departure time interval splitting were tried, but the runtime was only a few percent better than the splitting into equal subintervals. The conclusion is that the equal subintervals, although also more simple, work better.

The last test demonstrates the suitability of parallelization. The dataset G1 with $\varepsilon = 0.001$ was used. The algorithm was performed with three settings: without splitting into subintervals, with four subintervals in the serial case (1 thread) and with four subintervals in four threads. The code was performed on one computer with multi-core processor. The following runtimes were measured (Table 3).

Table 3. Paralellization runtimes on G1 with $\varepsilon = 0.001$.

1 Interval, 1 Thread	4 Intervals, 1 Thread	4 Intervals, 4 Threads
8.6 s	7.6 s	2.6 s

The testing shows that the parallelization is suitable so it is recommended.

5. Conclusions

Two algorithms for ε-approximation of TDSP and their heuristic improvement for real data were presented. The algorithms significantly reduce the memory use. When the maximum relative error is a sufficiently large value (in our case 10^{-3}), the algorithms save the computational time, too. From this point of view, the algorithms are suitable for precomputing the TTFs for the next use (e.g., time-dependent distance oracles and time-dependent contraction hierarchies).

It has been shown that the splitting into departure time subintervals can further reduce the runtime and this splitting is suitable for parallelization or distribution because the subproblems are independent.

In future work, it would be useful to utilize some technique for graph size reduction based on static shortest path search. The goal is to remove the edges that will certainly not be used. In a one-to-one problem case the developed algorithms can be combined with other speed-up techniques that reduce the graph (e.g., time-dependent-sampling [14]).

Author Contributions: Conceptualization, František Kolovský, Jan Ježek and Ivana Kolingerová; investigation, František Kolovský, Jan Ježek and Ivana Kolingerová; writing—original draft, František Kolovský and Ivana Kolingerová; supervision, Ivana Kolingerová and Jan Ježek; software, František Kolovský.

Funding: This work has been supported by the Project SGS-2019-015 (”Využití matematiky a informatiky v geomatice IV”) and by the Ministry of Education, Youth and Sports of the Czech Republic, project PUNTIS (LO1506) under the program NPU I.

Acknowledgments: In this section you can acknowledge any support given which is not covered by the author contribution or funding sections. This may include administrative and technical support, or donations in kind (e.g., materials used for experiments).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Foschini, L.; Hershberger, J.; Suri, S. On the Complexity of Time-Dependent Shortest Paths. *Algorithmica* **2014**, *68*, 1075–1097. [CrossRef]
2. Orda, A.; Rom, R. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *J. ACM (JACM)* **1990**, *37*, 607–625. [CrossRef]
3. Dehne, F.; Omran, M.T.; Sack, J.R. Shortest Paths in Time-Dependent FIFO Networks. *Algorithmica* **2012**, *62*, 416–435. [CrossRef]
4. Omran, M.; Sack, J.R. Improved approximation for time-dependent shortest paths. In Proceedings of the International Computing and Combinatorics Conference, Atlanta, GA, USA, 4–6 August 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 453–464.
5. Geisberger, R.; Sanders, P. *Engineering Time-Dependent Many-To-Many Shortest Paths Computation*; OASIcs-OpenAccess Series in Informatics; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Wadern, Germany, 2010; Volume 14.
6. Batz, G.V.; Geisberger, R.; Sanders, P.; Vetter, C. Minimum time-dependent travel times with contraction hierarchies. *J. Exp. Algorithmics* **2013**, *18*, 1.1–1.43. [CrossRef]
7. Geisberger, R. Engineering Time-dependent One-To-All Computation. *arXiv* **2010**, arXiv: 1010.0809.
8. Bellman, R. On a routing problem. *Q. Appl. Math.* **1958**, *16*, 87–90. [CrossRef]
9. Kolovský, F.; Ježek, J.; Kolingerová, I. The e-approximation of the Label Correcting Modification of the Dijkstra’s Algorithm. In Proceedings of the 5th International Conference on Geographical Information Systems Theory, Applications and Management—Volume 1: GISTAM, Crete, Greece, 3–5 May 2019; SciTePress, Setubal, Portugal, 2019; pp. 26–32. [CrossRef]
10. Gilsinn, J.F.; Witzgall, C. *A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees*; US National Bureau of Standards, Washington, DC, USA, 1973; Volume 772.
11. Pape, U. Implementation and efficiency of Moore-algorithms for the shortest route problem. *Math. Program.* **1974**, *7*, 212–222. [CrossRef]
12. Bardossy, M.G.; Shier, D.R. Label-correcting shortest path algorithms revisited. In *Perspectives in Operations Research*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 179–197.
13. Imai, H.; Iri, M. An optimal algorithm for approximating a piecewise linear function. *J. Inf. Process.* **1986**, *9*, 159–162.
14. Strasser, B. *Dynamic Time-Dependent Routing in Road Networks through Sampling*; OASIcs-OpenAccess Series in Informatics; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Wadern, Germany, 2017; Volume 59.

15. Dean, B.C. *Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms*; Rapport Technique; Massachusetts Institute of Technology: Cambridge, MA, USA, 2004.
16. Dehne, F.; Omran, M.T.; Sack, J.R. Shortest paths in time-dependent FIFO networks using edge load forecasts. In Proceedings of the Second International Workshop on Computational Transportation Science, Seattle, WA, USA, 3 November 2009; ACM: New York, NY, USA, 2009; pp. 1–6.

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).