Space and Time Trade-Off for the k Shortest Simple Paths Problem

Ali Al Zoobi
Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis Cedex, France
http://www-sop.inria.fr/members/Ali.Al-Zoobi/
ali.al-zoobi@inria.fr

David Coudert
Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis Cedex, France
http://www-sop.inria.fr/members/David.Coudert/
david.coudert@inria.fr

Nicolas Nisse
Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis Cedex, France
http://www-sop.inria.fr/members/David.Coudert/
nicolas.nisse@inria.fr

Abstract

The k shortest simple path problem (kSSP) asks to compute a set of top-k shortest simple paths from a vertex s to a vertex t in a digraph. Yen (1971) proposed the first algorithm with the best known theoretical complexity of $O(kn(m + n \log n))$ for a digraph with n vertices and m arcs. Since then, the problem has been widely studied from an algorithm engineering perspective, and impressive improvements have been achieved.

In particular, Kurz and Mutzel (2016) proposed a *sidetracks-based* (SB) algorithm which is currently the fastest solution. In this work, we propose two improvements of this algorithm.

We first show how to speed up the SB algorithm using dynamic updates of shortest path trees. We did experiments on some road networks of the 9th DIMAC'S challenge with up to about half a million nodes and one million arcs. Our computational results show an average speed up by a factor of 1.5 to 2 with a similar working memory consumption as SB. We then propose a second algorithm enabling to significantly reduce the working memory at the cost of an increase of the running time (up to two times slower). Our experiments on the same data set show, on average, a reduction by a factor of 1.5 to 2 of the working memory.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Shortest paths; Theory of computation → Design and analysis of algorithms

Keywords and phrases k shortest simple paths, graph algorithm, space-time trade-off

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.18

Supplementary Material The code of our algorithms is publicly available at `https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths`.

Funding This work has been supported by the French government, through the UCA^{JEDI} Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-15-IDEX-01, the ANR project MULTIMOD with the reference number ANR-17-CE22-0016, the ANR project Digraphs with the reference number ANR-19-CE48-0013, and by Région Sud PACA.

1 Introduction

The classical k shortest paths problem (kSP) returns the top-k shortest paths between a pair of source and destination nodes in a graph. This problem has numerous applications in various kinds of networks (road and transportation networks, communications networks,

social networks, etc.) and is also used as a building block for solving optimization problems. Let $D = (V, A)$ be a digraph, an s-t path is a sequence $(s = v_0, v_1, \cdots, v_l = t)$ of vertices starting with s and ending with t, such that $(v_i, v_{i+1}) \in A$ for all $1 \leq i < l$. It is called simple if it has no repeated vertices, i.e., $v_i \neq v_j$ for all $0 \leq i < j \leq l$. The length of a path is the sum of the weights of its arcs and the top-k shortest paths is therefore the set containing a shortest s-t path, a second shortest s-t paths, *etc.* until the k^{th} shortest s-t path.

Several algorithms for solving kSP have been proposed. In particular, Eppstein [5] proposed an exact algorithm that computes k shortest paths (not necessarily simple) with time complexity in $O(m + n \log n + k)$, where m is the number of arcs and n the number of vertices of the graph. An important variant of this problem is the k shortest *simple* paths problem (kSSP) introduced in 1963 by Clarke *et al.* [3] which adds the constraint that reported paths must be simple. This variant of the problem has various applications in transportation network when paths with repeated vertices are not desired by the user. It is also a subproblem of other important problems like constrained shortest path problem, vehicle and transportation routing [10, 12, 19]. It can be applied successfully in bio-informatics [1], especially in biological sequence alignment [17] and in natural language processing [2]. For more applications, see Eppstein's recent comprehensive survey on k-best enumeration [6].

The algorithm with the best known time complexity for solving the kSSP problem has been proposed by Yen [20], with time complexity in $O(kn(m + n \log n))$. Since, several works have been proposed to improve the efficiency of the algorithm in practice [10, 14, 11, 7, 16].

Recently, Kurz and Mutzel [16, 15] obtained a tremendous running time improvement, designing an algorithm with the same flavor as Eppstein's algorithm. The key idea was to postpone as much as possible the computation of shortest path trees. To do so, they define a path using a sequence of shortest path trees and *deviations*. With this new algorithm, they were able to compute hundreds of paths in graphs with million nodes in about one second, while previous algorithms required an order of tens of seconds on the same instances. For instance, Kurz and Mutzel's algorithm computed $k = 300$ hundred shortest paths in 1.15 second for the COL network [4] while it required about 80 seconds for the Yen's algorithm and about 30 seconds by its improvement by Feng [7].

Our contribution. We propose two variations of the algorithm proposed by Kurz and Mutzel. We first show how to speed up their algorithm using dynamic updates of shortest path trees resulting in an average speed up by a factor of 1.5 to 2 and with a similar working memory consumption (i.e., the total memory consumption excluding the memory allocated for the input and the output). We then propose a second algorithm enabling a significant reduction of the working memory at the cost of a small increase of the running time.

This paper is organized as follows. First, in Section 2.2, we describe Yen's algorithm and then show in Section 2.3, how Kurz and Mutzel's algorithm improves upon it. In Section 3, we present our algorithms by precisely describing how they differ from Kurz and Mutzel's algorithm. Finally, Section 4 presents our simulation settings and results.

2 Preliminaries

2.1 Definition and Notation

Let $D = (V, A)$ be a directed graph (digraph for short) with vertex set V and arc set A. Let $n = |V|$ be the number of vertices and $m = |A|$ be the number of arcs of D. Given a vertex $v \in V$, $N^+(v) = \{w \in V \mid vw \in A\}$ denotes the out-neighbors of v in D. Let $w_D : A \to \mathbb{R}^+$ be a length function over the arcs. For every $u, v \in V$, a *(directed) path* from

u to v in D is a sequence $P = (v_0 = u, v_1, \cdots, v_r = v)$ of vertices with $v_i, v_{i+1} \in A$ for all $0 \leq i < r$. Note that vertices may be repeated, i.e., paths are not necessarily simple. A path is *simple* if, moreover, $v_i \neq v_j$ for all $0 \leq i < j \leq r$. The length of the path P equals $w_D(P) = \sum_{0 \leqslant i < r} w_D(v_i, v_{i+1})$ (we will omit D when there is no ambiguity). The distance $d_D(u, v)$ between two vertices $u, v \in V$ is the shortest length of a u-v path in D (if any). Given two paths $(v_1, \cdots, v_r)$ and $Q = (w_1, \cdots, w_p)$ and $v_r w_1 \in A$, let us denote the v_1-w_p path obtained by the concatenation of the two paths by $(v_1, \cdots, v_r, Q)$.

Given $s, t \in V$, a *top-k set of shortest s-t paths* is any set S of (pairwise distinct) simple s-t paths such that $|S| = k$ and $w(P) \leq w(P')$ for every s-t path $P \in S$ and s-t path $P' \notin S$. The k shortest simple paths problem takes as input a weighted digraph $D = (V, A)$, $w_D : A \to \mathbb{R}^+$ and a pair of vertices $(s, t) \in V^2$ and asks to find a top-k set of shortest s-t paths (if they exist).

Let $t \in V$. An *in-branching* T rooted at t is any sub-digraph of D that induces a tree containing t, such that every $u \in V(T) \setminus \{t\}$ has exactly one out-neighbor (that is, all paths go toward t). An in-branching T is called a *shortest path (SP) in-branching* rooted at t if, for every $u \in V(T)$, the length of the (unique) u-t path P^T_{ut} in T equals $d_D(u, t)$. Note that an SP in-branching is sometimes called *reversed shortest path tree*.

In the forthcoming algorithms, the following procedure will often be used (and the key point when designing the algorithms is to limit the number of such calls and to optimize each of them). Given a sub-digraph H of D and $u, t \in V(H)$, we use Dijkstra's algorithm for computing an SP in-branching rooted in t that contains a shortest u-t path in H. Note that, the execution of the Dijkstra's algorithm may be stopped as soon as a shortest u-t path has been computed (when u is reached), i.e., the in-branching may only be partial (i.e., not spanning D). The key point will be that this way to proceed not only returns a shortest u-t path in H (if any) but an SP in-branching rooted in t, containing u. Note that any such call has worst-case time complexity $O(m + n \log n)$.

Let $P = (v_0, v_1, \cdots, v_r)$ be any path in D and $i < r$. Any arc $a = v_i v' \neq v_i v_{i+1}$ is called a *deviation* of P at v_i. Moreover, any path $Q = (v_0, \cdots, v_i, v', v'_1, \cdots, v'_\ell = v_r)$ is called an *extension* of P at a (or at v_i). Note that neither P nor Q is required to be simple. However, if Q is simple, it will be called a *simple extension* of P at a (or at v_i). In addition, Q is called a shortest (simple) extension at v_i if and only if Q is an extension with minimum length among all (simple) extensions of P at v_i. Furthermore, Q is called a shortest (simple) extension at a if and only if Q is an extension with minimum length among all (simple) extensions of P at a.

2.2 General framework: Yen's algorithm and Feng's improvements

We start by describing the general framework used by the kSSP algorithms in [10, 14, 11, 7, 16]. Precisely, let us describe Yen's algorithm [20] trying to give its main properties and drawbacks. Then, we explain how Kurz and Mutzel's algorithm improves upon it (Section 2.3). Finally, we will detail our adaptation of the latter method in Section 3.

All of the algorithms described below start by computing a shortest s-t path $P_0 = (s = v_0, v_1, \cdots, v_r = t)$ (in what follows we always assume that there is at least one such path). This is done by applying Dijkstra's algorithm from t (as described in previous section), so also computes an SP in-branching T_0 rooted at t and containing s. Note that P_0 is simple since weights are non-negative. Obviously, a second shortest s-t simple path must be a shortest simple extension of P_0 at one of its vertices. Yen's algorithm computes a shortest simple extension of P_0 at v_i for every vertex v_i in P_0 as follows. For every $0 \leq i < r$, let $D_i(P_0)$ be the graph obtained from D by removing the vertices $v_0, \cdots, v_{i-1}$ (this is to avoid

non simple extension) and the arc v_iv_{i+1} (to ensure that the computed path is a new one, i.e., different from P_0). For every $0 \leq i < r$, an SP in-branching in $D_i(P_0)$ rooted at t is computed using Dijkstra's algorithm until it reaches v_i and therefore returns a shortest path Q_i from v_i to t. Note that Yen's algorithm no longer uses the SP in-branchings and this will be one key improvement described further. For every $0 \leq i < r$, the extension $(v_0, \cdots, v_i, Q_i)$ of P_0 at v_i is added to a set *Candidate* (initially empty). Note that the index i (called below *deviation-index*) where the path $(v_0, \cdots, v_i, Q_i)$ deviates from P_0 is kept explicit[1]. Once $(v_0, \cdots, v_i, Q_i)$ has been added to *Candidate* for all $0 \leq i < r$, by remark above, a shortest path in *Candidate* is a second shortest s-t simple path.

More generally, by induction on $0 < k' < k$, let us assume that a top-k' set S of shortest s-t paths has been computed and the set *Candidate* contains a set of simple s-t paths such that there exists a shortest path $Q \in Candidate$ such that $S \cup \{Q\}$ is a top-$(k'+1)$ set of shortest s-t paths. Yen's algorithm pursues as follows. Let $Q = (v_0 = s, \cdots, v_r = t)$ be any shortest path in *Candidate*[2] and let $0 \leq j < r$ be its deviation-index. First, Q is extracted from *Candidate* and it is added to S (as the $(k'+1)^{th}$ shortest s-t path). Then, every shortest extension of Q is added to *Candidate* (since they are potentially a next shortest s-t path). For this purpose, for every $j \leq i < r$, let $D_i(Q)$ be the digraph obtained from D by first removing the vertices $v_0, \cdots, v_{i-1}$ (this is to avoid non simple extension). Then, here is one important bottleneck of Yen's algorithm, for every arc v_iv' such that *Candidate* already contains some path with prefix $(v_0, \cdots, v_i, v')$, then v_iv' is removed from $D_i(Q)$. This therefore ensures to compute only new paths. Indeed, the computed extensions are distinct from every path previously computed as they have different prefixes (this is the reason to keep explicitly the deviation-index). For every $j \leq i < r$, an SP in-branching rooted at t is computed using Dijkstra's algorithm until it reaches v_i and therefore returns a shortest path Q_i from v_i to t in $D_i(Q)$. For every $0 \leq i < r$, the extension $(v_0, \cdots, v_i, Q_i)$ of Q at v_i (together with its deviation index i) is added to the set *Candidate*. This process is repeated until k paths have been found (when $k' = k$).

Therefore, for each path Q that is extracted from *Candidate*, $O(|V(Q)|)$ applications of Dijkstra's algorithm are done. This gives an overall time-complexity of $O(kn(m + n \log n))$ which is the best theoretical (worst-case) time-complexity currently known (and of all algorithms described in this paper) to solve the k-shortest simple paths problem.

One expensive part in the pre-described framework of Yen's algorithm are the multiple calls of Dijkstra's algorithm. Feng [7] proposed a practical improvement by trying to avoid some calls. Essentially, when a path $Q = (v_0, \cdots, v_r)$ with deviation-index j is extracted, its extensions are computed from $i = j$ to $r - 1$. Roughly, for every $j < i \leq r$, the computation of the extension at v_i is actually done with the help of the initial SP in-branching T_0.

In practice, this process significantly accelerates the executions of Dijkstra's algorithm. At the price of a larger memory consumption, Kurz and Mutzel improved Yen's framework which leads to the fastest algorithm currently known (Section 2.3) for the k shortest simple paths problem.

---

[1] The deviation-index is not kept explicitly in Yen's algorithm but, since it is a trivial improvement already existing in the literature, we mention it here.

[2] Actually *Candidate* is implemented, using a pairing heap, in such a way that extracting a shortest path in it takes logarithmic time and insertions are done in constant time.

2.3 Kurz and Mutzel's algorithm

All of Yen's improvements aim at minimizing the time consumed by Dijktra's algorithm calls. Instead, Kurz and Mutzel [16] chose to use a smaller number of such calls. This can be done by memorizing the SP in-branchings previously computed by the algorithm. More precisely, instead of keeping the paths in the set $Candidate$, the algorithm keeps only a representation of it using a sequence of SP in-branchings and deviations. These representations will allow to extract any shortest path P in time $O(|P|)$ and the length of P in constant time. As a result, for each shortest s-t path P, the memorized SP in-branching can be used to extract a shortest extension of P at a vertex v_i in a pivot step. Unfortunately, there is no guarantee that the extracted shortest extension will be simple. If it is not simple, a new Dijsktra's algorithm call has to be done. However, in many cases the extracted extension is simple and a Dijsktra's algorithm call can be avoided.

Precisely, Kurz and Mutzel's algorithm mainly relies on two key improvements. First, following a principle of Eppstein's algorithm [5], it explicitly keeps the SP in-branchings computed during the execution of the algorithm (this is achieved at some non-negligible cost of working memory consumption, but leads to an improvement of the practical running-time). Moreover, instead of computing the extensions of the extracted path in each iteration, the algorithm adds to $Candidate$ a representation of each extension (together with a lower bound of its length). Then, only when such a representation is extracted from $Candidate$, the corresponding extension is explicitly computed. This way of postponing the computations allows to avoid the actual computation of many extensions (which are not used any further), which leads to a drastic improvement of the running time.

Let us describe the Kurz and Mutzel's algorithm whose pseudo code is given in Algorithm 1. As usual, the algorithm starts with the computation of a shortest s-t (simple) path $P_0 = (v_0 = s, v_1, \cdots, v_r = t)$ together with an SP in-branching tree T_0 rooted at t and containing s. T_0 is added to a set $\mathcal{T}$ initially empty (this set $\mathcal{T}$ will contain all computed SP in-branchings). Then, for every $0 \leq i < r$, and for every deviation e at v_i (i.e., arcs $e = v_iv'$ are considered for every $v' \in N^+(v_i) \setminus \{v_0, \cdots, v_{i+1}\}$), let $P_{v't}(T_0)$ be the shortest path from v' to t in T_0. Note that the path $Q(i, e) = (v_0, \cdots, v_i, v', P_{v't}(T_0))$ is a shortest extension of P_0 at e, but it is not necessarily simple (it is not simple if $P_{v't}(T_0)$ intersects $\{v_0, \cdots, v_i\}$). Hence, $lb(e) = w((v_0, \cdots, v_i)) + w(v_iv') + w(P_{v't}(T_0))$ is a lower bound on the length of any shortest simple extension of P_0 at e (and it is its actual length if the path $Q(i, e)$ is simple). The algorithm proceeds as follows. First, by categorizing the vertices of T_0 (using a trick due to Feng [7] that we do not detail here), it is possible to decide in constant time, for each $i < r$ and each deviation e at v_i, whether $Q(e, i)$ is simple or not. Then, for every $0 \leq i < r$, and for every deviation e at v_i, the algorithm adds $((T_0, e, T_0), lb(e))$ in a heap (ordered using lb) $Candidate_{simple}$ if $Q(e, i)$ is simple, and it adds $((T_0, e, T_i), lb(e))$ in a heap $Candidate_{not-simple}$ otherwise, where T_i is the name of a new SP in-branching rooted at t in $D \setminus \{v_0, \cdots, v_i\}$ whose actual computation is postponed. Hence, T_0 is the only SP in-branching that has been computed (using Dijkstra's algorithm) so far.

More generally, by induction on $0 < k' < k$, let us assume that a top-k' set S of shortest s-t paths and two heaps $Candidate_{simple}$ and $Candidate_{not-simple}$ have been computed. Following Eppstein's idea, each element of these heaps is of the form $((T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb)$ (describing a path as explained below) such that, for every $0 \leq i \leq h$, T_i is an SP in-branching that has already been computed and stored in $\mathcal{T}$, while (only if the element comes from $Candidate_{not-simple}$) T_{h+1} may not have already been computed but has a pointer associated to it stored in $\mathcal{T}$. That is, even if T_{h+1} has not yet been computed, it is defined and can be referred to. Observe that we may have $T_j = T_{j+1}$ for some $0 \leq j \leq h + 1$.

Algorithm 1 Kurz-Mutzel Sidetrack Based (SB) algorithm for the kSSP [16].

Require: A digraph $D = (V, A)$, a source $s \in V$, a sink $t \in V$, and an integer k
Ensure: k shortest simple s-t paths

Let $Candidate_{simple} \leftarrow \emptyset$, $Candidate_{not-simple} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
$T_0 \leftarrow$ an SP in-branching of D rooted at t containing s
Add $((T_0), w(P_{st}(T_0)))$ to $Candidate_{simple}$
while $Candidate_{simple} \cup Candidate_{not-simple} \neq \emptyset$ and $|Output| < k$ **do**
    $\varepsilon = ((T_0, e_0, \cdots, T_h, e_h = (u_h, v_h), T_{h+1}), lb) \leftarrow$ a shortest element in $Candidate_{simple}$ and $Candidate_{not-simple}$ // *with priority to* $Candidate_{simple}$
    if $\varepsilon \in Candidate_{simple}$ **then**
        Extract ε from $Candidate_{simple}$ and add it to the $Output$
        for every deviation $e = v_j v'$ with $v_j \in P_{v_h t}(T_{h+1})$ **do**
            $ext \leftarrow (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1})$
            $lb' \leftarrow lb - w(P_{v_j t}(T_{h+1})) + w(e) + w(P_{v' t}(T_{h+1}))$
            if ext represents a simple path **then**
                Add (ext, lb') to $Candidate_{simple}$
            else
                $T' \leftarrow$ the name of an SP in-branching of $D_h(P)$ // T' *is not computed yet*
                Add T' to $\mathcal{T}$
                Add $((T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T'), lb')$ to $Candidate_{not-simple}$
    else
        if T_{h+1} has not been computed yet **then**
            Compute T_{h+1}, an SP in-branching of $D_h(P)$ and add it to $\mathcal{T}$
        $\varepsilon' = ((T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb + w(P_{v' t}(T_{h+1})) - w(P_{v' t}(T_h)))$
        Add ε' to $Candidate_{simple}$
return $Output$

Let P_1 be the simple path that starts in s and, for every $0 \leq j \leq h$, follows the (already computed) tree T_j from the current vertex till the tail of the deviation e_j and then follows deviation e_j to its head. Hence, P_1 ends in the head z of e_h. Now, if the element is in $Candidate_{simple}$, we know by induction that the SP in-branching T_{h+1} has already been computed and that the path P obtained by concatenating P_1 and the shortest z-t path $P_{zt}(T_{h+1})$ is guaranteed to be simple and has length $w(P) = w(P_1) + w(P_{zt}(T_{h+1})) = lb(e_h)$. If the element is in $Candidate_{not-simple}$ (the shortest z-t path $P_{zt}(T_h)$ intersects P_1) the algorithm actually computes the SP in-branching T_{h+1} rooted at t (if not already done). Observe that the digraph in which T_{h+1} is computed is a subdigraph of the digraph in which T_h has been computed. Furthermore, $w(P_{zt}(T_{h+1})) \geq w(P_{zt}(T_h))$ (by setting $w(P_{zt}(T_{h+1})) = +\infty$ if there is no z-t path in T_{h+1}) and z is the only common vertex of P_1 and $P_{zt}(T_{h+1})$. Hence, the path P obtained by concatenating P_1 and the shortest z-t path $P_{zt}(T_{h+1})$ (if it exists) is guaranteed to be simple and has length $w(P) = w(P_1) + w(P_{zt}(T_{h+1})) \geq w(P_1) + w(P_{zt}(T_h)) = lb(e_h)$.

An iteration of Kurz and Mutzel's algorithm proceeds as follows. First an element $\varepsilon = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}), lb)$ with smallest lb is extracted from $Candidate_{simple}$ and $Candidate_{not-simple}$ (with priority for $Candidate_{simple}$ in case of equal lb). If ε was in $Candidate_{simple}$, the path P as defined above is the next shortest simple s-t path and it is added to the output. Then, all possible deviations of P along the path $P_{zt}(T_{h+1})$ are determined and added to $Candidate_{simple}$ or $Candidate_{not-simple}$ depending on whether they are simple or not (note that only a representation of them is build and not the actual path).

Otherwise, the algorithm actually computes the SP in-branching T_{h+1} (if not already done) to determine the shortest z-t path in T_{h+1} (if it exists), and adds T_{h+1} to $\mathcal{T}$. If such path exists, the algorithm adds to $Candidate_{simple}$ a new element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}), lb')$ describing a simple s-t path with length $lb' = w(P_1) + w(P_{zt}(T_{h+1})) = lb - w(P_{zt}(T_h)) + w(P_{zt}(T_{h+1}))$.

Actually, the same SP in-branching can be used for all deviations at the same vertex v_i of a given path P. So, for each vertex v_i in P, a single call of Dijkstra's algorithm is needed. As a result, finding all of the extensions of a given path P can be done in $O(|P|(m + n \log n))$ time. Therefore, the time complexity of Kurz and Mutzel's algorithm (in the worst case) is bounded by $O(kn(m + n \log n))$ as the algorithm extends no more than k paths and the number of vertices of each path is bounded by n.

Overall, Kurz and Mutzel's algorithm computes k shortest simple s-t paths with a much lower number of applications of Dijkstra's algorithm and so its running time is in general much better than the algorithms proposed by Yen or Feng. On the other hand, it requires to store many SP in-branchings previously computed which implies a larger working memory consumption.

3 Our contributions

We propose two independent variants of the SB algorithm (Algorithm 1). The first one, called SB*, gives, with respect to our experimental results, an average speed up by a factor of 1.5 to 2 compared with SB algorithm. The second one, called PSB (Parsimonious SB), is based on a different manner to handle non simple candidates in order to reduce the number of computed and stored SP in-branchings. This leads to a significant reduction of the working memory at the price of a slight increase in running time compared with SB algorithm.

3.1 The SB* algorithm

Here, we propose a variant of the SB algorithm, strongly based on Kurz and Mutzel's framework, that is a tiny modification of SB algorithm allowing to speed it up.

More precisely, each time a representation $(T_0, e_0, T_1 \cdots, e_{h-1} = (u_{h-1}, v_{h-1}), T_h, e_h = (u_h, v_h), T_{h+1})$ is extracted from $Candidate_{not-simple}$ and that T_{h+1} has not been computed yet (i.e., it is only a pointer), our algorithm does not compute T_{h+1} from scratch as SB algorithm does. Instead, the SB* algorithm creates a copy T of T_h, discards vertices of the path from v_{h-1} to u_h in T_h, and updates the SP in-branching T using standard methods for updating a shortest path tree [9]. Then, the pointer T_{h+1} is associated with the new in-branching T.

It is clear that the SB* algorithm computes (and stores) exactly the same number of in-branchings as the SB algorithm. The computational results presented in Section 4.2 show that this update procedure gives an average speed up by a factor of 1.5 to 2.

3.2 The PSB algorithm

Our main contribution is the Parsimonious SB algorithm (PSB) presented in this section whose main goal is to solve the k shortest simple paths problem with a good tradeoff between the running time and the working memory consumption. Indeed, a weak point of the SB algorithm comes from the fact that it keeps all the SP in-branchings that it computes throughout its execution in the memory . In order to reduce the working memory consumption, the main difference between the SB algorithm and the one presented here consists of the types of the elements that the PSB algorithm stores in the heap $Candidate_{not-simple}$ and

the way they are used. We now describe the PSB algorithm by detailing how its differs from the SB algorithm. Let us mention that the PSB algorithm uses a heap $Candidate_{simple}$ similar (i.e., containing exactly the same type of elements) to the one used by SB algorithm.

Let us start considering a step of PSB algorithm when an element $\varepsilon = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h = (u_h, v_h), T_{h+1}), lb)$ is extracted from $Candidate_{simple}$. The first difference between the SB algorithm is that T_{h+1} may have not yet been computed, in which case it must be computed at that step and stored in $\mathcal{T}$. Next, as the SB algorithm, the PSB algorithm first adds the (simple) path P corresponding to ε to the output. Then, it considers all deviations of P at the vertices between v_h and t, i.e., all arcs (not in P) with tail in $P_{v_h t}(T_{h+1})$. For every such deviation $e = uv$ with $u \in V(P_{v_h t}(T_{h+1}))$, by using Feng's "trick" (already mentioned without details), it can be decided in constant time whether it admits a simple extension, i.e., whether the path P_e that "follows" the path P_1 corresponding to ε from s to v_h, then follows the path $P_{v_h u}(T_{h+1})$, the arc e and finally the path $P_{vt}(T_{h+1})$ is simple or not. In the case when P_e is simple, then the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1}), lb_e)$ is added to $Candidate_{simple}$, where $lb_e = w(P_1) + w(P_{v_h u}(T_{h+1})) + w(e) + w(P_{vt}(T_{h+1}))$ (exactly as it is done by the SB algorithm). The second difference with the SB algorithm relies on the set $X = \{f_1, \cdots, f_r\}$ of deviations for which the extension using T_{h+1} is not simple. The key point is that we create a single element for all deviations in X. This ensures that the size of $Candidate_{not-simple}$ is at most k, as at most one element is added to $Candidate_{not-simple}$ per path added to the output. More precisely, let $X = \{f_1, \cdots, f_r\}$ be the set of "non simple" deviations ordered in such a way that, for every $1 \leq i < j < l \leq r$, the tail of f_j is between (or equal to) the tails of f_i and f_l on the path $P_{v_h t}(T_{h+1})$. For every $1 \leq i \leq r$ and $f_i = u_i v_i$, let $lb_i = lb_{f_i} = w(P_1) + w(P_{v_h, u_i}(T_{h+1})) + w(f_i) + w(P_{v_i, t}(T_{h+1}))$. The PSB algorithm then adds the element $\varepsilon' = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, X, T_{h+1}), \min_{1 \leq i \leq r} lb_i)$ to $Candidate_{not-simple}$, and so the weight of ε' in the heap $Candidate_{not-simple}$ is the smallest lower bound among all lower bounds related to the "non simple" deviations in X.

Let us now consider a step of the PSB algorithm when an element is extracted from $Candidate_{not-simple}$. This happens, as in the SB algorithm, when the smallest key (lower bound) of the elements in $Candidate_{simple} \cup Candidate_{not-simple}$ corresponds only to an element of $Candidate_{not-simple}$. Let $\varepsilon = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, X, T_{h+1}), lb)$ be this element and let $X = \{f_1, \cdots, f_r\}$. Let also $e_h = u_h v'_h$, let $P_1 = (s = x_1, \cdots, x_o = v'_h)$ be the prefix (from s to v'_h) of the path represented by ε, let $P_{v'_h t}(T_{h+1}) = (v'_h, v'_{h+1}, \cdots, v'_p = t)$ and let $f_j = v'_{i_j} v_j$ for every $1 \leq j \leq r$ (by the way the f_j's are ordered, $h \leq i_j \leq i_{j'} \leq p$ for all $1 \leq j \leq j' \leq r$). The fact that the type of the elements in $Candidate_{not-simple}$ is more involved (so, allowing to decrease a lot the working memory) requires an advanced way to treat them (and potentially more costly in term of running time). To limit the increase of the running time, the PSB algorithm proceeds in such a way that several deviations in $\{f_1, \cdots, f_r\}$ are somehow considered "simultaneously". More precisely, it proceeds as follows.

Let $1 \leq i^* \leq r$ be the smallest integer such that $lb_{i*} = lb$. The PSB algorithm proceeds as follows to deal with the deviations $f_r, f_{r-1}, \cdots, f_{i^*}$ in this order. First, it applies Dijkstra's algorithm to compute an SP in-branching T'_r rooted at t in $D_r = D - \{x_1, \cdots, x_o = v'_h, \cdots, v'_{i_r}\}$ until it reaches v_r. If v_r is reached, then the path $Q_r = P_1 P_{v'_h v'_{i_r}}(T_{h+1}) P_{v_r t}(T'_r)$ is a simple s-t path and the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, f_r, T'_r), w(Q_r))$ is added to $Candidate_{simple}$. However, the in-branching T'_r is not saved into $\mathcal{T}$ but only its name is kept (this allows to reduce the working memory size while it may require to recompute the tree T'_r later. The bet here is that it will not be necessary to actually redo this computation). Then, for $j = r - 1$ down to i^*, the SP in-branching T'_r is updated to become the SP in-branching T'_j rooted in t in $D_j = D - \{x_1, \cdots, x_o = v'_h, \cdots, v'_{i_j}\}$, possibly reaching v_j

and so providing a simple path Q_j. To speed up the computation of T'_j, it is actually computed by updating T'_{j+1} which is done using standard methods for updating a shortest path tree [9]. Finally, the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, f_j, T'_j), w(Q_j))$ is added to $Candidate_{simple}$. In the current implementation of our PSB algorithm (the one used in the experiments described in next section), the in-branching T'_j is saved in $\mathcal{T}$ only if $j = i^*$[3]. Finally, the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, X', T_{h+1}), \min_{1 \leq j < i^*} lb_j)$, with $X' = \{f_1, \cdots, f_{i^*-1}\}$, is added into $Candidate_{not-simple}$.

The correctness of the PSB algorithm follows from the one of the SB algorithm by noticing that the elements extracted from $Candidate_{simple} \cup Candidate_{not-simple}$ are the ones with smallest lower bound and the fact that, each time that a path is extracted, a shortest extension of each of its deviations is considered.

Finally, as already mentioned, the number of elements in the heap $Candidate_{not-simple}$ is bounded by k as each of its elements correspond to a path that has been added to the output, while with the SB algorithm, this heap may contain $O(km)$ elements. Furthermore, as for the SB algorithm, we may keep only the k elements with smallest lower bound in $Candidate_{simple}$. Hence, the working memory used by the PSB algorithm for the heaps is significantly smaller than for the SB algorithm. However, the largest part of the working memory is due to the number of SP in-branchings that are computed and stored in $\mathcal{T}$. Although this number seems difficult to evaluate, we observe experimentally (see Section 4) that it is significantly smaller with the PSB algorithm.

4 Experimental evaluation

4.1 Experimental settings

We have implemented[4] the algorithms NC (improvement of Yen's algorithm by Feng [7]), SB [16], SB* and PSB in C++14 and our code is publicly available at `https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths`.

Following [16], we have implemented a pairing heap data structure [8] supporting the decrease key operation, and we use it for the Dijkstra shortest path algorithm. Our implementation of the Dijkstra shortest path algorithm is lazy, that is it stops computation as soon as the distance from query node v to t is proved to be the shortest one. Further computations might be performed later for another node v' at larger distance from t. Our implementation of Dijkstra's algorithm supports an update operation when a node or an arc is added to the graph. In addition, we have implemented a special copy operation that enables to update the in-branching when a set of nodes is removed from the graph. This corresponds to the operations performed when creating an in-branching T_{h+1} from T_h. Observe that in our implementations of NC, SB, SB* and PSB, the parameter k is not part of the input, and so the sets of candidates are simply implemented using pairing heaps. This choice enables to use these methods as iterators able to return the next shortest path as long as one exists (Note that if k is part of the input, the data structure used to store candidates could be changed in order to contain only the k best candidates, but the algorithm would only return exactly k paths even if more exist). We have evaluated the performances of our algorithms on some road networks from the 9th DIMACS implementation challenge [4]. The characteristics of these graphs are reported in Table 1. In the following, we refer to the graphs ROME,

[3] A way to establish an even better space versus time tradeoff would be to determine a good threshold τ such that an in-branching T'_j is stored in $\mathcal{T}$ if and only if $w(Q_j) \leq \tau$. Due to lack of time we have not been able to establish such a parameter τ but it will be one of the objectives of our future works.

[4] Despite several queries, we have not been granted access to the code used for experiments in [7, 16].

Table 1 Characteristics of the TIGER graphs used in kSSP experiments.

Area	ROME	DC	DE	NY	BAY	COL
Number of vertices	3 353	9 559	49 109	264 346	321 270	435 666
Number of edges	8 870	29 682	119 520	733 846	800 172	1 057 066

DC and DE as the small networks, and to the graphs NY, BAY and COL as the large networks. We also generated random networks using method `RandomGNM` of SageMath [18] with $n \in \{5000, 10000, 20000\}$ and for each n, we let $m = 10n, 50n$ and $100n$. For each network (both the random and the road networks), we have measured the execution time and the number of stored SP in-branchings. Note that the number of stored in-branchings gives a rigorous indication of the memory consumption (in particular, this is independent of the implementation) [13]. For each network we run the algorithms on a thousand pairs of vertices randomly chosen. We report in Tables 2 and 7 the average and the median of their running times and number of stored SP in-branchings.

All reported computations have been performed on computers equipped with 2 quad-core 3.20GHz Intel Xeon W5580 processors and 64GB of RAM.

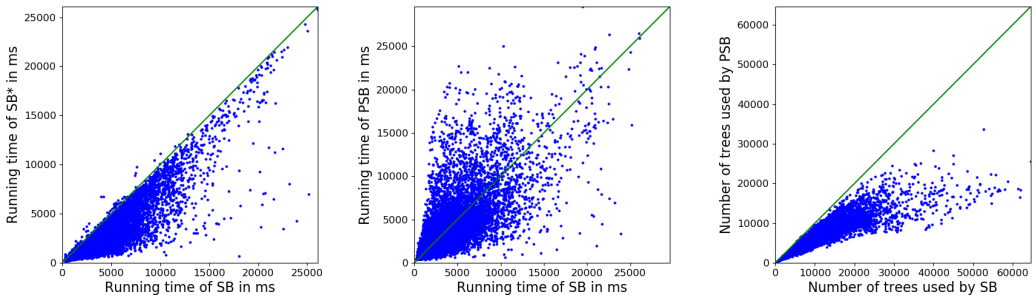
4.2 Experimental results

The simulations show that our tiny improvement SB* of SB algorithm allows to decrease the running-time by a factor between 1,5 (on NY) and 2 (on DE) on median (Tables 2 and 3). The fact that in each network a few queries are extremely slow makes the median a better indicator than the average. In particular, in all the networks considered, SB* algorithm is, for most of the queries, faster than SB algorithm (Figures 1a and 2a). By design, the number of stored in-branchings is the same in both algorithms. It is interesting to note that the gain increase with the size of the networks. It seems that the differences of performances depends on the structure of the queries and of the networks. In the future work, we plan to investigate the relationship between the kinds of queries and/or networks and the gain in running time.

The simulations comparing PSB algorithm and SB algorithm show a significant reduction of the working memory when using PSB (Tables 4 and 5 and Figures 1c and 2c). Again, the gain increases when considering larger networks. In term of running time, SB algorithm is slightly faster on average but Figures 1b and 2b indicate that globally, they are quite comparable. It would be interesting to understand the impact of the length of the queries on the performances of both algorithms.

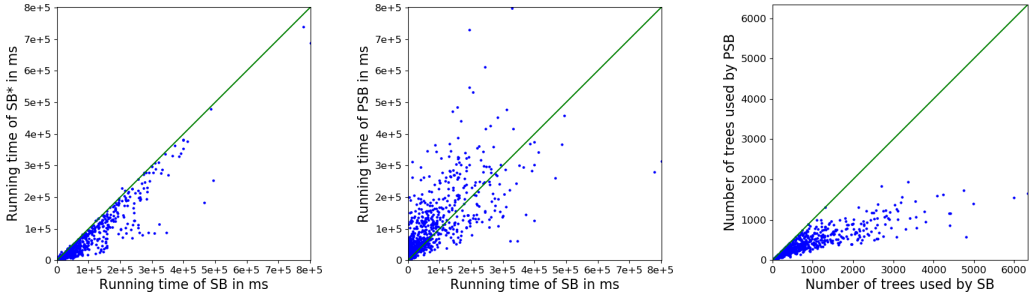
Finally, following some simulations in [16], we have also compared all the algorithms on random graphs (Edös-Rényi). Due to lack of time, we considered only one graph per setting (number of vertices, of edges and k) and the average is done on 1000 requests (note that this setting is similar to the one in [16]). The performances (Table 6) are similar than in the ones obtained for road networks. That is, the SB* algorithm is faster than all other algorithms (more than twice as fast in the case of large graphs when $k = 1000$). Surprisingly, the NC algorithm is sometimes (for dense graphs) faster than SB and PSB. Moreover, the PSB algorithm always uses less memory than SB algorithm (Table 7), with a more significant difference in the case of sparse large graphs.

In the future work, we will continue our experiments in order to discover which conditions (structure of graphs and queries...) make one of the prescribed algorithms faster or / and less memory consuming than the others.



(a) Running time of SB and SB*. **(b)** Running time of SB and PSB. **(c)** Number of trees of SB and PSB.

Figure 1 Comparison of the running time of SB versus SB* (Figure 1a) and SB versus PSB (Figure 1b) on ROME, and comparison of the number of stored trees for SB versus PSB (Figure 1c). Each dot corresponds to one pair source/destination ($k = 10,000$).



(a) Running time of SB and SB*. **(b)** Running time of SB and PSB. **(c)** Number of trees of SB and PSB.

Figure 2 Comparison of the running time of SB versus SB* (Figure 2a) and SB versus PSB (Figure 2b) on COL, and comparison of the number of stored trees for SB versus PSB (Figure 2c). Each dot corresponds to one pair source/destination ($k = 1,000$).

Table 2 Time consuming (average and median in ms) of SB, SB* and PSB on small networks.

Area		ROME			DC			DE		
k		100	1000	10,000	100	1000	10,000	100	1000	10,000
Avg	SB	43	440	4,502	15	175	2,051	773	7,867	82,916
	SB*	**24**	**272**	**2,939**	**11**	**118**	**1,388**	**532**	**5,721**	**61,294**
	PSB	42	427	4,380	21	248	2,859	865	8,762	90,226
Med	SB	33	347	3,663	8	74	893	403	5,382	42,613
	SB*	**15**	**185**	**2,105**	**6**	**45**	**538**	**196**	**2,184**	**28,294**
	PSB	30	314	3,252	9	101	1,185	654	6,517	73,046

Table 3 Time consuming (average and median in ms) of SB, SB* and PSB on big networks.

Area		NY			BAY			COL		
k		100	500	1000	100	500	1000	100	500	1000
Avg	SB	904	4,334	8,741	3,270	18,464	38,346	5,216	28,262	59,717
	SB*	**581**	**2,787**	**5,707**	**2,395**	**13,669**	**28,545**	**3,723**	**20,313**	**43,373**
	PSB	1,822	9,417	19,166	5,083	27,438	55,711	7,371	39,078	80,696
Med	SB	156	600	1,230	695	4,073	9,443	1,412	8,737	19,664
	SB*	**148**	**356**	**676**	**340**	**2,075**	**4,934**	**722**	**5,051**	**11,620**
	PSB	336	2,324	5,278	1,934	12,000	25,760	3,072	19,219	42,114

Table 4 Number of SP in-branching genereted and stored by SB and PSB on small networks.

Area		ROME			DC			DE		
k		100	1000	10,000	100	1000	10,000	100	1000	10,000
Avg	SB	106	1,108	11,446	14.9	209	2,594	88	928	9,945
	PSB	**53**	**667**	**6,956**	**10.6**	**140**	**1,716**	**36**	**390**	**4,212**
Med	SB	87	961	10,164	6	105	1,555	48	557	6,551
	PSB	**56**	**615**	**6,570**	**5**	**80**	**1,106**	**25**	**287**	**3,154**

Table 5 Number of SP in-branching genereted and stored by SB and PSB on big networks.

Area		NY			BAY			COL		
k		100	500	1000	100	500	1000	100	500	1000
Avg	SB	14.9	81	171	44.6	266	562	47	266	570
	PSB	**9.8**	**51**	**106**	**22**	**124**	**259**	**22**	**123**	**259**
Med	SB	3	21	45	13	90	209	13	101	234
	PSB	**2**	**16**	**35**	**9**	**57**	**124**	**9**	**63**	**142**

Table 6 Average time consuming (in ms) of NC, SB, SB* and PSB on random digraph with different densities.

Digraph		$n = 5,000$			$n = 10,000$			$n = 20,000$		
	m	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$
$k = 100$	NC	40	64	88	37	114	158	191	304	388
	SB	12	39	53	23	47	79	805	412	363
	SB*	**35**	**30**	**48**	**17**	**41**	**70**	**132**	**153**	**204**
	PSB	30	40	51	24	46	78	554	435	389
$k = 500$	NC	159	275	354	311	470	652	789	1211	1498
	SB	49	182	250	74	180	332	3869	1924	1637
	SB*	**54**	**121**	**213**	**40**	**139**	**271**	**690**	**573**	**787**
	PSB	66	175	229	74	166	314	3692	2005	1727
$k = 1000$	NC	313	546	697	617	924	1285	1545	2368	2920
	SB	98	370	512	144	356	669	7709	3890	3264
	SB*	**79**	**239**	**430**	**71**	**267**	**533**	**1010**	**1100**	**1528**
	PSB	112	349	463	145	322	620	5209	3978	3412

Table 7 Average of number of SP in-branching computed and stored using SB and SB* on random digraph with different densities.

Digraph		$n = 5,000$			$n = 10,000$			$n = 20,000$		
	m	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$
$k = 100$	SB	2.332	3.726	2.08	1.992	1.566	1.753	33.142	9.149	5.09
	PSB	**2.275**	**3.657**	**2.04**	**1.952**	**1.538**	**1.724**	**25.95**	**8.529**	**4.882**
$k = 500$	SB	8.88	16.07	7.477	6.287	4.615	5.493	161.844	42.866	22.094
	PSB	**8.434**	**15.57**	**7.175**	**6.041**	**4.465**	**5.269**	**126.23**	**39.603**	**21.018**
$k = 1000$	SB	17.477	32.207	14.98	12.023	8.623	10.532	323.231	85.178	43.193
	PSB	**16.538**	**31.132**	**14.34**	**11.471**	**8.307**	**10.059**	**252.151**	**78.28**	**40.948**

References

1 M. Arita. Metabolic reconstruction using shortest paths. *Simulation Practice and Theory*, 8(1-2):109–125, 2000. doi:10.1016/S0928-4869(00)00006-9.

2 M. Betz and H. Hild. Language models for a spelled letter recognizer. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 856–859. IEEE, 1995. doi:10.1109/ICASSP.1995.479829.

3 S. Clarke, A. Krikorian, and J. Rausen. Computing the n best loopless paths in a network. *Journal of the Society for Industrial and Applied Mathematics*, 11(4):1096–1102, 1963. doi:10.1137/0111081.

4 C. Demetrescu, A. Goldberg, and D. Johnson. 9th dimacs implementation challenge - shortest paths, 2006. URL: http://users.diag.uniroma1.it/challenge9/.

5 D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998. doi:10.1137/S0097539795290477.

6 David Eppstein. *k-Best Enumeration*, pages 1003–1006. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4_733.

7 G. Feng. Finding k shortest simple paths in directed graphs: A node classification algorithm. *Networks*, 64(1):6–17, 2014. doi:10.1002/net.21552.

8 M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. doi:10.1007/BF01840439.

9 D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000. doi:10.1006/jagm.1999.1048.

10 Eleni Hadjiconstantinou and Nicos Christofides. An efficient implementation of an algorithm for finding k shortest simple paths. *Networks*, 34(2):88–101, 1999. doi:10.1002/(SICI)1097-0037(199909)34:2<88::AID-NET2>3.0.CO;2-1.

11 J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms*, 3(4):45, 2007. doi:10.1145/1290672.1290682.

12 W. Jin, S. Chen, and H. Jiang. Finding the k shortest paths in a time-schedule network with constraints on arcs. *Computers & operations research*, 40(12):2975–2982, 2013. doi:10.1016/j.cor.2013.07.005.

13 David S. Johnson. A theoretician's guide to the experimental analysis of algorithms. *Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges*, 59:215–250, 2002. doi:10.1090/dimacs/059/11.

14 N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982. doi:10.1002/net.3230120406.

15 D. Kurz. *k-best enumeration - theory and application*. Theses, Technischen Universität Dortmund, March 2018. doi:10.17877/DE290R-19814.

16 D. Kurz and P. Mutzel. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. In *Int. Symp. on Algorithms and Computation (ISAAC)*, volume 64 of *LIPIcs*, pages 49:1–49:13. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.ISAAC.2016.49.

17 T. Shibuya and H. Imai. New flexible approaches for multiple sequence alignment. *Journal of Computational Biology*, 4(3):385–413, 1997. doi:10.1089/cmb.1997.4.385.

18 The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.9)*, 2019. https://www.sagemath.org.

19 W. Xu, S. He, R. Song, and S. S. Chaudhry. Finding the k shortest paths in a schedule-based transit network. *Computers & Operations Research*, 39(8):1812–1826, 2012. doi:10.1016/j.cor.2010.02.005.

20 J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971. doi:10.1287/mnsc.17.11.712.