Finding the k Shortest Simple Paths: Time and Space Trade-Offs

ALI AL ZOOBI, DAVID COUDERT, and NICOLAS NISSE, Université Côte d'Azur, Inria, CNRS, I3S, France

The k shortest simple path problem (kSSP) asks to compute a set of top-k shortest simple paths from a source to a sink in a digraph. Yen (1971) proposed an algorithm with the best known polynomial time complexity for this problem. Since then, the problem has been widely studied from an algorithm engineering perspective. The most noticeable proposals are the *node-classification* (NC) algorithm (Feng, 2014) and the *sidetracks-based* (SB) algorithm (Kurz, Mutzel, 2016). The latest offers the best running time at the price of a significant memory consumption.

We first show how to speed up the SB algorithm using dynamic updates of shortest path trees resulting in a faster algorithm (SB*) with the same memory consumption. We then propose the *parsimonious SB* (PSB) algorithm that significantly reduces the memory consumption of SB at the cost of a small increase of the running time. Furthermore, we propose the *postponed node-classification* (PNC) algorithm that combines the best of the NC and the SB algorithms. It offers a significant speed up compared to the SB algorithm while using the same amount of memory as the NC algorithm.

Our experimental results on complex networks show that all the considered algorithms have low memory consumption, and that the PSB algorithm is the fastest. On road networks, the relative performances of the algorithms depend on the number k of requested paths. Indeed, when the number k of requested paths is small (i.e., $k \leq 20$ in our experiments), the SB* algorithm is the fastest among the considered algorithms, but it suffers from a large memory consumption and it offers very bad performances on some queries. When the number of requested paths is large (i.e., larger than 20 according to our experiments), the PNC algorithm is the fastest among the considered algorithms on road networks and it has a low memory footprint. The PNC algorithm is therefore a better choice on road networks.

CCS Concepts: • **Theory of computation → Shortest paths**.

Additional Key Words and Phrases: k shortest simple paths; graph algorithm; space-time trade-off

1 INTRODUCTION

The classical k shortest paths problem (kSP) aims at finding 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 $0 \leq i < l$. 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 weight of a path is the sum of the weights of its arcs and

This work has been supported by the French government, through the UCA$^{\text{JEDI}}$ 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, by Région Sud PACA and by the STIC-AmSud project GALOP.
An extended abstract of this work has been presented in the 18th International Symposium on Experimental Algorithms [1].
Authors' address: Ali Al Zoobi, ali.al-zoobi@inria.fr; David Coudert, david.coudert@inria.fr; Nicolas Nisse, nicolas.nisse@inria.fr, Université Côte d'Azur, Inria, CNRS, I3S, France.

1084-6654/2023/10-ART $15.00
https://doi.org/10.1145/3626567

the top-k shortest paths therefore comprise a set containing a shortest s-t path, a second shortest s-t path, *etc.*, until a k^{th} shortest s-t path.

Several algorithms for solving kSP have been proposed. In particular, Eppstein [9] proposed an exact algorithm that computes k shortest paths (not necessarily simple) in time $O(m + n \log n + k)$, 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.* [7] which adds the constraint that reported paths must be simple. This variant of the problem has various applications in transportation networks 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 [17, 20, 36]. It can be applied successfully in bio-informatics [3], especially in biological sequence alignment [31] and in natural language processing [5]. For more applications, see Eppstein's recent comprehensive survey on k-best enumeration [10].

The most famous algorithm for solving the kSSP problem has been proposed by Yen [37] and has time complexity $O(kn(m+n \log n))$. It has been proved that the kSSP problem can be solved in $O(k)$ iterations of APSP (All Pairs Shortest Paths) [16]. This means that the kSSP problem can be solved in time $O(kn(m + n \log \log n))$ on sparse digraphs and in time $O(kn^3(\log \log n)/\log^2 n)$ on dense digraphs using the fastest APSP algorithms [18, 29]. Vassilevska Williams and Williams [34] show that a subcubic kSSP algorithm would also result in a subcubic algorithm for APSP, which seems unlikely at the moment. Recently, Eppstein and Kurz [11] proved that on digraphs with bounded treewidth, the kSSP problem can be solved in time $O(n + k \log n)$.

On the other hand, several works have been proposed to improve the efficiency of the algorithm in practice [12, 15, 17, 19, 22–24, 32]; they all feature the same worst-case running time as Yen's algorithm, *i.e.*, $O(kn(m+n \log n))$.

In particular, Feng [12] proposed an improvement of Yen's algorithm called Node Classification (NC) algorithm. With the help of a pre-computed shortest path tree of the digraph, the NC algorithm classifies the vertices of the digraph in order to speed up the computation of shortest simple paths. An interesting fact about the NC algorithm is that its memory consumption is almost the same as Yen's algorithm (only a shortest path tree is kept in the memory).

Recently, Kurz and Mutzel [23, 24] obtained a tremendous improvement of the practical running time, designing an algorithm called Sidetrack Based (SB) with the same flavor as Eppstein's algorithm. The key ideas are to define a path using a sequence of shortest path trees and *deviations*, and to postpone as much as possible the computation of shortest path trees. 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 $k = 1\,000$ shortest simple paths in 50 milliseconds for the DC network [8] while it required about 5 seconds with Yen's algorithm and about 0.3 seconds by the improvement proposed by Feng [12]. The drawback of the SB algorithm is the need for storing all computed shortest path trees, thus leading to a large usage of memory.

To sum up, the fastest algorithm with low memory consumption (*i.e.*, the same memory consumption as Yen's algorithm) is the Node Classification (NC) algorithm, proposed independently by [12] and [15]. With larger memory consumption, the Sidetrack Based algorithm (SB) [24] can achieve a tremendous speed up.

Our contributions. We propose a new algorithm with low memory consumption, called Postponed Node Classification (PNC), that combines the best of the NC and the SB algorithms. More precisely, while the PNC algorithm has the same memory consumption than the NC algorithm, our experimental results show that it is the fastest kSSP algorithm among all considered algorithms on road networks.

Considering large memory consumption, we show how to speed up the SB algorithm using dynamic updates of shortest path trees resulting with the SB* algorithm, that is, the fastest kSSP algorithm (on median) with large memory consumption among the considered algorithms on road networks. Moreover, we propose a new algorithm, called Parsimonious Sidetrack Based (PSB), that is based on the SB algorithm. The PSB algorithm gives, on road networks, a space time trade-off between SB and NC algorithms. That is, on road networks, it significantly reduces the memory consumption of SB at the cost of an increase of the running time. Nonetheless, on complex networks, the PSB algorithm gives the best running time among all the considered algorithms. We further propose parameterized variants of PSB (called PSBv2 and PSBv3) that improve its performance, both in terms of memory consumption and of running time, on road networks.

We analyse the behavior of all of the aforementioned algorithms on different types of networks (road, biological, Internet and social networks). We have also investigated the relationships between the performances of the algorithms and the Dijkstra rank of the queries. Finally, we end up with a first empirical framework for selecting the most suitable kSSP algorithm for a given use case.

This paper is organized as follows. We start recalling the Yen and NC algorithms in Section 2. Then, we present, in Section 3, the SB algorithm with our improvement SB*. In Section 4, we present the PSB algorithm and its two variants PSBv2 and PSBv3. Section 5 is devoted to the presentation of the PNC algorithm. We present our experimental evaluation of all these algorithms on various networks in Section 6. Finally, we conclude this paper in Section 7.

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 set of the out-neighbors of v in D. Let $\omega_D : A \rightarrow \mathbb{R}^+$ be a weight function over the arcs. For every $u, v \in V$, a *(directed) path* from u to v in D is a sequence $P = (u = v_0, v_1, \cdots, v_r = v)$ 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 *weight* of the path P equals $\omega_D(P) = \sum_{0 \leq i < r} \omega_D(v_i, v_{i+1})$ (we will omit D when there is no ambiguity). The *distance* $d_D(u, v)$ between two vertices $u, v \in V$ is the smallest weight of a u-v path in D (if any). Given two paths $P = (v_0, \cdots, v_r)$, $Q = (w_0, \cdots, w_p)$ and $v_r w_0 \in A$, we denote by $P.Q$ the v_0-w_p path obtained by the concatenation of P and Q. *i.e.*, $P.Q = (v_0, \cdots, v_r, w_0, \cdots, w_p) = (v_0, \cdots, v_r, Q) = (P, w_1, \cdots, w_p)$.

Given $s, t \in V$, a *top-k set of shortest simple s-t paths* is any set S of (pairwise distinct) simple s-t paths such that $|S| = k$ and $\omega(P) \leq \omega(P')$ 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 $D = (V, A)$, $\omega_D : A \longrightarrow \mathbb{R}^+$ and a pair of vertices $(s, t) \in V^2$ and asks to find a top-k set of shortest simple 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 (not necessarily spanning) 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 weight 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*. Similarly, a *shortest path (SP) out-branching* T rooted at s is any sub-digraph of D inducing a tree containing s, such that every $u \in V(T) \setminus \{s\}$ has exactly one in-neighbor and the weight of the (unique) s-u path P^T_{su} in T equals $d_D(s, u)$. Any algorithm able to compute an SP in-branching (or an SP out-branching) is called an SP algorithm. As we consider directed weighted digraphs, we use the algorithm of Dijkstra. However, it is possible to use any suitable shortest path algorithm instead.

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 an SP algorithm to compute an SP in-branching rooted at t that contains a shortest u-t path in H. Note that, the execution of an SP 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 (not necessarily spanning H). The key point will be that this way to proceed not necessarily only returns a shortest u-t path in H (if any) but an SP in-branching rooted in t, containing u. Recall 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 $P' = (v_0, \cdots, v_i, v', v'_1, \cdots, v'_\ell = v_r)$ is called a *detour* of P at a (or at v_i). Note that neither P nor P' is required to be simple. However, if P' is simple, it will be called a *simple detour* of P at a (or at v_i). In addition, P' is called a shortest (simple) detour at v_i (or at a) if and only if P' is a detour with minimum weight among all (simple) detours of P at v_i (or at a).

2.2 Yen's algorithm

We start by describing Yen's algorithm [37] trying to give its main properties and drawbacks.

All the algorithms described below start by computing a shortest s-t path $P_0 = (s = v_0, v_1, \cdots, v_r = t)$, and assume that there is always at least one such path. This is done by applying an SP algorithm from s. We clearly may assume that P_0 is simple since the weights are non-negative. A second shortest s-t simple path must be a shortest simple detour of P_0 at one of its vertices. Yen's algorithm computes a shortest simple detour 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 detours) and the arc $v_i v_{i+1}$ (to ensure that the computed path is a new one, *i.e.*, different from P_0). For every $0 \leq i < r$, an SP out-branching in $D_i(P_0)$ rooted at v_i is computed using an SP algorithm until it reaches t and therefore returns a shortest path Q_i from v_i to t. For every $0 \leq i < r$, the detour $(v_0, \cdots, v_{i-1}, Q_i)$ 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 $(v_0, \cdots, v_{i-1}, Q_i)$ deviates from P_0 is kept explicit[1], *i.e.*, the path is stored with its deviation index. Once $(v_0, \cdots, v_{i-1}, Q_i)$ has been added to *Candidate* for all $0 \leq i < r$, by remark above, a path with minimum weight 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 simple s-t paths has been computed and that *Candidate* is a set of simple s-t paths. Moreover, let us assume that there exists a shortest path $Q \in Candidate$ such that $S \cup \{Q\}$ is a top-$(k'+1)$ set of shortest simple s-t paths. Moreover, let us assume by induction that, for every path R in *Candidate*, with deviation index j, all detours of $R = (v_0, \cdots, v_{|R|})$ at vertices v_i for $0 \leq i < j$ have already been computed and added to *Candidate*. 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, for each vertex v in Q, a shortest simple detour of Q at v is added to *Candidate* (since potentially one of these detours is a next shortest s-t path). For this purpose, for every $j \leq i < r$, let $\pi_i = (v_0, \cdots, v_{i-1})$ ($\pi_i = \emptyset$ if $i = 0$) and let $D_i(Q)$ be a subdigraph of D containing a shortest v_i-t path Q_i in D such that $Q_i \cap \pi_i = \emptyset$ and the path $\pi_i.Q_i$ is new ($\pi_i.Q_i \notin S$). After the construction of $D_i(Q)$ (described below), an SP out-branching of $D_i(Q)$ rooted at v_i is computed using an SP algorithm until it reaches t and therefore returns a shortest path Q_i from v_i to t in $D_i(Q)$. For every $0 \leq i < r$, the shortest simple detour $\pi_i.Q_i$ 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, *i.e.*, when $k' = k$. Indeed, the computed detours of Q are distinct from every previously computed paths as they have different prefixes (this is the reason to keep explicitly the deviation-index).

The procedure of constructing $D_i(Q)$ is the following. First, to avoid non-simple detours, *i.e.*, any intersection between Q_i and π_i, the vertices $v_0, \cdots, v_{i-1}$ (if $i > 0$) are removed from D. Second, to ensure that the computed path $(\pi_i.Q_i)$ is new (different from those in S), each arc $v_i v'$ such that S already contains a path with prefix $(v_0, \cdots, v_i, v')$ is removed from $D_i(Q)$.

[1]The deviation-index is not kept explicitly in Yen's algorithm. But, since it is a trivial improvement already existing in the literature [25], we mention it here.

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

Therefore, for each path Q that is extracted from $Candidate$, $O(|V(Q)|)$ calls of an SP 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 kSSP problem.

2.3 A Node Classification algorithm

In this section, we present the Node Classification (NC) algorithm, an improvement of Yen's algorithm proposed independently by Feng [12] and Gao et al. [15].

The most expensive part of Yen's algorithm is its large number of calls to an SP algorithm. The NC algorithm aims at reducing the computing time of each of these calls, and possibly to avoid some of them.

Precisely, during the process of finding a detour, the search area of an SP algorithm is restricted to a digraph that is smaller than D with the help of a pre-computed shortest path in-branching. The NC algorithm starts by computing a shortest path in-branching T of D rooted at t (used to extract a first shortest path P_0). Then, when a path $Q = (v_0, \cdots, v_r)$ with deviation-index j is extracted, its detours are computed from $i = j$ to $r - 1$. The NC algorithm classifies the vertices as `red`, `yellow`, and `green`: a vertex on the prefix (*i.e.*, the path $(v_0, \cdots, v_{i-1})$) is colored `red`, a vertex u that can reach t through T without visiting a `red` vertex (*i.e.*, $P^T_{ut} \cap (v_0, \cdots, v_{i-1}) = \emptyset$) is colored `green`, and all other vertices are colored `yellow`. This coloring can be computed in linear time using a DFS in T. Moreover, the coloring used to compute the detour at v_{i+1} can be obtained faster by updating the coloring for the detour at v_i.

Another important ingredient of the NC algorithm is the notion of *residual weight*. For each arc $e = uv$ not in T, the residual weight of e is the cost of deviating from T at e. Precisely, it is the weight of the path $u.P^T_{vt}$ minus the weight of the path P^T_{ut}. Formally, the residual weight of arc uv is $\delta(u, v) = \omega(u, v) + \omega(P^T_{vt}) - \omega(P^T_{ut})$. The residual weight is computed only once (after computing T) and remains valid till the end of the execution of the NC algorithm. Note that an arc in T has residual weight equals to 0, and so the residual weight of the path P^T_{ut} from any `green` vertex u to t in T equals 0.

Recall that to compute a detour of Q at v_i, Yen's algorithm executes an SP algorithm to compute a shortest path from v_i to t in $D_i(Q)$. In the case of the NC algorithm, Feng proved that it is sufficient to execute an SP algorithm using the residual weights and to stop its execution as soon as a green vertex is reached. This results in restricting the execution of the SP algorithm to the `yellow` subgraph, which is expected to be smaller than $D_i(Q)$, and so to speed up the computation of the detours.

In Section 5, we propose an adaptation of the NC algorithm (using ideas from the SB algorithm presented in the next section) that allows us to speed it up.

3 SIDETRACK BASED (SB) ALGORITHM

We now present the Sidetrack Based (SB) algorithm, proposed by Kurz and Mutzel [24]. We start by describing the data structure used in the SB algorithm. Then, we explain it and provide a pseudo code (Algorithm 1). Finally we analyse a few of its aspects. Note that our contributions in Section 4 strongly rely on this algorithm and that is why we describe it in detail.

3.1 Compact representation of a path

The SB algorithm is based on a data structure generalizing the representation of a path proposed by Eppstein [9]. Such compact representation uses sequences of in-branchings $T_0, T_1, \cdots, T_h$ and deviations $e_0, e_1, \cdots, e_h$ (recall that a deviation of a path P is any arc not in P but with its tail in P).

Precisely, the sequence $\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1})$ with $e_i = v_i w_i$ for all $0 \leq i \leq h$, represents the path P starting at s, following T_0 until the tail v_0 of e_0, then the deviation e_0, then T_1 from the head w_0 of e_0 until it reaches the tail v_1 of e_1, *etc.*, until it reaches the head w_h of e_h, plus (possibly) the path from w_h to t in T_{h+1}. That is, P is the sequence of vertices of the paths $P^{T_0}_{sv_0}, P^{T_1}_{w_0 v_1}, \cdots, P^{T_h}_{w_{h-1} v_h}$ followed by the vertices of $P^{T_{h+1}}_{w_h t}$ if this latter path exists. Two consecutive in-branchings T_i and T_{i+1} are not necessarily distinct. The SB algorithm ensures that, if P is an s-t path (*i.e.*, if $P^{T_{h+1}}_{w_h t}$ exists), then the subpath π of P going from s to w_h $(v_0, \cdots, w_h)$ is always simple and P is not simple only if $P^{T_{h+1}}_{w_h t}$ intersects π.

3.2 The SB algorithm

We are now ready to present the SB algorithm, whose pseudocode is presented in Algorithm 1. The main idea of the algorithm is to postpone the computation of the detours. That is, the in-branchings T_{h+1} are computed only if needed. The subtlety is to know when to extract a path (without knowing its actual weight if T_{h+1} has not been computed yet) from the set *Candidate* (where the paths are encoded using the format presented above). For this purpose, a lower bound on the distance from w_h to t is used to get a lower bound on the weight of the path.

Roughly, the SB algorithm uses a set *Candidate* to manage candidate paths that are encoded using the above data structure. Sequentially, it extracts a shortest element ε from *Candidate*. If ε represents a simple path, this path is added to the output and the representations of its detours (which will be found using the last tree in the representation of ε) are added to *Candidate*. Otherwise, the SB algorithm attempts to modify ε by instantiating its last in-branching (see below). If this computation leads to a representation of a simple path, then it is added to *Candidate*. Otherwise, ε is discarded. The SB algorithm goes on iteratively until it has found k paths. The initialization consists in computing a first in-branching T_0 rooted at t in D (using an SP algorithm) and so a shortest s-t (simple) path $P^{T_0}_{st}$ and adding its representation to *Candidate*.

More precisely, the set *Candidate* is a min-heap in which the weight (the key) of an element is a lower bound on the weight of the path it represents. Each element μ in *Candidate* has the form $\mu = (\varepsilon = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), lb, \zeta)$ where each in-branching $T_{h'}$ (with $h' \leq h$) is already computed and lb is a lower bound of the weight of the path represented by ε. The value ζ is a boolean indicating whether the path represented by ε is known to be simple. If so, it will follow from the construction that T_{h+1} has already been computed. Else T_{h+1} must be first computed to know if ε represents a simple path. For the initialization, the in-branching T_0 is computed and the element $((T_0), \omega(P^{T_0}_{st}), \zeta = 1)$ is inserted in *Candidate*.

The SB algorithm iteratively extracts elements from *Candidate* by minimum weight (with a priority to representation of simple paths to break ties) until k paths are obtained or *Candidate* is empty. When an element $\mu = (\varepsilon = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), lb, \zeta)$ is extracted from *Candidate*, two cases are distinguished. Let i be the index of w_h in the path P represented by ε (note that i plays the same role as the deviation-index in Yen's algorithm).

Case $\zeta = 1$. Then, ε represents a simple path $P = (v_0 = s, \cdots, v_i = w_h, \cdots, v_r = t)$ and all of its in-branchings have already been computed. In this case, the path P is added to the output. Then, the algorithm considers all deviations of P starting from a vertex v_j of the suffix $(v_i, \cdots, v_r = t)$ of P. More precisely, for every arc $e = (v_j, w)$, with $i \leq j < r$ and $w \notin \{v_0 = s, v_1, \cdots, v_j, v_{j+1}\}$, let $P^{T_{h+1}}_{wt}$ be a shortest path from w to t in T_{h+1} (if any) and let $Q(v_j, e) = (v_0, \cdots, v_j, P^{T_{h+1}}_{wt})$. If $Q(v_j, e)$ is simple, the representation $\mu' = ((T_0, e_0, \cdots, e_h, T_{h+1}, e = (v_j, w), T_{h+1}), lb(e), \zeta = 1)$ is added to *Candidate* with $lb(e) = \omega(Q(v_j, e))$ as a key (note that the computation of $lb(e)$ is done in constant time since, in particular, T_{h+1} is already computed). Otherwise, *i.e.*, $Q(v_j, e)$ is not simple, the representation $\mu'' = (\varepsilon'' = (T_0, e_0, \cdots, e_h, T_{h+1}, e = (v_j, w), T'), lb(e), \zeta = 0)$ is added to *Candidate*, where T' is the *name* of the in-branching of $D \setminus \{v_0, \cdots, v_j\}$ whose actual computation is postponed, and $lb(e) = \omega(Q(v_j, e))$ is a lower bound on the weight of the path represented by ε''.

Case $\zeta = 0$. In this case, the algorithm checks for the existence of a w_h-t path $P^{T_{h+1}}_{w_h t}$. To do so, the in-branching T_{h+1} (whose computation had been postponed) is computed. Note that T_{h+1} is an in-branching in $D \setminus$

Algorithm 1 Sidetrack Based (SB) algorithm for the kSSP [24]

```
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
 1: Let $Candidate \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
 2: $T_0 \leftarrow$ an SP in-branching of $D$ rooted at $t$ containing $s$
 3: Add $((T_0), \omega(P_{st}(T_0)), \zeta = 1)$ to $Candidate$
 4: while $Candidate \neq \emptyset$ and $|Output| < k$ do
 5:     $\mu = (\varepsilon = ((T_0, e_0, \cdots, T_h, e_h = (u_h, v_h), T_{h+1}), lb, \zeta) \leftarrow$ a shortest element in $Candidate$
 6:     if $\zeta = 1$ then
 7:         Extract $\mu$ from $Candidate$ and add $\varepsilon$ to $Output$
 8:         for every deviation $e = v_j v'$ with $v_j \in P^{T_{h+1}}_{v_h t}$ do
 9:             $\varepsilon' \leftarrow (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1})$
10:             $lb' \leftarrow lb - \omega(P^{T_{h+1}}_{v_j t}) + \omega(e) + \omega(P^{T_{h+1}}_{v' t})$
11:             if $\varepsilon'$ represents a simple path then
12:                 Add $\mu' = (\varepsilon', lb', \zeta = 1)$ to $Candidate$
13:             else
14:                 $T' \leftarrow$ the name of an SP in-branching of $D_j(P)$          // $T'$ is not computed yet
15:                 Add $\mu'' = (\varepsilon'' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T'), lb', \zeta = 0)$ to $Candidate$
16:     else
17:         if $T_{h+1}$ has not been computed yet then
18:             Compute $T_{h+1}$, an SP in-branching of $D_h(P)$
19:         Let $\mu' = (\varepsilon' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb + \omega(P^{T_{h+1}}_{v_h t}) - \omega(P^{T_h}_{v_h t}), \zeta = 1)$
20:         Add $\mu'$ to $Candidate$
21: return $Output$
```

$\{v_0, \cdots, v_h\}$, which ensures that, if $P^{T_{h+1}}_{w_h t}$ is found, the path $P_{\text{new}} = (s = v_0, \cdots, v_h, P^{T_{h+1}}_{w_h t})$ is guaranteed to be simple. Moreover, P_{new} has weight $\omega(P_{\text{new}}) = \omega((s = v_0, \cdots, v_h, w_h)) + \omega(P^{T_{h+1}}_{w_h t})$. Then, the representation $\mu' = (\varepsilon' = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), \omega(P_{\text{new}}), \zeta = 1)$ is added to $Candidate$. Finally, if no w_h-t path can be found in T_{h+1}, μ is discarded.

Analysis. Let us first analyze the time complexity of the operations performed after the extraction of an element $\mu = (\varepsilon = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), lb, \zeta)$ from $Candidate$. When $\zeta = 0$, the time complexity is dominated by the computation of the in-branching T_{h+1}, which is done in time $O(m + n \log n)$ using an SP algorithm. When $\zeta = 1$, the algorithm first extracts the path $P = (s = v_0, v_1, \cdots, v_i = w_h, \cdots, v_r = t)$ from the representation ε in time $O(n)$. Then, it associates a label $\lambda(v)$ to every vertex $v \in V$ as follows. First, every vertex v_j (for $i \leq j \leq r$) of P is labelled with j. Then, each vertex $u \in V(T_{h+1})$ such that v_j is the first vertex of P reached from u in $P^{T_{h+1}}_{ut}$ is labelled with j. Finally, the vertices not in $V(T_{h+1}) \cup \{v_0, \cdots, v_{i-1}\}$ are labelled with 0 (these are the vertices for which all shortest paths to t go through $(v_0, \cdots, v_{i-1})$). This labeling operation can be done in overall time $O(n)$ using DFS on T_{h+1} to propagate the labels from the vertices $v_i, \cdots, v_r$. Now, using the label $\lambda(u)$ associated to the vertex u, it is possible to check in constant time if the path $Q(v_j, e)$, with $i \leq j < r$ and $e = (v_j, w)$, is simple. Indeed, if $\lambda(w) \leq j$, the path $P^{T_{h+1}}_{wt}$ intersects the subpath $(v_0, \cdots, v_j)$ of P and so $Q(v_j, e)$ is not simple. Otherwise, the path $Q(v_j, e)$ is simple. Altogether, the operations performed when $\zeta = 1$ can be done in time $O(n + m)$.

In the worst case scenario, each first extraction of a path from $Candidate$ leads to a non-simple detour, and then a call of an SP algorithm. Note that no more than one call to an SP algorithm is done per vertex on a path, thanks to the test of line 17 of Algorithm 1. So, the complexity of the SB algorithm is bounded by $O(kn(m + n\log n))$ as the number of vertices of a simple path is bounded by n and the algorithm stops once k paths have been added to $Output$.

There are two key improvements for which the SB algorithm has, in practice, out-performed all other algorithms for solving the kSSP problem so far. First, it saves an SP algorithm call if the detour is simple. Second, if the detour is not simple it is inserted with a lower bound on its weight and the corresponding call to an SP algorithm is postponed. This way, if this detour leads to a long path (path with weight larger than the k^{th} shortest path), the call to an SP algorithm will never be performed.

However, as the SB algorithm stores complete in-branchings in memory, it has the drawback of possibly consuming a large amount of memory (much more than the NC algorithm, which stores a single in-branching, while keeping the whole description of the paths it computes).

3.3 The SB* algorithm

Here, we propose the SB* algorithm, a variant of the SB algorithm that is a tiny modification of the SB algorithm but leading to a significant speed up (see Section 6.2).

In fact, 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 $Candidate$ with $\zeta = 0$ and T_{h+1} has not been computed yet (*i.e.*, it is only a name), our algorithm does not compute T_{h+1} from scratch as the 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 [14]. Then, the name 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 6.2 show that this update procedure gives an average speed-up factor of 1.1 to 1.2 on road networks.

4 SPACE - TIME TRADE-OFFS

4.1 The Parsimonious Sidetrack Based algorithm

Here, we present the Parsimonious Sidetrack Based (PSB) algorithm, which is an adaptation of the SB algorithm allowing to reduce the memory consumption due to the storage of all in-branchings computed by the SB algorithm. Here, we only focus on the differences between the SB and the PSB algorithm.

The main difference is that the PSB algorithm stores two types of elements in $Candidate$. The first type, of the form $(\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h = (v_h, w_h), T_{h+1}), lb)$, represents a simple s-t path P of weight lb. Contrary to the SB algorithm, the in-branching T_{h+1} has not necessarily been computed yet. The second type, of the form (ε, Dev, lb), contains an extra field Dev (explained below) and, in this case, all of the in-branchings $T_1, \cdots, T_{h+1}$ are already computed.

Let us start by considering a step of the PSB algorithm when an element $\mu = (\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h = (v_h, w_h), T_{h+1}), lb)$ representing a simple path P is extracted from $Candidate$. T_{h+1} is computed at this step (if not already done) which allows to get P explicitly. Then, the PSB algorithm adds $P = (s = v_0, \cdots, v_i = w_h, \cdots, v_r = t)$ to $Output$ and (as for the SB algorithm), for every $v \in \{v_i, \cdots, v_r\}$, and every deviation e with tail v, the detour $Q(v, e)$ of P at e is considered. If $Q(v, e)$ is simple (this test is done as explained in the first paragraph of the analysis in Section 3.2), then $\mu' = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1}), \omega(Q(v, e)))$ is added to $Candidate$. Otherwise, the deviation e is added to a list Dev (initially empty). Once all deviations have been considered, the (unique) element (ε, Dev, lb') is added to $Candidate$, where $lb' = \min_{f_j=(u_j,u'_j)\in Dev} \omega(Q(u_j, f_j))$. That is, Dev is the list of all “non-simple deviations” of P at the vertices between w_h and t, ordered with respect to the index of their tail on P, *i.e.*, for two deviations $f_i = (u_i, u'_i), f_j = (u_j, u'_j) \in Dev$, $f_i \leq f_j$ if and only if $i \leq j$. Finally, let lb' be a lower bound on the weight of the detours at a deviation in Dev. The important difference between the SB and the PSB algorithms comes from the fact that non-simple detours are considered as a unique object by the PSB algorithm.

Now, let us consider a step when the PSB algorithm extracts an element $\mu = (\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}), Dev = \{f_1, \cdots, f_j = (u_j, u'_j), \cdots, f_l\}, lb)$ from $Candidate$. As mentioned above, in this case, ε encodes a simple s-t path $(v_0, \cdots, v_r)$. Let $1 \leq min \leq l$ be the smallest integer such that $lb = \omega(Q(u_{min}, f_{min}))$.

Then, the PSB algorithm proceeds as follows. For j decreasing from l to min, an in-branching T'_j in $D \setminus \{v_0, \cdots, v_{i_j} = u_j\}$ is computed (but not stored!) until a path $P^{T'_j}_{u'_j t}$ from u'_j to t is discovered (if no such path exists, j is decreased by one). If $P^{T'_j}_{u'_j t}$ exists, then $\varepsilon_j = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, f_j, T'_j)$ represents a simple s-t path of weight $lb_j = \omega((v_0, \cdots, v_{i_j})) + \omega(f_j) + \omega(P^{T'_j}_{u'_j t})$. Then, the element $\mu_j = (\varepsilon_j, lb_j)$ is added to $Candidate$, but T'_j is not stored (the PSB algorithm might have to recompute it later). A second key improvement is that to speed up the computation, T'_j is actually computed by updating T'_{j+1}, which is done using standard tools from [14]. Then, only when $j = min$, the in-branching T'_{min} is stored and $\mu_{min} = (\varepsilon_{min}, lb_{min})$ is added to $Candidate$. The reason why T'_{min} is stored (while other T'_j are not) is that μ_{min} is expected to be extracted soon from $Candidate$ (because the path represented by ε_{min} is expected to be short) and we want to avoid the recomputation of T'_{min}. Finally, the element $\mu' = (\varepsilon, Dev' = \{f_1, \cdots, f_{min-1}\}, lb')$ is added to $Candidate$, where lb' is the minimum weight over the non-simple detours in Dev'.

The correctness follows from that of the SB algorithm. Moreover, since most of the computed in-branchings are not stored, the memory used by the PSB algorithm is significantly smaller than for the SB algorithm.

4.2 Special variants of the PSB algorithm

A better space and time trade-off than the PSB algorithm can be achieved if each computed and stored in-branching is going to be used in the future steps, *i.e.*, it is used to extract a simple candidate path that is going to be extracted from $Candidate$ (before the k^{th} shortest path). Unfortunately, such information cannot be afforded as the weight of the k^{th} path is not previously known. However, if computing an in-branching leads to constructing a path with weight relatively short, *e.g.*, less than a threshold value times the weight of the currently considered path, then storing such in-branching is meaningful as the extraction of its corresponding element form $Candidate$ is expected soon and storing it leads to save its redundant computation. Here, we present two variants of the PSB algorithm: PSBv2 and PSBv3. The PSBv2 algorithm is a tiny improvement of the PSB algorithm leading to consume less memory by storing less in-branching while the PSBv3 algorithm gives an adaptable trade-off depending on the value of the threshold.

Let us consider, again, a step when the PSB algorithm extracts an element $\mu = (\varepsilon, Dev = \{f_1, \cdots, f_j = (u_j, u'_j), \cdots, f_l\}, lb)$ from $Candidate$ with $1 \leq min \leq l$ the smallest integer such that $lb = \omega(Q(u_{min}, f_{min}))$. The PSB algorithm algorithm iterates on j, decreasing from l to min as explained above, a corresponding in-branching T'_j is computed for each j (but not stored). Then, only when $j = min$, the in-branching T'_{min} is stored.

The PSBv2 algorithm does not naively store T'_{min}. Instead, T'_{min} is stored only if the weight of its corresponding detour is less than a threshold value θ times the weight of the shortest simple path in $Candidate$. That is, $lb_{min} = \omega((v_0, \cdots, v_{i_{min}})) + \omega(P^{T'_{min}}_{u_{min} t}) \leq \theta * \omega(P_{next})$, where P_{next} is the shortest simple path in $Candidate$. As a result, if the in-branching T'_{min} leads to a (relatively) long path that is not expected to be extracted very soon from $Candidate$, then it is freed from the memory.

The PSBv3 algorithm behaves on every deviation in Dev between min and l the same way the PSBv2 algorithm behaves with f_{min}. Precisely, for each deviation f_j with $min \leq j \leq l$, the PSBv3 algorithm computes its corresponding in-branching T'_j. This in-branching (T'_j) is stored only if the weight of its corresponding detour is less than a threshold value θ times the weight of the shortest simple path in *Candidate*.

The value of the threshold θ could change dynamically during the execution. For instance, it could be related to the ratio between the two upcoming paths, *i.e.*, the two elements in *Candidate* with minimum weight.

5 POSTPONING THE DETOURS' COMPUTATION

In this section, we present the Postponed Node Classification (PNC) algorithm. The PNC algorithm has time complexity $O(kn(m + n \log n))$ with a memory consumption similar to the one of the NC algorithm. However, it is faster in practice than the NC algorithm.

Even though the NC algorithm consumes less time during each SP algorithm call than Yen's algorithm, the total number of calls remains equal to Yen's algorithm. Here, with the help of a lower bound on the weight of a simple detour, we propose (using an idea similar to the one of the SB algorithm) to postpone the calls in order to avoid some of them. We prove that such postponement does not hurt the correctness of the algorithm.

Let us describe our algorithm PNC.

As in the NC algorithm, our algorithm starts by computing an SP in-branching T rooted at t that will be used throughout the execution of the algorithm. Then, like the NC algorithm, the PNC algorithm proceeds by phases where a new path is added to the output and its detours are computed and added in the set *Candidate*. Our algorithm differs from the NC algorithm in how and when it computes the detours of the paths but also in the structure of the elements in the heap *Candidate*.

Let us consider a phase when a s-t path $P = (s = u_0, u_1, \cdots, u_i, \cdots, u_{r-1}, u_r = t)$ is extracted from *Candidate*. Let $0 \leq i < r$ and consider the step when a shortest simple detour of P at u_i is computed. Let N be the set of neighbors v of u_i such that the paths with prefix $(u_0, \cdots, u_i, v)$ have already been added to *Output*. Let $D' = D \setminus \{(u_i, v) \mid v \in N\}$ and let $D_i(P) = D' \setminus (u_0, \cdots, u_{i-1})$.

Let us describe how the PNC algorithm finds a new shortest simple detour P' of P at u_i. Recall that a detour P' is said new if P' has not been added to the *Output* yet. Let $v_{LB} \in D_i(P)$ be a neighbor of u_i (neither in N nor in the prefix of P) such that the residual weight of (u_i, v_{LB}) is minimum, *i.e.*, $\delta(u_i, v_{LB}) \leq \delta(u_i, v')$ for every $v' \in N^+_{D_i(P)}(u_i)$ (recall that $\delta(u, v) = \omega(u, v) + \omega(P^T_{vt}) - \omega(P^T_{ut})$ denotes the residual weight of arc uv as defined in Section 2.2). Note that, by definition of the residual weight, the path $P_{LB} = (s, u_1, \cdots, u_i, P^T_{v_{LB}t})$ is a shortest new detour (not necessarily simple) of P at u_i, so in particular:

Claim 1. *$\omega(P_{LB}) \leq \omega(P')$ for any new simple detour P' of P at u_i*

Similarly to the SB algorithm (and in contrast to the NC algorithm), the PNC algorithm may add non-simple paths to the set *Candidate*. Precisely, each element in *Candidate* has the form $(P = (s = u_0, \cdots, u_r = t), \omega(P), i, \zeta)$ where i is its deviation index and ζ is a boolean flag indicating whether the path P is simple or not.

The main idea of the PNC algorithm (Algorithm 2) is the following. Instead of computing naively all of the shortest simple detours of P, *i.e.*, a shortest simple detour at u_j for all $i \leq j < r$, the following procedure is used. We first label the vertices of the graph with respect to P using the procedure presented in Section 3.2 for testing whether a deviation $Q(v_j, e)$ is simple or not. Then, for each vertex $u_j \in P$, with $i \leq j < r$, we find the neighbor v_{LB} of u_j (if any) such that the residual weight of (u_j, v_{LB}) is minimum and we check whether $\lambda(v_{LB}) > j$ or not. If this is the case, the detour $P_{LB} = (s, u_1, \cdots, u_j, P^T_{v_{LB}t})$ is simple and we add $(P_{LB}, \omega(P_{LB}), j, \zeta = 1)$ to the set *Candidate*. Otherwise, *i.e.*, if $\lambda(v_{LB}) \leq j$, the detour P_{LB} is added to the set *Candidate* (even though it is not simple) with its weight $\omega(P_{LB})$ as a key, *i.e.*, the element $(P_{LB}, \omega(P_{LB}), j, \zeta = 0)$ is added to *Candidate*. The idea is that, in the latter case, the non-simple path added to *Candidate* may never be extracted from *Candidate*, in which case a call to an SP algorithm is saved.

When an element $(P, \omega(P), i, \zeta)$ is extracted from *Candidate*. If $\zeta = 1$, the simple path P is added to the *Output* and its detours are added to *Candidate* as explained above. Otherwise, *i.e.*, P is not simple, it will be "repaired" into

Algorithm 2 Postponed Node Classification (PNC)

```
1: **Input** A digraph $D = (V, A)$, source $s \in V$, sink $t \in V$ and an integer $k$
2: **Output** $k$ shortest simple $s$-$t$ paths
3: Let $Candidate \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
4: $T \leftarrow$ an SP in-branching of $D$ rooted at $t$
5: Add $(P_{st}(T), \omega(P_{st}(T)), 0, 1)$ to $Candidate$
6: **while** $Candidate \neq \emptyset$ and $|Output| < k$ **do**
7:     $(P = (s, u_1, \cdots, t), \omega(P), i, \zeta) \leftarrow$ extract a shortest element from $Candidate$
8:     $\pi \leftarrow (s, u_1, \cdots, u_{i-1})$
9:     $Dev_{old} = \{e = (u_i, v)$ s.t. there is a path in $Output$ having $\pi.e$ as prefix$\}$
10:    **if** $\zeta = 1$ ($P$ is simple) **then**
11:        Add $P$ to $Output$
12:        $\lambda \leftarrow$ labelling of the vertices of $D$ with respect to $P$ and $T$
13:        **for** each vertex $u_j$ in $(u_i, \cdots, t)$ **do**
14:            $(u_j, v_{LB}) \leftarrow$ an arc in $A \setminus Dev_{old}$ with minimum $\delta$ among those with tail $u_j$
15:            $P_{LB} \leftarrow (s, \cdots, u_j, v_{LB}, P^T_{v_{LB}t})$
16:            $\zeta' \leftarrow 0$
17:            **if** $\lambda(v_{LB}) > j$ (*i.e.*, $P_{LB}$ is simple) **then**
18:                $\zeta' \leftarrow 1$
19:            Add $(P_{LB}, \omega(P_{LB}), j, \zeta')$ to $Candidate$
20:    **else**
21:        Compute a shortest $u_i$-$t$ path $Q$ in $D' = (V \setminus \pi, A \setminus Dev_{old})$
22:        **if** $Q$ exists **then**
23:            Add $(P' = \pi.Q, \omega(P'), i, 1)$ to $Candidate$
24: **return** $Output$
```

a simple path and re-added to *Candidate*. More precisely, after the extraction of P from *Candidate*, an SP algorithm is called to find a shortest (simple) path Q from u_i to t in $D_i(P)$ and P is replaced by $P' = (s, u_1, \cdots, u_{i-1}, Q)$. Claim 1 ensures that the order of extraction of the simple paths from *Candidate* remains valid. And finally, such postponement of this SP algorithm call may end up by skipping it.

6 EXPERIMENTAL EVALUATION

In this section we describe our experimental evaluation of the algorithms presented in this paper. We start by describing our implementation and experimental settings (Section 6.1). Then, we discuss our experimental results on road in Section 6.2 and present in Section 6.3 a refined comparison of the PNC and the SB* algorithms on road networks with respect to the Dijkstra rank of the queries. We finally discuss our experimental results on complex networks in Section 6.4.

6.1 Experimental settings

Here, we specify the details of the implementation and the setting used in our experiments.

We have implemented[3] all the algorithms presented in this paper (Yen, NC [12], PNC, SB [24], SB* and PSB) in C++ and our code is publicly available [2].

[3]Despite several queries, we have not been granted access to the code used for experiments in [12, 24].

Following [24], we have implemented a pairing heap data structure [13] supporting the decrease key operation, and we use it for Dijkstra's shortest path algorithm. Our implementation of the Dijkstra's shortest path tree algorithm is lazy. That is, it stops the computation as soon as the distance from query node v to t is proved to be the smallest one. Further computations might be performed later for another node v' at a larger distance from t starting from this partial shortest path tree already computed. Our implementation of Dijkstra's algorithm supports an update operation when a node or an arc is added to the graph. Moreover, we have implemented a special copy operation that updates 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 in the SB* algorithm.

Observe that, in our implementations, the parameter k is not part of the input, and so the sets of candidates are simply implemented using pairing heaps. This choice enables us to use these methods as generators able to return the next shortest path as long as one exists. Note that, if k were part of the input, the data structure used to store the candidates could be changed in order to contain only the k best candidates, but the algorithm would only be able to return k paths even if more exist. Moreover, for the SB, SB*, PSB, PSBv2 and PSBv3 algorithms, following [24], we store the candidates in two heaps. The first heap stores the simple candidates ($Candidate_{simple}$) and the second one stores the non-simple candidates ($Candidate_{not-simple}$). Then, we extract candidates from $Candidate_{simple}$ as long as the weight of a shortest simple path in $Candidate_{simple}$ is smaller or equal than the weight of a shortest non-simple path in $Candidate_{not-simple}$. This way, we prioritize the extraction of simple paths.

Concerning the PSBv2 and PSBv3 algorithms, based on preliminary experiments, we choose to update the value of the threshold θ dynamically as follows. Recall that, when looking for the i^{th} path, a corresponding in-branching will be stored only if the weight of that path is at most θ times the weight of the $(i-1)^{th}$ path. Precisely, let ℓ_s be the weight of a smallest element in $Candidate_{simple}$ and let ℓ_{ns} be the weight of a smallest non-simple element in $Candidate_{not-simple}$. If at least one of these sets is empty, θ keeps its previous value. Otherwise, let $c = \max(\frac{\ell_s}{\ell_{ns}}, \frac{\ell_{ns}}{\ell_s})$. The value of θ is set to $1 + \alpha(c-1)$, for some constant $\alpha > 0$. The intuition is to store an in-branching only if it is expected to be used soon, that is, while extracting one of the upcoming paths. Observe that in our experiments we have set the factor α in the formula for computing θ to $\alpha = 11$, based on preliminary experiments.

Test instances. We have evaluated the performances of our algorithms on some road networks from the 9th DIMACS implementation challenge [8] and on several complex networks.

A road network of a city is the digraph modeling its roads, *i.e.*, a vertex is associated to each crossroad, and there is an arc of weight w between two vertices if and only if there is a road of physical length w (in km) between their corresponding crossroads. Road networks are known to be sparse, almost planar and to have a bounded degree [35]. The road networks ROME, DC and DE are referred to as *small road networks*, while the road networks NY, BAY and COL are referred to as *big* ones. The characteristics of these graphs are reported in Table 1.

On the other hand, complex networks model different types of networks. Generally, they are characterized by being small-world, *i.e.*, they have a logarithmic diameter, a power-law degree distribution and a high clustering coefficient [6]. For instance, the BIOGRID network represents mutation/deletion of genes resulting in lethality when combined in a same cell [28]. The DIP network represents protein to protein interactions [30]. The FB network represents social circles from Facebook [27]. Likewise, the LOC network is a graph provided from Brightkite location-based social networking [26]. Finally, the P2P network is the peer-to-peer network of the Gnutella file sharing network [26] and the CAIDA (2013.11.01) network is the graph of the relationships between the autonomous systems of the Internet [33]. As these networks are unweighted, we only consider the number of edges as the weight of a path. For each network, we work only on its largest biconnected component. The characteristics of these graphs are depicted in Table 1

network	n	m	D	$\langle d\rangle$	$-\alpha$	$\langle cc\rangle$	Description
ROME	3 353	8 870	57	5.2	-	0.025	Road network of Rome [8]
DC	9 559	29 682	140	6.2	-	0.039	Road network of Washington DC [8]
DE	49 108	119 520	573	4.8	-	0.024	Road network of Delware [8]
NY	264 346	733 846	664	5.5	-	0.02	Road network of New York [8]
BAY	321 270	800 172	791	4.9	-	0.016	Road network of San Francisco Bay area [8]
COL	435 666	1 057 066	1 219	4.8	-	0.017	Road network of Colorado area [8]
BIOGRID	2 318	12 580	7	21.7	1.96	0.20	Mutation/deletion of genes resulting in cell lethality [28]
FB	3 698	85 963	6	93.0	-	0.61	Social circles from Facebook [27]
P2P	5 606	23 510	8	16.7	-	0.014	Peer-to-peer network of the Gnutella file sharing network [26]
DIP	13 969	60 621	17	17.4	2.38	0.11	Protein-protein interaction network [30]
CAIDA	29 432	143 000	9	19.4	2.06	0.42	Relationships between Autonomous Systems of the Internet [33]
LOC	33 187	188 577	11	22.7	2.25	0.29	Brightkite location-based social networking service provider [26]

Table 1. Characteristics of the graphs used in kSSP experiments: number of nodes (n), number of edges (m), diameter (D), average degree ($\langle d\rangle$), exponent $-\alpha$ of the power-law degree distribution, and average clustering coefficient ($\langle cc\rangle$). For the complex networks, these statistics refer to the largest connected component.

In our experiments, we have randomly selected 100 destinations per network. For each destination, we have chosen one source with *Dijkstra rank* r for every $r \in \{2, 10, 10^2, \cdots, 10^{\lfloor \log_{10} n\rfloor}, n\}$. That is, we have selected about $100\lfloor \log_{10} n\rfloor$ queries (source-destination pairs of vertices) per network of order n. Recall that, during the computation of an in-branching T rooted at t using Dijkstra's algorithm, all vertices are settled in a fixed order (the order in which Dijkstra's algorithm removes the vertices from its priority queue). The Dijkstra rank $r_t(v)$ of vertex v with respect root vertex t is its position in this order and we have $r_t(t) = 1 < r_t(v) \leq n$ for all $v \in V \setminus \{t\}$. Then, for each network and for each source-destination pair, we have run each algorithm for values of k up to 1 000 on road networks and up to 10 000 on complex networks. Notice that, thanks to our choice of implementing the kSSP algorithms as generators, we are able to measure the running time for obtaining k' paths, for several values of $k' \leq k$, while processing a query for extracting k paths. Because of the excessive running time of Yen's algorithm, we have chosen to run it only on small 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 an indication of the memory consumption that is independent from the implementation and the architecture of the machine [21].

All reported computations have been performed on a computer equipped with a 3.70GHz Intel Core i9-10900K processor, 64GB of RAM, and with operating system fedora 35.

6.2 Experimental results on road Networks

In this section, we describe and analyze our experimental results on road networks.

In Table 2, we have reported the average and the median of the algorithms' running times in all considered road networks when $k = 1\,000$, and the number of stored in-branchings is reported in Table 3. Moreover, in Figure 1, we report the evolution of the average and median running times of the algorithms in the COL and DC networks when the number k of reported paths increases (the results are similar for the other road networks). Finally, Figure 2 presents pairwise comparisons of the running times and number of stored in-branchings for all source-destination pairs on COL network.

We first observe from the results reported in Table 2 and Figures 1a and 1b that all algorithms are faster than Yen's algorithm on small road networks (the average speed up is between one and two orders of magnitude). Similar experiments described in [12, 24] lead to the same conclusion on big road networks.

		Rome	DC	DE	NY	BAY	COL
Y	avg	1 053	4 470	73 727	-	-	-
	med	388	423	9 434	-	-	-
	max	16 382	67 434	1 611 504	-	-	-
NC	avg	229	471	5 565	34 915	45 447	66 048
	med	112	220	3 686	14 654	24 854	39 043
	max	1 310	2 387	16 978	137 266	136 564	212 174
PNC	avg	**120**	**238**	**1 924**	**12 699**	**16 082**	**23 428**
	med	**89**	165	**1 216**	**6 677**	**9 260**	**16 114**
	max	**690**	**2 011**	**10 251**	**91 333**	**108 320**	**133 414**
SB	avg	339	325	8 574	43 394	79 667	110 382
	med	286	166	3 808	24 006	36 014	43 076
	max	1 324	3 203	74 172	451 181	789 828	2 021 857
SB*	avg	269	298	7 208	36 433	68 001	97 679
	med	218	**150**	3 381	18 290	31 930	37 263
	max	1 275	3 167	58 293	449 570	663 995	1 453 920
PSB	avg	294	322	4 956	43 182	46 466	63 961
	med	242	219	3 932	34 868	35 275	44 990
	max	1 118	2 645	21 282	257 135	273 211	362 595
PSBv2	avg	302	332	4 845	42 003	44 426	60 089
	med	258	230	3 888	34 481	32 966	43 096
	max	1 134	2 538	22 811	228 794	257 373	325 434
PSBv3	avg	299	328	4 770	41 724	44 088	59 496
	med	252	232	3 796	34 022	32 394	41 705
	max	1 138	2 572	22 750	229 333	256 810	325 700

Table 2. Running time (ms) of the algorithms on road networks, ($k = 1,000$)

	Rome	DC	DE	NY	BAY	COL
NC, PNC and PNC*	1	1	1	1	1	1
SB and SB*	1 397	651	1 698	1 550	1 727	1 737
PSB	827	**447**	633	700	629	655
PSBv2	**766**	454	**578**	**604**	**569**	**580**
PSBv3	776	470	598	615	580	594

Table 3. Average number of stored trees using some *kSSP* algorithms on road networks, ($k = 1,000$)

Next, the simulation results reported in Table 2 and Figure 1 confirm that the use of shortest path tree update procedures in the SB* algorithm helps reducing the running time compared to the SB algorithm. More precisely, the average and median running times of the SB* algorithm are significantly smaller than the ones of the SB algorithm on all road networks. Note that, by design, the number of stored in-branchings is the same for both algorithms. Concerning the PSB algorithm, as shown in Table 2 and Figures 2c and 2d, it is faster than the NC algorithm and consumes less memory than the SB algorithm. It is also faster on average than the SB* algorithm on some networks (DE, BAY, COL) but its median running time is always larger than the one of the SB* algorithm.

Moreover, from Tables 2 and 3, we observe that the two special variants of the PSB algorithm, namely the PSBv2 and the PSBv3 algorithms, lead to a small improvement of the running time (up to 5% time reduction) but to a significant reduction of the memory consumption (up to 15% memory reduction).

The most important observation from the results reported in Table 2 and Figures 1a and 1b is that the PNC algorithm outperforms all other algorithms on road networks when the number of requested paths is at least 20, except for the median running time on DC. When $k = 1,000$, the average and median running times of the PNC algorithm are between 2 and 5 times smaller than for the other algorithms. This is particularly interesting considering that the memory consumption of the PNC algorithm is very small. The conclusion is however different when the number of requested paths is less than 20. Indeed, we observe that the SB-like algorithms (SB, SB*, PSB, etc.) offer better performances for small values of k.

Another important observation from the results reported in Table 2 and Figures 1a and 1b is that, for all algorithms, there is a significant gap between the average and the median running times. To better analyze this observation, Figure 2 shows pairwise comparisons of the running times and number of stored in-branchings for all source-destination pairs on COL network when $k = 1,000$. In these scatter plots, we observe, for all algorithms, huge variations in the running times depending on the source-destination pairs. These variations are sufficient to explain the gap between the average and the median running times. Actually, Table 2 shows that the maximum running time for a source-destination pair can be up to 20 times larger than the average running time of an algorithm and up to 50 times larger than its median running time (e.g., SB on COL). To better understand this behavior, we analyse in Section 6.3 the impact of the Dijkstra rank of the queries on the running time of the algorithms.

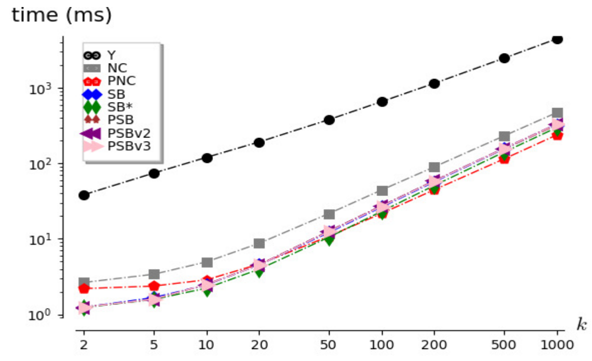
To conclude, among the considered algorithms, the PNC and the SB* algorithms are the fastest on road networks. The PNC algorithm has a better average running time and is more stable than the SB* algorithm whose performances are seriously affected by a few outliers. Furthermore, the memory consumption of the PNC algorithm is very small compared to all SB-like algorithms. Hence, the PNC algorithm seems to be the best compromise on road networks.

6.3 Impact of the Dijkstra rank of the queries

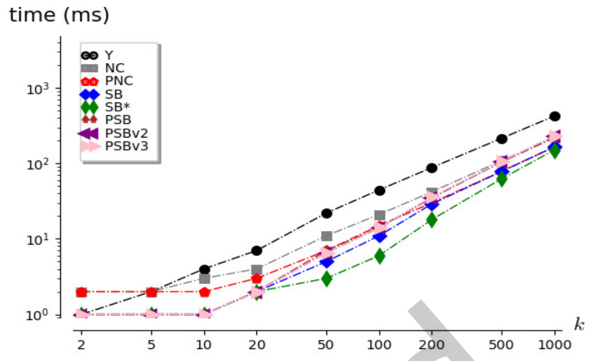
As noticed in Section 6.2, some algorithms have different behaviors with respect to the structure of the network and the query's properties. We therefore investigate in this section whether the Dijkstra rank of an s-t query may explain the variations of the running time of the algorithms. To this end, we have reported in Figure 3 the average and median running times of the algorithms on the COL road network for increasing values of the Dijkstra rank of the queries. We have also reported the average numbers of stored trees and Dijkstra calls. Observe also that the colors of the points in the scatter plots of Figure 2 correspond to the Dijkstra ranks of the queries.

We observe in Figures 3a and 3b that the Dijkstra rank of the queries has little impact on the running time of the PNC algorithm. This can also be concluded from the scatter plots of Figures 2a and 2e. On the other hand, we observe for the SB-like algorithms, and in particular for the SB* algorithm, that the running time decreases drastically when the Dijkstra rank is large. More precisely, we observe almost one order of magnitude reduction on average and up to two orders of magnitude on median. To explain these behaviors, let us compare the operations performed by the PNC and the SB* algorithms after each extraction of a path P from $Candidate$.

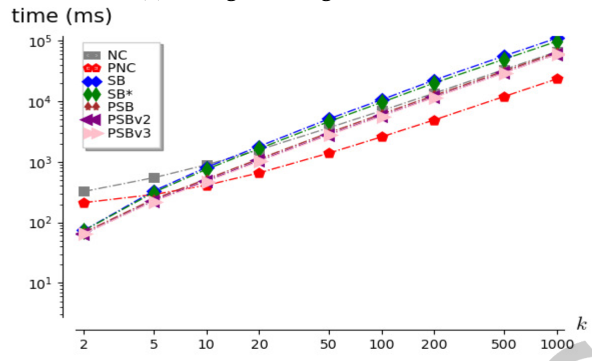
We start when a simple path $P = (s = v_1, v_1, \cdots, v_r = t)$ with deviation index i is extracted. In this case, both algorithms start computing a labeling λ of the vertices that will be used to determine if a considered deviation leads to a simple detour or not. Then, while the SB* algorithm computes all its detours at once, the PNC algorithm considers only one deviation per vertex v_j with $i \leq j < r$. More precisely, the SB* algorithm considers all the detours starting from deviation (v_j, w), with $i \leq j < r$ and $w \in N^+(v_j)$, while the PNC algorithm considers for vertex v_j only one detour (v_j, v_{LB}) with minimum residual weight, where $v_{LB} \in N^+(v_j)$. Overall, although the SB*



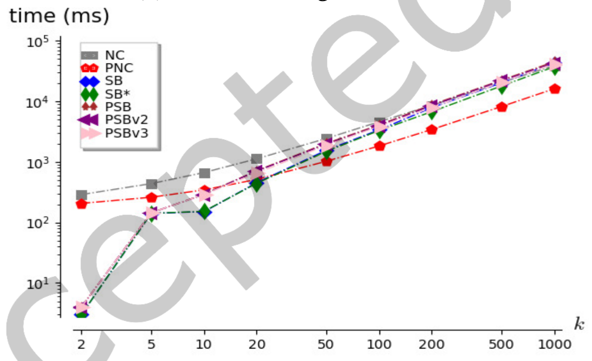
(a) Average running time on DC



(b) Median running time on DC



(c) Average running time on COL



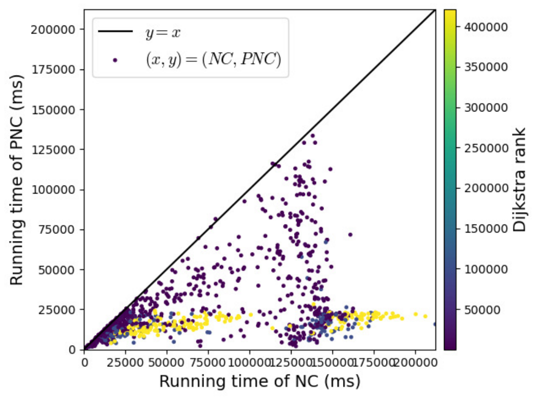
(d) Median running time on COL

Fig. 1. The running time of the kSSP algorithms on road networks with respect to the values of k.

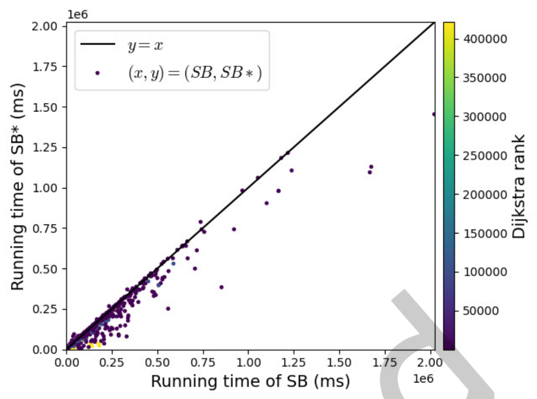
algorithm do slightly more work than the PNC algorithm, the computational cost of the operations performed by both algorithms when a simple path is extracted from *Candidate* is similar and cannot be used to explain the differences of behaviors of the algorithms.

Let us now consider the case when a non-simple path P with deviation index i is extracted from *Candidate*. In this case, the PNC algorithm searches for a shortest simple v_i-t path Q (line 21 of Algorithm 2) in the digraph using the residual weights. This is done using Dijkstra's algorithm to compute an out-branching from v_i and computations are stopped as soon as t is extracted. Now, observe that, thanks to the use of the residual weights, after the extraction of a vertex v from the priority queue of Dijkstra's algorithm, the next vertex v' to be extracted is likely to be an out-neighbor of v such that $\delta(v, v') = 0$. In other words, the algorithm tries to reach t as early as possible while keeping the total number of extractions from the priority queue as small as possible. Hence, the running time of the operations performed by the PNC algorithm to "repair" a path is rather independent from the distance between v_i and t and also from the Dijkstra rank of the source-destination pair.

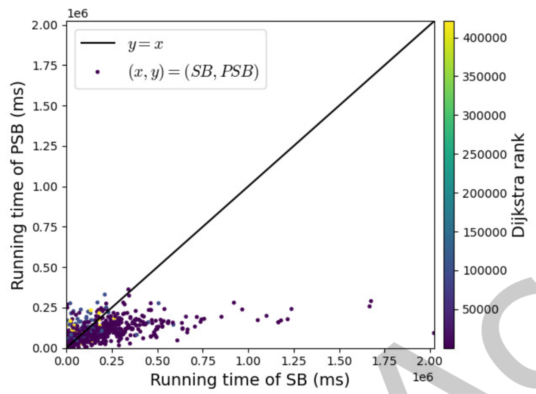
Concerning the SB* algorithm, it has to compute the SP in-branching T_{h+1} of the sequence $\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1})$, representing the non-simple path P, until reaching the head w_h of arc $e_h = (v_h, w_h)$. Clearly, the time needed to find the path $P^{T_{h+1}}_{w_h t}$ depends on the Dijkstra rank of w_h in the in-branching T_{h+1}, which increases with the distance to t. Therefore, when considering a source-destination pair with large Dijkstra rank, the running time of the SB* algorithm will be very small if all computed detours involve deviations that are near t (*i.e.*, with small



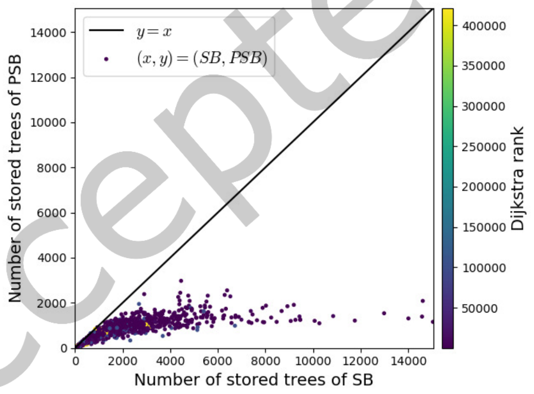
(a) Running time of NC and PNC



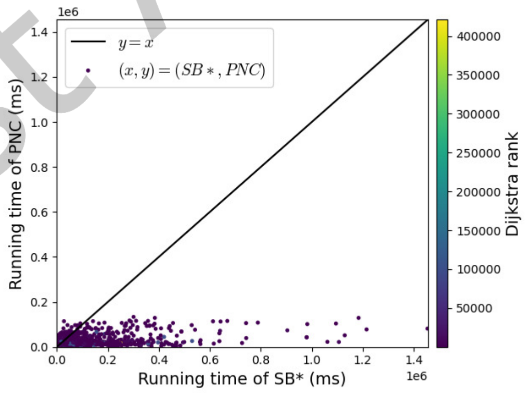
(b) Running time of SB and SB*



(c) Running time of SB and PSB

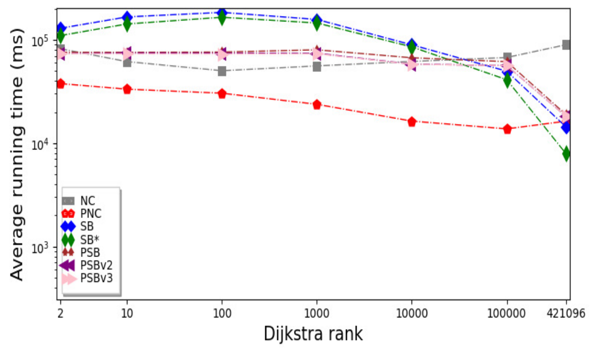


(d) Number of trees of SB and PSB

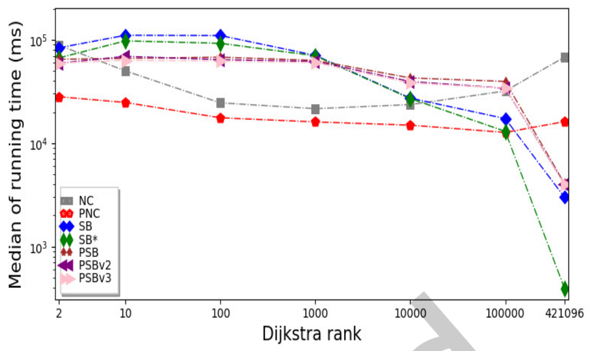


(e) Running time of SB* and PNC

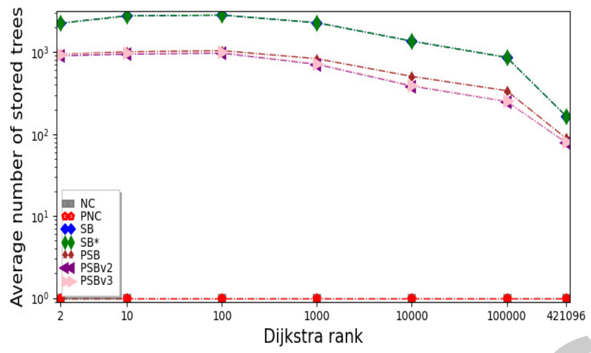
Fig. 2. Comparison of the running time and the number of stores trees on COL. Each dot corresponds to one pair source/destination ($k = 1,000$).



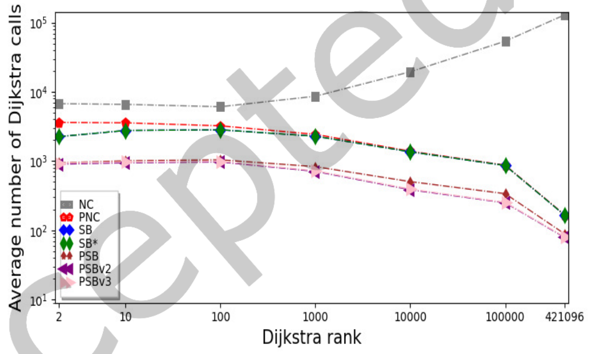
(a) Average running time



(b) Median running time



(c) Average number of stored trees



(d) Average number of Dijkstra calls

Fig. 3. Influence of the Dijkstra rank on the performances of the kSSP algorithms on COL ($k = 1\,000$)

Dijkstra ranks) and very large if the deviations are far from t (*i.e.*, with large Dijkstra ranks). This explains the huge difference between the median and the average running times of the SB* algorithm for source-destination pairs with large Dijkstra rank observed in Figures 3a and 3b. This also explains the fact that, in the scatter plots of Figure 2e, the outliers (*i.e.*, the queries which require much more time for the SB* algorithm than for the PNC algorithm) are mostly of small Dijkstra ranks. To emphasis this latter fact, Table 4 represents the proportion of the queries, for each Dijkstra rank, that the SB* algorithm solves faster than the PNC algorithm (recall that there are exactly 100 queries per rank). Finally, the fact that the PNC algorithm is faster than the SB* algorithm on both average and median running times for queries with small Dijkstra rank is due to the use of the residual weights that speeds up the computation of the deviations.

Dijkstra rank	2	10	100	1000	10000	100000	421096
SB*	22.5	12.5	11.3	23.8	35.4	50.3	86.8
PNC	77.5	87.5	88.7	76.2	64.6	49.7	13.2

Table 4. Percentage of the number of times one of the SB* or PNC algorithms is faster than the other on COL per Dijkstra ranks ($k = 1\,000$)

6.4 Complex Networks

Here we analyse and we try to explain the behavior of the algorithms on complex networks, except for Yen's algorithm because of its excessive running time (as already mentioned above).

We have reported in Table 5 the average and the median of the algorithms' running time in all considered complex networks when $k = 10\,000$, and in Table 6 the number of stored in-branchings. Furthermore, we have reported in Figure 4 the evolution of the average and median running times of the algorithms in the CAIDA and LOC networks when the number k of reported paths increases (the results are similar for the other complex networks).

On complex networks, the PSB algorithm and its two variants PSBv2 and PSBv3 are the fastest kSSP algorithms among the considered algorithms (Table 5 and Figure 4). Considering the memory consumption, all of the kSSP algorithm have a small memory consumption even for $k = 10,000$. Indeed, as shown in Table 6, the number of stored in-branchings does not exceed 59 and can be as small as 3 on FB. It is also shown in Table 5 that the running time of PSB and its two variants (PSBv2 and PSBv3) is similar.

In what follows, we give some qualitative arguments that may explain the fact that the PSB algorithm is the fastest among all considered algorithms on complex networks.

Suppose that P is a shortest simple path from s to t. On complex networks, it is likely to have a vertex v on P with high degree. Let us study the behavior of the different variants of the PSB algorithm when they compute the detours of P at v, in contrast with the other algorithms.

First, the Yen's, NC and PNC algorithms may compute, independently, for each vertex $v' \in N^+(v)$ neighbor of v a shortest path from v' to t resulting with $|N^+(v)|$ shortest path algorithm calls to find the shortest simple detours at the neighbors of v. On the other hand, the SB* and the PSB algorithms compute at most one shortest path in-branching T at v, that works for each neighbor v' of v. In other words, the SB* and the PSB algorithms are favorable to iterate on v. Moreover, as v has a high degree, it is supposed that a large number of the neighbors v' of v leads to simple candidates, *i.e.*, $P^T_{v't} \cap (s, \cdots, v) = \emptyset$, where $(s...v)$ is the prefix of the current path. So, the number of shortest path in-branchings computed and/or stored using the SB* and the PSB algorithms is expected to be small.

In addition, as the number of hops (*i.e.*, number of edges) of a shortest path is "small" (remember that complex network are small-worlds and so have small diameters), the number of calls of the shortest path in-branching update is expected to be small for the PSB algorithm. As this procedure is faster for the PSB algorithm than for the SB* algorithm and that the number of calls is similar, the PSB algorithm is faster than the SB* algorithm on complex networks. This is not valid on road network because the number of hops of a shortest path may be big and the shortest path in-branching update is called many more times.

To conclude, on complex networks, the PSB algorithm is the fastest among the considered algorithms. It has a reasonable memory consumption and this seems to be related to the structural properties of complex networks.

7 CONCLUSION

In this paper, we have presented several algorithms for the kSSP problem. In particular, we have proposed several new algorithms for this problem with the aim of reducing the running time and/or the memory consumption of the algorithms.

Our simulation results show that the best algorithm to be chosen for solving the kSSP problem depends on the use case. For instance, on the considered complex networks, the PSB algorithm achieves the best results. Indeed, it is the fastest among the considered algorithms, and, similarly to the other algorithms, it has low memory consumption. Besides, on road networks, the PNC algorithm seems very promising. Indeed, it is the fastest (on average, when $k \geq 20$) among the considered algorithms and it has a low memory consumption (it stores only one in-branching in memory). Furthermore, it is very stable as its performances are barely influenced by the

		BIOGRID	FB	P2P	DIP	CAIDA	LOC
NC	avg	1 431	753	2 679	7 766	30 522	15 598
	med	859	695	1 605	3 853	10 432	7 416
	max	10 346	15 504	27 028	79 790	219 322	285 726
PNC	avg	748	678	1 333	3 467	10 665	7 258
	med	670	636	1 260	3 252	9 109	6 912
	max	3 734	2 160	4 069	61 036	79 339	62 142
SB	avg	523	1 093	316	443	5 090	1 418
	med	441	648	298	368	3 911	1 172
	max	1 660	5 996	734	6 833	19 845	4 624
SB*	avg	495	1 125	242	404	4 740	1 314
	med	425	776	238	343	3 808	1 113
	max	1 508	5 654	480	7 047	17 963	4 250
PSB	avg	**215**	462	**146**	**240**	**1 971**	**653**
	med	**202**	396	**145**	**200**	**1 973**	**608**
	max	427	1 505	272	**3 588**	**8 550**	1 983
PSBv2	avg	220	447	147	**240**	1 981	657
	med	203	361	**145**	201	2 005	612
	max	532	**1 310**	**253**	3 594	8 581	2 018
PSBv3	avg	221	**441**	147	245	1 983	661
	med	205	**357**	**145**	203	1 988	609
	max	**417**	1 454	266	3 795	8 760	**1 907**

Table 5. Running time (ms) of the algorithms on Complex networks, ($k = 10,000$)

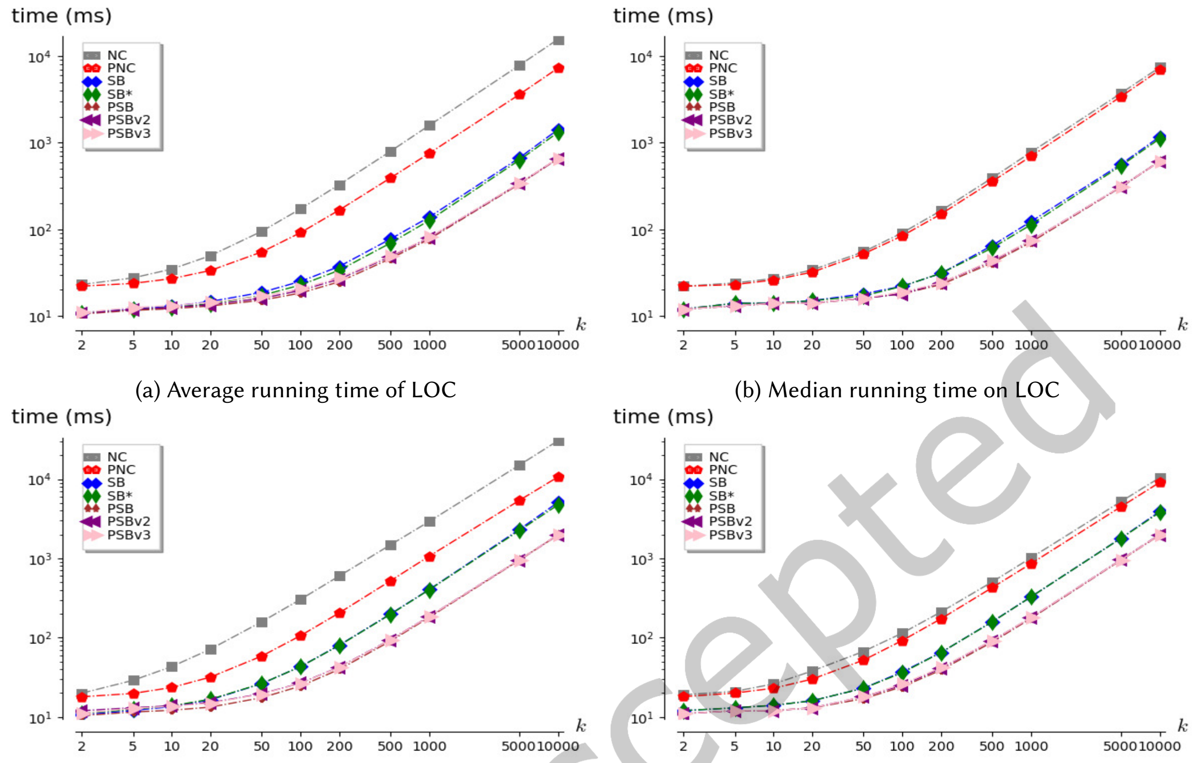
	BIOGRID	FB	P2P	DIP	CAIDA	LOC
NC, PNC and PNC*	1	1	1	1	1	1
SB and SB*	**12**	4	41	39	**7**	**9**
PSB	**12**	**3**	**33**	**31**	**7**	**9**
PSBv2	13	4	34	32	**7**	**9**
PSBv3	25	6	58	59	12	18

Table 6. Average number of stored trees using some *kSSP* algorithms on complex networks, ($k = 10,000$)

Dijkstra rank of the source with respect to the destination. However, if we know that the Dijkstra rank of the queries is large compared to the number of vertices of the digraph, and if large memory consumption is allowed, then the SB* algorithm might be a good choice. Moreover, the SB* algorithm is faster than the PNC algorithm for reporting a small number of paths (e.g., less than 20).

An empirical framework for the selection of the most appropriate *k*SSP algorithm with respect to the use case is suggested in Figure 5.

An open problem is how to handle the *k*SSP problem on networks with arbitrary arc weights (including negative weights). Another interesting question is how to design a data structure enabling to quickly answer *k*SSP queries similarly to the data structures used by the hub labelling and contraction hierarchy schemes to answer distance queries [4]. A probably more difficult question would be to address dynamic networks, *i.e.*, where



(a) Average running time of LOC

(b) Median running time on LOC

(c) Average running time of CAIDA

(d) Median running time on CAIDA

Fig. 4. The running time of the kSSP algorithms on complex network with respect to the values of k

the weights of the arcs evolve along time (e.g., in road networks where the traversal time of an arc may vary). Would it be possible to quickly update the solutions after a modification in the network?

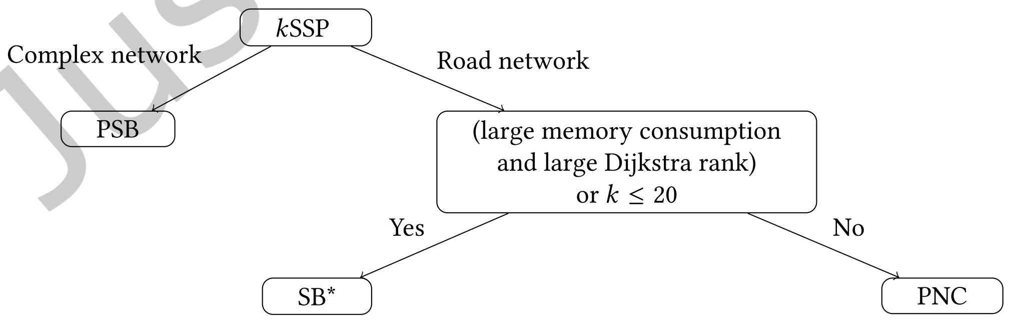

kSSP
Complex network
Road network
PSB
(large memory consumption and large Dijkstra rank) or $k \leq 20$
Yes
No
SB*
PNC

Fig. 5. A framework of the appropriate kSSP algorithm with respect to the use case

REFERENCES

[1] Ali Al Zoobi, David Coudert, and Nicolas Nisse. 2020. Space and Time Trade-Off for the k Shortest Simple Paths Problem. In *18th International Symposium on Experimental Algorithms (SEA) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 160)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 18:1–18:13. https://doi.org/10.4230/LIPIcs.SEA.2020.18

[2] Ali Al Zoobi, David Coudert, and Nicolas Nisse. 2021. *k shortest simple paths (Version 2.0)*. https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths.

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

[4] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. In *Algorithm engineering*. Springer, 19–80.

[5] M. Betz and H. Hild. 1995. Language models for a spelled letter recognizer. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, Vol. 1. IEEE, 856–859. https://doi.org/10.1109/ICASSP.1995.479829

[6] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. 2006. Complex networks: Structure and dynamics. *Physics reports* 424, 4-5 (2006), 175–308.

[7] S. Clarke, A. Krikorian, and J. Rausen. 1963. Computing the n best loopless paths in a network. *J. Soc. Indust. Appl. Math.* 11, 4 (1963), 1096–1102. https://doi.org/10.1137/0111081

[8] Camil Demetrescu, Andrew V. Goldberg, and D. S. Johnson. 2006. 9th DIMACS Implementation Challenge - Shortest Paths. http://users.diag.uniroma1.it/challenge9/

[9] D. Eppstein. 1998. Finding the k Shortest Paths. *SIAM J. Comput.* 28, 2 (1998), 652–673. https://doi.org/10.1137/S0097539795290477

[10] David Eppstein. 2016. *Encyclopedia of Algorithms*. Springer New York, Chapter k-Best Enumeration, 1003–1006. https://doi.org/10.1007/978-1-4939-2864-4_733

[11] David Eppstein and Denis Kurz. 2017. K-Best Solutions of MSO Problems on Tree-Decomposable Graphs. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, Vol. 89. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:13. https://doi.org/10.4230/LIPIcs.IPEC.2017.16

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

[13] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. 1986. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica* 1, 1 (1986), 111–129. https://doi.org/10.1007/BF01840439

[14] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. 2000. Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. *Journal of Algorithms* 34, 2 (2000), 251–281. https://doi.org/10.1006/jagm.1999.1048

[15] Jun Gao, Huida Qiu, Xiao Jiang, Tengjiao Wang, and Dongqing Yang. 2010. Fast top-k simple shortest paths discovery in graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*. 509–518.

[16] Zvi Gotthilf and Moshe Lewenstein. 2009. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Inform. Process. Lett.* 109, 7 (2009), 352–355.

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

[18] Yijie Han and Tadao Takaoka. 2016. An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths. *Journal of Discrete Algorithms* 38 (2016), 9–19.

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

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

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

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

[23] D. Kurz. 2018. *k-best enumeration - theory and application.* Theses. Technischen Universität Dortmund. https://doi.org/10.17877/DE290R-19814

[24] D. Kurz and P. Mutzel. 2016. A Sidetrack-Based Algorithm for Finding the k Shortest Simple Paths in a Directed Graph. In *Int. Symp. on Algorithms and Computation (ISAAC) (LIPIcs, Vol. 64)*. Schloss Dagstuhl, 49:1–49:13. https://doi.org/10.4230/LIPIcs.ISAAC.2016.49

[25] Eugene L Lawler. 1972. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science* 18, 7 (1972), 401–405.

[26] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2–42. https://doi.org/10.1145/1217299.1217301

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Rose Oughtred, Chris Stark, Bobby-Joe Breitkreutz, Jennifer Rust, Lorrie Boucher, Christie Chang, Nadine Kolas, Lara O'Donnell, Genie Leung, Rochelle McAdam, et al. 2019. The BioGRID interaction database: 2019 update. *Nucleic acids research* 47, D1 (2019), D529–D541.

[29] Seth Pettie. 2004. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science* 312, 1 (2004), 47–74.

[30] Lukasz Salwinski, Christopher S Miller, Adam J Smith, Frank K Pettit, James U Bowie, and David Eisenberg. 2004. The database of interacting proteins: 2004 update. *Nucleic acids research* 32, suppl_1 (2004), D449–D451.

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

[32] Douglas R Shier. 1979. On algorithms for finding the k shortest paths in a network. *Networks* 9, 3 (1979), 195–214.

[33] The Cooperative Association for Internet Data Analysis (CAIDA). 2013. The CAIDA AS Relationships Dataset. http://www.caida.org/data/active/as-relationships/.

[34] Virginia Vassilevska Williams and Ryan Williams. 2010. Subcubic equivalences between path, matrix and triangle problems. In *IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 645–654.

[35] Feng Xie and David Levinson. 2007. Measuring the structure of road networks. *Geographical analysis* 39, 3 (2007), 336–356.

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

[37] J. Y. Yen. 1971. Finding the k Shortest Loopless Paths in a Network. *Management Science* 17, 11 (1971), 712–716. https://doi.org/10.1287/mnsc.17.11.712