Finding the k Shortest Simple Paths: Time and Space trade-offs

Ali Al Zoobi, David Coudert, Nicolas Nisse

Finding the k Shortest Simple Paths: Time and Space trade-offs*

Ali Al Zoobi and David Coudert and Nicolas Nisse

Université Côte d'Azur, Inria, CNRS, I3S, France

April 13, 2021

Abstract

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 working memory.

We first show how to speed up the SB algorithm using dynamic updates of shortest path trees resulting in a faster algorithm (SB*) with same working memory. We then propose the *parsimonious SB* (PSB) algorithm that significantly reduces the working memory 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 NC and SB. It offers a significant speed up compared to NC while using the same amount of working memory of NC.

Our experimental results on complex networks show that all of the considered algorithms have low working memory, and that the PSB algorithm is the fastest. On road networks, the SB* algorithm is the fastest (on median) among the considered algorithms, but it suffers from a large working memory. The PNC algorithm has comparable running time to SB* on road networks while using the same working memory as NC.

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

*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.

Several algorithms for solving kSP have been proposed. In particular, Eppstein [8] proposed an exact algorithm that computes k shortest paths (not necessarily simple) in $O(m+n\log n+k)$-time, 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.* [6] 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 [16, 19, 35]. It can be applied successfully in bio-informatics [2], especially in biological sequence alignment [30] and in natural language processing [4]. For more applications, see Eppstein's recent comprehensive survey on k-best enumeration [9].

The most famous algorithm for solving the kSSP problem has been proposed by Yen [36] and has time complexity in $O(kn(m + n\log n))$. It has been proved that the kSSP problem can be solved in $O(k)$ iterations of APSP (All Pairs Shortest Paths) [15]. This means that the kSSP problem can be solved in $O(kn(m + n\log\log n))$-time on sparse digraphs and $O(kn^3(\log\log n)/\log^2 n)$-time on dense digraphs using the fastest APSP algorithms [17, 28]. Vassilevska Williams and Williams [33] 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 [10] 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 [11, 14, 16, 18, 21–23, 31]; they all feature the same worst-case running time as Yen's algorithm i.e, $O(kn(m + n\log n))$.

In particular, Feng [11] proposed an improvement of Yen's algorithm called Node Classification (NC) algorithm. With the help of a precomputed shortest path tree of the digraph, NC algorithm classifies the vertices of the digraph with respect to their validity. Thus, computing a shortest simple detour will be restricted to the non-valid sub-digraph that is smaller than the original one. By this restriction, the running time of computing a shortest path remarkably decreases. 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 [22, 23] 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 $k = 1000$ shortest paths in 50 (ms) for the DC network [7] while it required about 5 (s) with Yen's algorithm and about 0.3 (s) by the improvement proposed by Feng [11]. The drawback of the SB algorithm is the need for storing all computed shortest path trees, thus leading to a large usage of working memory.

To conclude, the fastest algorithm with low working memory consumption (i.e, the same working memory as Yen's algorithm) is the Node Classification (NC) algorithm, proposed independently by [11] and [14]. With larger memory consumption, the Sidetrack Based algorithm (SB) [23] can achieve a tremendous speed up.

Our contributions. We propose a new algorithm with low working memory consumption, called Postponed Node Classification (PNC), that is much faster than the NC algorithm while using the same working memory. Our experimental results show that the PNC algorithm is the fastest kSSP algorithm with low working memory among all considered algorithms. Further-

more, it competes with the algorithms with large memory consumption on road networks.

Considering large working memory, 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 network, a space time trade off between SB and NC algorithms. That is, it enables to significantly reduce the working memory 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 PSB-v2 and PSB-v3) that improve its performances, both in terms of working 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 some properties of the queries, such as the number of hops and the stretch from the center. 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 PSB-v2 and PSB-v3. 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 out-neighbors of v in D. Let $w_D : A \to \mathbb{R}^+$ 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 *length* of the path P equals $w_D(P) = \sum_{0 \leq i < r} w_D(v_i, v_{i+1})$ (we will omit D when there is no ambiguity). The *distance* $d_D(u, v)$ between two vertices $u, v \in V$ is the smallest length 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 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 (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 length of the (unique) u-t path P_{ut}^T 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 length

of the (unique) s-u path P_{su}^T 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 Dijkstra's algorithm. 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 in 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 length among all (simple) detours of P at v_i (or at a).

2.2 Yen's algorithm

We start by describing Yen's algorithm [36] trying to give its main properties and drawbacks.

All of the algorithms described below start by computing a shortest s-t path $P_0 = (s = v_0, v_1, \cdots, v_r = t)$, and assume that there is always at least one such path. This is done by applying an SP algorithm from s. Note that P_0 is simple since 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 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 $Q \in Candidate$ with $S \cup \{Q\}$ a top-$(k'+1)$ set of shortest 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

[1] The deviation-index is not kept explicitly in Yen's algorithm. But, since it is a trivial improvement already existing in the literature [24], 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.

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_iv' such that S already contains a path with prefix $(v_0, \cdots, v_i, v')$ is removed from $D_i(Q)$.

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 [11] and Gao et al. [14].

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 precomputed 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 in P^T_{ut}. Formally, the residual weight of arc uv is $\delta(u, v) = w(u, v) + w(P^T_{vt}) - w(P^T_{ut})$. The residual weight is computed only once (after computing T) and remains valid till the end of the execution of the algorithm. Note that an arc in T has residual weight equals 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 execute an SP algorithm to compute a shortest path from v_i to t in $D_i(Q)$. In the case of 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 result in restricting the execution of the SP algorithm to the `yellow` subgraph that 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 of 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 [23]. 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 few aspect of it. 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 [8]. 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 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 $P_{sv_0}^{T_0}, P_{w_0v_1}^{T_1}, \cdots, P_{w_{h-1}v_h}^{T_h}$ followed by the vertices of $P_{w_ht}^{T_{h+1}}$ if this latter path exists. Two consecutive in-branchings T_i and T_{i+1} are not necessarily distinct. SB algorithm ensures that, if P is an s-t path (i.e., if $P_{w_ht}^{T_{h+1}}$ 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 $P_{w_ht}^{T_{h+1}}$ intersects π.

3.2 The SB algorithm

We are now ready to present the SB algorithm, whose pseudocode is presented in Algorithm 1. 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 (that 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 $P_{st}^{T_0}$ 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 $((T_0), \omega(P_{st}^{T_0}), \zeta = 1)$ 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 the 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, for every deviation tailing at the suffix, i.e, for every deviation $e = (v_j, w)$ at v_j with $i \leq j < r$ (i.e, e is tailing at the suffix $(v_i, \cdots, t)$ of P), let $P_{wt}^{T_{h+1}}$ be a shortest path from w to t in T_{h+1} (if any) and let $Q(v_j, e) = (v_0, \cdots, v_j, P_{wt}^{T_{h+1}})$. 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; $Q(v_j, e)$ 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 $P_{wt}^{T_{h+1}}$. 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 $D \setminus \{v_0, \cdots, v_h\}$, which ensures that, if $P_{wt}^{T_{h+1}}$ is found, the path $P_{new} = (s = v_0, \cdots, v_h, P_{wt}^{T_{h+1}})$ is guaranteed to be simple. Moreover, P_{new} has weight $\omega(P_{new}) = \omega((s = v_0, \cdots, v_h, w_h)) + \omega(P_{wt}^{T_{h+1}})$. Then, the representation $\mu' = (\varepsilon' = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), \omega(P_{new}), \zeta = 1)$ is added to *Candidate*. Finally, if no w_h-t path can be found in T_{h+1}, μ is discarded.

Analysis. In the a 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 checks of line 18. 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 are 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 into memory, it has the drawback to possibly have an important consumption of working memory (much more than the NC algorithm that stores a single in-branching, while keeping the whole description of 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 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 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

Algorithm 1 Sidetrack Based (SB) algorithm for the kSSP [23]

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 $Candidate \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
$T_0 \leftarrow$ an SP in-branching of D rooted at t containing s
Add $((T_0), w(P_{st}(T_0)), \zeta = 1)$ to $Candidate$
while $Candidate \neq \emptyset$ and $|Output| < k$ **do**
    $\mu = (\varepsilon = ((T_0, e_0, \cdots, T_h, e_h = (u_h, v_h), T_{h+1}), lb, \zeta) \leftarrow$ a shortest element in $Candidate$
    if $\zeta = 1$ **then**
        Extract μ from $Candidate$ and add ε to $Output$
        for every deviation $e = v_j v'$ with $v_j \in P_{v_h t}^{T_{h+1}}$ **do**
            $\varepsilon' \leftarrow (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1})$
            $lb' \leftarrow lb - w(P_{v_j t}^{T_{h+1}}) + w(e) + w(P_{v' t}^{T_{h+1}})$
            if ε' represents a simple path **then**
                Add $\mu' = (\varepsilon', lb', \zeta = 1)$ to $Candidate$
            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 $\mu'' = (\varepsilon'' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T'), lb', \zeta = 0)$ to $Candidate$
    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 $\mathcal{T}$
        Let $\mu' = (\varepsilon' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb + w(P_{v' t}^{T_{h+1}}) - w(P_{v' t}^{T_h}), \zeta = 1)$
        Add μ' to $Candidate$
return $Output$

u_h in T_h, and updates the SP in-branching T using standard methods for updating a shortest path tree [13]. 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 6.2 show that this update procedure gives an average speed up by a factor of 1.5 to 2 on road networks.

4 Space - time tradeoffs

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 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 considering a step of 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, PSB algorithm adds $P = (s = v_0, \cdots, v_i = w_h, \cdots, v_r = t)$ to *Output* and (as 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, 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 set *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 set 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 SB and PSB algorithms comes from the fact that non simple detours are considered as a unique object by PSB algorithm.

Now, let us consider a step when 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, PSB algorithm proceeds as follows. For j decreasing from l to min, an in-branching T'_j in $D \backslash \{v_0, \cdots, v_{i_j} = u_j\}$ 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 (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 [13]. 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 the one of the SB algorithm. Moreover, since most of the computed in-branchings are not stored, the working memory used by PSB is significantly smaller than for SB algorithm.

4.2 Special variants of PSB

A better space and time trade-off than PSB 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 length relatively short, say less than a threshold value time the current outputted path, then storing such in-branching is meaningful as the extraction of its corresponding element form *Candidate* is expected soon and saving it leads to save its redundant computation. Here, we present two variants of the PSB; PSB-v2 and PSB-v3. PSB-v2 is a tiny improvement of PSB leading to consume less memory by storing less in-branching while PSB-v3 gives an adaptable trade off depending on the value of the threshold.

Let us consider, again, a step when 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 PSB-v2 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.

PSB-v3 behaves on every deviation in Dev between min and l the same way PSB-v2 behaves with f_{min}. Precisely, for each deviation f_j with $min \leq j \leq l$, PSB-v3 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 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's computation

In this section, we present the Postponed Node Classification (PNC) algorithm. The PNC algorithm have a $O(kn(m + n \log n))$ time complexity with a similar working memory as 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 a similar idea as 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_0 rooted at t that will be used throughout the execution of the algorithm. Then, as 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 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 the 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 $\delta(u, v) = w(u, v) + w(P^{T_0}_{vt}) - w(P^{T_0}_{ut})$ denotes the residual weight of arc uv as defined in Section 2.3). 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. *$w(P_{LB}) \leq w(P')$ for any new simple detour P' of P at u_i*

Similarly to the SB algorithm (and in contrast to the NC algorithm), PNC algorithm may add non-simple paths to the set *Candidate*. Precisely, each element in *Candidate* has the form $(P = (s = u_0, \cdots, u_r = t), w(P), i, \zeta)$ 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_i for all $i \leq j < r$, the following procedure is used. For each vertex $u_i \in P$, the SP in-branching T_0 is colored (`yellow`, `red`, `green`, as in the NC algorithm) with respect to P and u_i. If the color of v_{LB} is `green`, it implies that the path P_{LB} is simple, and a shortest simple detour is found (by the remark above). In this case, $(P_{LB}, w(P_{LB}), i, \zeta = 1)$ is added to the set *Candidate*. Otherwise, i.e., if v_{LB} is `yellow`, the detour P_{LB} is added to the set *Candidate* (even though it is not simple) with its weight $w(P_{LB})$ as a key, i.e, the element $(P_{LB}, w(P_{LB}), i, \zeta = 0)$ 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* and so a call to an SP algorithm is saved.

When an element $(P, w(P), i, \zeta)$ 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. If not, i.e., P is not simple, it will be "repaired" into 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.

Algorithm 2 Postponed Node Classification (PNC)

Input A digraph $D = (V, A)$, source $s \in V$, sink $t \in V$ and an integer k
Output k shortest simple s-t paths
Let $Candidate \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
$T \leftarrow$ an SP in-branching of D rooted at t
Add $(P_{st}(T), w(P_{st}(T)), 0, 1)$ to *Candidate*
while $Candidate \neq \emptyset$ and $|Output| < k$ **do**
    $(P = (s, u_1, \cdots, t), w(P), i, \zeta) \leftarrow$ extract a shortest element from *Candidate*
    Color the vertices `yellow`, `red`, `green` with respect to P and u_i
    $\pi \leftarrow (s, u_1, \cdots, u_{i-1})$
    $Dev_{old} = \{e = (u_i, v)$ s.t there is a path in *Output* having $\pi.e$ as prefix$\}$
    if $\zeta = 1$ (P is simple) **then**
        add P to *Output*
        for each vertex u_j in $(u_i, \cdots, t)$ **do**
            $(u_j, v_{LB}) \leftarrow$ an arc in $A \setminus Dev_{old}$ with minimum δ among those tailing at u_j
            $P_{LB} \leftarrow (s, \cdots, u_j, v_{LB}, P^T_{v_{LB}t})$
            $\zeta' \leftarrow 0$
            if v_{LB} is `green` **then**
                $\zeta' \leftarrow 1$
            add $(P_{LB}, w(P_{LB}), j, \zeta')$ to *Candidate*
    else
        Compute a shortest u_i-t path Q in $D' = (V \setminus \pi, A \setminus Dev_{old})$
        if Q exists **then**
            Add $(P' = \pi.Q, w(P'), i, 1)$ to *Candidate*
return *Output*

6 Experimental evaluation

In this section we describe our experimental evaluation. First, we start by describing our implementation and settings (Section 6.1), then we discuss our experimental results on road and complex networks (Section 6.2). Finally, Section 6.3 contains an experimental study to some parameters related to the queries.

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 [11], PNC, SB [23], SB* and PSB) in C++ and our code is publicly available [1]. Following [23], we have implemented a pairing heap data structure [12] supporting the decrease key operation, and we use it for Dijkstra's shortest path algorithm. Our implementation of the Dijkstra shortest path tree 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 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 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 in SB*.

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 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. Moreover, for the SB, SB*, PSB, PSB-v2 and PSB-v3 algorithms, following [23], we store the candidates into two heaps, the first one to store the simple candidates ($Candidate_{simple}$) and the second one to store the non-simple candidates ($Candidate_{not-simple}$). Then, we extract candidates from $Candidate_{simple}$ as long as the length of the shortest simple path is smaller or equal to the length of the shortest non-simple path in $Candidate_{not-simple}$. This way, we prioritize the extraction of simple paths.

Concerning the PSB-v2 and PSB-v3 algorithms, based on preliminary experiments, we choose to update the value of θ dynamically with respect to the ratio between the pair of the upcoming paths. Recall that when looking for the i^{th} path, a corresponding in-branching will be stored only if the length of that path is at most θ times the length of the $(i-1)^{th}$ path.

Precisely, let ℓ_s be the length of the smallest element in $Candidate_{simple}$ and let ℓ_{ns} be the length of the smallest non simple element in $Candidate_{not-simple}$. These values are both set to 1 if any of the corresponding sets is empty. Let also $c = \max(\frac{\ell_s}{\ell_{ns}}, \frac{\ell_{ns}}{\ell_s})$ and $\rho = c - 1$. The value of θ is set to $1 + \alpha\rho$, 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. The choice of these values (ℓ_s and ℓ_{ns}) gives us a meaningful indication and the value of θ is easy to compute. Observe that in our experiments we have set the factor α in the formula for computing θ to $\alpha = 11$, based on preliminary experiments.

Besides, a special implementation of the PNC algorithm is proposed and referred to as PNC*. As shown in Algorithm 2, the PNC algorithm computes a shortest simple detour (line 21) with an SP algorithm call that may visit the whole sub-digraph. The PNC* algorithm tries to reduce the size of this sub-digraph in order to speed up these calls. For this purpose, it proceeds the

[3]Despite several queries, we have not been granted access to the code used for experiments in [11, 23].

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 [7]
DC	9 559	29 682	140	6.2	-	0.039	Road network of Washington DC [7]
DE	49 109	119 520	573	4.8	-	0.024	Road network of Delware [7]
NY	264 346	733 846	664	5.5	-	0.02	Road network of New York [7]
BAY	321 270	800 172	791	4.9	-	0.016	Road network of San Francisco Bay area [7]
COL	435 666	1 057 066	1219	4.8	-	0.017	Road network of Colorado area [7]
BIOGRID	2 318	12 580	7	21.7	1.96	0.20	Mutation/deletion of genes resulting in cell lethality [27]
FB	3 698	85 963	6	93	-	0.61	Social circles from Facebook [26]
P2P	5 606	23 510	8	16.7	-	0.014	Peer-to-peer network of the Gnutella file sharing network [25]
DIP	13 969	60 621	17	17.4	2.38	0.11	Protein-protein interaction network [29]
CAIDA	29 432	143 000	9	19.4	2.06	0.42	Relationships between Autonomous Systems of the Internet [32]
LOC	33 187	188 577	11	22.7	2.25	0.29	Brightkite location-based social networking service provider [25]

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 ($\langle cc \rangle$).

same way as the NC algorithm. That is, it gives a color to each node (vertex) with respect to its end in the first pre-computed in-branching and call an SP algorithm visiting only the `yellow` vertices.

Networks setting We have evaluated the performances of our algorithms on some road networks from the 9th DIMACS implementation challenge [7] and on several complex networks.

A road network of a city is the digraph modelizing its roads, i.e, a vertex is associated to each crossroad, and there is an arc of length 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 [34]. We denote by small road networks the road network of ROME, DC and DE while big road networks denote NY, BAY and COL. The characteristics of these graphs are reported in Table 1.

On the other side, complex networks modelize different types of networks. Generally, they are characterized by being small-world, i.e, they have a logarithmic diameter, by a power-law degree distributions and a high clustering coefficient [5]. For instance, BIOGRID system synthetic lethality represents mutation/deletion of genes resulting in lethality when combined in a same cell [27]. DIP represents protein to protein interactions [29]. The FB network represents social circles from Facebook [26]. Likewise, LOC is a graph provided from Brightkite location-based social networking [25]. Finally, P2P is the peer-to-peer network of the Gnutella file sharing network [25] and CAIDA (2013.11.01) is the graph of the relationships between the autonomous system of the Internet [32]. As these networks are unweighted, we only consider the number of hops as the length of a path. We consider for each network its largest biconnected component. The characteristics of these graphs are depicted in Table 1

In our experiments, we have randomly chosen 1000 queries (source-destination pairs of vertices) for each network, and we have run each algorithm for each of these pairs for $k = 1,000$ on road networks and $k = 10,000$ on complex networks. 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 [20].

We also attempt in Section 6.2 to explain the performance of the algorithms with respect to the structure of the network and / or some properties of the queries (e.g., the hop distance between the source and the destination, their stretch from the "center" of the graph...).

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

6.2 Experimental results

In this section, we first describe and analyze our experimental results on road networks and then on complex networks. We will see that the behavior of the algorithms' differ from one type of network to the other.

First, we have measured the average and the median of the algorithms' running time in all considered networks. The results on road networks are reported in Table 2 and the ones on complex networks are in Table 4. The data in Tables 2 to 4 corresponds to the biggest experienced value of k ($k = 1,000$ for road networks and $10,000$ for complex network).

Then, we have measured the number of stored trees for all networks. As we will discuss below, in the case of complex networks, there are very few differences in the number of in-branchings. The results on road networks are described in Table 3. Moreover, in Figures 3 and 4, we report the evolution of the average and median running times of the algorithms when the number k of reported paths increases. For the latter figures, the results we obtained differ depending on the class (road or complex) of the networks but among a same class, they do not differ much depending on the considered networks, so we only report them in the case of the networks COL, DC (see Figures 1 and 2), BIOGRID and CAIDA (see Figure 4).

We have then performed a refined comparison of the algorithms on the road networks. The results we obtained does not differ much depending on the considered networks, so we only report the comparison of the algorithms on the DC and COL networks in Figures 1 and 2. More precisely, we have plot pairwise comparisons of the running times and number of stored trees for each source-destination pairs on DC and COL network.

Finally, some statistics about the queries' properties and their impact on the algorithms' performances are depicted in Figures 5 and 6.

Road Networks. Note that the algorithms can be classified as three sets: Algorithms with low memory consumption i.e, the ones storing no more than a single in-branching, that are Yen's, NC, PNC and PNC*. Algorithms storing big number of in-branchings (SB and SB*). And algorithms establishing a space - time tradeoff (PSB and its two variants). We first compare the experimental results of the first class together, then we do the same with the second class. We finally compare the PSB algorithm with the others and we give a conclusion on our experiments on road networks.

- We first observe, based on Table 2 and Figures 3a and 3b , 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 [11, 23] leads to the same conclusion on big road networks.

  Let us now compare NC, PNC and PNC* algorithms together, as they all store no more than a single in-branching in the memory. The average and median running times reported in Table 2 show that the PNC algorithm is significantly faster than the NC algorithm for all road networks (up to 5 times faster on average). Moreover, a refined comparison of the NC and PNC algorithms on DC and COL networks (Figures 1b and 2b) show that this is true for all queries.

  Surprisingly, PNC* algorithm is slower than PNC (based on Table 2, Figures 1d and 2d) even though its was expected to be faster (see Section 6.1). This is due to the extra time

consumed during the coloring procedure that is computed from scratch each time (unlike to PNC where no such operation is needed).

- The simulation results reported in Table 2 and Figure 3 confirm that the use of shortest path tree update procedures in SB* helps to significantly reduce the running time compared to SB. More precisely, the average and median running times of the SB* algorithm are significantly smaller than for the SB algorithm on all road networks (SB* is up to twice faster than SB). A more refined comparison on DC and COL network (Figures 1a and 2a) show that SB* is faster than SB for almost all queries (more than 90% of the queries). The same behavior was observed on all road networks. Finally, we recall that by design, the number of stored in-branchings is the same in both algorithms.

- Concerning the PSB algorithm, as shown in Table 2 and Figures 1e, 1f, 2e and 2f, PSB gives a space time tradeoff between SB and NC i.e, it is faster than NC and consumes less memory than SB. However, PSB algorithm is, sometimes, beaten by the PNC algorithm as it is faster while consuming less memory (only on in-branching is stored). As a result, PSB algorithm could give a space time tradeoff only on DC and BAY. Note that, the special two variants of PSB (PSB-v2 and PSB-v3) 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 30% memory reduction), see Tables 2 and 3.

  An unexpected observation (in Table 2) is the gap between the average and the median running time of the SB like algorithms (SB, SB*, PSB, PSB-v2 and PSB-v3). That is, the median could be up to 5 times smaller than the average while it is not the case with the remaining algorithm. We discuss this observation further in Section 6.3.

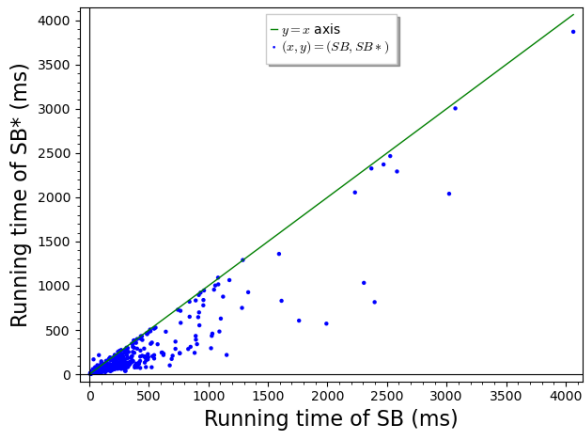
  Among the considered algorithms, none of them clearly outperforms all the others on road network. As shown in Table 2 and Figure 3, on most of the road networks (Rome, DE, BAY and COL), PNC algorithm is the fastest on average (slightly faster than SB*). However, SB* is, by far, the fastest on median on DC, NY, BAY and COL. This is justifiable by the big difference between the average and the median running time of SB* algorithm and by the fact that, on a small number of queries, SB* algorithm is extremely slow, as shown in Figures 1c and 2c where a small number of points are very far from the majority of the other points. Note that the results may change with respect to the value of k (Figure 3).

To conclude, among the considered algorithms, PNC and SB* algorithms are the fastest on road networks. SB* has a better average running time while PNC algorithm has a running time that is more stable.

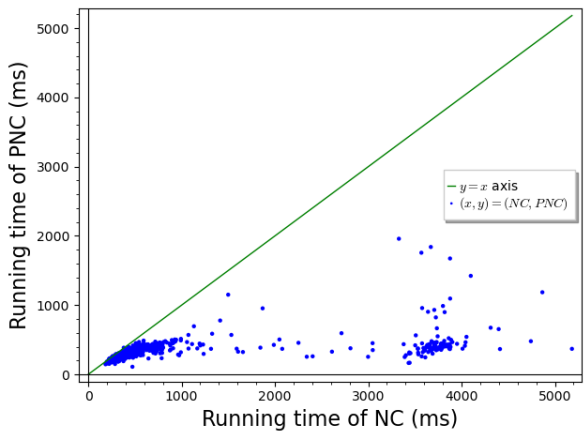
Complex Networks. Here we analyse and we try to explain the behavior of the algorithms on complex networks, except for Yen's algorithm because of its excessive running time (as already mentioned above).

On complex networks, the PSB algorithm is the fastest kSSP algorithm among the considered algorithms (Table 4 and Figure 4). Considering the space consumption, all of the kSSP algorithm have a tiny memory consumption (for $k = 10,000$, the number of stored in-branchings does not exceed 50). It is also shown in Table 4 that the running time of PSB and its two variants (PSB-$v1$ and PSB-$v2$) is similar.

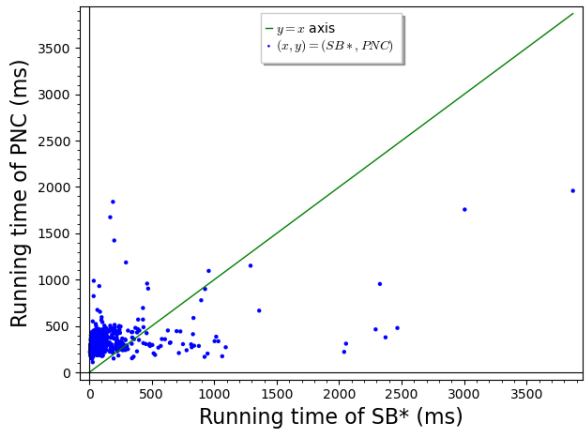
In what follows, we give some qualitative arguments that may explain the fact that the PSB algorithm is the fastest among all considered algorithms on complex networks.



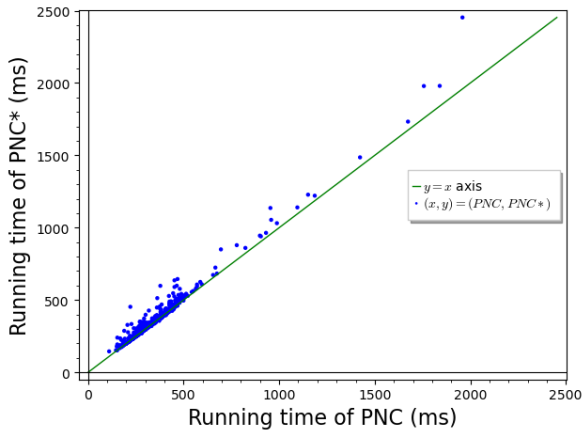
(a) Running time of SB and SB*



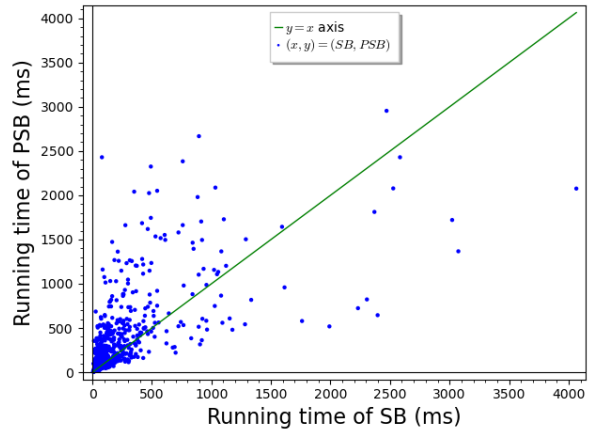
(b) Running time of NC and PNC



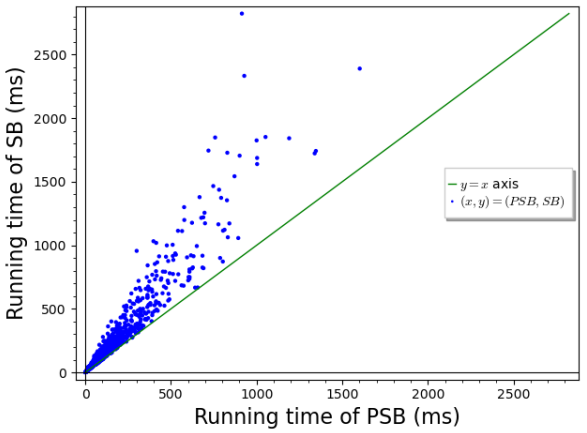
(c) Running time of SB* and PNC



(d) Running time of PNC and PNC*

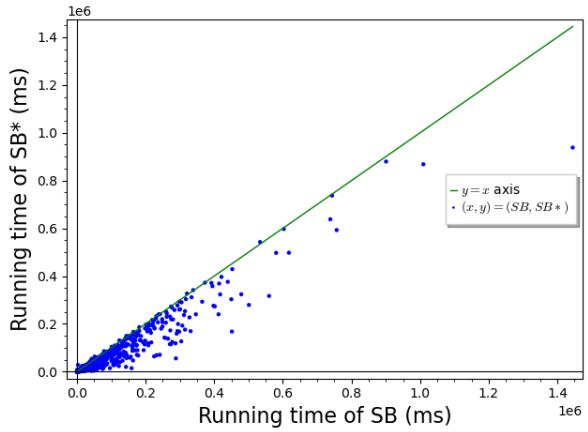


(e) Running time of SB and PSB

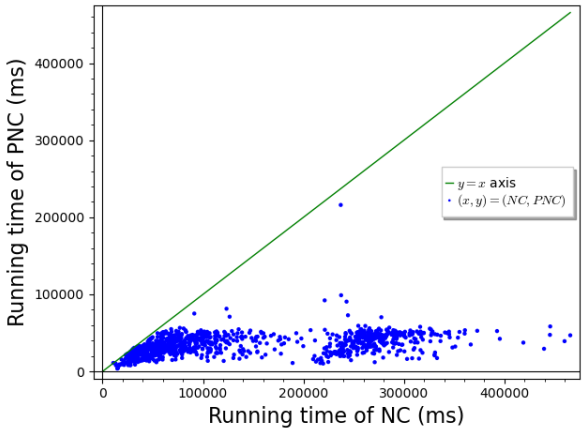


(f) Number of trees of SB and PSB

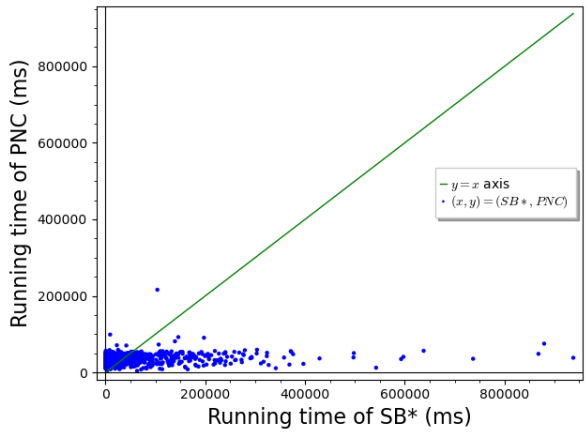
Figure 1: Comparison of the running time and the number of stores trees on DC. Each dot corresponds to one pair source/destination ($k = 1,000$).



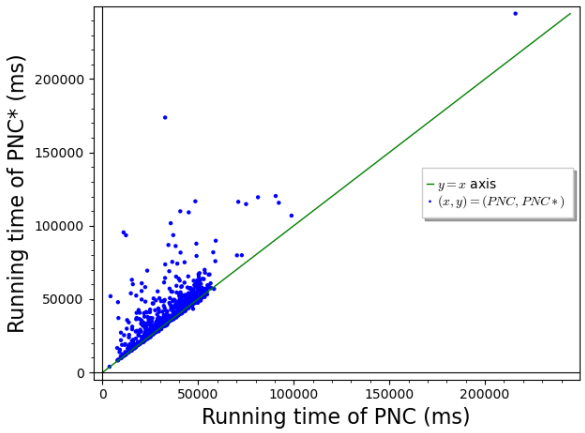
(a) Running time of SB and SB*



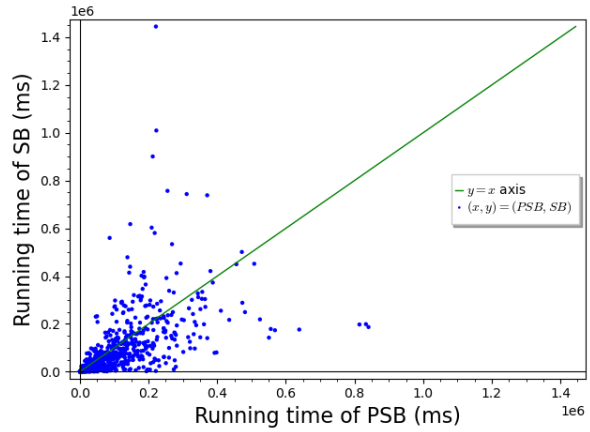
(b) Running time of NC and PNC



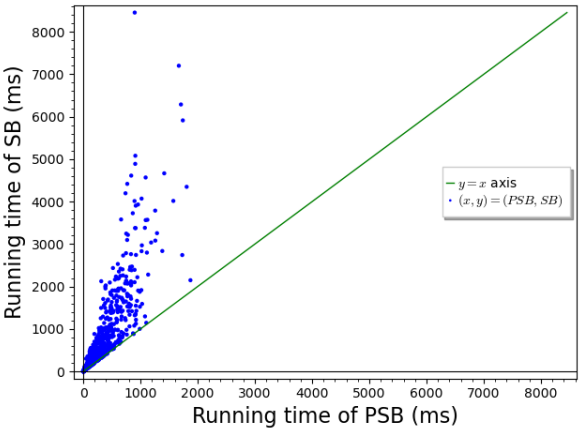
(c) Running time of SB* and PNC



(d) Running time of PNC and PNC*



(e) Running time of PSB and SB



(f) Number of trees of SB and PSB

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

		Rome	DC	DE	NY	BAY	COL
Yen	avg	3389	11316	159785	-	-	-
	med	1439	4945	59463	-	-	-
NC	avg	407	823	10620	99521	94149	146025
	med	178	404	6129	64465	56136	99540
PNC	avg	**181**	326	**1972**	41923	**24970**	**35265**
	med	**155**	299	**1644**	39389	24064	35039
PNC*	avg	203	336	2434	69913	28481	40045
	med	173	305	1997	58446	26552	38952
SB	avg	451	184	8469	53423	38390	68077
	med	356	75	4321	30300	8783	20535
SB*	avg	282	**117**	5428	**33704**	28693	49859
	med	199	**43**	2139	**18659**	**4977**	**11060**
PSB	avg	447	269	7939	106040	53286	81321
	med	340	117	6148	81927	23574	40640
PSB-v2	avg	447	265	7513	100377	49683	76766
	med	347	117	5849	75914	21812	38732
PSB-v3	avg	446	265	7471	100390	49653	77185
	med	346	115	5785	75681	21770	38709

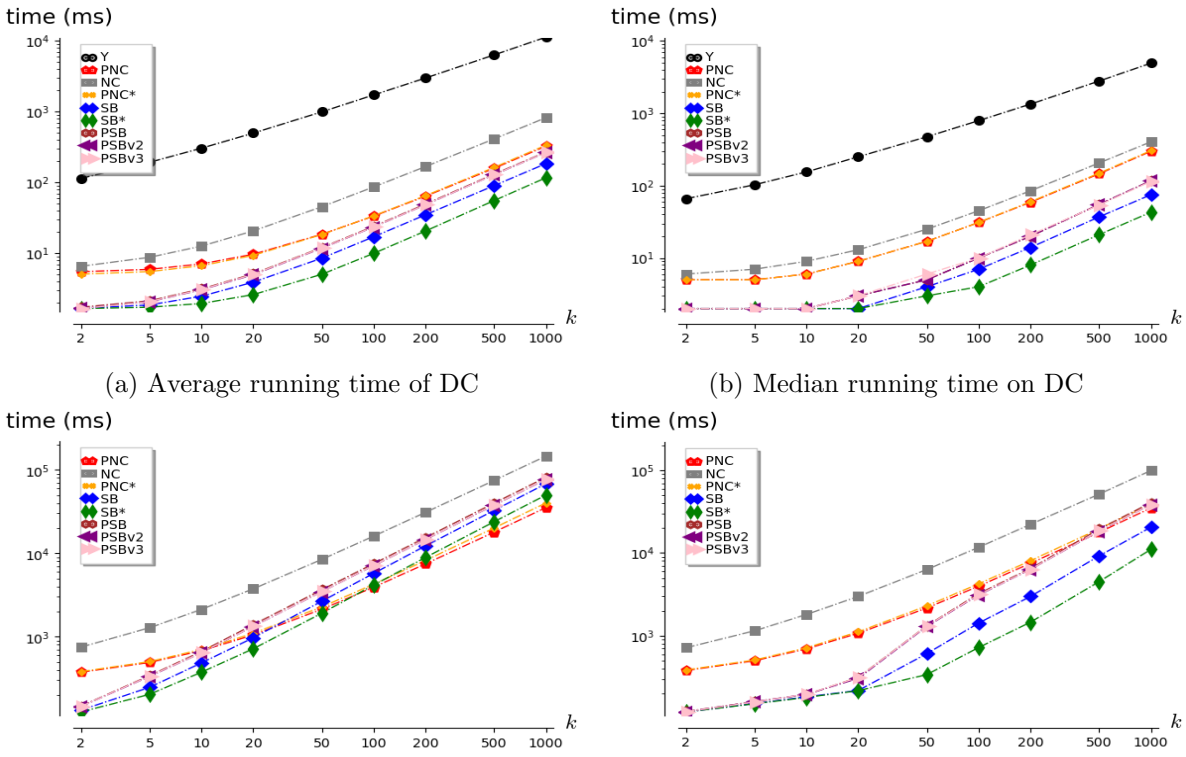
Table 2: Running time (ms) of the algorithms on road networks, ($k = 1,000$)

	Rome	DC	DE	NY	BAY	COL
NC, PNC and PNC*	1	1	1	1	1	1
SB and SB*	1135	243	948	1669	541	585
PSB	694	160	335	536	246	249
PSB-v2	**580**	**138**	**274**	**373**	**194**	**208**
PSB-v3	588	147	290	380	200	212

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

		BIOGRID	FB	P2P	DIP	CAIDA	LOC
NC	avg	1905	1493	3247	8325	25367	26659
	med	1458	1442	3014	7590	18583	23846
PNC	avg	1336	1523	2475	6361	22459	21474
	med	1303	1492	2386	6354	21824	21065
PNC*	avg	1302	1478	2479	6148	21869	18910
	med	1281	1449	2396	6087	21122	18332
SB	avg	993	2986	596	870	8637	2524
	med	821	2472	575	771	6724	2200
SB*	avg	980	3070	481	802	8506	2508
	med	810	2507	476	725	6661	2173
PSB	avg	431	**1062**	293	456	3644	1237
	med	394	**971**	282	422	3666	1190
PSB-v2	avg	**421**	1082	294	439	**3541**	**1163**
	med	**387**	988	282	410	**3574**	**1122**
PSB-v3	avg	431	1081	**292**	**437**	3654	1210
	med	399	991	**280**	**410**	3664	1167

Table 4: Running time (ms) of the algorithms on Complex networks, ($k = 10,000$)



(a) Average running time of DC

(b) Median running time on DC

(c) Average running time of COL

(d) Median running time on COL

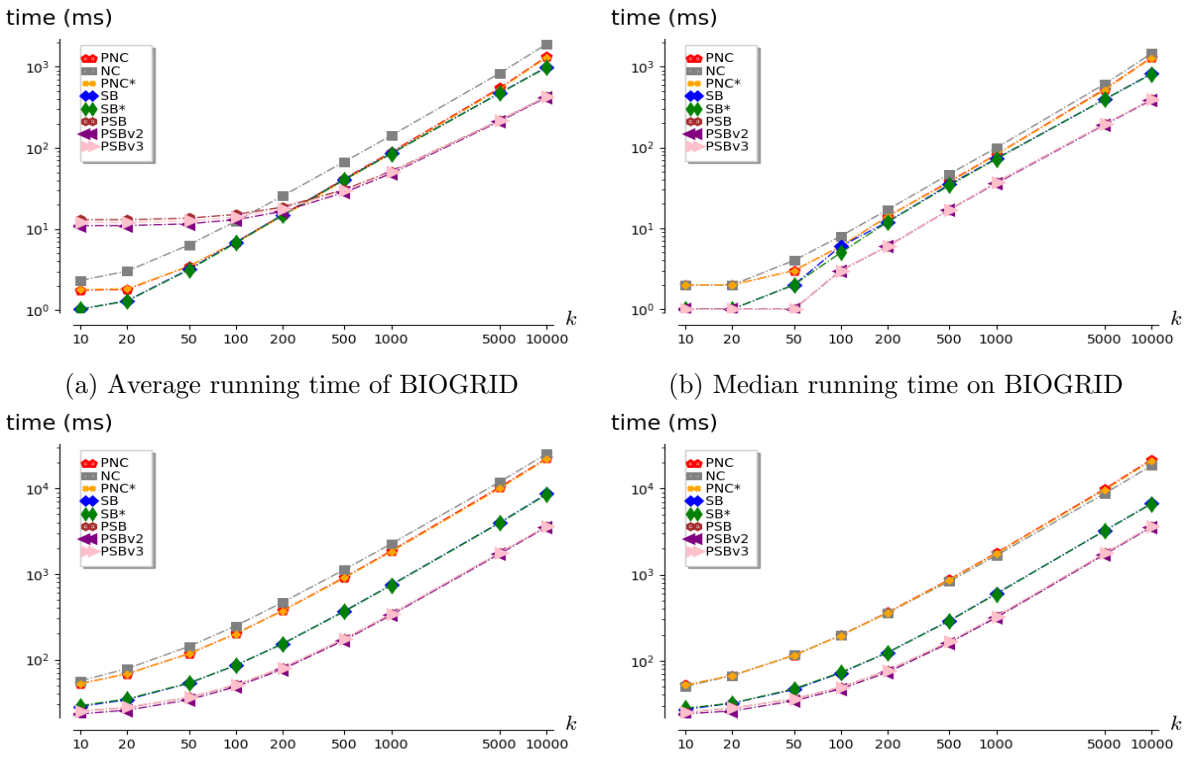
Figure 3: The running time of the kSSP algorithms on road network with respect to the values of k

Suppose P is a shortest 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, Yen's, NC, PY 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, SB* and PSB algorithms compute at most one shortest path in-branching T at v, that works for each neighbor v' of v. In another word, SB* and PSB 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, $P^T_{v't} \cap (s, \cdots, v) = \emptyset$. So, the number of shortest path in-branchings computed and/or stored using SB* and PSB algorithms is expected to be small.

In addition, as the number of hops of a shortest path is "small" (remember that complex network are small-worlds). The number of calls of the shortest path in-branching update is expected to be small for PSB. As this procedure is faster in PSB than SB* and the number of calls is similar, PSB algorithm is faster than SB* 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 feasible working memory and this seems to be related to structural properties of complex networks.



(a) Average running time of BIOGRID

(b) Median running time on BIOGRID

(c) Average running time of CAIDA

(d) Median running time on CAIDA

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

6.3 Impact of the properties of the queries

Here we study the impact of other parameters (related to the properties of the queries) on the running time of the algorithms. In particular, the number of hops of a shortest path and the stretch of a shortest path from the "center" (defined below).

As noticed in Section 6.2, some algorithms have different behaviors with respect to the structure of the network and the query's properties. In this section, we are investigating whether some properties of an s-t query may explain the variations of the running time of the algorithms. For this purpose, we have considered two criteria of each s-t query, the number of hops between s and t of a shortest s-t path and the stretch of a shortest s-t path from the "center" of the graph.

A similar indicator to the number of hops of a shortest path is the maximum number of hops, that is the number of hops of a path with maximum number of hops among the k shortest paths given by an algorithm. Formally, the maximum number of hops of a kSSP query from s to t using an algorithm A is M if and only if for each path P given by A, $|P| \leq M$. We studied this parameter and the obtained results are almost the same as those obtained while studying the number of hops. Therefore, we only describe the results corresponding to the number of hops.

The number of hops of the shortest path (the one given by an SP algorithm) is a meaningful criterion to be studied, as each of the algorithms has the common routine of iterating and eventually calling an SP algorithm for each vertex on the first shortest path. As the obtained results are similar on different road networks, we only add and discuss the results of NC, PNC, SB* and PSB algorithm on NY road network.

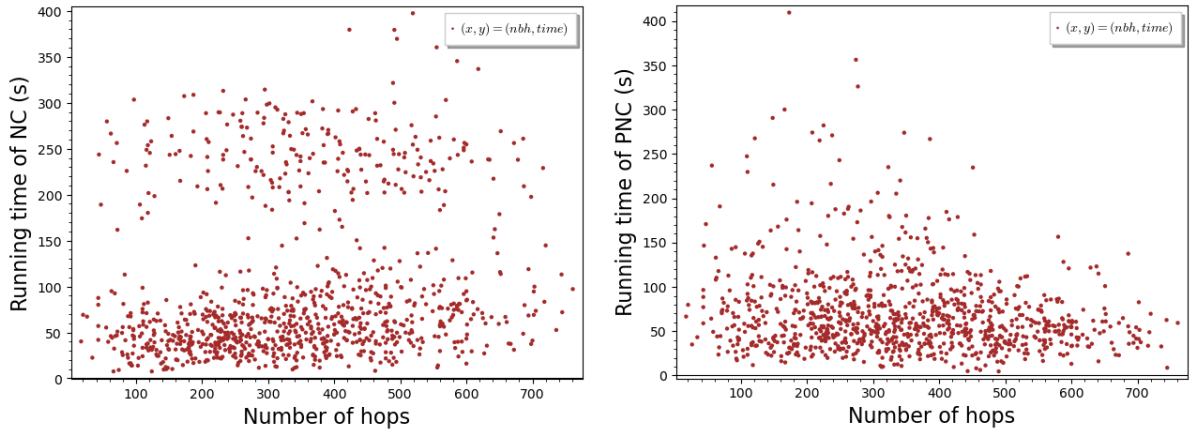
As shown in Figure 5, no clear pattern can lead to a establish a concrete relationship between the running time for one of these algorithms and the number of hops. For instance, the SB* algorithms Figure 5c has big running time for queries with relatively small number of hops. However, this is not flagrant and it does not hold on the remaining algorithms (Figures 5a, 5b and 5d).

Another interesting parameter to be studied in road networks is the stretch from the center (by center we mean a vertex of minimum eccentricity in the network). So, the stretch of an s-t path P from the center c is defined as the ratio between the length of a shortest s-t path passing through c and the length of P. This gives an indicator on how far a path can be from the center. In order to establish a relation between the running time and the stretch of the center, we plotted (Figure 6) the running time with respect to the stretch value. Clearly, no flagrant pattern related to the stretch value is found. Then, no concrete relation between the running time of these algorithms and the stretch from the center can be established based on our experiments.

To conclude, no clear relationship between the running time of an algorithm and the number of hops, neither the stretch from the center is experimentally found. However, it seems that not all queries perform the same. For instance, in Figures 1b, 2b, 5a and 6a, it can be observed two distinct clouds of points. It would be interesting to understand whether these two clouds correspond to some specific properties of the queries. This could help us to design improvements of our algorithms.

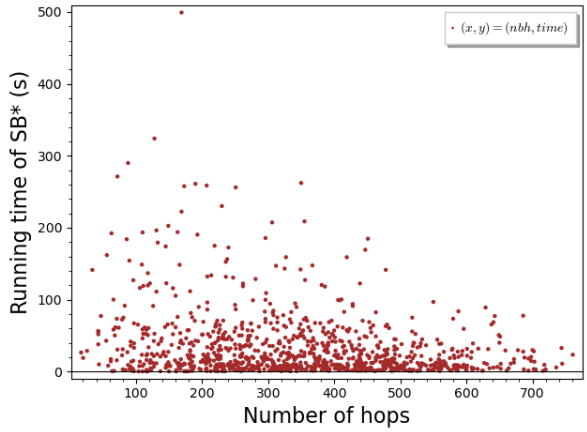
7 Conclusion

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 working memory consumption of the algorithms.

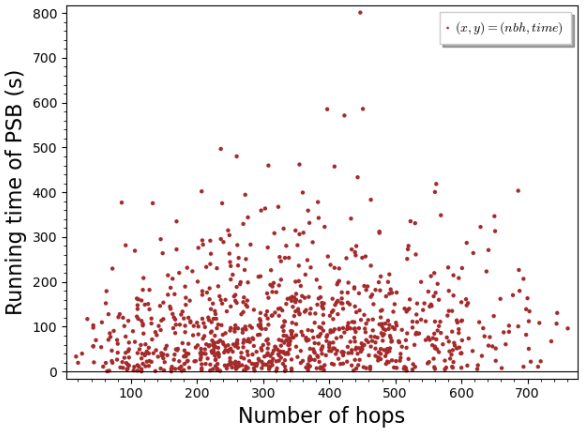


(a) Running time of NC

(b) Running time of PNC

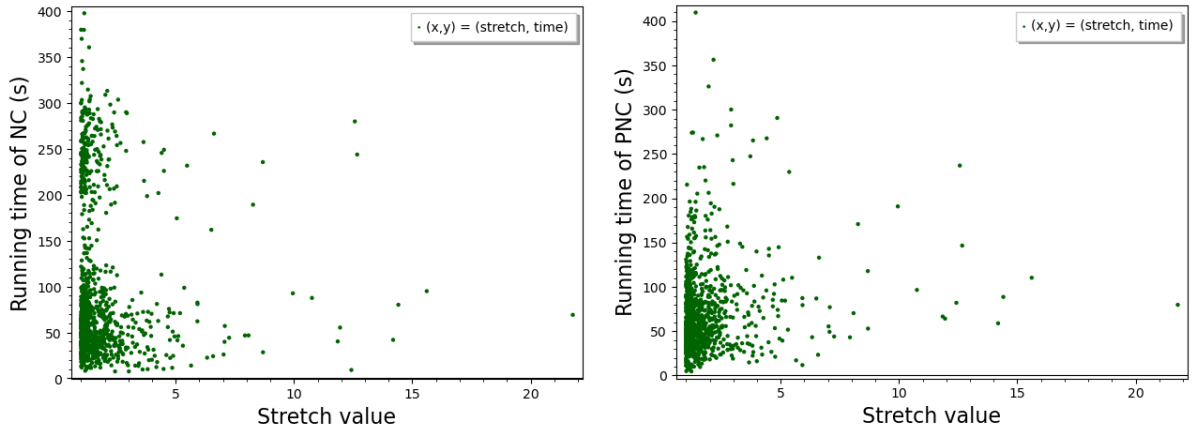


(c) Running time of SB*



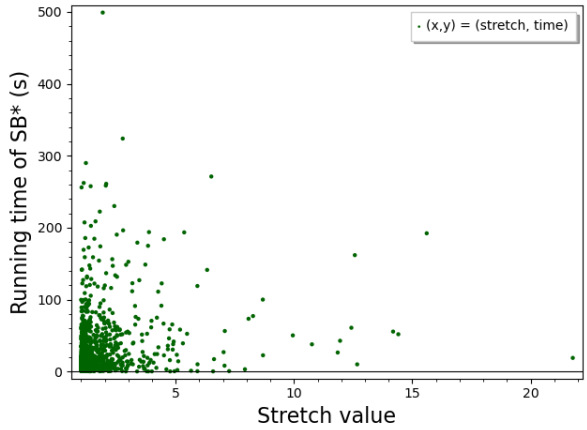
(d) Running time of PSB

Figure 5: The running time with respect to the number of hops of the shortest path of some kSSP algorithms on NY. Each dot corresponds to one pair source/destination ($k = 1,000$).

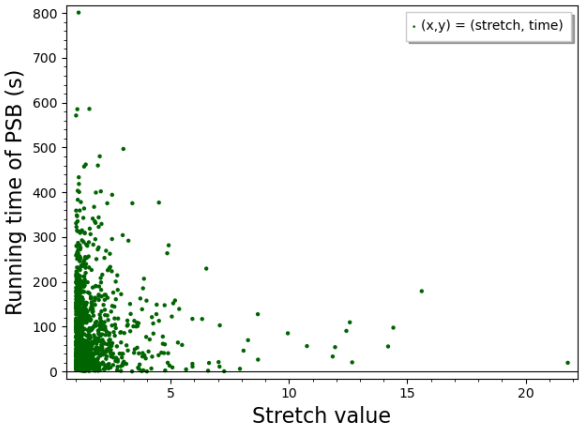


(a) Running time of NC

(b) Running time of PNC



(c) Running time of SB*



(d) Running time of PSB

Figure 6: The running time with respect to the stretch from the center of some kSSP algorithms on NY. Each dot corresponds to one pair source/destination ($k = 1,000$).

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, if large memory consumption is allowed, the SB* algorithm is the fastest among the considered algorithms on most of the queries. However, the PNC algorithm has a running time that is more stable and it has low working memory. Therefore, the PNC algorithm seems to offer a better space time trade-off than the other considered algorithms on road networks.

An empirical framework for the selection of the most appropriate kSSP algorithm with respect to the use case is suggested in Figure 7.

An open problem is how to handle the kSSP problem on networks with arbitrarily arc weights (including negative weights). Another interesting question is how to design a data structure enabling to quickly answer kSSP queries similarly to the data structures used by the hub labelling and contraction hierarchy schemes to answer distance queries [3]. A probably more difficult question would be to address dynamic networks, i.e., where 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?

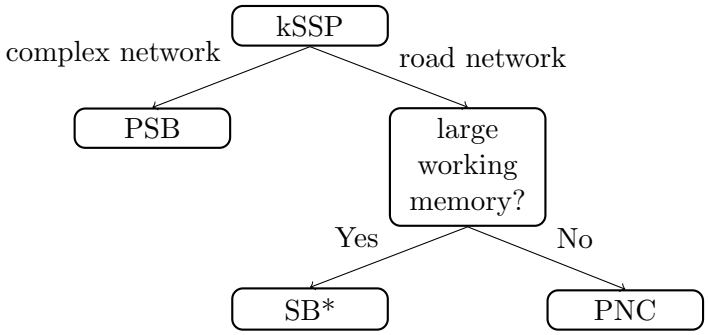


Figure 7: A framework of the appropriate kSSP algorithm with respect to the use case

References

[1] A. Al Zoobi, D. Coudert, and N. Nisse. *k shortest simple paths (Version 2.0)*, 2021. `https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths`.

[2] M. Arita. Metabolic reconstruction using shortest paths. *Simulation Practice and Theory*, 8(1-2):109–125, 2000.

[3] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer, 2016.

[4] 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.

[5] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006.

[6] 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.

[7] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS implementation challenge - shortest paths, 2006.

[8] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[9] D. Eppstein. *Encyclopedia of Algorithms*, chapter k-Best Enumeration, pages 1003–1006. Springer New York, 2016.

[10] D. Eppstein and D. Kurz. K-best solutions of MSO problems on tree-decomposable graphs. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, volume 89, pages 16:1–16:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[11] G. Feng. Finding k shortest simple paths in directed graphs: A node classification algorithm. *Networks*, 64(1):6–17, 2014.

[12] 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.

[13] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[14] J. Gao, H. Qiu, X. Jiang, T. Wang, and D. Yang. Fast top-k simple shortest paths discovery in graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 509–518, 2010.

[15] Z. Gotthilf and M. Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Information Processing Letters*, 109(7):352–355, 2009.

[16] E. Hadjiconstantinou and N. Christofides. An efficient implementation of an algorithm for finding k shortest simple paths. *Networks*, 34(2):88–101, 1999.

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

[18] 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.

[19] 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.

[20] D. 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.

[21] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982.

[22] D. Kurz. *k-best enumeration - theory and application.* Theses, Technischen Universität Dortmund, Mar. 2018.

[23] 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.

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

[25] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–42, 2007.

[26] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[27] R. Oughtred, C. Stark, B.-J. Breitkreutz, J. Rust, L. Boucher, C. Chang, N. Kolas, L. O'Donnell, G. Leung, R. McAdam, et al. The biogrid interaction database: 2019 update. *Nucleic acids research*, 47(D1):D529–D541, 2019.

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

[29] L. Salwinski, C. S. Miller, A. J. Smith, F. K. Pettit, J. U. Bowie, and D. Eisenberg. The database of interacting proteins: 2004 update. *Nucleic acids research*, 32(suppl_1):D449–D451, 2004.

[30] T. Shibuya and H. Imai. New flexible approaches for multiple sequence alignment. *Journal of Computational Biology*, 4(3):385–413, 1997.

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

[32] The Cooperative Association for Internet Data Analysis (CAIDA). The CAIDA AS relationships dataset. `http://www.caida.org/data/active/as-relationships/`, 2013.

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

[34] F. Xie and D. Levinson. Measuring the structure of road networks. *Geographical analysis*, 39(3):336–356, 2007.

[35] 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.

[36] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.