



# A Scalable Algorithm for Single-Source Top- $k$ Loopless Shortest Paths

Mattia D’Emidio

Dept. of Information Engineering,  
Computer Science, and Mathematics  
University of L’Aquila  
L’Aquila, Italy  
mattia.demidio@univaq.it 

Gabriele Di Stefano

Dept. of Information Engineering,  
Computer Science, and Mathematics  
University of L’Aquila  
L’Aquila, Italy  
gabriele.distefano@univaq.it 

**Abstract**—In this paper, we study the problem of computing *top- $k$  loopless (a.k.a. simple) shortest paths* from a source vertex in weighted digraphs. While the problem of determining top- $k$  simple shortest paths for a single pair of vertices of a graph has been widely investigated in the last decades, with Yen’s algorithm and some of its heuristic improvements emerging as the best performing solutions, surprisingly little or no effort has been devoted to designing efficient algorithms to handle the more general version of the problem where all top- $k$  simple shortest paths from a distinguished source vertex to all other vertices of a digraph have to be discovered. Motivated by the plethora of applications ranked shortest paths finds in practice, we introduce the first algorithm natively designed to address such more general version of the problem. On the one hand, we show our solution runs in polynomial time and is on par, in terms of time complexity, with the best known alternative in the literature to attack the considered problem, which consists in applying Yen’s algorithm with the source paired to all other vertices as inputs. On the other hand, we provide an extensive experimental evaluation that considers both real and artificially generated inputs and shows our algorithm outperforms the state-of-the-art by up to orders of magnitude, in terms of running time.

**Index Terms**—Top- $k$  Loopless Shortest Paths, Graph Algorithms, Algorithm Engineering, Experimental Algorithms.

## I. INTRODUCTION

Graph-based modeling of data has become ubiquitous and, thanks to their effectiveness in capturing the networked nature of information, a standard choice for representation and manipulation of information itself in a large variety of application domains of computer science and engineering, such as digital social networks, communication infrastructures, scientific computing, protein-protein interaction analysis, road navigation, databases and transportation systems [1], [2].

Computing shortest paths is considered one of the most fundamental primitive to be performed on graph datasets, since they can be exploited for a variety of purposes, including routing protocols, trip planning, scheduling, and network design [3]–[5]. In addition, shortest paths provide a unifying framework for many combinatorial and constrained optimization problems, e.g. knapsack, packing, sequence alignment, polygon construction, clustering, length-limited Huffman-coding [1], [3], [6], [7]. The  $k$  shortest paths (or top- $k$  shortest paths) problem is a natural and long-studied generalization of

the shortest path problem in which not one, but collections of  $k$  paths, in non-decreasing order of weight, are sought [8], [9]. Such generalization finds even more applications than classical shortest paths computation, specifically is exploited in all those domains where several connection possibilities between vertices of a graph must be computed and examined according to some ranking, e.g. in robust network design, adaptive motion planning, multi-channel communications, web indexing and ranking, fraud detection, or tools for automatic translation between natural languages [1], [8], [10]–[12]. Moreover, top- $k$  shortest paths are known to represent a more natural, informative and robust measure of distance/similarity between vertices (and graphs, by extension) w.r.t. classical shortest paths and hence are widely adopted for graph learning tasks and by link prediction methods [1], [13].

Similarly to classical shortest path problems, the  $k$  shortest paths problem and corresponding solving algorithms have been investigated in many variants, the most relevant of which are related to the input graph having arc orientations and weights or not and to whether the aim is computing collections of  $k$  shortest paths for a pair of vertices (*single-pair* variant), for all vertices from a given source vertex (*single-source* variant), and for all pairs (*all-pairs* variant) [7]–[9], [14]. Besides the above variants, a significant difference between the  $k$  shortest paths problem and the computation of shortest paths, in terms of inherent computational complexity, is induced by two properties of the desired output: first, while by definition any shortest path for a pair of vertices is necessarily *simple* (i.e. it does not contain *loops*), collections of  $k$  shortest paths are generally allowed to include non-simple paths [7]; second,  $k$  shortest paths do not admit any optimal sub-structure property that can be exploited to derive solutions to smaller sub-problems from solutions of larger ones [14].

When paths of a solution are not constrained to be simple, the reference algorithm in terms of worst-case running time is due to Eppstein [7] who proposed a method that computes  $k$  shortest paths (not necessarily simple) for a single pair of vertices in time  $\mathcal{O}(m + n \log n + k)$  for any  $m$ -edge  $n$ -vertex weighted graph. The algorithm can be improved to run in  $\mathcal{O}(n + m + k)$  time when the graph to be managed is unweighted, and extended to find the  $k$  shortest paths from a

given source to each vertex in the graph in  $\mathcal{O}(m+n \log n+kn)$  total time [7]. When instead one requires paths to be simple (i.e. loopless), the problem becomes computationally more expensive to be solved. In fact, the most efficient method for the variant called *Single-Pair Top- $k$  Simple Shortest Paths problem* remains, unbeaten since its introduction in 1971, Yen's algorithm [15], which runs in  $\mathcal{O}(kn(m+n \log n))$  time. Moreover, it is known that the problem is not harder than the all-pairs shortest paths problem, since it can be solved through  $\mathcal{O}(k)$  iterations of any all-pairs shortest paths algorithm [16]. Furthermore, Vassilevska et al. [17] showed that a subcubic algorithm for the Single-Pair Top- $k$  Simple Shortest Paths problem would result in a subcubic algorithm for all-pairs shortest paths.

Several attempts at improving Yen's algorithm empirically have been designed in the last two decades, either for special graph classes or for typical inputs or by employing a pre-processing phase [8], [9], [18], [19]. The most noticeable advancements are the node-classification algorithm by Feng [19] and the sidetracks-based algorithm by Kurz and Mutzel [9] with the latter offering the best running time at the price of a significant memory overhead. We refer the reader to [8] and references therein for a recent overview on strategies to accelerate Yen's algorithm. To the best of our knowledge, no algorithm different than applying  $n-1$  times any method for the single-pair variant is known to address the single-source version of the  $k$  simple shortest paths problem. Similarly, very little is known about the all-pairs variant, for both the cases of simple and non-simple paths. The most recent progress is due to Argawal et al. [14] who devise algorithms that, for selected values of  $k$  and for simple paths, are faster than computing  $k$  shortest paths for all vertex pairs via any single-pair algorithm.

**Our Contribution.** In this paper, we introduce a new algorithm for solving the *Single-Source Top- $k$  Simple Shortest Paths problem* in polynomial time in general digraphs. To this aim, we provide a strong theoretical characterization of solutions to the problem, similar to what is known for the computation of classical shortest paths from a single source. We prove the correctness of the method and analyze its worst-case time complexity. On the one hand, we show our solution is on par, in terms of time complexity, with the best known option to attack the problem at hand, i.e. executing Yen's algorithm  $n-1$  times, for all vertex pairs where the source is one of the two endpoints. On the other hand, we provide an extensive experimental evaluation, that considers both real and artificially generated inputs, showing our new solution outperforms the state-of-the-art and always computes solutions to the Single-Source Top- $k$  Simple Shortest Paths problem faster (up to orders of magnitude) than the reference method in the literature.

## II. PRELIMINARIES

In this section, we give the notation and the definitions used throughout the paper. We are given a directed non-negatively weighted graph  $G = (V, E, \omega)$ , having  $n = |V|$  vertices and  $m = |E|$  edges, with  $E \subseteq V \times V$  and  $\omega : E \rightarrow \mathbb{R}^+$  being

a weight function on the edges. Given a vertex  $v \in V$ , we denote by  $N(v) = \{u \in V | (v, u) \in E\}$  the set of (*outgoing*) neighbors of  $v$ . A *path*  $P = (s = v_1, v_2, \dots, t = v_\eta)$  in  $G$ , connecting a pair of vertices  $s, t \in V$  (called its *endpoints*), is a sequence of  $\eta$  vertices such that  $(v_i, v_{i+1}) \in E$  for all  $i \in [1, \eta-1]$ . We say a path  $P$  is *simple* (or *loopless*) whenever it does not self-intersect, i.e. when there are no vertex repetitions in  $P$ . The *weight*  $\omega(P)$  of a path  $P = (s = v_1, v_2, \dots, t = v_\eta)$  is the sum of the weights of the edges induced by the vertices in  $P$ , i.e.  $\omega(P) = \sum_{i=1}^{\eta-1} \omega(v_i, v_{i+1})$ . In an unweighted graph, the weight coincides with the length of the path, i.e. is equal to the number of its edges (unweighted graphs can be treated as weighted ones with unitary weights). Given a graph  $G = (V, E)$  and a subset  $V' \subseteq V$  of its vertices, we denote by  $G[V']$  the sub-graph of  $G$  induced by the vertices in  $V'$ , i.e. the graph  $G' = (V', E')$  with  $E' = \{(u, v) \in E | u, v \in V'\}$ . Similarly, given a graph  $G$  and a path  $P$  in  $G$ , we denote by  $V(P)$  and  $E(P)$ , resp., the vertices and edges of  $G(P)$  and use  $G(P)$  to identify the sub-graph  $G[V(P)]$ .

A *shortest path*  $P$ , for a pair of vertices  $s, t \in V$ , is a path having minimum weight among all paths in  $G$  that connect  $s$  to  $t$ . The *distance*  $d(s, t)$  between  $s$  and  $t$  is the weight  $\omega(P)$  of a shortest path  $P$  from  $s$  to  $t$  in  $G$ . Given two paths, say  $P' = (v_1, v_2, \dots, v_\eta)$  and  $P'' = (u_1, u_2, \dots, u_\zeta)$ , we denote by  $P' \oplus P''$  the *concatenation* of the two paths, that is the path  $(v_1, v_2, \dots, v_\eta, u_1, u_2, \dots, u_\zeta)$  obtained by appending all vertices of  $P''$  to  $P'$ . Moreover, whenever a path  $P$  is the concatenation of two paths, e.g.  $P = P' \oplus Q$  (with  $Q$  possibly being the empty path), we say that  $P'$  is a *prefix* of  $P$ .

Given a non-negatively weighted digraph  $G = (V, E, \omega)$ , two vertices  $s, t \in V$ , and an integer  $k > 0$ . The Single-Pair Top- $k$  Simple Shortest Paths (SP-kSiSP for short) problem asks to compute a collection  $\mathcal{C}_{st} = \{P_1, P_2, \dots, P_{k'}\}$  of  $k' \leq k$  paths from  $s$  to  $t$  such that: (i)  $P_i$  is simple for any  $i \in \{1, 2, \dots, k'\}$ ; (ii)  $P_i \neq P_j$  for any  $i, j \in \{1, 2, \dots, k'\}, i \neq j$ ; (iii)  $\omega(P') \geq \omega(P_i)$  for any path  $P'$  in the graph, from  $s$  to  $t$ , that is not in  $\mathcal{C}_{st}$  and for any  $P_i \in \mathcal{C}_{st}$ ; (iv)  $\mathcal{C}_{st}$  is maximal (there does not exist a collection of size  $k'' > k'$ ). Observe that, if there exist  $k' < k$  simple paths from  $s$  to  $t$  in  $G$ , then the problem requires the computation of a collection  $\mathcal{C}_{st}$  that contains all and only such  $k' < k$  simple paths. Otherwise, the problem asks to compute a collection  $\mathcal{C}_{st}$  of  $k$  pair-wisely distinct simple paths from  $s$  to  $t$  such that the weight  $\omega(P')$  of any path  $P'$  that is in  $G$  but not in  $\mathcal{C}_{st}$  is larger than or equal than the weight of any path in  $\mathcal{C}_{st}$ . In either case, a collection  $\mathcal{C}_{st}$  of paths from  $s$  to  $t$ , satisfying the constraints imposed by the SP-kSiSP problem, is called a collection of top- $k$  simple shortest paths for pair  $(s, t)$ . Observe that there can be several feasible collections (e.g. if for a pair there exist a number  $k'' > k$  of paths of a same weight).

A generalization of the SP-kSiSP problem is the Single-Source Top- $k$  Simple Shortest Paths (SS-kSiSP for short) problem which asks to compute, for a given root vertex  $r$  and for each vertex  $v \in V$ , a collection  $\mathcal{C}_v = \{P_1, P_2, \dots, P_{k'}\}$  of  $k' \leq k$  paths from  $r$  to  $v$  such that: (i)  $P_i$  is simple for any  $i \in \{1, 2, \dots, k'\}$ ; (ii)  $P_i \neq P_j$  for any  $i, j \in \{1, 2, \dots, k'\}$  :

$i \neq j$ ; (iii)  $\omega(P') \geq \omega(P_i)$  for any path  $P'$  in the graph, from  $r$  to  $v$ , that is not in  $\mathcal{C}_v$  and for any  $P_i \in \mathcal{C}_v$ ; (iv)  $\mathcal{C}_v$  is maximal (there does not exist a collection of size  $k'' > k'$ ). Similarly to SP-kSiSP, for any vertex  $v$ , if there exist  $k' < k$  simple paths from  $r$  to  $v$  then the problem SS-kSiSP asks to compute a collection  $\mathcal{C}_v$  that contains all and only such  $k'$  simple paths. Vice versa, SS-kSiSP requires to determine a collection  $\mathcal{C}_v$  of  $k$  pair-wisely distinct simple paths, from  $r$  to  $v$ , such that the weight  $\omega(P')$  of any path  $P'$  that is in  $G$  but not in  $\mathcal{C}_v$  is larger than or equal to the weight of any path in  $\mathcal{C}_v$ . A collection  $\mathcal{C}_v$  of paths from  $r$  to  $v$ , satisfying the constraints imposed by the SS-kSiSP problem, is called a collection of (single-source) top- $k$  simple shortest paths for  $v$ .

Observe that, for each vertex  $v \in V$ , there can be several feasible collections (e.g. if there exist a number  $k'' > k$  of paths of a same weight from the root to a vertex  $v$ ). We call  $\Gamma_v$  the set of all such collections for a vertex  $v$  and use  $S = \{\mathcal{C}_{v_1}^S, \mathcal{C}_{v_2}^S, \dots, \mathcal{C}_{v_n}^S\}$  to identify any solution to SS-kSiSP, where  $\mathcal{C}_{v_i}^S \in \Gamma_{v_i} \ \forall v_i \in V$ . When  $S$  is clear by the context, we simply write  $\mathcal{C}_v$  instead of  $\mathcal{C}_v^S$ .

**Brute-Force Algorithm for SS-kSiSP.** A baseline brute-force approach for solving the SS-kSiSP problem is given in Algorithm 1. The algorithm is a simple modification of the

---

**Algorithm 1:** Algorithm EXHAUSTIVE for SS-kSiSP.

---

**Input:** Graph  $G = (V, E, \omega)$ , root  $r \in V$ , integer  $k > 0$ .

**Output:** Top- $k$  simple shortest paths  $T_v$  from  $r$  to each  $v \in V \setminus \{r\}$ .

---

```

1  $PQ \leftarrow \emptyset;$                                  $\triangleright$  Empty priority queue
2 foreach  $v \in V$  do  $T_v \leftarrow [];$            $\triangleright$  Empty lists
3  $PQ.enqueue((r));$                              $\triangleright$  Priority is  $\omega((r)) = 0$ 
4 while  $PQ \neq \emptyset$  do
5    $\Pi = (r, \dots, v) \leftarrow PQ.dequeueMin();$   $\triangleright$  Path w/
      $\text{min priority}$ 
6   foreach  $u \in N(v) \setminus V(\Pi)$  do           $\triangleright$  Simple paths
7      $PQ.enqueue(\Pi \oplus (u));$                  $\triangleright$  Priority is
        $\omega(\Pi) + \omega(v, u)$ 
8   if  $|T_v| < k$  then Add  $\Pi$  to  $T_v$ ;
```

---

brute-force method for finding all simple paths in a graph for a pair of vertices [1], [20] that solves SS-kSiSP by exhaustively discovering, through a modified Dijkstra's-like visit, all simple paths from a given root vertex to any other vertex  $v \in V$  of the graph and by storing, in a data structure  $T_v$ , only a subset of such paths that form the sought collection of top- $k$  simple shortest paths for  $v$ . The algorithm finds and enqueues and dequeues paths in a priority queue  $PQ$ , where the priority is the weight of the paths, and terminates when either  $PQ$  is empty or when  $|T_v| = k$  for all  $v \in V \setminus \{r\}$ . Note that, the same algorithm can be easily modified to achieve a solution to the SP-kSiSP problem, for a pair  $s, t \in V$  of vertices, by starting the visit from either  $s$  or  $t$  and by modifying the termination criterion to stop the algorithm when either the queue is empty or  $|T_t| = k$ . The correctness of Algorithm 1 is shown by the following results.

**Lemma II.1.** Algorithm EXHAUSTIVE computes all simple paths from the root vertex to any other vertex.

*Proof.* By contradiction, assume there exists a simple path from the root  $r$  to a vertex that is not found by algorithm EXHAUSTIVE, i.e. that is not dequeued from  $PQ$  in line 5. Among all simple paths that are not found by the algorithm, let  $\pi = (r, \dots, u, v)$  be the shortest. Since all paths that are enqueued are eventually extracted from  $PQ$ , it follows that prefix  $\pi' = (r, \dots, u)$  must have been enqueued and dequeued, as otherwise  $\pi$  would not be the shortest path among those that are not found. It follows that  $\pi$  is enqueued in line 7 when  $\pi'$  is dequeued, which is a contradiction since it implies that  $\pi$  is enqueued and eventually dequeued from  $PQ$ .  $\square$

**Lemma II.2.** Algorithm EXHAUSTIVE dequeues simple paths from the root vertex to any other vertex in non-decreasing order of weight.

*Proof.* Assume by contradiction that there exist two paths, say  $\pi$  and  $\pi' = (r, \dots, v)$ , such that  $\omega(\pi) > \omega(\pi')$  and  $\pi$  is dequeued from  $PQ$  before  $\pi'$ . Call  $F$  the set of paths that are dequeued by the algorithm before  $\pi$  is dequeued. Observe that  $F$  contains at least path  $(r)$  which is enqueued and dequeued in any case by the algorithm in its first steps. Now, focus on when path  $\pi$  is dequeued. Since  $\omega(\pi) > \omega(\pi')$  we have that  $\pi' \notin PQ$ , as otherwise we would have dequeued  $\pi'$ . Since the algorithm extracts the minimum. Moreover,  $\pi' \notin F$  as well by hypothesis. Hence, the prefix of  $\pi'$ , say  $\pi''$ , such that  $\pi' = \pi'' \oplus (v)$ , cannot be in  $PQ$  and neither in  $F$ . In fact,  $\pi''$  cannot be in  $PQ$ , as otherwise  $\pi''$  would have been dequeued in place of  $\pi$  (note that  $\pi''$  is shorter than  $\pi$  since  $\omega(\pi'') < \omega(\pi)$ ) and hence  $\pi'$  would have been enqueued in line 7 and therefore extracted before  $\pi$ , which is a contradiction. Moreover,  $\pi''$  cannot be in  $F$ , as otherwise  $\pi'$  would have been added to  $PQ$  when  $\pi''$  is dequeued. By repeating recursively this argument for any prefix of  $\pi'$ , we reach the contradiction of path  $(r)$  not being in both  $PQ$  and  $F$ .  $\square$

**Theorem II.3.** Algorithm EXHAUSTIVE solves SS-kSiSP.

*Proof.* The proof follows by Lemmas II.1 and II.2. In particular, by Lemma II.2 we know that algorithm EXHAUSTIVE discovers all simple paths in order of weight. Hence, in line 8, data structure  $T_v$  is filled with paths that follow a non-decreasing order of weight, for each  $v \in V$ . Moreover, by Lemma II.1 we know algorithm EXHAUSTIVE determines all simple paths from the root, hence at termination  $T_v = \mathcal{C}_v$  for each  $v \in V$  and for some  $\mathcal{C}_v \in \Gamma_v$ .  $\square$

For the sake of completeness, observe that algorithm EXHAUSTIVE can be easily modified to terminate earlier by adding an additional test in the while loop of line 4. Specifically, the while loop can be changed to break if, for all vertices, we have  $|T_v| = k$ . This modification does not change the time complexity of the algorithm since in the worst case there can be a vertex  $w$  such that  $k$  simple paths from  $r$  to  $w$  do not exist in the graph. In the reminder of the paper we refer to this variant, with a little abuse of notation, by algorithm EXHAUSTIVE as well. While algorithm EXHAUSTIVE is simple

and elegant, it is easy to see how its time complexity is exponential w.r.t. input size.

**Theorem II.4.** There exists an  $n$ -vertex,  $m$ -edge graph that force EXHAUSTIVE to run for  $\Omega(2^n)$  time to solve SS-kSiSP.

*Proof.* The algorithm in the worst case adds to  $PQ$  all simple paths from the root to any vertex  $v \in V$ . It is easy to construct an input instance where the number of such paths is exponential w.r.t. the number of vertices  $n$ , regardless of  $k$ . For instance, consider the undirected, unweighted graph of Fig. 1. It is easy to see that, if we call  $d$  the number of vertices of  $c_i$  type, the number of simple paths from the root  $r$  to vertex  $v$  is  $2^d$ . Since  $d = \Theta(n)$  the claim follows.  $\square$

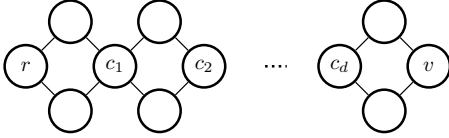


Fig. 1. Input instance considered in the proof of Theorem II.4.

**Polynomial-time Algorithm for SS-kSiSP.** The state-of-the-art for solving the SS-kSiSP problem, in terms of time complexity, consists in performing  $n$  times the algorithm of Yen [15], which is the algorithm with the best time complexity for solving the SP-kSiSP problem [8]. Specifically, to obtain a solution to SS-kSiSP in a graph  $G = (V, E, \omega)$ , it suffices to execute Yen's algorithm for each pair  $(r, v)$  where  $r \in V$  is the root vertex while  $v \in V \setminus \{r\}$  is any other vertex of the graph. To solve SP-kSiSP for a given pair of vertices  $(s, t)$ , such method first computes a shortest path, say  $p$ , from  $s$  to  $t$  as first path. Then, it considers each vertex in  $p$  as the *deviation* vertex to determine candidates (*detours*) for the next shortest path using a single-source shortest path visit that uses at least one edge that is not in  $p$ . The next shortest path is then selected among the candidates with minimum cost. The process continues until  $k$  different shortest paths are determined. The total time cost of Yen's algorithm is thus  $\mathcal{O}(kn(m + n \log n))$  for any  $n$ -vertex,  $m$ -edge weighted digraph, which comes from  $\mathcal{O}(n)$  single-source shortest path discoveries for each of the  $k$  simple shortest paths. Specifically, the following holds:

**Theorem II.5** ([7]). Yen's algorithm solves SP-kSiSP in  $\mathcal{O}(kn(m + n \log n))$  time for any an  $n$ -vertex,  $m$ -edge weighted digraph and  $k > 0$ .

Therefore, the following result can be derived.

**Corollary II.5.1.** Given an  $n$ -vertex,  $m$ -edge weighted digraph  $G = (V, E, \omega)$  and a distinguished root vertex  $r \in V$ , the SS-kSiSP problem can be solved in  $\mathcal{O}(kn^2(m + n \log n))$  time by executing Yen's algorithm for all pairs  $(r, v)$  of  $G$  such that  $v \in V \setminus \{r\}$ , for any  $k > 0$ .

### III. PROPERTIES OF SOLUTIONS TO SS-kSiSP

In this section, we provide a detailed characterization of some properties of solutions to the SS-kSiSP problem which

we exploit to design efficient algorithms to solve it. To this aim, we first provide the following definitions.

**Definition III.1** (PREDECESSOR). Let  $S = \{C_{v_1}^S, C_{v_2}^S, \dots, C_{v_n}^S\}$  be a solution to SS-kSiSP. Consider the relation  $R_S \subseteq V \times V$  such that  $uR_Sv$  if and only if vertex  $u \in V \setminus \{v\}$  belongs to a path of  $C_v^S$  for  $v \in V$ , i.e.  $u \in V(P) \setminus \{v\}$  for some  $P \in C_v^S$ . Then, we say vertex  $u$  is a *predecessor* of  $v$  and denote by  $V(C_v^S) = \{u \mid uR_Sv\}$  a set of *predecessors* of a vertex  $v$ .

**Definition III.2** (GENERAL PREDECESSOR). Let  $S$  be a solution to SS-kSiSP. Let  $T_S$  be the transitive closure of relation  $R_S$  of Def. III.1. Then, we say a vertex  $u \in V$  is a *general predecessor* of a vertex  $v \in V$  if and only if  $uT_Sv$ , and denote by  $A_{S,v} = \{u \mid uT_Sv\} \cup \{v\}$  a set of *general predecessors* of a vertex  $v \in V$ .

The following two lemmata establish important structural properties that hold for certain paths belonging to solutions to SS-kSiSP and that will be used to design our algorithms to solve SS-kSiSP.

**Lemma III.1** (BASIC LEMMA). Let  $S = \{C_{v_1}^S, C_{v_2}^S, \dots, C_{v_n}^S\}$  be a solution to SS-kSiSP for a given graph  $G = (V, E, \omega)$ , root vertex  $r \in V$ , and  $k > 0$ . Let  $P_{rw} = P_{rv} \oplus P_{vw}$  any path from the root  $r$  to some vertex  $w \in V$  such that  $P_{rw} \in C_w$ ,  $\forall C_w \in \Gamma_w$  and  $P_{rv} \notin C_v^S$ . Then, there exists a vertex  $t \in V(C_v^S) \cap V(P_{vw})$ .

*Proof.* Notice that, by definition of  $V(C_v^S)$ , we have  $t \neq v$  while  $t$  might coincide with  $w$  or not. By contradiction, assume that  $V(C_v^S) \cap V(P_{vw})$  is empty. Consider the set of simple paths, from  $r$  to  $w$ :  $\Pi_{rw} = \{Q \oplus P_{vw} \mid Q \in C_v^S\}$ . Since number of paths in  $C_v^S$  is  $k$  (otherwise  $P_{rv}$  should belong to  $C_v^S$ ), also the size of  $\Pi_{rw}$  is  $k$ . Furthermore, the weight of each path in  $\Pi_{rw}$  is less than or equal to  $\omega(P_{rw})$ . Hence, there exists a solution  $S'$  to SS-kSiSP such that  $P_{rw} \notin C_w^{S'}$ , a contradiction.  $\square$

In other words, Lemma III.1 shows that the existence of a path  $P_{rv}$  that: a) is not part of a solution  $C_v \in \Gamma_v$  for some vertex  $v \in V$ ; b) is a prefix of path  $P_{rw}$  that is in all solutions for some other vertex  $w \in V$ ; implies the existence of a predecessor  $t \in V(C_v)$  that belongs to  $P_{rw}$ .

We can derive the following result that characterizes the structure of simple paths to  $w$  under the hypotheses of Lemma III.1:

**Corollary III.1.1** (BASIC COROLLARY). Let  $S = \{C_{v_1}^S, C_{v_2}^S, \dots, C_{v_n}^S\}$  be a solution to SS-kSiSP for a given graph  $G = (V, E, \omega)$ , root vertex  $r \in V$ , and  $k > 0$ . Let  $P_{rw} = P_{rv} \oplus P_{vw}$  be any path from the root  $r$  to some vertex  $w \in V$  such that  $P_{rw} \in C_w^S$  and  $P_{rv} \notin C_v^S$ . Then, there must exist a path  $P'_{rw} = P_{rv} \oplus P'_{vw}$  in  $C_w^S$  such that  $P'_{vw}$  is a shortest path in  $G[V \setminus V(P_{rv}) \cup \{v\}]$ .

A stronger characterization on the structure of paths that belong to solutions to SS-kSiSP is provided through the following result. In particular, the next lemma asserts that

if a path  $P_{rw} = P_{rv} \oplus P_{vw}$  is necessarily in a solution for a vertex  $w$ , but the prefix  $P_{rv}$  is not necessary to complete a solution for  $v$ , then there must be a vertex  $y$ , among the general predecessors of  $v$ , that lies on  $P_{vw}$  and such that the prefix  $P_{ry}$  of  $P_{rw}$  is among the paths in a solution for  $y$ .

**Lemma III.2 (PATH LEMMA).** Let  $S = \{C_{v_1}^S, C_{v_2}^S, \dots, C_{v_n}^S\}$  be a solution to SS-kSiSP for a given input graph  $G = (V, E, \omega)$ , root vertex  $r \in V$ , and  $k > 0$ . Let  $P_{rw} = P_{rv} \oplus P_{vw}$  be a path from the root  $r$  to a vertex  $w \in V$  such that  $P_{rw} \in \mathcal{C}_w, \forall C_w \in \Gamma_w$  and  $P_{rv} \notin C_v^S$ . Then, there exist two vertices  $t, y$  in  $V(P_{vw}) \cap A_{S,v}$  such that: i)  $y \in V(C_t^S)$ ; ii)  $P_{rt} = P_{rv} \oplus P_{vt} \notin C_t^S$ ; and iii)  $P_{ry} = P_{rv} \oplus P_{vy} \in C_y^S$ , where  $P_{vt}$  and  $P_{vy}$  are both prefixes of  $P_{vw}$ .

*Proof.* By Lemma III.1, there is a vertex  $t_0$  in  $P_{vw}$  that belongs to  $V(C_{t_0}^S)$ . If  $P_{rt_0} = P_{rv} \oplus P_{vt_0} \in C_{t_0}^S$  then we are done, since  $v$  and  $t_0$  play the roles of  $t$  and  $y$ , respectively. On the contrary, we can consider, recursively, vertex  $t_0$  and path  $P_{rt_0} \notin C_{t_0}^S$  and apply again Lemma III.1. Hence, there exists a vertex  $t_1$  in  $P_{vw}$  belonging to  $V(C_{t_1}^S)$ . Now, if  $P_{rt_1} = P_{rt_0} \oplus P_{t_0t_1} \in C_{t_1}^S$ , again we are done since  $t_0$  and  $t_1$  play the roles of  $t$  and  $y$ , respectively. Vice versa, we can repeat the reasoning above and either find two internal vertices  $t_i$  and  $t_{i+1}$  of  $P_{rw}$  that play the roles of  $t$  and  $y$ , or an internal vertex  $t_i$  of  $P_{rw}$  and  $w$  itself playing the roles of  $t$  and  $y$ , respectively, since  $P_{rw} \in \mathcal{C}_w, \forall C_w \in \Gamma_w$  by hypothesis.  $\square$

Notice that Figs 2 and 3 provide a visual representation of the claims of Lemma III.1 and III.2, respectively. The

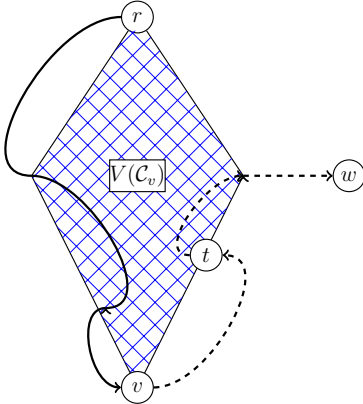


Fig. 2. Solution  $S'$  in the proof of Lemma III.1: the solid path is  $P_{rv} \notin C_v$  while the dashed path is  $P_{vw}$  such that  $P_{rw} \in \mathcal{C}_w$ ; vertex  $t$  must be a predecessor of  $v$  w.r.t.  $S$ . The blue grid identifies the vertices of  $V(C_v)$ .

following lemma shows that sets of general predecessors form a nested set collection.

**Lemma III.3 (RECURSIVE LEMMA).** Let  $S$  be a solution to SS-kSiSP for a given graph  $G = (V, E, \omega)$ , root vertex  $r \in V$ , and  $k > 0$ . Then, for any  $v, x \in V$  such that  $x \in A_{S,v}$ , we have  $A_{S,x} \subseteq A_{S,v}$ .

*Proof.* By definition III.2, any vertex  $y \in A_{S,x}$  is a general predecessor of  $x$  hence  $y$  is such that  $yT_S x$ . Similarly, since

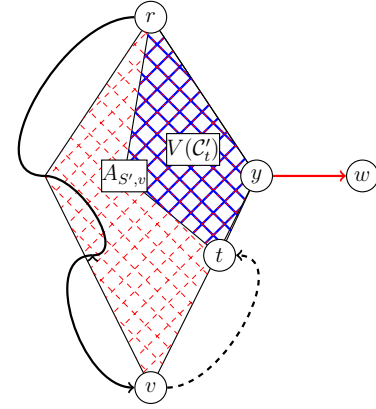


Fig. 3. Construction of solution  $S'$  of Lemma III.2: black solid path is  $P_{rv}$ , while the black dashed one is  $P_{vy}$ ; the red path, instead, is a simple path  $P_{yw}$  in  $G[V \setminus A_{S,v} \cup \{y\}]$ ; vertices  $t, y$  are in  $V(P_{vw}) \cap A_{S',v}$ . The blue grid shows  $V(C_t)$  while the red dashed grid identifies the vertices of  $A_{S',v}$  (which includes  $V(C_t)$ ).

$x \in A_{S,v}$ , we have  $xT_S v$ . Then, by transitivity, it follows that  $yT_S v$  which implies  $y \in A_{S,v}$ .  $\square$

Finally, we derive another relation between paths in any solution to SS-kSiSP and general predecessors of vertices in such paths. Such relation allows to characterize the structural properties of paths in solutions to SS-kSiSP and of their prefixes, and to understand whether certain paths can be safely discarded as not being part of a solution for an instance of SS-kSiSP on the basis of conditions holding on one of their prefixes.

**Lemma III.4 (PRUNING LEMMA).** Let  $S = \{C_{v_1}^S, C_{v_2}^S, \dots, C_{v_n}^S\}$  be a solution to SS-kSiSP for a given input graph  $G = (V, E, \omega)$ , root vertex  $r \in V$ , and  $k > 0$ . Let  $P_{rv}$  any path from the root  $r$  to a vertex  $v \in V$ . If (i)  $|C_u^S| = k$  for all vertices  $u \in A_{S,v}$  and (ii)  $P_{rv}$  is not a prefix of any path in  $\bigcup_{u \in A_{S,v}} C_u^S$ , then there exists a solution  $S' = \{C'_{v_1}, C'_{v_2}, \dots, C'_{v_n}\}$  such that  $P_{rv}$  is not a prefix of any path in  $\bigcup_{u \in V} C'_u$ .

*Proof.* We construct a solution  $S' = \{C'_{v_1}, C'_{v_2}, \dots, C'_{v_n}\}$  by modifying  $S$  as follows. By hypothesis,  $P_{rv}$  is not a prefix of any path in  $\bigcup_{u \in A_{S,v}} C_u^S$  hence we assign  $C'_u = C_u^S$  for all vertices  $u \in A_{S,v}$ . Now, consider any vertex  $w \in V \setminus A_{S,v}$  such that  $w \neq r$  and let  $P_{rw} = P_{rv} \oplus P_{vw} \in \mathcal{C}_w$  be a path having  $P_{rv}$  as prefix in  $\mathcal{C}_w$ . Let  $y$  be the vertex that lies on  $P_{vw}$ , belongs to set  $A_{S,v}$  and is closest to  $w$ , that is a vertex such that  $P_{vw} = P_{vy} \oplus P_{yw}$  where  $P_{yw}$  is a path from  $y$  to  $w$  in  $G[V \setminus A_{S,v} \cup \{y\}]$ . Note that  $y$  can be  $v$  itself. Let  $P_{ry}^{max}$  be the heaviest path in  $\mathcal{C}_y$ . Since  $P_{rv}$  is not a prefix of any path in  $\bigcup_{u \in A_{S,v}} C_u^S$ , it follows that the weight of path  $P_{ry} = P_{rv} \oplus P_{vy}$

must be at least that of  $P_{ry}^{max}$ . Hence, we have:

$$\begin{aligned}\omega(P_{rw}) &= \omega(P_{rv} \oplus P_{vw}) = \omega(P_{rv} \oplus P_{vy} \oplus P_{yw}) \\ &= \omega(P_{rv} \oplus P_{vy}) + \omega(P_{yw}) \geq \omega(P_{ry}^{max}) + \omega(P_{yw}) \\ &\geq \omega(P_{ry}^{max} \oplus P_{yw}).\end{aligned}$$

Since  $y \in A_{S,v}$ , we have  $|\mathcal{C}_y^S| = k$  by hypothesis (i). Moreover, by Lemma III.3,  $A_{S,y} \subseteq A_{S,v}$ . Then there must exist, in  $G$ ,  $k$  simple paths in the form  $P_{ry} \oplus P_{yw}$ , one for each path  $P_{ry} \in \mathcal{C}_y^S$ , that have weight at most that of  $P_{rw}$ . Then, we can construct set  $\mathcal{C}_w'$  by replacing  $P_{rw} \in \mathcal{C}_w^S$  with a path  $P_{ry} \oplus P_{yw}$  for some  $P_{ry} \in \mathcal{C}_y^S$ .  $\square$

#### IV. NEW ALGORITHMS FOR SS-KSiSP

In this section, we introduce our new algorithm for solving the SS-kSiSP problem. Our design is incremental: we first exploit the properties of Lemma III.4 to design algorithm SS-TOP- $k$ . We prove its correctness but show its running time can be exponential w.r.t. the graph size. Then, we improve it by incorporating the results of Lemma III.2 to obtain algorithm BOUND-SS-TOP- $k$ , and show it solves SS-kSiSP in worst-case polynomial time with respect to graph size and  $k$ .

While describing the two algorithms we assume that, during their execution, they build a data structure  $T$  (e.g., an associative array of lists) that has one entry, say  $T_v$ , for each vertex  $v$  of the graph, that stores a collection of simple paths found for said vertex. For both SS-TOP- $k$  and BOUND-SS-TOP- $k$  we show that, at termination,  $\{T_v\}_{v \in V}$  is a solution to the SS-kSiSP problem, i.e. each  $T_v$  is a collection of top- $k$  simple shortest paths from  $r$  to  $v$ , belonging to  $\Gamma_v$ , for any vertex  $v \in V \setminus \{r\}$ . In the description of the procedures, given a collection of paths, say  $T_v$ , we use  $V(T_v)$  to denote the set of vertices belonging to paths in  $T_v$ , similarly to the notation used for solutions to SS-kSiSP. Moreover, we say a vertex  $v$  is *saturated* whenever  $|T_v| = k$ .

##### A. Algorithm SS-TOP- $k$

In this section we introduce algorithm SS-TOP- $k$  to solve SS-kSiSP. The algorithm, whose pseudo-code is given in Algorithm 2, fills data structure  $T_v$  with simple paths emanating from the root until a solution to the instance of SS-kSiSP is achieved. In detail, the method works as follows. First, data structure  $T_v$  for each  $v \in V$ , is initialized to be the empty list while priority queue  $PQ$  contains only path  $(r)$ , for the given root vertex  $r \in V$ . Paths are added to  $PQ$  with a priority equal to their weight; hence initially we add  $(r)$ , which is the only simple path from the root to itself, with zero priority. Then, until the queue is not empty or  $|T_w| < k$  for some  $w \in V \setminus \{r\}$ , a path  $\Pi$ , terminating at some vertex  $v$ , is dequeued from  $PQ$  due to having minimum priority (i.e. weight) among those in  $PQ$  and an *iteration* of the algorithm is executed. Specifically, in a generic iteration, if the test of line 6 succeeds (i.e. the conditional statement evaluates to true) then the visit proceeds as in the exhaustive algorithm, by enqueueing, for each vertex  $u$  neighboring  $v$  that does not intersect  $\Pi$ , the simple path  $\Pi \oplus (u)$  in the graph that has

$\Pi$  as prefix and terminates at  $u$ , with priority equal to its weight, namely  $\omega(\Pi \oplus (u)) = \omega(\Pi) + \omega(v, u)$ . Vice versa, no enqueue operation is performed. In both cases, if the number of paths in  $T_v$  is less than  $k$ , then path  $\Pi$  is added to  $T_v$  to be part of a solution to the problem. Observe that the test of line 6 evaluates to true when either the number of paths in  $T_v$  is less than  $k$  or whenever routine PRUNING( $v, r$ ) (whose pseudo-code is given in Procedure 3) returns true. The latter routine verifies whether there exists a general predecessor of  $v$  that is not saturated or not. To this aim, the procedure starts a breadth-first-search-like visit of the graph that begins by enqueueing  $v$  into a FIFO queue  $Q$  (see line 14). Whenever a vertex is removed from  $Q$ , following the order of insertion, the visit proceeds by considering only the part of the sub-graph induced by the predecessors of the dequeued vertex (see line 9). Hence, by transitivity, it explores the set of general predecessors of  $v$ . If one of the encountered vertices, say  $x$ , is found to be not saturated (i.e. such that  $|T_x| < k$ ) then the routine returns true. Visited vertices are stored into a set SEEN, as in a classic breadth-first-search, to avoid evaluating a vertex to be not saturated more than once. If the routine terminates without finding any of such vertices, then it returns false. Algorithm SS-TOP- $k$  stops whenever either the queue becomes empty or when condition  $|T_w| = k$  is reached for all  $w \in V \setminus \{r\}$ , i.e. when all vertices are saturated.

---

#### Algorithm 2: Algorithm SS-TOP- $k$ .

---

**Input:** Graph  $G = (V, E, \omega)$ , root  $r \in V$ , integer  $k > 0$ .

**Output:** Top- $k$  simple shortest paths  $T_v$  from  $r$  to each  $v \in V \setminus \{r\}$ .

---

```

1  $PQ \leftarrow \emptyset;$  ▷ Empty priority queue
2 foreach  $v \in V$  do  $T_v \leftarrow [];$  ▷ Empty lists
3  $PQ.enqueue((r));$  ▷ Priority is  $\omega((r)) = 0$ 
4 while  $PQ \neq \emptyset \wedge \exists w \neq r : |T_w| < k$  do
5    $\Pi = (r, \dots, v) \leftarrow PQ.dequeueMin();$  ▷ Path w/
   min priority
6   if  $|T_v| < k \vee \text{PRUNING}(v, r)$  then
7     foreach  $u \in N(v) \setminus V(\Pi)$  do ▷ No pruning
8        $PQ.enqueue(\Pi \oplus (u));$  ▷ Priority is
        $\omega(\Pi) + \omega(v, u)$ 
9   if  $|T_v| < k$  then Add  $\Pi$  to  $T_v;$  ▷  $v$  not saturated
```

---



---

#### Procedure 3: Procedure PRUNING.

---

**Input:** A vertex  $v$ .

**Output:** True iff  $\exists$  non-saturated general predecessor of  $v$ .

---

```

1  $Q \leftarrow \{v\};$  ▷ Add  $v$  to empty FIFO queue
2  $SEEN \leftarrow \{v\};$ 
3 while  $Q \neq \emptyset$  do
4    $x \leftarrow Q.pop();$  ▷ Dequeue head of queue
5   if  $|T_x| < k$  then
6     return true; ▷ Non saturated vertex  $x \in A_{S,v}$ 
7   foreach  $u \in N(x) : u \notin SEEN$  do
8     if  $u \in V(T_x)$  then
9        $Q.append(u);$ 
10     $SEEN \leftarrow SEEN \cup \{u\};$ 
11 return false;
```

---

We now give the proof of correctness of Algorithm 2.



*Theorem IV.1.* Algorithm SS-TOP- $k$  solves SS-kSiSP.

*Proof.* Observe that, by removing line 6, and hence executing lines 7–8 regardless of the result of the execution of routine PRUNING, algorithm SS-TOP- $k$  is equivalent to algorithm EXHAUSTIVE (see Algorithm 1) which exhaustively computes a collection of top- $k$  simple shortest paths from the root vertex to all other vertices. Therefore, in this case, the claim trivially follows by Theorem II.3. Thus, to complete the proof we need to show that, if the conditional statement of line 6 is false in some cases, and the visit does not proceed from some vertices as their neighbors are not enqueued into  $PQ$ , then the algorithm's correctness is not altered i.e., at termination, collection  $\{T_{v_1}, T_{v_2}, \dots, T_{v_n}\}$  is a solution  $S$  to SS-kSiSP, meaning that each  $T_v$  is an element of  $\Gamma_v$ .

By contradiction, assume that there exists a non-empty subset of vertices  $W \subseteq V$  such that, at termination,  $T_u \notin \Gamma_u$  for all  $u \in W$ , meaning that: either (i)  $|T_u| = k' < k$  while all elements of  $\Gamma_u$  have size  $k'' > k'$  with  $k'' \leq k$ , i.e. collection  $T_u$  is not maximal; or (ii)  $|T_u| = k$  but there exist a path  $P'$  in the graph that has been added to  $T_u$  but it should not have been, i.e. there exists at least a path  $P$  whose weight is smaller than that of  $P'$  in the graph. In both cases, it follows that for any vertex  $s \in W$  there exists at least a path  $P_{rs}$  in the graph that should be in  $T_s$  but is not added to  $T_s$  during the execution of the algorithm (i.e.,  $P_{rs} \in \mathcal{C}_s, \forall \mathcal{C}_s \in \Gamma_s$ ).

Now, call  $M$  the set of all such paths and let  $P_{rw} \in M$  be a path in  $M$ , from  $r$  to some vertex  $w \in W$ , that contains a vertex  $v \in P_{rw}$  which is the first vertex such that the test of line 6 is false, i.e. the first vertex during the execution of the algorithm such that  $|T_v| < k \vee \text{PRUNING}(v, r)$  false and  $v$  lies on a path in  $M$ . This implies that: (i)  $|T_v| \geq k$  and, by Theorem II.3,  $T_v \in \Gamma_v$ , i.e.  $T_v$  is a set of  $k$  simple shortest paths from  $r$  to  $v$ ; (ii) there is no other vertex lying on a path in  $M$  that is dequeued from  $PQ$  before  $v$  during the execution of the algorithm and such that the test of line 6 is false; (iii)  $P_{rv}$  is a prefix of  $P_{rw}$ , if we call  $P_{rv}$  the path that is dequeued from  $PQ$  together with  $v$ ; (iv)  $P_{rw} \in \mathcal{C}_w$  for each  $\mathcal{C}_w \in \Gamma_w$ . Moreover, if we denote by  $X$  the set of vertices that are visited by routine PRUNING( $v, r$ ) in the corresponding execution of line 6 then for any  $x \in X$  we have that: (i)  $x$  is saturated, i.e.  $|T_x| = k$ ; (ii)  $T_x \in \Gamma_x$  by Theorem II.3. Thus, set  $X \equiv A_{S,v}$  for some solution  $S$ . Hence, by Lemma III.4, since (i)  $|\mathcal{C}_u| = k$  for all vertices  $u \in A_{S,v}$  and (ii)  $P_{rv}$  is not a prefix of any path in  $\bigcup_{u \in A_{S,v}} \mathcal{C}_u$ , then there must exist a solution  $S' = \{\mathcal{C}'_{v_1}, \mathcal{C}'_{v_2}, \dots, \mathcal{C}'_{v_n}\}$  such that  $P_{rv}$  is not a prefix of any path in  $\bigcup_{u \in V} \mathcal{C}'_u$ , which contradicts the hypothesis of  $P_{rv}$  being a prefix of  $P_{rw}$  and  $P_{rw}$  being in  $\mathcal{C}_w$  for any  $\mathcal{C}_w \in \Gamma_w \forall \mathcal{C}_w \in \Gamma_w$ .  $\square$

Unfortunately, while the pruning test of line 6 in Algorithm 2 appears to be helpful for reducing, w.r.t. algorithm EXHAUSTIVE, the number of paths that have to be enqueued and dequeued into/from the queue to determine solutions to SS-kSiSP, it is easy to see that alone it does not suffice to

guarantee polynomial running time in the worst case, as shown by the following result:

*Theorem IV.2.* There exist an  $n$ -vertex,  $m$ -edge graph and an integer  $k > 0$  that force SS-TOP- $k$  to run in  $\Omega(2^n)$  time to solve SS-kSiSP.

*Proof.* Consider the graph of Fig. 4, which is a slightly modified input, w.r.t. that of Fig. 1, and analyze the case of  $k = 3$  and of the root vertex being  $r$ . We observe that all simple shortest paths from  $r$  to vertex  $x_1$  can be classified in two categories: in the first category we have (two) simple paths that do not share any vertex with the internal vertices of the red dashed path, namely  $(r, x_1)$ ,  $(r, x_2, c_1, x_1)$ , and in the second we have all simple paths formed by concatenating all simple paths from  $r$  to vertex  $v$  with the red path from  $v$  to  $x_1$ . If we call  $d$  the number of vertices of  $c_i$  type in the given graph, then we have  $2^d$  of such simple paths from  $r$  to  $x_1$ . Assume the dashed red path has weight  $2d$ . Then, algorithm SS-TOP- $k$  finds the two simple paths of the first category after at most three dequeue and at most nine dequeue operations, resp., depending on the order in which neighbors are considered. Similarly, after at most nine dequeue operations, paths terminating at  $x_3$  and  $x_4$  are dequeued and the four paths  $(r, x_1, c_1, x_3, c_2)$ ,  $(r, x_1, c_1, x_4, c_2)$ ,  $(r, x_2, c_1, x_3, c_2)$ ,  $(r, x_2, c_1, x_4, c_2)$  terminating at vertex  $c_2$  are then enqueued. Observe that, such enqueue operations follow from the test of line 6 evaluating to true for both  $c_1$ , since  $|T_{c_1}| < k$ , and  $x_3$  and  $x_4$ , since both  $T_{x_1}$  and  $T_{x_3}$  have size  $k = 3$  but calls to routine PRUNING( $x_1, r$ ) and PRUNING( $x_3, r$ ) both return true ( $x_1$  or  $x_2$  are found to be not saturated). This behavior proceeds unchanged for all vertices in all simple paths from  $r$  to  $v$  that do not share vertices with the red path, with routine PRUNING( $*, r$ ) always returning true since  $x_1$  is a predecessor of all such vertices and is not saturated. Hence, a number  $\mathcal{O}(2^d) = \mathcal{O}(2^n)$  of paths that terminate at  $v$  eventually is enqueued and must be dequeued before any of the  $2^d$  simple paths from  $r$  to  $x_1$  of the same category can be enqueued/dequeued and therefore before the algorithm can successfully terminate.  $\square$

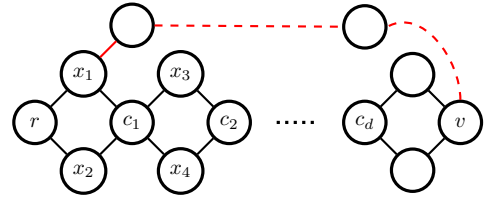


Fig. 4. Input instance considered in the proof of Theorem IV.2.

#### B. Algorithm BOUND-SS-TOP- $k$

In this section, we design algorithm BOUND-SS-TOP- $k$  that solves SS-kSiSP in polynomial time. The algorithm is built upon algorithm SS-TOP- $k$  and exploits the characterization of solutions to SS-kSiSP, shown in Sections II–III, in order to limit the maximum number of paths that can be enqueued and

hence dequeued from the priority queue, for each terminal vertex other than the root. To obtain such a bound, the main modification consists in replacing procedure PRUNING, which simply searches for a non-saturated general predecessor to decide whether a visit must be pruned at a given path  $\Pi = (r, \dots, v) \notin \mathcal{C}_v$  or not, with an explicit computation of top- $k$  simple shortest paths from the root to all non-saturated general predecessors. This is done to avoid pathological behaviors of algorithm SS-TOP- $k$  which is forced, due to a non saturated general predecessor, to discover an exponential number of paths before completing the computation of a solution to SS-kSiSP (see Fig. 4). In more detail, the algorithm, whose

---

**Algorithm 4:** Algorithm BOUND-SS-TOP- $k$ .

---

**Input:** Graph  $G = (V, E, \omega)$ , root  $r \in V$ , integer  $k > 0$ .  
**Output:** Top- $k$  simple shortest paths  $T_v$  from  $r$  to each  $v \in V \setminus \{r\}$ .

```

1  $PQ \leftarrow \emptyset;$  ▷ Empty priority queue
2 foreach  $v \in V$  do  $T_v \leftarrow [];$  ▷ Empty lists
3  $PQ.enqueue((r));$  ▷ Priority is  $\omega(r) = 0$ 
4 while  $PQ \neq \emptyset \wedge \exists w \neq r : |T_w| < k$  do
5    $\Pi = (r, \dots, v) \leftarrow PQ.dequeueMin();$  ▷ Path w/
   min priority
6   if  $|T_v| < k$  then
7     Add  $\Pi$  to  $T_v$ ;
8     foreach  $u \in N(v) \setminus V(\Pi)$  do
9       if  $u$  not super-saturated  $\wedge \Pi \oplus (u) \notin PQ$  then
10         $PQ.enqueue(\Pi \oplus (u));$  ▷ Priority is
         $\omega(\Pi) + \omega(v, u)$ 
11   else if  $v$  not super-saturated then
12     Compute  $A_{S,v}$  and a  $\mathcal{C}_w \supseteq T_w, \forall w \in A_{S,v};$ 
13     foreach  $w \in A_{S,v}$  do
14       foreach  $p \in \mathcal{C}_w \setminus T_w : p \notin PQ$  do
15         $PQ.enqueue(p);$  ▷ Priority is  $\omega(p)$ 
16       Mark  $w$  super-saturated;
```

---

pseudo-code is given in Algorithm 4, works as follows. The initialization phase is similar to algorithm SS-TOP- $k$ : we set up  $T_v$  to be equal the empty list, for each  $v \in V$ , and priority queue  $PQ$  to contain only path  $(r)$ , for the root vertex  $r \in V$ , with priority equal to zero. Then, until the queue  $PQ$  is not empty or  $|T_w| < k$  for some  $w \in V \setminus \{r\}$ , a path  $\Pi$ , terminating at some vertex  $v$ , is dequeued from  $PQ$  due to having minimum priority (i.e. weight) among those in  $PQ$  and an *iteration* of the algorithm is executed. In a generic iteration, if the conditional statement of line 6 evaluates to true then the execution proceeds as in algorithm SS-TOP- $k$  with two main differences: first, if the size of  $T_v$  is less than  $k$ , then path  $\Pi$  is added to  $T_v$  to be part of a solution; second, we enqueue into  $PQ$  simple path  $\Pi \oplus (u)$ , having  $\Pi$  as prefix and terminating at  $u$ , with priority equal to its weight  $\omega(\Pi \oplus (u)) = \omega(\Pi) + \omega(v, u)$ , only for the neighboring vertices of  $v$  in  $N(v)$  that are not *super-saturated*. A vertex  $x$  becomes *super-saturated* whenever the algorithm has computed a collection of top- $k$  simple shortest paths from  $r$  to  $w$  for each general predecessor of  $x$ , including  $x$  itself, in some solution.

In particular, initially, all vertices are not super-saturated. Then, when the conditional statement of line 6 evaluates

to false, i.e. when we extract a path  $\Pi = (r, \dots, v)$  and  $|T_v| \geq k$  (meaning we have found  $k$  paths whose weight is less or equal than  $\omega(\Pi)$ ) we distinguish two cases: either  $v$  is super-saturated or not. In the former case, no further enqueue operation is performed. In the latter, instead, we compute explicitly a set of general predecessors  $A_{S,v}$  of  $v$  and a collection  $\mathcal{C}_w \supseteq T_w, \forall w \in A_{S,v}$ . On the one hand, this is easy to achieve for vertices that are saturated, for which we can show  $T_w = \mathcal{C}_w$  for some  $\mathcal{C}_w \in \Gamma_w$ ; on the other hand, it requires to compute a collection of top- $k$  simple shortest paths from  $r$  to  $w$  (a solution to SP-kSiSP for pair  $(r, w)$ ) for each vertex  $w$  in  $A_{S,v}$  that is not saturated. For the easiness of understanding, we indicate this solution by  $\mathcal{C}_w$  as well, with a little abuse of notation. Such solution can be obtained, for instance, by any algorithm (e.g. Yen's) that solves the SP-kSiSP problem for pair  $(r, w)$ . Once set  $A_{S,v}$  and a collection  $\mathcal{C}_w \supseteq T_w, \forall w \in A_{S,v}$  is available, we enqueue into  $PQ$  any path  $p \in \mathcal{C}_w \setminus T_w$  with priority  $\omega(p)$  and mark every vertex in  $A_{S,v}$  super-saturated. Observe that, whenever we need to add a path  $p$  to  $PQ$  during the algorithm, we check that  $p \notin PQ$  before performing any enqueue operation. This is necessary to avoid inserting twice a same path in the queue and to achieve correct results, as proved further in this section.

In Algorithm 5 we provide a more detailed description of algorithm BOUND-SS-TOP- $k$ . The determination of set  $A_{S,v}$  consists in an iterative implementation of the computation of the transitive closure of the relation predecessors (see Def. III.2) that uses a membership set SEEN in order to terminate and not to visit a vertex more than once (note that a vertex can be a predecessor of many vertices in a solution). In such description we use notation  $Yen(G, r, x, k)$  to denote a call to Yen's algorithm that returns a collection of top- $k$  simple shortest paths from a pair  $(r, x)$  of vertices in graph  $G$ . Super-saturated vertices are stored in a set named S-SAT. We are able to show the correctness of Algorithm 4 as stated in the following result. We provide a sketched proof: the complete proof of correctness is omitted due to space limitations and will be given in a full version of this paper.

*Theorem IV.3.* Algorithm BOUND-SS-TOP- $k$  solves SS-kSiSP.

*Proof.* In what follows, we use  $T_u^t$  to represent the content of data structure  $T_u$  at the  $t$ -th iteration of the main loop at line 4. Similarly, we denote by  $PQ^t$  the priority queue  $PQ$  after the  $t$ -th execution of such loop. We assume  $t = 0$  if no iteration of the loop has been performed and  $t = 1$  when one iteration has been performed. If  $A$  is a collection of paths, we call  $L(A)$  the *profile* of  $A$ , that is the ordered list of the weights of the paths in  $A$ . Hence, a pair of profiles can be naturally compared by using the lexicographical ordering, i.e. for any two profiles  $L(A')$  and  $L(A'')$  of two collections of paths  $A'$  and  $A''$  we can write comparison expressions such as  $L(A') \geq L(A'')$ ,  $L(A') < L(A'')$  or  $L(A') = L(A'')$ .

First of all, we observe that  $L(\mathcal{C}_u) = L(\mathcal{C}'_u)$  even if  $\mathcal{C}_u$  and  $\mathcal{C}'_u$  are different solutions to SS-kSiSP for a given vertex  $u$ , i.e.  $\mathcal{C}_u \in \Gamma_u$  and  $\mathcal{C}'_u \in \Gamma_u$  with  $\mathcal{C}_u \neq \mathcal{C}'_u$ . Moreover, if



$L(T_u^t) \leq L(C_u)$  then all paths in  $T_u^t$  must belong to a solution in  $\Gamma_u$  for vertex  $u$ . Finally, if  $L(T_u^t) = L(C_u)$  then  $T_u^t$  belongs to a solution  $C_u$  for  $u$ .

The proof is by induction on the number  $t$  of times that the main loop at line 4 is performed. We use  $P_{ru}$  to denote a generic path from a vertex  $r$  to a vertex  $u$  in the graph. We prove that, at each time  $t$  and for each vertex  $w \in V$ , the following two properties hold:

- 1)  $L(T_w^t) \leq L(C_w)$ ;
- 2) if there exists a path  $P_{rw}$  such that  $P_{rw} \notin T_w^t$  and  $L(T_w^t \cup \{P_{rw}\}) \leq L(C_w)$  then there exists a prefix  $P'$  of  $P_{rw}$  such that  $P' \in PQ^t$ .

On the one hand, the first property ensures that paths in  $T_w^t$  computed until time  $t$  are correct, i.e. paths in  $T_w^t$  belong to a solution to SS-kSiSP for each vertex  $w \in V$ , and that such paths are found in the due non-decreasing order of weight. The second property, on the other hand, guarantees that any next path that has to be added to  $T_w^t$  to complete a solution in  $\Gamma_w$  has a prefix in  $PQ^t$  for each vertex  $w$ . Note that by the definition of prefix, any path is a prefix of itself. If we focus on time  $t = 0$ , we have that  $L(T_w^0) = [ ]$  for each vertex  $w \in V$  while  $PQ^0 = \{(r)\}$ . Hence, both properties 1) and 2) hold since path  $(r)$  is prefix of any path in a solution of SS-kSiSP.

Now we assume that properties 1) and 2) are true at time  $t$  and show that they remain true after the execution of the loop for the  $(t + 1)$ -th time. We first consider the case when the condition controlling the main loop is true, then  $PQ^t$  is not empty. Let  $P_{rv}$  the path dequeued at line 5. Notice that, like  $P_{rv}$ , any path in  $PQ$  can be inserted either at line 10 or at line 15. Call *normal* the first kind of insertion and *exceptional* the second one.

We first prove that property 1) holds at time  $t + 1$ . We distinguish two cases. If  $P_{rv}$  was inserted in  $PQ$  as a consequence of an exceptional insertion, then we have  $L(T_v^{t+1}) = L(T_v^t \cup \{P_{rv}\}) \leq L(C_v)$ , as exceptional insertions are dictated by an execution of Yen's algorithm which returns a solution  $C_v$  for any vertex  $v$ . Moreover, since  $L(T_w^{t+1}) = L(T_w^t)$  for any other vertex  $w \neq v$ , then property 1) holds. Consider, instead, that  $P_{rv}$  was inserted in  $PQ$  as a consequence of a normal insertion. If  $|L(T_v^t)| = |L(C_v)|$  it follows that property 1) clearly remains true as  $P_{rv}$  is not added to  $T_v^t$ . If, instead,  $|L(T_v^t)| < |L(C_v)|$ , we suppose, by contradiction, that  $L(T_v^t \cup \{P_{rv}\}) \not\leq L(C_v)$ . Since  $|L(T_v^t)| < |L(C_v)|$ , there must exist a path  $P'_{rv}$  such that  $L(T_v^t \cup \{P'_{rv}\}) \leq L(C_v)$  and we can distinguish two cases, since  $\omega(P_{rv}) \neq \omega(P'_{rv})$ : either (a)  $\omega(P_{rv}) > \omega(P'_{rv})$ ; or (b)  $\omega(P_{rv}) < \omega(P'_{rv})$ .

For case (a): since  $L(T_v^t \cup \{P'_{rv}\}) \leq L(C_v)$ , then, by property 2), there exists a prefix  $P'$  of  $P'_{rv}$  such that  $P' \in PQ^t$ . Since  $P'$  is a prefix of  $P'_{rv}$  we have  $\omega(P') \leq \omega(P'_{rv})$ . Moreover  $\omega(P'_{rv}) < \omega(P_{rv})$ , which implies that  $P'$  should have been extracted from  $PQ$  in place of  $P_{rv}$ , since we extract in order of weight.

For case (b): If  $\omega(P_{rv}) < \omega(P'_{rv})$  then, by property 1) at time  $t$ ,  $P_{rv}$  is already in  $T_v^t$  and then the algorithm has inserted

$P_{rv}$  in  $PQ$  twice. This leads to a contradiction. In fact, first of all, there cannot be two insertions of  $P_{rv}$  that are both normal, since this would imply that prefix  $P''$  of  $P_{rv}$ , obtained by removing  $v$  from  $P_{rv}$  would have been inserted twice in  $PQ$  at line 7, which in turn implies  $L(T_v^t) \not\leq L(C_v)$ . Moreover, if first insertion of  $P_{rv}$  is exceptional and the second one is normal, then  $v$  is marked *super-saturated* by the algorithm and no path terminating at  $v$  is inserted again in  $PQ$  (see lines 9 and 11). Finally, if the first insertion of  $P_{rv}$  is normal and the second is exceptional, we obtain a contradiction since an exceptional insertion concerns only paths in some solution  $C_v$  that are not already in  $T_v$  (see line 14).

We now prove that property 2) holds at time  $t + 1$ .

Assume first that the condition at line 6 is true. Hence, path  $P_{rv}$  is removed from  $PQ$  but, for each non super-saturated vertex  $u$  in  $N(v) \setminus V(P_{rv})$ , path  $P_{ru} = P_{rv} \oplus (u)$ , is inserted in  $PQ$ . Then, if  $u$  is not super-saturated, we have that  $P_{ru}$  is now in  $PQ^{t+1}$  and it is a prefix of any path  $P_{rw}$  from the root vertex to a vertex  $w$  that had  $P_{rv}$  as prefix. Vice versa, if  $u$  is super-saturated, the algorithm correctly does not enqueue  $P_{ru}$  in  $PQ^t$ . Indeed,  $P_{ru} \notin C_u$  and if there were a path  $P_{rw}$ , as in property 2), having  $P_{ru}$  as prefix, then by Lemma III.2 there would exist a vertex  $y$  in  $P_{rw}$  such that a path  $P_{ry} = P_{ru} \oplus P_{uy}$  is in  $C_y$ , for some solution  $C_y$ . By the same lemma,  $y$  is a general predecessor of  $u$ , hence path  $P_{ry}$  has been determined and added to  $PQ$  via an exceptional insertion when  $u$  has become super-saturated. Path  $P_{ry}$  is a prefix of  $P_{rw}$  and, since  $\omega(P_{ry}) > \omega(P_{rv})$  and  $PQ$  is a priority queue,  $P_{ry}$  is still in  $PQ^{t+1}$ .

Now consider the case when the condition of line 6 is false. If  $v$  is super-saturated, then simply  $P_{rv}$  is removed from  $PQ$  and no path is inserted. However, if  $P_{rv}$  was a prefix of a path  $P_{rw}$ , as above, any prefix of  $P_{rw}$  has been computed and inserted as exceptional in  $PQ$  when  $v$  has become super-saturated. If  $v$  is not super-saturated and  $P_{rv}$  is a prefix of a path  $P_{rw}$ , again by Lemma III.2 there exists a vertex  $y$  in  $P_{rw}$  such that  $P_{ry} = P_{rv} \oplus P_{vy}$  is in  $C_y$ , for some solution  $C_y$ . As  $y$  is in the set of the general predecessors of  $v$  (indicated as  $A_{S,v}$  in the algorithm), then path  $P_{ry}$  is inserted in  $PQ^{t+1}$  in this iteration of the loop and this suffices to show that property 2) holds.

To complete the proof we now consider the case when the condition controlling the main loop (see line 4) is false at time  $t_e$ . If  $\forall w \neq r : |T_w^{t_e}| = k$  then, by property 1), the algorithm has computed a correct solution for SS-kSiSP. If instead  $PQ^{t_e}$  is empty, since there is no path in  $PQ^{t_e}$  then the conclusion of property 2) is false which implies that the premise is also false. Hence, for each path  $P_{rw}$  we have either  $P_{rw} \in T_w^{t_e}$  or  $L(T_w^{t_e} \cup \{P_{rw}\}) \not\leq L(C_w)$ . In both cases it follows that  $L(T_w^{t_e}) = L(C_w)$  and therefore that the algorithm has computed a correct solution for SS-kSiSP.  $\square$

For the sake of clarity, it is easy to observe how applying algorithm BOUND-SS-TOP- $k$  to the input graph of Fig. 4, with  $k = 3$ , prevents the discovery of an exponential number of

paths, which instead happens if SS-TOP- $k$  is used. In fact, when a fourth path to  $c_2$  is extracted and we have  $|T_{c_2}| = k$ , the algorithm searches for non-saturated general predecessors, and enqueues only the paths that are necessary to complete a solution for such vertices.

**Theorem IV.4.** Algorithm BOUND-SS-TOP- $k$  runs in  $\mathcal{O}(k(m \log km + n^2 \text{SP}(n, m)))$  time in the worst case, where  $\text{SP}(n, m)$  is the worst case running time of a shortest-path algorithm in an  $n$ -vertex  $m$ -edge graph.

*Proof.* Observe that number of times the loop in line 4 of algorithm 4 is upper bounded by the number of insertions in the queue  $PQ$ , which is given by the sum of the number of normal insertions and the number of exceptional insertions. The former is  $\mathcal{O}(km)$  since, for each vertex  $v \in V$ , we perform at most  $k\delta_v$  insertions (see line 10), where  $\delta_v = |N(v)|$  is the degree of the vertex. By summing up for all vertices, we have  $\sum_{v \in V} k\delta_v = k \sum_{v \in V} \delta_v = 2km = \mathcal{O}(km)$ . The latter, instead, is  $\mathcal{O}(nk)$  since, in the worst case, the size of any  $A_{S,v}$  is at most  $n-1$  and we execute Yen's algorithm to compute and enqueue at most  $k$  paths per vertex of  $A_{S,v}$ . Now, for each insertion, the algorithm enqueues a path in  $PQ$  which is a priority queue whose size is at most the total number of insertions, bounded by  $\mathcal{O}(nk + mk)$ . Thus, each insertion requires  $\mathcal{O}(\log(k(n + m)))$  operations and hence the running time due to insertions is  $\mathcal{O}(k(m + n) \log(k(m + n)))$ . Similarly, the running time for corresponding dequeue operations is  $\mathcal{O}(k(m + n) \log(k(m + n)))$ . Moreover, for each of the at most  $n - 1$  vertices for which we perform at most  $k$  exceptional insertions, we need to run once Yen's algorithm which costs a total of  $\mathcal{O}(n \cdot kn \cdot \text{SP}(n, m)) = \mathcal{O}(kn^2 \text{SP}(n, m))$  worst-case running time [8]. Therefore, the total running time of the algorithm is  $\mathcal{O}(k(m + n) \log(k(m + n)) + kn^2 \text{SP}(n, m)) = \mathcal{O}(km \log km + kn^2 \text{SP}(n, m)) = \mathcal{O}(k(m \log km + n^2 \text{SP}(n, m)))$ .  $\square$

Observe that, for unweighted graphs,  $\text{SP}(n, m)$  is the running time of the breadth-first-search algorithm, which is  $\mathcal{O}(m + n)$ . Vice versa, for weighted graphs,  $\text{SP}(n, m)$  is upper bounded by the running time of Dijkstra's algorithm, i.e.  $\mathcal{O}(m + n \log n)$ . Thus, we can derive the following corollary:

**Corollary IV.4.1.** The worst-case running time of algorithm BOUND-SS-TOP- $k$  is upper bounded by the time taken by  $\Theta(n)$  executions of Yen's algorithm in the worst-case.

*Proof.* Algorithm BOUND-SS-TOP- $k$  runs for  $T(n, m, k) = \mathcal{O}(k(m \log km + n^2 \text{SP}(n, m)))$  worst-case time. If we expand the product we have that  $T(n, m, k) = \mathcal{O}(k(m \log km + n^2 \text{SP}(n, m))) = \mathcal{O}(km \log km + kn^2 \text{SP}(n, m))$ . Since  $\text{SP}(n, m) \in \mathcal{O}(m + n \log n)$  (for weighted graphs) we obtain that  $T(n, m, k) = \mathcal{O}(km \log km + kn^2(m + n \log n))$ . Moreover, notice that, in the worst case,  $k$  can be exponential in  $n$ , therefore  $\log km = \mathcal{O}(n + \log m)$  and hence  $km \log km = \mathcal{O}(km(n + \log m))$ . This implies  $\mathcal{O}(km(n + \log m))$  is upper bounded by  $\mathcal{O}(kn^2(m + n \log n))$  in any graph such that  $m = \Omega(n)$  and hence by the time taken by  $n$  executions of Yen's algorithm in the worst-case.  $\square$

---

#### Algorithm 5: Detailed description of algorithm 4

---

**Input:** Graph  $G = (V, E, \omega)$ , root  $r \in V$ , integer  $k > 0$ .  
**Output:** Top- $k$  simple shortest paths  $T_v$  from  $r$  to each  $v \in V \setminus \{r\}$ .

```

1  $PQ \leftarrow \emptyset;$  ▷ Empty priority queue
2 foreach  $v \in V$  do  $T_v \leftarrow [];$  ▷ Empty lists
3  $PQ.enqueue((r));$  ▷ Priority is  $\omega((r)) = 0$ 
4  $S-SAT \leftarrow \{r\};$  ▷ Set of super-saturated vertices
5 while  $PQ \neq \emptyset \wedge \exists w \neq r : |T_w| < k$  do
6    $\Pi = (r, \dots, v) \leftarrow PQ.dequeueMin();$  ▷ Path w/ min priority
7   if  $|T_v| < k$  then
8     add  $\Pi$  to  $T_v;$ 
9     foreach  $u \in N(v) \setminus V(\Pi)$  do
10      if  $u \notin S-SAT$  and  $\Pi \oplus (u) \notin PQ$  then
11         $PQ.enqueue(\Pi \oplus (u));$  ▷ Priority is  $\omega(\Pi) + \omega(v, u)$ 
12    else if  $v \notin S-SAT$  then
13       $A_{S,v} \leftarrow \emptyset;$ 
14       $Q \leftarrow \{v\};$  ▷ Add  $v$  to empty FIFO queue
15       $SEEN \leftarrow \{v\};$ 
16      while  $Q \neq \emptyset$  do
17         $x \leftarrow Q.pop();$  ▷ Dequeue head of queue
18        if  $x \notin S-SAT$  then
19          if  $|T_x| < k$  then  $C_x \leftarrow V(\text{Yen}(G, r, x, k));$ 
20          else  $C_x \leftarrow T_x;$ 
21           $A_{S,v} \leftarrow A_{S,v} \cup V(C_x);$ 
22          foreach  $p \in C_x \setminus T_x : p \notin PQ$  do
23             $PQ.enqueue(p);$  ▷ Priority is  $\omega(p)$ 
24          foreach  $w \in V(C_x)$  do
25            if  $w \notin SEEN$  and  $w \notin S-SAT$  then
26               $SEEN \leftarrow SEEN \cup \{w\};$ 
27               $Q.append(w);$ 
28               $S-SAT \leftarrow S-SAT \cup \{w\};$ 
29             $S-SAT \leftarrow S-SAT \cup \{x\};$ 

```

---

## V. EXPERIMENTAL EVALUATION

In this section, we present the results of an extensive experimental evaluation, whose main objective is to assess the performance of our new algorithms for SS-kSiSP, namely SS-TOP- $k$  and BOUND-SS-TOP- $k$ . To this aim, we implement our methods and compare them to an implementation of the reference approach to address SS-kSiSP, that is the application of Yen's algorithm for all  $n - 1$  pairs of vertices  $(r, v)$  for a given root  $r \in V$  and for each  $v \in V \setminus \{r\}$  of a given graph  $G = (V, E, \omega)$ . We call SS-YEN such generalization of Yen's algorithm that solves SS-kSiSP. We remark that SS-YEN and BOUND-SS-TOP- $k$  are on par in terms of time complexity hence the ultimate purpose of our experimental evaluation is to establish which is the best performing solution to adopt in practice. We also emphasize that we implement and test variants of all algorithms able to handle both undirected and unweighted graphs by either properly removing orientations or by assigning unitary weights to edges or by replacing Dijkstra's like behaviors with breadth-first-search like ones, similarly to other studies on shortest paths [1], [3], [7], [13].

### A. Test Environment

Our test environment is based on NetworKit [21] and NetworkX [22], two widely adopted toolkits for testing graph algorithms and performing network analytics tasks at scale. Specifically, through such toolkits, we implement SS-TOP- $k$ , BOUND-SS-TOP- $k$  and SS-YEN algorithms in Python. Our implementation of SS-YEN was inspired by the very effective implementation of Yen’s algorithm provided within NetworkX, that considers bidirectional shortest-path computations to achieve very fast running times. All our trials have been executed, through the Python 3.10 interpreter, under Linux (Kernel 6.8.0-47), on a workstation equipped with an Intel® Xeon® CPU E5-2643 3.40GHz and 128 GB of RAM. As input to our experiments, following other empirical studies on graph algorithms [1], [3], [5], [23], we employ a large graph dataset, including both real-world instances, of heterogeneous sizes and topologies, taken from publicly available repositories [24], [25] and artificial graphs, constructed via well-established parametrical random generators, such as, e.g., *Erdős-Rényi* or *Barabási-Albert* models [26]. Details on used inputs are given in Table I.

### B. Executed Tests

For each of the mentioned graphs, and for each value of parameter  $k \in \{2, 4, 8, 16\}$  we execute both SS-TOP- $k$ , BOUND-SS-TOP- $k$ , and SS-YEN and measure the running time to find a solution to SS-kSiSP. Depending on the graph being directed/undirected or weighted/unweighted, we use the corresponding implementation of the two methods able to handle the corresponding case. The choice of values of parameter  $k$  to consider was inspired by applications of SS-kSiSP, where the number of sought simple shortest paths rarely exceeds few tens [1], [13], [27], and by other studies on SS-kSiSP and variants [1], [7], [8]. Nonetheless, to assess the scalability properties of the two methods against  $k$ , we complete our experimental evaluation by testing algorithms BOUND-SS-TOP- $k$  and SS-YEN for a selection of graphs when  $k \in \{2^i\}_{i=1,2,\dots,8}$ . Details about graphs considered for this part of our experimentation are given in Table II. In order to reduce the bias due to the considered root vertex, for each combination of graph and  $k$  we execute 3 runs for 3 different roots, selected uniformly at random among the vertices of the graph, and compute average values of running times. Note that preliminary experimentation showed SS-TOP- $k$ ’s running time to be very large in some graphs, most likely due to the presence of components resembling the topological configuration of Fig. 4, hence we exclude such algorithm from our experimental framework and, from here onward, we focus on BOUND-SS-TOP- $k$  and SS-YEN only. Notice also that, for the sake of validity, in each trial we check the correctness of solutions to SS-kSiSP, computed by each algorithm, by testing that each resulting collection of paths, from the root to each vertex, matches, in terms of profile, that obtained by the standard, consolidated implementation of Yen’s algorithm offered by NetworkX.

### C. Analysis

The results of our experimentation for  $k \in \{2, 4, 8, 16\}$  are summarized in Table III where, for each graph and  $k$ , we report the average running time to solve an instance of SS-kSiSP by SS-YEN and BOUND-SS-TOP- $k$ , resp., and the average *speed-up*, i.e. the average ratio of the running time of SS-YEN to that of BOUND-SS-TOP- $k$ , which provides a measure of how the latter is faster than the former at finding solutions to SS-kSiSP. For the sake of readability, in what follows we use acronyms SS-Y and B-SS to refer, resp, to algorithms SS-YEN and BOUND-SS-TOP- $k$ . All running times are expressed in seconds<sup>1</sup>. The main conclusion that can be derived by the data of Table III is that algorithm B-SS unquestionably outperforms the state-of-the-art w.r.t. the running time to solve an instance of SS-kSiSP. Specifically, the observed average speed-up ranges from a minimum of 1.15 (observed on input DER for  $k = 16$ ) to a maximum of several thousands (see e.g. graphs LUX for  $k = 2$  or graph CAI when  $k = 8$ ) with the large majority of experiments showing algorithm B-SS being orders of magnitude faster than SS-Y. This conclusion is supported also by the alternative views of the data of Table III given in Figs 5, 6 and 7. In particular, in Fig. 5 we plot,

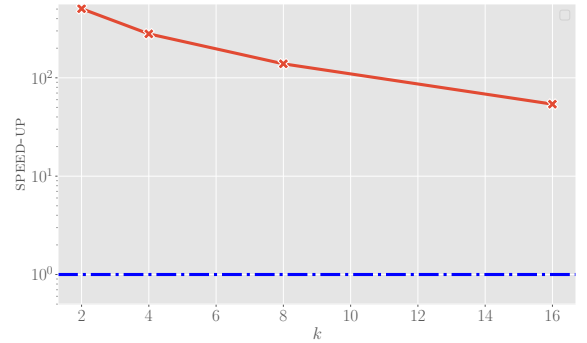


Fig. 5. Average speed-up over all graphs as a function of  $k$ .

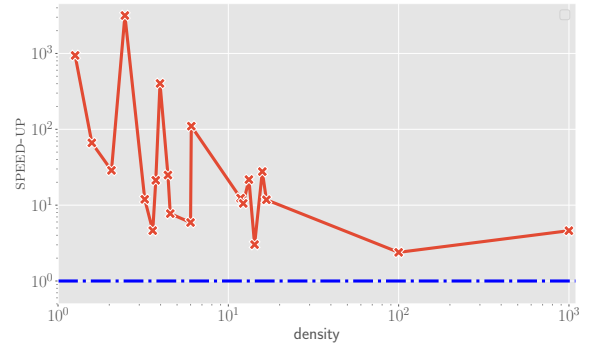


Fig. 6. Average speed-up over all graphs as a function of graph density.

on a log-scaled axis to magnify the differences, the average measured speed-up over all graphs as a function of  $k$ . We notice that the speed-up, offered by B-SS, tends to decrease as

<sup>1</sup>Source code available at <https://shorturl.at/1dkoc>.

Dataset	Short	Type	$ V $	$ E $	DIRECTED	WEIGHTED	$d_{AVG}$	$d_{MED}$	$d_{MAX}$	SYNTHETIC
LINUX	LIN	BLOGGING	913	4162	●	○	4.56	2	61	○
CA-GrQC	CAG	COLLABORATION	4158	13422	○	○	6.46	3	81	○
OREGON-AS	ORE	AUTONOM. SYSTEM	10670	22002	○	○	4.12	2	2312	○
CAIDA	CAI	ETHERNET	32000	40204	○	●	2.51	2	203	○
P2P-GNUTELLA	GNU	PEER2PEER	14149	50916	●	○	3.60	3	32	○
LUXEMBOURG	LUX	ROAD NETWORK	30647	75546	●	●	2.47	3	9	○
D-ERDŐS-RÉNYI	DER	RANDOM UNIFORM	1000	100025	●	●	100.03	100	134	●
WIKI-VOTE	WIV	VOTING	7066	100736	○	○	28.51	4	1065	○
EMAIL EU	EMA	EMAIL	34203	151132	●	○	4.42	1	715	○
BRIGHTKITE	BRI	LOCATION	56739	212945	○	○	7.51	2	1134	○
D-BARABÁSI-ALB.	DBR	RANDOM POWER LAW	50000	299964	○	○	12.00	8	950	●
BARABÁSI-ALB.	BAR	RANDOM POWER LAW	252765	400534	○	●	3.17	2	1966	●
ARXIV	ARX	CITATION	34401	420784	●	○	24.46	15	846	○
EPINIONS	EPI	REVIEWS	41441	693507	●	○	16.73	3	1820	○
SLASHDOT	SLD	BLOG/NEWS	71307	841201	●	○	11.80	2	2510	○
WIKITALK	WKT	CHAT NETWORK	111881	1477893	●	○	13.21	1	7973	○
YOUTUBE	YTB	MEDIA	453956	1801420	○	○	7.94	2	26229	○
AMAZON	AMZ	RATING	403364	2443311	○	○	12.11	10	2752	○
ERDŐS-RÉNYI	ERD	RANDOM UNIFORM	4000	3997931	○	●	1998.97	1999	2102	●
FACEBOOK	FAC	SOCIAL NETWORK	1238865	19611264	○	○	31.66	10	4866	○

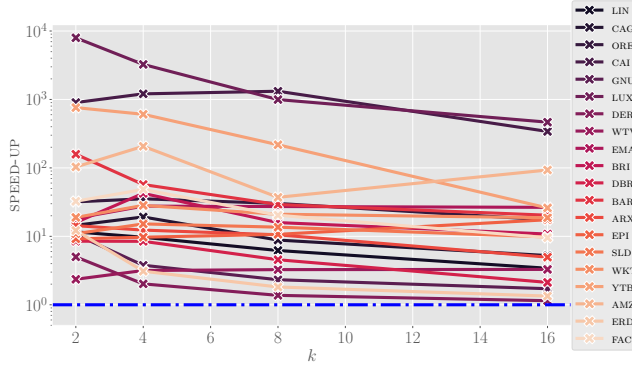
TABLE I

OVERVIEW OF INPUT DIGRAPHS: FIRST THREE COLUMNS PROVIDE DATASET NAME, ACRONYM, AND TYPE; 4TH AND 5TH COLUMNS SHOW NUMBER OF VERTICES AND ARCS, RESP.; COLUMNS 6TH AND 7TH REPORT WHETHER THE GRAPH IS DIRECTED AND WEIGHTED (● = TRUE, ○ = FALSE); COLUMNS 8TH, 9TH AND 10TH PROVIDE AVERAGE, MEDIAN AND MAXIMUM (OUTGOING) DEGREE, RESP., WHILE THE LAST COLUMN INDICATES WHETHER THE GRAPH IS SYNTHETIC (●) OR REAL (○). INPUTS ARE SORTED BY  $|E|$ , NON-DECREASING.

Dataset	Short	Type	$ V $	$ E $	DIRECTED	WEIGHTED	$d_{AVG}$	$d_{MED}$	$d_{MAX}$	SYNTHETIC
MAY-FAA	MAY	FLIGHT NETWORK	792	1898	●	○	2.40	1	21	○
ITALY	ITA	TRAIN NETWORK	1329	3862	●	●	2.91	3	13	○
BITCOIN	BTC	FINANCIAL	5875	21489	○	●	7.32	2	795	○
GOOGLE	GOO	WEB INDEX	12354	164046	●	○	13.28	10	120	○

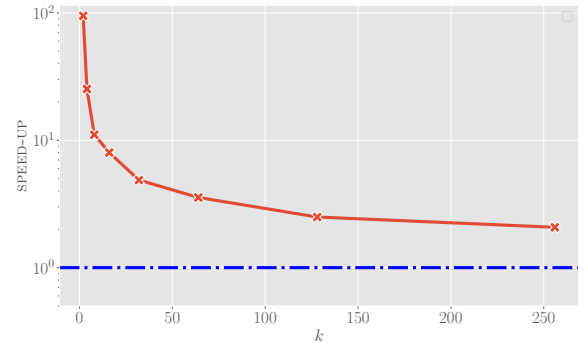
TABLE II

OVERVIEW OF INPUT DIGRAPHS USED FOR ASSESSING THE SCALABILITY AGAINST  $k$ . COLUMNS ARE AS IN TABLE I.

Fig. 7. Average speed-up per graph as a function of  $k$ .

$k$  increases. However, even in the largest graphs and for the largest considered values of  $k$ , B-SS remains always faster than SS-Y and often hundreds of times faster, as also shown by the plots, per graph, of Fig. 7. Such effectiveness is confirmed also by the results of the experimentation for  $k \in \{2^i\}_{i=1,2,\dots,8}$ , shown in Table IV and Figs 8–9, which suggest that even for very large  $k$ , B-SS remains the method that should be preferred in practice to solve SS-kSiSP. Finally, if we focus on Fig. 6, where we report how the average speed-up changes as a function of graph *density* (the ratio of the number of edges to the number of vertices), we notice that increases of the

density of the graph tend to influence negatively the measured speed-up, which however remains significantly larger than one (shown through a blue line) even for the densest instance. Moreover, it is worth remarking that dense graphs, say with a density above  $\approx 10$ , very infrequently in practice [1], [5], [28] therefore the identified trend is of theoretical interest but of little significance w.r.t. the objective of the experimental study at hand.

Fig. 8. Average speed-up over all graphs as a function of  $k$  when  $k$  scales up from 2 to 256.

graph	$k = 2$			$k = 4$			$k = 8$			$k = 16$		
	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP
LIN	0.21	0.02	11.62	0.73	0.07	9.69	1.50	0.25	6.21	3.48	1.01	3.42
CAG	2.37	0.16	14.52	6.31	0.33	19.19	13.52	1.61	8.82	34.71	6.93	5.24
ORE	8.47	0.27	31.91	22.79	0.65	35.34	81.08	4.11	30.24	150.35	10.35	17.40
CAI	809.56	1.00	901.71	2417.18	2.40	1207.32	5402.10	4.52	1318.57	11172.46	113.86	340.96
GNU	19.95	1.91	10.63	58.33	15.51	3.78	146.48	63.24	2.32	330.45	192.79	1.72
LUX	25233.54	3.08	7953.22	52575.95	22.33	3251.14	189334.10	236.13	995.86	254976.42	547.32	465.86
DER	32.96	6.57	5.01	103.16	51.29	2.02	278.52	202.56	1.37	619.84	540.91	1.15
WIV	2.88	1.23	2.36	8.06	2.54	3.17	18.29	5.60	3.26	49.69	15.10	3.29
EMA	15.97	0.88	18.36	50.55	1.82	27.62	115.65	4.23	27.26	320.15	12.00	26.53
BRI	50.25	3.17	15.93	282.77	6.67	42.45	369.34	23.32	15.88	888.25	83.07	10.80
DBR	41.53	4.89	8.49	123.97	14.77	8.46	287.91	63.65	4.55	679.34	323.49	2.12
BAR	27952.38	178.51	158.40	84794.61	1485.57	57.16	210048.88	7196.32	29.19	565406.15	27784.33	20.35
ARX	86.52	6.04	14.32	159.56	12.92	12.34	408.64	38.95	10.53	964.04	197.72	4.96
EPI	40.25	4.35	9.20	94.31	9.68	9.71	234.33	21.81	10.69	1042.70	53.76	17.47
SLD	69.82	6.39	10.94	199.10	13.11	15.19	488.48	36.09	13.62	1058.14	109.64	9.80
WKT	210.55	11.17	18.89	694.11	24.20	28.32	1083.53	51.53	21.04	2376.39	127.45	18.70
YTB	23922.69	30.93	760.11	55878.71	106.35	606.97	129191.89	903.62	217.70	221415.22	8513.77	26.01
AMZ	5187.75	50.24	103.26	20085.63	97.08	206.91	31561.91	848.90	37.18	96201.26	1032.81	93.15
ERD	18438.85	1510.75	12.17	57502.40	18569.73	3.09	149296.23	81859.59	1.82	304514.72	227387.14	1.34
FAC	12935.07	398.52	32.54	39244.64	811.92	48.43	98925.03	5596.61	20.35	155744.86	16963.92	9.55

TABLE III  
RESULTS OF THE EXECUTIONS OF SS-Y AND B-SS FOR  $k \in \{2, 4, 8, 16\}$ .

graph	$k = 2$			$k = 4$			$k = 8$			$k = 16$		
	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP
MAY	0.28	0.02	16.72	0.86	0.07	12.45	1.39	0.38	3.84	3.27	1.34	2.48
ITA	15.61	0.06	261.25	33.61	0.71	48.07	70.35	3.82	18.84	182.21	10.43	17.52
BTC	34.18	0.44	77.85	93.43	3.43	27.86	255.63	16.05	15.99	538.53	66.83	8.16
GOO	13.06	0.54	24.23	37.64	2.94	12.83	81.29	14.98	5.58	189.63	48.66	3.91

graph	$k = 32$			$k = 64$			$k = 128$			$k = 256$		
	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP	SS-Y	B-SS	SPEED-UP
MAY	0.28	0.02	16.72	0.86	0.07	12.45	1.39	0.38	3.84	3.27	1.34	2.48
ITA	15.61	0.06	261.25	33.61	0.71	48.07	70.35	3.82	18.84	182.21	10.43	17.52
BTC	34.18	0.44	77.85	93.43	3.43	27.86	255.63	16.05	15.99	538.53	66.83	8.16
GOO	13.06	0.54	24.23	37.64	2.94	12.83	81.29	14.98	5.58	189.63	48.66	3.91

TABLE IV  
RESULTS OF THE EXECUTIONS OF SS-Y AND B-SS FOR  $k \in \{2, 4, 8, 16, 32, 64, 128, 256\}$ .

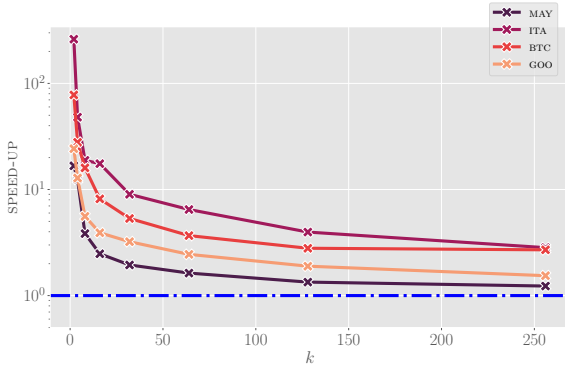


Fig. 9. Average speed-up per graph as a function of  $k$  when  $k$  scales up from 2 to 256.

#### D. Limitations of our Approach

Two main limitations can be identified in our contribution. First and foremost, our algorithm must rely on selected solution to the SP-kSiSP problem under specific circumstances

(see line 19 of Algorithm 5). In such cases, we employ Yen's algorithm to complete the determination of the entire set  $A_{S,v}$  for some vertex  $v \in V$  that is not saturated; hence we do not consider and discard any partial solution to SS-kSiSP computed by B-SS for such vertex. It would be interesting to investigate whether such partially computed solutions can be exploited to further accelerate the computation of  $A_{S,v}$  in these cases. Second, on a related note, in our experimental evaluation we do not consider heuristics, without guarantees, that have been designed to accelerate Yen's algorithm [8] and use, as building block for algorithm SS-Y we used as comparison, the version of Yen's algorithm that performs bidirectional shortest paths computations [22]. This choice was dictated by the complexity of implementations of such heuristics and by the fact that their running times were showed to be faster than those of Yen's algorithm either by relatively small factors, or only in specific instances, or at the price of large memory overheads. On the one hand, incorporating such methods in our experimental settings could be beneficial to understand whether the speed-up offered by algorithm B-SS is preserved even against accelerated versions of Yen's algorithm. On the

other hand, we expect also the running time of B-SS would be reduced by employing the same versions of Yen’s algorithm in line 19 of Algorithm 5. In fact, in our experimentation we measured the number of times line 19 of Algorithm 5 is executed (data are omitted due to space limitations), and observed that the running time of B-SS decreases when this number increases. Hence, we expect that using a faster single-pair algorithm will induce a corresponding reduction of the running time of B-SS, and we are confident the conclusions of our experimental evaluation would remain roughly unchanged even by considering heuristically accelerated versions of Yen’s algorithm.

## VI. CONCLUSION AND FUTURE WORK

In this paper we have studied the Single-Source Top- $k$  Simple Shortest Paths problem. We have designed the first algorithm that is natively designed to address such problem in polynomial time. We have shown our solution is on par, in terms of time complexity, with the reference method in the literature to attack the considered problem, i.e. executing Yen’s algorithm with the source paired to all other vertices as inputs. We have provided the results of an extensive experimental evaluation that showed how our new solution runs up to orders of magnitude faster than the state-of-the-art method.

As future work, we plan to address the limitations highlighted in Section V-D, thus: (i) expanding the experimental framework to consider refinements and heuristics to accelerate Yen’s algorithm and more and larger input graphs; (ii) investigating whether partial solutions to SS- $k$ SiSP, computed throughout the execution of algorithm BOUND-SS-TOP- $k$ , can be reused or exploited to further reduce its running time, especially when it comes to solve instances of SP- $k$ SiSP (see line 19 of algorithm 5). Moreover, with the same aim, it would be interesting to understand whether parallel sections can be incorporated in our solution. Finally, it would be certainly worth to study the relationships, computationally speaking, that exist between SS- $k$ SiSP and other variants of  $k$  shortest paths problems with the purpose of better characterizing their inherent complexity and possibly to design faster and more scalable algorithms.



## REFERENCES

- [1] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida, "Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 2–8. [Online]. Available: <https://doi.org/10.1609/aaai.v29i1.9154>
- [2] C. Zhang, A. Bonifati, H. Kapp, V. I. Haprian, and J. Lozi, "A reachability index for recursive label-concatenated graph queries," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 67–81. [Online]. Available: <https://doi.org/10.1109/ICDE55515.2023.00013>
- [3] M. D'Emidio, L. Forlizzi, D. Frigioni, S. Leucci, and G. Proietti, "Hardness, approximability, and fixed-parameter tractability of the clustered shortest-path tree problem," *J. Comb. Optim.*, vol. 38, no. 1, pp. 165–184, 2019. [Online]. Available: <https://doi.org/10.1007/s10878-018-00374-x>
- [4] S. Anirban, J. Wang, and M. S. Islam, "Experimental evaluation of indexing techniques for shortest distance queries on road networks," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 624–636. [Online]. Available: <https://doi.org/10.1109/ICDE55515.2023.00054>
- [5] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Robust distance queries on massive networks," in *Algorithms-ESA 2014: 22th Annual European Symposium, Wrocław, Poland, September 8-10, 2014. Proceedings 21*. Springer, 2014, pp. 321–333.
- [6] B. A. Miller, Z. Shafi, W. Ruml, Y. Vorobeychik, T. Eliassi-Rad, and S. Alfeld, "Attacking shortest paths by cutting edges," *ACM Trans. Knowl. Discov. Data*, vol. 18, no. 2, pp. 35:1–35:42, 2024. [Online]. Available: <https://doi.org/10.1145/3622941>
- [7] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput.*, vol. 28, no. 2, pp. 652–673, 1998. [Online]. Available: <https://doi.org/10.1137/S0097539795290477>
- [8] A. A. Zoobi, D. Coudert, and N. Nisse, "Finding the k shortest simple paths: Time and space trade-offs," *ACM J. Exp. Algorithmics*, vol. 28, pp. 1.11:1–1.11:23, 2023. [Online]. Available: <https://doi.org/10.1145/3626567>
- [9] D. Kurz and P. Mutzel, "A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph," in *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*, ser. LIPIcs, S. Hong, Ed., vol. 64. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 49:1–49:13.
- [10] J. Li, X. Xiong, L. Li, D. He, C. Zong, and X. Zhou, "Finding top-k optimal routes with collective spatial keywords on road networks," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 368–380. [Online]. Available: <https://doi.org/10.1109/ICDE55515.2023.00035>
- [11] Z. Luo, L. Li, M. Zhang, W. Hua, Y. Xu, and X. Zhou, "Diversified top-k route planning in road network," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 3199–3212, Jul. 2022. [Online]. Available: <https://doi-org.univqa.idm.oclc.org/10.14778/3551793.3551863>
- [12] H. Liu, C. Jin, B. Yang, and A. Zhou, "Finding top-k shortest paths with diversity," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 1761–1762. [Online]. Available: <https://doi.org/10.1109/ICDE.2018.00238>
- [13] A. D'Ascenzo and M. D'Emidio, "Top-k distance queries on large time-evolving graphs," *IEEE Access*, vol. 11, pp. 102 228–102 242, 2023. [Online]. Available: <https://doi.org/10.1109/ACCESS.2023.3316602>
- [14] U. Agarwal and V. Ramachandran, "Finding k simple shortest paths and cycles," in *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*, ser. LIPIcs, S. Hong, Ed., vol. 64. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 8:1–8:12. [Online]. Available: <https://doi.org/10.4230/LIPIcs.ISAAC.2016.8>
- [15] J. Y. Yen, "Finding the k shortest loopless paths in a network," *management Science*, vol. 17, no. 11, pp. 712–716, 1971.
- [16] Z. Gotthilf and M. Lewenstein, "Improved algorithms for the k simple shortest paths and the replacement paths problems," *Information Processing Letters*, vol. 109, no. 7, pp. 352–355, 2009. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S002001900800361X>
- [17] V. Vassilevska Williams and R. Williams, "Subcubic equivalences between path, matrix and triangle problems," in *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, 2010, pp. 645–654. [Online]. Available: <https://doi.org/10.1109/FOCS.2010.67>
- [18] 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*, ser. CIKM '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 509–518. [Online]. Available: <https://doi-org.univqa.idm.oclc.org/10.1145/1871437.1871504>
- [19] G. Feng, "Finding k shortest simple paths in directed graphs: A node classification algorithm," *Networks*, vol. 64, no. 1, pp. 6–17, 2014. [Online]. Available: <https://doi.org/10.1002/net.21552>
- [20] R. Sedgewick, *Algorithms in c, part 5: graph algorithms, third edition*, 3rd ed. Addison-Wesley Professional, 2001.
- [21] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "Networkit: A tool suite for large-scale complex network analysis," *Netw. Sci.*, vol. 4, no. 4, pp. 508–530, 2016.
- [22] A. Hagberg, P. Swart, and D. S. Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.
- [23] E. Angriman, A. van der Grinten, M. von Looz, H. Meyerhenke, M. Nöllenburg, M. Predari, and C. Tzovas, "Guidelines for experimental algorithmics: A case study in network analysis," *Algorithms*, vol. 12, no. 7, p. 127, 2019. [Online]. Available: <https://doi.org/10.3390/a12070127>
- [24] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*. AAAI Press, 2015, p. 4292–4293.
- [25] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," <http://snap.stanford.edu/data>, jun 2014.
- [26] B. Bollobás, *Random Graphs, Second Edition*, ser. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2011, vol. 73. [Online]. Available: <https://doi.org/10.1017/CBO9780511814068>
- [27] S. M. M. Rashid, M. E. Ali, and M. A. Cheema, "DeepAltTrip: Top-K Alternative Itineraries for Trip Recommendation," *IEEE Transactions on Knowledge & Data Engineering*, vol. 35, no. 09, pp. 9433–9447, Sep. 2023. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/TKDE.2023.3239595>
- [28] D. Coudert, A. D'Ascenzo, and M. D'Emidio, "Indexing graphs for shortest beer path queries," in *24th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2024, September 5-6, 2024, Royal Holloway, London, United Kingdom*, ser. OASIcs, P. C. Bouman and S. C. Kontogiannis, Eds., vol. 123. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 2:1–2:18. [Online]. Available: <https://doi.org/10.4230/OASIcs.ATMOS.2024.2>