# On Computing Top-$k$ Simple Shortest Paths from a Single Source

Mattia D'Emidio* Gabriele Di Stefano†

**Abstract**

We investigate the problem of computing the *top-k simple shortest paths* in weighted digraphs. While the single-pair variant – finding the top-$k$ simple shortest paths between two specified vertices – has been extensively studied over the past decades, with Yen's algorithm and its heuristic improvements emerging as the most effective solving strategies, relatively little attention has been devoted to the more general single-source version, where the goal is determining top-$k$ simple shortest paths from a source vertex to all other vertices. Motivated by the numerous practical applications of ranked shortest paths, in this paper we provide new insights and algorithmic contributions to this problem. In particular, we first present a theoretical characterization of the structural properties of its solutions. Then, we introduce the first polynomial-time algorithm specifically designed to handle it. On the one hand, we prove our new algorithm is on par, in terms of time complexity, with the best (and only) polynomial-time approach known in the literature to solve the problem, that is applying the fastest single-pair algorithm independently to each vertex pair formed by the source and the remaining vertices. On the other hand, through an extensive experimental evaluation on both real-world and synthetic graphs, we demonstrate that our algorithm consistently and significantly outperforms the latter baseline in terms of running time, achieving speed-ups of up to several orders of magnitude. These results establish our new algorithm as the solution to be preferred for computing $k$ simple shortest paths from a single source in practical settings.

## 1 Introduction

Computing shortest paths is universally considered one of the most fundamental operations to be performed on graph data, as it serves a broad spectrum of applications in computer science and engineering, including communication protocols [10], trip planning [14], vehicle routing [5], network design [6], timetabling [23] and artificial reasoning [4]. The *k shortest paths* problem is a natural and long-studied generalization of the problem of determining shortest paths in which the goal is to compute not just a single path but a collection of so-called *top-k shortest paths*, in non-decreasing order of weight [16,44,46]. Compared to classical shortest paths, this generalization supports more complex applications, particularly all those that require determining and evaluating, according to some metric, multiple connection options between the vertices of a graph such as, e.g., adaptive motion planning, secure communications in multi-hop networks, fraud detection, natural language processing [2,13,16,29–31,46]. Moreover, top-$k$ shortest paths are considered a more informative measure of distance between vertices and, by extension, between graphs. Consequently, they are widely used for graph learning tasks [16], link prediction and social network analytics [2,12].

Computationally speaking, on the one hand the problem of determining $k$ shortest paths presents similarities with the computation of standard shortest paths; in fact, variants of the problem based on the input graph being directed or not (weighted or not, respectively) [26,37] and/or on the scope of the computation, namely whether the objective is to compute paths for a pair of vertices (*single-pair* variant) [16,20,27,46],

*Department of Information Engineering, Computer Science and Mathematics, and Centre of Excellence Ex-EMERGE (Centre of EXcellence on Connected, Geo-localized and Cyber-secure vehicles), University of L'Aquila (Italy) – mattia.demidio@univaq.it, http://www.mattiademidio.com.

†Department of Information Engineering, Computer Science and Mathematics, and Centre of Excellence Ex-EMERGE (Centre of EXcellence on Connected, Geo-localized and Cyber-secure vehicles), University of L'Aquila (Italy) – gabriele.distefano@univaq.it, http://gs.ing.univaq.it/.

from a single source to all other vertices (*single-source* variant) [16], or for all pairs of vertices (*all-pairs* variant) [1], have been defined and studied. On the other hand, finding $k$ shortest paths presents additional algorithmic challenges, primarily due to structural properties of the sought output. In fact, while a shortest path is, by definition, *simple* (i.e. it does not contain *loops*), a collection of $k$ shortest paths may include non-simple paths [16]. Furthermore, $k$ shortest paths lack of optimal sub-structure, and this precludes the possibility of efficiently constructing solutions to problem instances by combining solutions to sub-problems [1, 2].

When paths are not constrained to be simple, the reference approach is Eppstein's algorithm, which computes $k$ shortest paths for a single vertex pair in $\mathcal{O}(m + n \log n + k)$ time on any $n$-vertex $m$-edge weighted graph [16]. This algorithm can be improved to run in $\mathcal{O}(n+m+k)$ time when the managed graph is unweighted, and it can be extended to find the $k$ shortest paths from a specified source vertex to all other vertices, or for all vertex pairs, in $\mathcal{O}(m + n \log n + kn)$ and $\mathcal{O}(mn + n^2 \log n + kn^2)$ time, respectively [16].

When paths are required to be simple, the problem becomes more computationally challenging [1]. For the single-pair version (commonly referred to as the *Single-Pair Top-k Simple Shortest Paths* problem, or P-KSSP for short), the most efficient known algorithm for general graphs remains the method designed by Yen in 1971 [44], which computes $k$ simple shortest paths between two vertices in $\mathcal{O}(kn(m + n \log n))$ time. Despite extensive research over the past decades [17, 18, 20, 21, 27, 35, 36, 46], no asymptotic improvement over Yen's result has been achieved for general graphs, while some progress has been made either for special graph classes or from a practical viewpoint. Concerning special graph classes, for example, Pascoal et al. [35] showed that in directed acyclic graphs (DAGs), $k$ shortest paths can be computed in $\mathcal{O}(m+k(n+\log k))$ time using $\mathcal{O}(k+m)$ additional space. If the DAG has also integer edge weights, the time complexity improves to $\mathcal{O}(m+kn)$, with the same space requirements. Similarly, Gotthilf at al. [21] slightly improved Yen's bound for sparse graphs, achieving a complexity of $\mathcal{O}(k(mn + n^2 \log \log n))$ while Katoh et al. designed a method that solves P-KSSP in $\mathcal{O}(k \cdot c(n, m))$ time in undirected non-negatively-weighted graphs, where $c(n, m) \in \Omega(m)$ is the time complexity of any algorithm for single-source shortest paths [26]. In addition, Coudert et al. [11] showed that $k$ shortest simple distances (paths, respectively) can be computed in $\mathcal{O}(k+n)$ time ($\mathcal{O}(kn)$ time, respectively) on graphs with treewidth at most 2 (on bounded treewidth graphs, respectively). Finally, from an inherent complexity viewpoint, Gotthilf at al. [21] showed that P-KSSP is not asymptotically harder than the all-pairs shortest paths (APSP) problem, as it can be solved through $\mathcal{O}(k)$ iterations of any algorithm for APSP. Moreover, Vassilevska et al. [43] proved that any sub-cubic algorithm for P-KSSP would imply a sub-cubic algorithm for APSP. Concerning empirical progresses, approaches worth to be mentioned are the *node-classification* algorithm, proposed independently in [17] and [20], the *sidetrack-based* method by Kurz and Mutzel [27], the *postponed node classification* (PNC) algorithm in [46] and the very recent *PeeK* framework [18]. Although all these approaches match Yen's time complexity, they have been shown, through experimentation, to run faster than Yen's for several, practically relevant combinations of graph types and values of $k$ (e.g. road networks for $k$ in the order of hundreds) [20, 46], with PNC emerging as the method that provides the largest speed-ups in the majority of the tested cases, without increasing the space occupancy. Instead, the performance gains of other algorithms come at the price of a high memory overhead (see [27, 46]) or via parallelism [18].

In contrast to the non-simple case and to classical shortest paths, both the single-source and the all-pair versions of the problem of computing $k$ *simple* shortest paths have only marginally been investigated [1, 41], notwithstanding numerous real-world applications rely on solutions to such problems [8, 9, 19, 24, 25, 30, 33, 40, 45]. In particular, for the single-source variant (called the *Single-Source Top-k Simple Shortest Paths* problem or S-KSSP for short), to the best of our knowledge, no *dedicated* polynomial-time algorithm is currently available in the literature for general graphs and $k$. The only polynomial-time approach is, straightforwardly, to execute any polynomial-time algorithm for P-KSSP independently for each of the $n-1$ vertex pairs formed by the source and every other vertex of the graph. For the all-pairs variant, instead, a dedicated solving approach is that of Argawal et al. [1], which is asymptotically better than computing $k$ shortest paths for all vertex pairs by $\Theta(n^2)$ executions of Yen's algorithm, but works only under the very restrictive hypothesis of $k \in \{2, 3\}$. Similarly to S-KSSP, the only polynomial-time solving strategy to handle the all-pairs variant for general $k$ is to repeat any polynomial-time algorithm for P-KSSP for all $\binom{n}{2}$ vertex pairs. However,

empirical evidence [18, 20, 46] suggests that adopting the above-mentioned approaches induces prohibitively long running times even for small graphs and values of $k$, rendering it impractical for the application domains in which $k$ simple shortest paths for many pairs must be computed routinely.

**Our Contribution.** In this paper, we take a step forward in addressing these practical limitations by introducing a novel algorithm to solve S-KSSP on general digraphs. Our approach is based on a theoretical characterization of the structural properties of the solutions to the problem. We prove the correctness of the algorithm and analyze its worst-case time complexity, showing that it runs in polynomial time with respect to both the graph size and $k$. On the one hand, this is on par, in terms of asymptotical time complexity, with the state-of-the-art option to attack the problem, that is running the most efficient, polynomial-time algorithm for P-KSSP $n-1$ times from the source to every other vertex. On the other hand, we provide the results of an extensive experimental evaluation, on both real-world and synthetic datasets, that demonstrates that our new algorithm consistently outperforms this state-of-the-art method, since, in all cases, it computes solutions to S-KSSP significantly faster, often by orders of magnitude.

## 2 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 $\omega : E \to \mathbb{R}^+$. In the remainder of the paper, we use terms graph and digraph interchangeably. We use $N^+(v) = \{u \in V | (v, u) \in E\}$ ($N^-(v) = \{u \in V | (u, v) \in E\}$, respectively) to denote the set of *outgoing* (*incoming*, respectively) neighbors of a vertex $v \in V$. A *path* $P = (s \equiv v_1, v_2, \ldots, t \equiv v_\eta)$ 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 $i \in [\eta - 1]$. Given an integer $k > 0$, we let $[k] := \{1, 2, \ldots, k\}$. Given a path $P = (v_1, v_2, \ldots, v_\eta)$ in $G$, we denote by $V(P)$ ($E(P)$, respectively) the vertices (edges, respectively) of $P$. Moreover, we say a path $P$ is *simple* (or *loopless*) when there are no vertex repetitions in $V(P)$. Finally, we call $\omega(P) = \sum_{i=1}^{\eta-1} \omega(v_i, v_{i+1})$ the *weight* of a path $P$. In an unweighted graph, the weight is equal to the *length* of the path, that is, to the number of its edges. Given a graph $G = (V, E, \omega)$ and a subset $V' \subseteq V$ of its vertices, we denote by $G[V']$ the sub-graph of $G$ induced by vertices in $V'$, i.e. graph $G' = (V', E', \omega)$ where $E' = \{(u, v) \in E \mid u, v \in V'\}$.

A *shortest path*, for two vertices $s, t \in V$, is a path having minimum weight among all paths connecting $s$ to $t$. Given two paths, say $P' = (v_1, v_2, \ldots, v_\eta)$ and $P'' = (u_1, u_2, \ldots, u_\zeta)$ such that $v_\eta \equiv u_1$, we denote by $P' \oplus P''$ the *concatenation* of the two paths, that is, path $(v_1, v_2, \ldots, v_\eta \equiv u_1, u_2, \ldots, u_\zeta)$ obtained by appending all vertices of $P''$ to $P'$. If $P = P' \oplus Q$ (with $Q$ possibly being the empty path), we call $P'$ a *prefix* of $P$.

Given a graph $G = (V, E, \omega)$ with $\omega : E \to \mathbb{R}^+$, two vertices $s, t \in V$, and an integer $k > 0$, the *Single-Pair Top-k Simple Shortest Paths* problem (P-KSSP, for short) asks to compute a collection $S_{st} = \{P_1, P_2, \ldots, P_{k'}\}$ of $k' \leq k$ paths from $s$ to $t$ such that: (i) $P_i$ is simple for any $i \in [k']$; (ii) $P_i \neq P_j$ for any $i, j \in [k'], i \neq j$; (iii) $\omega(P') \geq \omega(P_i)$ for any path $P'$ from $s$ to $t$ that is not in $S_{st}$ and for any $P_i \in S_{st}$; (iv) $S_{st}$ is maximal (there does not exist a collection of size $k'' > k'$). By the above, it follows that: if there exist $k' < k$ simple paths from $s$ to $t$ in $G$, then the problem requires the computation of a collection $S_{st}$ that contains all and only such $k' < k$ paths; otherwise, the problem asks to compute a collection $S_{st}$ 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 $S_{st}$ is larger than or equal to the weight of any path in $S_{st}$. In either case, a collection of paths from $s$ to $t$, satisfying the constraints imposed by P-KSSP, is called a collection of top-$k$ simple shortest paths for pair $(s, t)$. Observe that there can be several feasible collections (e.g., there exist $k'' > k$ paths of same weight for a pair).

A generalization of P-KSSP is the *Single-Source Top-k Simple Shortest Paths* problem (S-KSSP) which, given a *source* (or *root*) vertex $r \in V$, asks to compute, for each $v \in V \setminus \{r\}$, a collection $S_{rv} = \{P_1, P_2, \ldots, P_{k'}\}$ of $k' \leq k$ paths from $r$ to $v$ which is a solution to P-KSSP for pair $(r, v)$. Each $S_{rv}$ is called a collection of top-$k$ simple shortest paths from $r$ to $v$ (or for $v$, when the root is clear from the context). As in P-KSSP, for each vertex $v$, there can be several feasible collections (e.g., if there exist $k'' > k$ paths of same weight from the root $r$ to a vertex $v$). We call $\Gamma_{rv}$ the family of all such collections for a vertex $v$ and use $S_r = \{S_{rv_1}, S_{rv_2}, \ldots, S_{rv_n}\}$ to identify any solution to S-KSSP, where $S_{rv_i} \in \Gamma_{rv_i} \quad \forall\, v_i \in V$. We

**Algorithm 1:** Algorithm EXH-S-KSSP.

**Input:** Graph $G = (V, E, \omega)$, root $r \in V$, $k \in \mathbb{N}$.
**Output:** Top-$k$ simple shortest paths $\mathrm{T}_v$ from $r$ to each $v \in V \setminus \{r\}$.

1 $PQ \leftarrow \emptyset$; ▷ Empty priority queue
2 **foreach** $v \in V$ **do** ▷ Empty lists
3 $\quad \mathrm{T}_v \leftarrow [\,]$;
4 $PQ.enqueue((r), \omega((r)) = 0)$;
5 **while** $PQ \neq \emptyset$ **do**
6 $\quad \Pi = (r, \ldots, v) \leftarrow PQ.dequeueMin()$ ;
7 $\quad$ **foreach** $u \in N^+(v) \setminus V(\Pi)$ **do** ▷ Simple paths
8 $\quad\quad PQ.enqueue(\Pi \oplus (u), \omega(\Pi) + \omega(v, u))$;
9 $\quad$ **if** $|T_v| < k$ **then**
10 $\quad\quad$ Add $\Pi$ to $\mathrm{T}_v$;

write $S$ instead of $S_r$ when $r$ is clear from the context.

# 3 Characterizing Solutions to S-KSSP

In this section, we first formalize a brute-force approach, named Algorithm EXH-S-KSSP, for solving S-KSSP and show that its time complexity is exponential with respect to the input size in the worst case. Then, we present a characterization of some relevant properties of solutions to S-KSSP which we leverage to improve Algorithm EXH-S-KSSP and obtain an algorithm that solves the problem in polynomial time.

## 3.1 Solving S-KSSP Exhaustively

In what follows we introduce Algorithm EXH-S-KSSP, an algorithm for S-KSSP obtained by modifying the brute-force method to find all simple paths in a graph for a pair of vertices [2, 39]. The algorithm (whose pseudo-code is given in Algorithm 1) solves S-KSSP 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 named $\mathrm{T}_v$, only a subset of such paths that form the sought collection of top-$k$ simple shortest paths for $v$. To this end, the algorithm: (i) finds, enqueues and dequeues simple paths from the root in a minimum priority queue $PQ$, where the priority is the weight of discovered paths; (ii) terminates when $PQ$ is empty. In the description of the procedures, given a collection of paths $\mathrm{T}_v$, we use $V(\mathrm{T}_v)$ to denote the set of vertices that belong to paths in $\mathrm{T}_v$, i.e. $V(\mathrm{T}_v) = \cup_{P \in \mathrm{T}_v} V(P)$. Moreover, in the remainder of the paper, we assume that routine $PQ.enqueue(p, \delta)$ adds to $PQ$ an element $p$ with priority $\delta$, while routine $PQ.dequeueMin()$ returns the element of $PQ$ having minimum priority and removes it from $PQ$.

Now, we first introduce and prove two lemmas that are necessary to show the correctness of algorithm EXH-S-KSSP.

**Lemma 1.** *Algorithm EXH-S-KSSP computes all simple paths from the root vertex to any other vertex.*

*Proof.* By contradiction, assume that there exists a simple path from the root $r$ to a vertex that is not found by Algorithm EXH-S-KSSP, which means that it is never dequeued from $PQ$ in line 6. Among all such simple paths, let $\pi = (r, \ldots, v, u)$ be the shortest. Since every path enqueued is eventually extracted from $PQ$, the prefix $\pi' = (r, \ldots, v)$ must have been both enqueued and dequeued; otherwise, $\pi$ would not be the shortest path that is not found by the algorithm. It follows that, when $\pi'$ is dequeued, $\pi$ is enqueued in line 8. Moreover, $\pi$ is simple since $u \notin V(\pi')$. This contradicts our hypothesis, as it implies that $\pi$ is enqueued and eventually dequeued from $PQ$. □

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

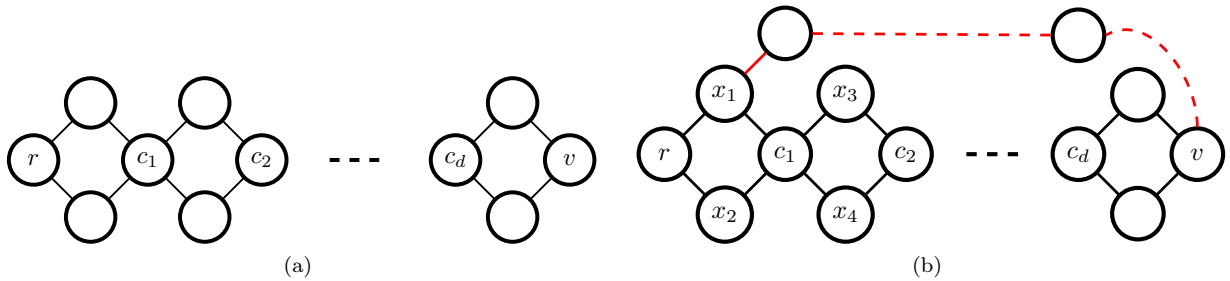


Figure 1: (a) and (b): input instances considered in the proofs of Theorems 2 and 4, respectively.

*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 $\pi' \notin PQ$, 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 therefore $\pi'$ would have been enqueued in line 8 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. Recursively repeating this argument for any prefix of $\pi'$, we reach the contradiction of the path $(r)$ not being in both $PQ$ and $F$. □

The correctness of Algorithm 1 is shown by the following results.

**Theorem 1.** *Algorithm 1 solves S-KSSP.*

*Proof.* The proof follows by Lemmas 1 and 2. In particular, by Lemma 2 we know that Algorithm EXH-S-KSSP discovers all simple paths in order of weight. Hence, in line 10, data structure $\mathrm{T}_v$ is filled with paths that follow a non-decreasing order of weight, for each $v \in V$. Moreover, by Lemma 1 we know that Algorithm EXH-S-KSSP determines all simple paths from the root, hence at termination $\mathrm{T}_v = S_{rv}$ for each $v \in V$ and for some $S_{rv} \in \Gamma_{rv}$. □

We emphasize that all methods introduced in this paper are designed for directed weighted graphs but can be applied to both undirected or unweighted graphs by considering bidirectional edges (i.e. $N^+(v) = N^-(v)$ for each $v \in V$) or unitary weights (i.e. $\omega(u, v) = 1$ for each $(u, v) \in E$). Moreover, note that Algorithm 1 can be modified to terminate earlier by adding a conditional statement inside the while loop of line 5. Specifically, if we define a vertex $v$ to be *saturated* when $|\mathrm{T}_v| = k$, the loop can be stopped when all vertices are saturated. It is easy to see that this modification does not affect the algorithm's correctness (since simple paths are found in non-decreasing order of weight) nor its time complexity (in the worst case, there may be a vertex $w$ for which fewer than $k$ simple paths from $r$ to $w$ exist in the graph). In the remainder of this paper, with a little abuse of notation, we refer to this variant as Algorithm EXH-S-KSSP as well. Note also that, while Algorithm 1 is simple and elegant, its time complexity is exponential in the input size, as shown below:

**Theorem 2.** *There exists an n-vertex graph that forces algorithm EXH-S-KSSP to run for $\Omega(2^n)$ time to achieve a solution to S-KSSP.*

*Proof.* Consider the undirected, unweighted graph in Fig. 1a: call $d$ the number of vertices of type $c_i$. Then, the number of paths that are enqueued in $PQ$, before extracting a shortest path from the root $r$ to $v$, is at least the number of simple paths in the graph from $r$ to $c_d$, that is, $2^d$. Since $d = \Theta(n)$ the claim follows. □

We now provide a characterization of some important properties of solutions to S-KSSP to exploit to design efficient algorithms to solve the problem. In particular, we are interested at identifying conditions that can be tested to stop early the search of algorithm EXH-S-KSSP, to reduce its time complexity. To this aim, we first provide the following definition.

**Definition 1** (PREDECESSOR)**.** *Let $S = \{S_{rv_1}, S_{rv_2}, \ldots, S_{rv_n}\}$ be a solution to S-KSSP for a given graph $G = (V, E, \omega)$, root vertex $r \in V$, and $k > 0$. 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 $S_{rv}$ for $v \in V$, i.e. $u \in V(P) \setminus \{v\}$ for some $P \in S_{rv}$. Then, we say vertex $u$ is a predecessor of $v$ in $S$.*

We denote by $V(S_{rv}) = \{u \mid uR_Sv\}$ a set of *predecessors* of a vertex $v$. The next definition introduces the notion of *general predecessor* of a vertex in a solution $S$ to S-KSSP. Informally, $v$ is a general predecessor of itself and, if a vertex $u$ is a general predecessor of $v$, then the predecessors of $u$ are also general predecessors of $v$. Formally:

**Definition 2** (GENERAL PREDECESSOR)**.** *Let $S$ be a solution to S-KSSP for a given graph $G = (V, E, \omega)$, root vertex $r \in V$, and $k > 0$. Let $T_S$ be the transitive closure of relation $R_S$ of Def. 1. A vertex $u \in V$ is a general predecessor of a vertex $v \in V$ if and only if $uT_Sv$ or $u = v$.*

We denote by $A_{S,v} = \{u \mid uT_Sv\} \cup \{v\}$ a set of *general predecessors* of a vertex $v \in V$. The following lemma shows that sets of general predecessors form a nested set collection.

**Lemma 3.** *Let $S$ be a solution to S-KSSP 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 2, any vertex $y \in A_{S,x}$ is a general predecessor of $x$. Hence $y$ is such that $yT_Sx$. Similarly, since $x \in A_{S,v}$, we have $xT_Sv$. By transitivity, it follows that $yT_Sv$ and hence that $y \in A_{S,v}$. □

The following two lemmata establish important structural properties that hold for certain paths belonging to solutions to S-KSSP. In particular, Lemma 4 shows that if there exists a path $P_{rv}$ that **is not in any** solution, for some vertex $v \in V$, and such that $P_{rv}$ is a prefix of a path $P_{rw}$ that is in **all** solutions for some other vertex $w \in V$, then a vertex of $P_{rw}$ must be a predecessor of $v$. Figure 2a provides a visual representation of the claim of Lemma 4.

**Lemma 4** (BASIC LEMMA)**.** *Let $S = \{S_{rv_1}, S_{rv_2}, \ldots, S_{rv_n}\}$ be a solution to S-KSSP 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 $P_{rw} \in S_{rw}$, $\forall\, S_{rw} \in \Gamma_{rw}$ and $P_{rv} \notin S_{rv}$. Then, there exists a vertex $t \in V(S_{rv}) \cap V(P_{vw})$.*

*Proof.* Notice that, by the definition of $V(S_{rv})$, we have $t \neq v$ while $t$ might coincide with $w$ or not. By contradiction, assume that $V(S_{rv}) \cap V(P_{vw})$ is empty. Consider the set of simple paths $\Pi_{rw} = \{Q \oplus P_{vw} \mid Q \in S_{rv}\}$. Since the number of paths in $S_{rv}$ is $k$ – otherwise $P_{rv}$ should belong to $S_{rv}$ – 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 S-KSSP such that $P_{rw} \notin S'_{rw}$, a contradiction. □

A stronger characterization of the structure of paths that belong to solutions to S-KSSP is provided through the following result. 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 $y$ in 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 $y$. Figure 2b provides a visual representation of the statement of Lemma 5.

**Lemma 5** (PATH LEMMA)**.** *Let $S = \{S_{rv_1}, S_{rv_2}, \ldots, S_{rv_n}\}$ be a solution to S-KSSP 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 a path from $r$ to a vertex $w \in V$ such that $P_{rw} \in S_{rw}, \forall\, S_{rw} \in \Gamma_{rw}$, and $P_{rv} \notin S_{rv}$. Then, there exist two vertices $t, y$ in $V(P_{vw}) \cap A_{S,v}$ such that: i) $y \in V(S_{rt})$; ii) $P_{rt} = P_{rv} \oplus P_{vt} \notin S_{rt}$; and iii) $P_{ry} = P_{rv} \oplus P_{vy} \in S_{ry}$, where $P_{vt}$ and $P_{vy}$ are both prefixes of $P_{vw}$.*

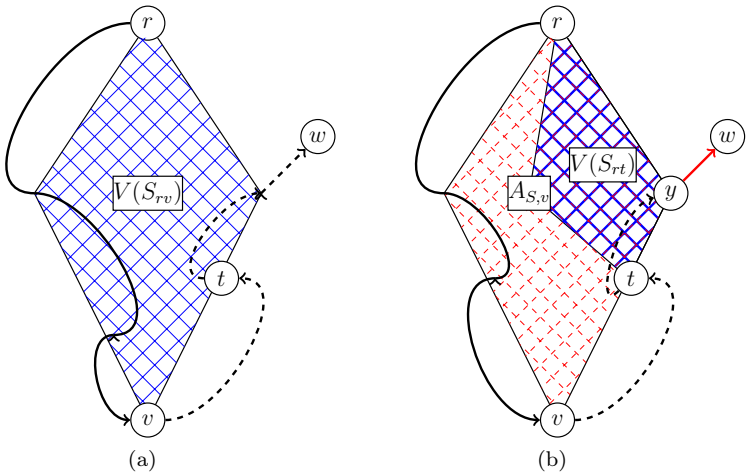


Figure 2: (a): Structure of solution $S$ of Lemma 4: the solid path is $P_{rv} \notin S_{rv}$ while the dashed path is $P_{vw}$; vertex $t$ must be a predecessor of $v$ in $S$; the blue grid highlights set $V(S_{rv})$. (b): Structure of solution $S$ of Lemma 5: the black solid path is $P_{rv}$ while the black dashed one is $P_{vy}$; the red path 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 identifies the vertices of $V(S_{rt})$ while the red dashed grid indicates set $A_{S,v}$, which includes $V(S_{rt})$.

*Proof.* By Lemma 4, there is a vertex $t_0$ in $P_{vw}$ that belongs to $V(S_{rv})$. If $P_{rt_0} = P_{rv} \oplus P_{vt_0} \in S_{rt_0}$ 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 $P_{rt_0} \notin S_{rt_0}$ and apply again Lemma 4. Hence, there exists a vertex $t_1$ in $P_{vw}$ belonging to $V(S_{rt_0})$. Now, if $P_{rt_1} = P_{rt_0} \oplus P_{t_0t_1} \in S_{rt_1}$, again we are done as $t_0$ and $t_1$ play the roles of $t$ and $y$, respectively. Vice versa, we can repeat the reasoning above and find either 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 $P_{rw} \in S_{rw}, \forall\, S_{rw} \in \Gamma_{rw}$ by hypothesis. □

Finally, we derive another relation between the paths in any solution to S-KSSP and the general predecessors of the vertices on such paths. When searching a graph for a solution to S-KSSP, this relation is helpful to safely discard certain paths, as they certainly are not part of any solution, based on conditions holding on their prefixes.

**Lemma 6** (PRUNING LEMMA)**.** *Let $S = \{S_{rv_1}, S_{rv_2}, \ldots, S_{rv_n}\}$ be a solution to S-KSSP for a given graph $G = (V, E, \omega)$, root vertex $r \in V$, and $k > 0$. Let $P_{rv}$ any path from $r$ to a vertex $v \in V$. If (i) $|S_{ru}| = k$ for all vertices $u \in A_{S,v} \setminus \{r\}$ and (ii) $P_{rv}$ is not a prefix of any path in $\bigcup_{u \in A_{S,v}} S_{ru}$, then there exists a solution $S' = \{S'_{rv_1}, S'_{rv_2}, \ldots, S'_{rv_n}\}$ such that $P_{rv}$ is not a prefix of any path in $\bigcup_{u \in V} S'_{ru}$.*

*Proof.* We construct a solution $S' = \{S'_{rv_1}, S'_{rv_2}, \ldots, S'_{rv_n}\}$ by modifying $S$ as follows. By hypothesis, $P_{rv}$ is not a prefix of any path in $\bigcup_{u \in A_{S,v}} S_{ru}$ hence we assign $S'_{ru} = S_{ru}$ for all vertices $u \in A_{S,v}$. Now, consider any vertex $w \in V \setminus A_{S,v}$ and let $P_{rw} = P_{rv} \oplus P_{vw} \in S_{rw}$ be a path having $P_{rv}$ as prefix and such that $P_{rv}$ satisfies properties (i) and (ii). We now show that $P_{rw}$ can be replaced with another path that does not have $P_{rv}$ as prefix. 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 $P^{max}_{ry}$ be the heaviest path in $S_{ry}$. Since $P_{rv}$ is not a prefix of any path in $\bigcup_{u \in A_{S,v}} S_{ru}$, it follows that the weight of path $P_{ry} = P_{rv} \oplus P_{vy}$ must be at least that of $P^{max}_{ry}$. Hence, we have:

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

**Algorithm 2:** Algorithm PRUNED-S-KSSP.

**Input:** Graph $G = (V, E, \omega)$, root $r \in V$, $k \in \mathbb{N}$.
**Output:** Top-$k$ simple shortest paths $\mathrm{T}_v$ from $r$ to each $v \in V \setminus \{r\}$.

```
1  PQ ← ∅;                                                          ▷ Empty priority queue
2  foreach v ∈ V do                                                 ▷ Empty lists
3  |  T_v ← [ ];
4  PQ.enqueue((r), ω((r)) = 0);
5  while PQ ≠ ∅ and ∃ w ≠ r : |T_w| < k do
6  |  Π = (r, ..., v) ← PQ.dequeueMin();
7  |  if not PRUNING(v) then                                        ▷ Pruning Test
8  |  |  foreach u ∈ N⁺(v) \ V(Π) do
9  |  |  |  PQ.enqueue(Π ⊕ (u), ω(Π) + ω(v, u));
10 |  if |T_v| < k then                                            ▷ Not saturated
11 |  |  Add Π to T_v;
```

**Procedure 3:** Procedure PRUNING.

**Input:** A vertex $v$.
**Output:** True iff all general predecessors of $v$ are saturated.

```
1  Q ← {v};                                                         ▷ Add v to empty FIFO queue
2  SEEN ← {v};
3  while Q ≠ ∅ do
4  |  x ← Q.pop();                                                  ▷ Dequeue head of queue
5  |  if |T_x| < k and x ≠ r then
6  |  |  return false;                                             ▷ x is a general predecessor of v and is not saturated
7  |  foreach u ∈ N⁻(x) : u ∉ SEEN do
8  |  |  if u ∈ V(T_x) then
9  |  |  |  Q.append(u);
10 |  |  |  SEEN ← SEEN ∪ {u};
11 return true;
```

Since $y \in A_{S,v}$, we have $|S_{ry}| = k$ by hypothesis $(i)$. Moreover, by Lemma 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 $P_{ry} \in S_{ry}$, that all have weight at most $\omega(P_{rw})$. Thus, we can construct set $S'_{rw}$ by replacing $P_{rw} \in S_{rw}$ with a path $P_{ry} \oplus P_{yw}$ for some $P_{ry} \in S_{ry}$. □

# 4 New Algorithms for S-KSSP

In this section, we introduce new algorithms to solve S-KSSP. Our design is incremental: starting from Algorithm EXH-S-KSSP, we first exploit the properties of the PRUNING LEMMA (Lemma 6) to design Algorithm PRUNED-S-KSSP. In particular, note that when Algorithm EXH-S-KSSP extracts from the priority queue a path $P$ ending at a *saturated* vertex $v$ (such that $|\mathrm{T}_v| = k$); clearly, we know $P$ is not part of the top-$k$ shortest paths for $v$, but is it a prefix of a path in any solution of S-KSSP? If, for some solution $S$, all vertices in $A_{S,v}$ are saturated, then by the PRUNING LEMMA the answer is negative: hence, we can avoid the computation of paths with this prefix. By incorporating this test into Algorithm EXH-S-KSSP we design Algorithm PRUNED-S-KSSP. We prove its correctness, but also show that there are graphs such that its running time grows exponentially in the graph size. This occurs when, in the above situation, $P$ is a prefix of a path belonging to *each* solution $S$. In fact, by the PATH LEMMA (Lemma 5), in this case $A_{S,v}$ contains at least a non-saturated vertex (namely vertex $y$ in the statement of the lemma). To overcome this limitation, we then proceed to design Algorithm BOUNDED-S-KSSP which improves over Algorithm PRUNED-S-KSSP by handling such pathological cases with the explicit computation of top-$k$ simple shortest paths for these non-saturated

vertices, without searching for further paths having $P$ as prefix. Thanks to this improvement, we will show that Algorithm BOUNDED-S-KSSP solves S-KSSP in polynomial time w.r.t. both the graph' size and $k$.

### 4.1 An Exponential-time Algorithm for S-KSSP

In this section, we present Algorithm PRUNED-S-KSSP to solve S-KSSP. This algorithm, whose pseudo-code is given in Algorithm 2, fills data structure $\mathrm{T}_v$ with simple paths emanating from the root until a solution to the instance of S-KSSP is achieved and works exactly as Algorithm EXH-S-KSSP except for the test of line 7, which serves the purpose of reducing the search space of solutions. In particular, such test evaluates to true whenever routine PRUNING($v$) returns false for a given vertex $v$, terminal vertex of a path $\Pi = (r, \ldots, v)$ extracted from the priority queue. This routine (see Procedure 3) checks whether all general predecessors of $v$ are saturated. In this case, by the PRUNING LEMMA, the algorithm can avoid to enqueue paths that have the extracted path $\Pi$ as prefix, since these are not part of a solution. In more detail, to determine whether all general predecessors of $v$ are saturated, Procedure PRUNING($v$) performs a breadth-first-search (BFS) visit of the transpose graph that starts by enqueueing $v$ into a FIFO queue $Q$ (see line 1). Whenever a vertex $x$ is removed from $Q$ the visit proceeds by considering only the incoming neighbors of $x$ that are also its predecessors (see line 9). Hence, by transitivity, it explores a set of general predecessors of $v$. If one of the encountered vertices, say $x$, is found to be not saturated (i.e., such that $|\mathrm{T}_x| < k$) then the routine returns false. If instead the search is concluded without finding any non-saturated general predecessor, then the routine returns true. Visited vertices are stored in a set SEEN to avoid evaluating vertices more than once. Algorithm PRUNED-S-KSSP stops either when the queue becomes empty or when $|\mathrm{T}_w| = k$ for all $w \in V \setminus \{r\}$. As in Algorithm EXH-S-KSSP, paths in $PQ$ have a priority equal to their weight. The correctness of Algorithm 2 is shown by the following result.

**Theorem 3.** *Algorithm 2 solves S-KSSP.*

*Proof.* Observe that, by removing line 7, and hence executing lines 8–9 regardless of the result of the execution of procedure PRUNING, Algorithm PRUNED-S-KSSP is equivalent to Algorithm EXH-S-KSSP (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 1. To complete the proof, we need to show that, if the conditional statement of line 7 is false in some cases, and the visit does not proceed from some vertices, then the algorithm's correctness is not altered; that is, at termination, collection $\{\mathrm{T}_{v_1}, \mathrm{T}_{v_2}, \ldots, \mathrm{T}_{v_n}\}$ is a solution $S$ to S-KSSP, or equivalently that each $\mathrm{T}_v$ is an element of $\Gamma_{rv}$. By contradiction assume that there exists a vertex, say $u$, such that at termination, $\mathrm{T}_u \notin \Gamma_{ru}$ meaning that: either (i) $|\mathrm{T}_u| = k' < k$ while all elements of $\Gamma_{ru}$ have size $k'' > k'$ with $k'' \leq k$, that is collection $\mathrm{T}_u$ is not maximal; or (ii) $|\mathrm{T}_u| = k$ but there exists a path $P'$ in the graph that has been added to $\mathrm{T}_u$ whereas it should not have been, that is there exists at least a path $P$ in the graph such that $\omega(P) < \omega(P')$. In both cases, it follows that there exists at least a path in the graph that should be in $\mathrm{T}_u$ but is not added to $\mathrm{T}_u$ during the execution of the algorithm. Among these paths, call $P_{ru}$ the one that contains a vertex $v \in P_{ru}$ such that $v$ is the first vertex for which the test in line 7 is false, that is, the first vertex during the execution of the algorithm such that PRUNING($v$) returns true. If we denote by $X$ the set of vertices that are visited by routine PRUNING($v$) executed in line 7, then, for any $x \in X$ we have that: (i) $x$ is saturated, i.e. $|\mathrm{T}_x| = k$; (ii) $\mathrm{T}_x \in \Gamma_{rx}$ by Theorem 1. Thus, set $X \equiv A_{S,v}$ for some solution $S$. Hence, according to Lemma 6, there must exist a solution $S' = \{S'_{rv_1}, S'_{rv_2}, \ldots, S'_{rv_n}\}$ such that $P_{rv}$ is not a prefix of any path in $\bigcup_{w \in V} S'_{rw}$, which contradicts the hypothesis of $P_{rv}$ being a prefix of $P_{ru}$ and $P_{ru} \subseteq S_{ru}$ for any $S_{ru} \in \Gamma_{ru}$. □

Clearly, intuitively, the pruning test of line 7 should allow one to reduce the number of paths that Algorithm PRUNED-S-KSSP enqueues in (dequeues from, respectively) the priority queue to determine a solution to S-KSSP, compared to Algorithm EXH-S-KSSP. However, the following result shows that the introduction of the pruning test does not suffice to guarantee a polynomial time complexity for Algorithm PRUNED-S-KSSP:

**Theorem 4.** *There exist an $n$-vertex graph and an integer $k > 0$ that force algorithm PRUNED-S-KSSP to run in $\Omega(2^n)$ time to achieve a solution to S-KSSP.*

**Algorithm 4:** Algorithm BOUNDED-S-KSSP.

**Input:** Graph $G = (V, E, \omega)$, root $r \in V$, $k \in \mathbb{N}$.
**Output:** Top-$k$ simple shortest paths $\mathrm{T}_v$ from $r$ to each $v \in V \setminus \{r\}$.

```
1  PQ ← ∅;                                                    ▷ Empty priority queue
2  foreach v ∈ V do                                           ▷ Empty lists
3  |  T_v ← [ ]
4  PQ.enqueue((r), ω((r)) = 0);
5  while PQ ≠ ∅ and ∃ w ≠ r : |T_w| < k do
6  |  Π = (r, ..., v) ← PQ.dequeueMin();
7  |  if |T_v| < k then
8  |  |  Add Π to T_v;
9  |  |  foreach u ∈ N⁺(v) \ V(Π) do
10 |  |  |  if u not super-saturated and Π ⊕ (u) ∉ PQ then
11 |  |  |  |  PQ.enqueue(Π ⊕ (u), ω(Π) + ω(v, u));
12 |  else if v not super-saturated then
13 |  |  foreach w general predecessor of v do
14 |  |  |  Compute S_rw ⊇ T_w;
15 |  |  |  foreach p ∈ S_rw \ T_w  :  p ∉ PQ do
16 |  |  |  |  PQ.enqueue(p, ω(p));
17 |  |  |  Mark w super-saturated;
```

*Proof.* Consider the graph of Figure 1b and analyze the case of $k = 3$ with $r$ as root vertex. Observe that all simple shortest paths from $r$ to vertex $x_1$ can be classified into two categories: in former we have (two) simple paths that do not share any vertex with the internal vertices of the red dashed path, namely $(r, x_1)$, $(r, x_2, c_1, x_1)$; in the latter 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 type $c_i$ in the given graph, then we have $2^{d+1}$ of simple paths from $r$ to $x_1$ of the latter category. Now, assume the dashed red path has weight $2d$. Then, Algorithm PRUNED-S-KSSP finds the two simple paths of the first category after at most three dequeue and at most nine dequeue operations, respectively, depending on the order in which neighbors are considered. Similarly, after at most nine dequeue operations, the 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 $c_2$ are then enqueued. Observe that such enqueue operations follow from the test of line 7 evaluating to true for both $x_3$ and $x_4$: this occurs since $\mathrm{T}_{x_3}$ and $\mathrm{T}_{x_4}$ have size $k = 3$ and both calls to routine PRUNING$(x_3)$ and PRUNING$(x_4)$ return false – due to $x_1$ and $x_2$ being not saturated. This behavior continues unchanged for all vertices in all simple paths from $r$ to $v$ that do not share vertices with the red path, with routine PRUNING always returning false since $x_1$ is a predecessor of all such vertices and is not saturated. Hence, $2^{d+1} = \mathcal{O}(2^n)$ paths terminating at $v$ are enqueued and dequeued before any of the $2^{d+1}$ simple paths from $r$ to $x_1$ of the second category can be enqueued/dequeued, and therefore before the algorithm can successfully terminate. □

## 4.2 Solving S-KSSP in Polynomial Time

In this section, we present Algorithm BOUNDED-S-KSSP that solves S-KSSP in polynomial time. The algorithm, whose pseudo-code is given in Algorithm 4, is built upon Algorithm PRUNED-S-KSSP and uses the characterization of solutions to S-KSSP of Section 3 to limit the maximum number of paths that are enqueued and hence dequeued from the priority queue, for each vertex other than the root. To obtain such a bound, the main modification consists in replacing procedure PRUNING, which searches for a non-saturated general predecessor to decide whether a visit must be pruned at a given path $\Pi = (r, \ldots, v) \notin S_{rv}$ or not, with an explicit computation of top-$k$ simple shortest paths for non-saturated general predecessors of $v$. Consider again the graph in Fig. 1b and assume $k = 3$. Define a vertex $x$ to be *super-saturated* whenever the algorithm has computed a collection of top-$k$ simple shortest paths from the root to each general predecessor of $x$, in-

cluding $x$ itself. When Algorithm BOUNDED-S-KSSP extracts the fourth among the paths terminating at $c_2$ – namely $(r, x_1, c_1, x_3, c_2)$, $(r, x_1, c_1, x_4, c_2)$, $(r, x_2, c_1, x_3, c_2)$, and $(r, x_2, c_1, x_4, c_2)$ – then $|T_{c_2}| = k$ and $c_2$ is not supersaturated. Thus, in line 14, the algorithm immediately computes top-$k$ simple shortest paths $S_{rx_1}$ and $S_{rx_2}$ for vertices $x_1$ and $x_2$, predecessors of $c_2$, in such a way that $S_{rx_1}$ and $S_{rx_2}$ include the paths already stored in $T_{x_1}$ and $T_{x_2}$. This avoids the computation of an exponential number of paths before completing the computation of a solution to S-KSSP, as instead happens if PRUNED-S-KSSP is used. In more detail, the algorithm works as follows. The initialization phase is as in Algorithm PRUNED-S-KSSP: we set up $\mathrm{T}_v$ to be an 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 0. Then, as long as $PQ$ is not empty or $|\mathrm{T}_w| < k$ for some $w \in V \setminus \{r\}$, a path $\Pi$ terminating at some vertex $v$ is dequeued from $PQ$ and an *iteration* of the algorithm is executed. In a generic iteration, if the conditional statement of line 7 evaluates to true (i.e. no pruning is possible) then the execution proceeds as in Algorithm PRUNED-S-KSSP, with the difference that simple paths $\Pi \oplus (u)$ are enqueued into $PQ$ only for the vertices $u$ in $N^+(v)$ that are not *super-saturated*. In particular, initially all vertices are not super-saturated. Then, when the test of line 7 evaluates to false, that is when we extract a path $\Pi = (r, \ldots, v)$ and $|\mathrm{T}_v| = k$ we distinguish two cases: either $v$ is super-saturated or not. In the former case, no further enqueue operation is performed, for the same reasons that stop the visit of Algorithm PRUNED-S-KSSP when PRUNING returns true. In the latter, instead, we compute explicitly a set of general predecessors of $v$ and a collection $S_{rw}$ that contains paths in $\mathrm{T}_w$, for each general predecessor $w$ of $v$: on the one hand, this is easy to achieve for saturated vertices, for which we will show $\mathrm{T}_w = S_{rw}$ for some $S_{rw} \in \Gamma_{rw}$; on the other hand, it requires to compute a collection of top-$k$ simple shortest paths from $r$ to $w$, i.e., a solution to P-KSSP for pair $(r, w)$, for each non-saturated general predecessor $w$ of $v$. Such a solution can be obtained by any algorithm that solves P-KSSP for pair $(r, w)$ in polynomial-time (e.g., Yen's). Once this is done, we enqueue any path $p \in S_{rw} \setminus \mathrm{T}_w$ into $PQ$ and mark every general predecessor of $v$ to be super-saturated. Observe that, whenever we need to add a path $p$ to $PQ$ during the algorithm, we verify that $p \notin PQ$ before performing the enqueue operation, to avoid inserting twice a same path in $PQ$ and to achieve a correct result. The correctness of Algorithm 4 is given by the following result.

**Theorem 5.** *Algorithm 4 solves S-KSSP.*

*Proof.* In what follows, we use $\mathrm{T}_u^t$ to represent the content of data structure $\mathrm{T}_u$ at the $t$-th iteration of the main loop of line 5. Similarly, we denote by $PQ^t$ the priority queue $PQ$ after the $t$-th execution of such a loop. We assume $t = 0$ if no loop iteration 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 write comparison expressions such as $L(A') \geq L(A'')$, $L(A') < L(A'')$ or $L(A') = L(A'')$. First, observe that $L(S_{ru}) = L(S'_{ru})$ even if $S_{ru}$ and $S'_{ru}$ are different solutions to S-KSSP for a given vertex $u$, i.e. $S_{ru} \in \Gamma_{ru}$ and $S'_{ru} \in \Gamma_{ru}$ with $S_{ru} \neq S'_{ru}$. Moreover, if $L(\mathrm{T}_u^t) \leq L(S_{ru})$ then all paths in $\mathrm{T}_u^t$ must belong to a solution in $\Gamma_{ru}$ for vertex $u$. Finally, if $L(\mathrm{T}_u^t) = L(S_{ru})$ then $\mathrm{T}_u^t$ belongs to a solution $S_{ru}$ for $u$.

The proof is by induction on the number $t$ of times that the main loop of line 5 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 $w \in V$, the following two properties hold:

1. $L(\mathrm{T}_w^t) \leq L(S_{rw})$;
2. if there exists a path $P_{rw}$ such that $P_{rw} \notin \mathrm{T}_w^t$ and $L(\mathrm{T}_w^t \cup \{P_{rw}\}) \leq L(S_{rw})$ 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 $\mathrm{T}_w^t$ computed until time $t$ are correct, i.e. paths in $\mathrm{T}_w^t$ belong to a solution to S-KSSP 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 $\mathrm{T}_w^t$ to complete a solution in $\Gamma_{rw}$ 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 $L(\mathrm{T}_w^0) = [\,]$ 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 S-KSSP.

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 of the condition controlling the main loop being true, then $PQ^t$ is not empty. Let $P_{rv}$ the path dequeued in line 6. Notice that, like $P_{rv}$, any path in $PQ$ can be inserted either in line 11 or in line 16. We call *normal* (*exceptional*, respectively) the first (second, respectively) kind of insertion.

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 $L(\mathrm{T}_v^{t+1}) = L(\mathrm{T}_v^t \cup \{P_{rv}\}) \leq L(S_{rv})$, as exceptional insertions are dictated by an execution of Yen's (or any algorithm for P-KSSP) algorithm which returns a solution $S_{rv}$ for any vertex $v$. Moreover, since $L(\mathrm{T}_w^{t+1}) = L(\mathrm{T}_w^t)$ 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 $|L(\mathrm{T}_v^t)| = |L(S_{rv})|$ it follows that property 1) clearly remains true as $P_{rv}$ is not added to $\mathrm{T}_v^t$. If, instead, $|L(\mathrm{T}_v^t)| < |L(S_{rv})|$, we suppose, by contradiction, that $L(\mathrm{T}_v^t \cup \{P_{rv}\}) \not\leq L(S_{rv})$. Since $|L(\mathrm{T}_v^t)| < |L(S_{rv})|$, there must exist a path $P'_{rv}$ such that $L(\mathrm{T}_v^t \cup \{P'_{rv}\}) \leq L(S_{rv})$ 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 $L(\mathrm{T}_v^t \cup \{P'_{rv}\}) \leq L(S_{rv})$, 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 $\mathrm{T}_v^t$ 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 $PQ$ in line 8, which in turn implies $L(\mathrm{T}_v^t) \not\leq L(S_{rv})$. 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 10 and 12). 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 $S_{rv}$ that are not already in $\mathrm{T}_v$ (see line 15).

We now prove that property 2) holds at time $t+1$. Assume first that the condition of line 7 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, $P_{ru} \notin S_{ru}$ and if there were a path $P_{rw}$, as in property 2), having $P_{ru}$ as prefix, then by Lemma 5 there would exist a vertex $y$ in $P_{rw}$ such that a path $P_{ry} = P_{ru} \oplus P_{uy}$ is in $S_{ry}$, for some solution $S_{ry}$. 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 7 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 5 there exists a vertex $y$ in $P_{rw}$ such that $P_{ry} = P_{rv} \oplus P_{vy}$ is in $S_{ry}$, for some solution $S_{ry}$. As $y$ is in the set of the general predecessors of $v$, 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 5) is false at time $t_e$. If $\forall\, w \neq r : |\mathrm{T}_w^{t_e}| = k$ then, by property 1), the algorithm has computed a correct solution for S-KSSP. 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 $P_{rw} \in \mathrm{T}_w^{t_e}$ or $L(\mathrm{T}_w^{t_e} \cup \{P_{rw}\}) \not\leq L(S_{rw})$. In both cases it follows that $L(\mathrm{T}_w^{t_e}) = L(S_{rw})$ and therefore that the algorithm has computed a correct solution for S-KSSP. □

Now, if we call $\mathrm{SP}(n, m)$ the time complexity of a shortest-path algorithm in an $n$-vertex $m$-edge graph,

the time complexity of Algorithm 4 is as follows:

**Theorem 6.** *Algorithm* BOUNDED-S-KSSP *runs in* $\mathcal{O}(k(m \log km + n^2 \text{SP}(n,m)))$ *time.*

*Proof.* Observe that the number of times the loop in line 5 of algorithm 4 is executed 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 11), where $\delta_v = |N^+(v)|$ is the outdegree 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: (i) in the worst case, the number of general predecessors of a vertex is at most $n$; (ii) we execute an algorithm for P-KSSP and enqueue at most $k$ paths per general predecessor. 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 an algorithm for solving P-KSSP. For this task, if we select Yen's algorithm, the best algorithm in terms of time complexity for this purpose, which runs in $\mathcal{O}(n \cdot kn \cdot \text{SP}(n,m)) = \mathcal{O}(kn^2 \text{SP}(n,m))$ time [46], we obtain a total running time for algorithm BOUNDED-S-KSSP of $\mathcal{O}(k(m+n)\log(k(m+n)) + kn^2 \text{SP}(n,m)) = \mathcal{O}(km \log km + kn^2 \text{SP}(n,m)) = \mathcal{O}(k(m \log km + n^2 \text{SP}(n,m)))$. □

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, the following corollary can be derived:

**Corollary 1.** *The running time of Algorithm 4 is asymptotically upper bounded by the running time of* $n-1$ *executions of Yen's algorithm.*

*Proof.* Observe that Algorithm BOUNDED-S-KSSP runs for $T(n,m,k) = \mathcal{O}(k(m \log km + n^2 \text{SP}(n,m)))$ 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 $\text{SP}(n,m)$ is $\mathcal{O}(m+n \log n)$ 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 that, in any graph such that $m = \Omega(n)$, $\mathcal{O}(km(n + \log m))$ is upper bounded by $\mathcal{O}(kn^2(m + n \log n))$ and therefore by the time taken by $\Theta(n)$ executions of Yen's algorithm. A similar result can be obtained if one considers unweighted graphs, by replacing $\text{SP}(n,m)$ with $\mathcal{O}(m+n)$. □

Note that Corollary 1 holds for any polynomial-time algorithm for P-KSSP that matches Yen's time complexity (e.g. PNC [46] or that in [27]). Note also that a lower level, more detailed description of Algorithm BOUNDED-S-KSSP is provided in Algorithm 5. In detail, set $A_{S,v}$ is computed through an iterative computation of the transitive closure of the relation predecessors (see Definition 2) that uses a membership set SEEN 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 a description, we use notation P-KSSP-ALGO$(G,r,x,k)$ to denote a generic call to an algorithm that solves P-KSSP that returns a collection of top-$k$ simple shortest paths for a pair $(r,x)$ of vertices in graph $G$. Super-saturated vertices are stored in a set named S-SAT.

# 5 Experimental Evaluation

In this section, we present the results of an extensive experimental evaluation aimed at assessing the average performance of Algorithm BOUNDED-S-KSSP. In fact, although its worst-case time complexity matches that of the best-known polynomial-time solution to S-KSSP, obtained by running a polynomial-time P-KSSP algorithm once for each vertex pair $(r,v)$, our method can often achieve significantly better performance in practice, mainly thanks to the effectiveness of the strategy that exploits shared path prefixes to determine

**Algorithm 5:** Detailed description of Algorithm 4.

**Input:** Graph $G = (V, E, \omega)$, root $r \in V$, $k \in \mathbb{N}$.
**Output:** Top-$k$ simple shortest paths $\mathrm{T}_v$ from $r$ to each $v \in V \setminus \{r\}$.

$PQ \leftarrow \emptyset$; ▷ Empty priority queue
**foreach** $v \in V$ **do** ▷ Empty lists
  $\mathrm{T}_v \leftarrow [\,]$;
$PQ.enqueue((r), \omega((r)) = 0)$;
S-SAT $\leftarrow \{r\}$; ▷ Super-saturated vertices
**while** $PQ \neq \emptyset$ **and** $\exists\ w \neq r : |T_w| < k$ **do**
  $\Pi = (r, \ldots, v) \leftarrow PQ.dequeueMin()$; ▷ Min-weight path
  **if** $|T_v| < k$ **then**
    Add $\Pi$ to $\mathrm{T}_v$;
    **foreach** $u \in N^+(v) \setminus V(\Pi)$ **do**
      **if** $u \notin$ *s-sat* **and** $\Pi \oplus (u) \notin PQ$ **then**
        $PQ.enqueue(\Pi \oplus (u), \omega(\Pi) + \omega(v, u))$;
  **else if** $v \notin$ *s-sat* **then**
    $A_{S,v} \leftarrow \emptyset$;
    $Q \leftarrow \{v\}$; ▷ Add $v$ to empty FIFO queue
    SEEN $\leftarrow \{v\}$;
    **while** $Q \neq \emptyset$ **do**
      $x \leftarrow Q.pop()$; ▷ Dequeue head of queue
      **if** $x \notin$ *s-sat* **then**
        **if** $|T_x| < k$ **then** $S_{rx} \leftarrow V(\text{P-KSSP-ALGO}(G, r, x, k))$;
        **else** $S_{rx} \leftarrow \mathrm{T}_x$;
        $A_{S,v} \leftarrow A_{S,v} \cup V(S_{rx})$;
        **foreach** $p \in S_{rx} \setminus T_x \ :\ p \notin PQ$ **do**
          $PQ.enqueue(p, \omega(p))$;
        **foreach** $w \in V(S_{rx})$ **do**
          **if** $w \notin$ *seen* **then**
            **if** $w \notin$ *s-sat* **then**
              SEEN $\leftarrow$ SEEN $\cup \{w\}$;
              $Q.append(w)$;
            S-SAT $\leftarrow$ S-SAT $\cup \{w\}$;
        S-SAT $\leftarrow$ S-SAT $\cup \{x\}$;

different top-$k$ collections from the root to other vertices. Thus, our main goal is to evaluate the average-case effectiveness of this strategy and determine whether Algorithm BOUNDED-S-KSSP outperforms the naive baseline, which applies a polynomial-time P-KSSP algorithm repeatedly. This comparison is critical to establish whether BOUNDED-S-KSSP constitutes an improvement over the state-of-the-art for S-KSSP. To this end, we implemented BOUNDED-S-KSSP and compared it with a generalization of Yen's algorithm [44], which solves S-KSSP by running Yen's method $n-1$ times, once for each pair $(r, v)$, with $r \in V$ and $v \in V \setminus r$. We refer to this baseline as Algorithm SS-YEN. For completeness, we also include a second baseline that also solves S-KSSP by executing $n-1$ times algorithm PNC [46], to handle P-KSSP for the same pairs. This baseline is selected since algorithm PNC has been shown empirically to outperform Yen's, for P-KSSP, in several real-world scenarios, despite sharing the same worst-case time complexity. We refer to this variant as Algorithm SS-PNC. For improved performance, our implementation of PNC integrates the efficient heuristic introduced in [18], which reduces the search effort by computing an upper bound on the weight of the $k$-th heaviest shortest path.

We do not consider approaches for P-KSSP whose performance is generally dominated by or comparable to that of PNC [17, 20], and methods with high space overhead (due to storing many shortest-path trees to accelerate the computation) [27]. Moreover, we exclude from our evaluation the algorithm of [26] specific for P-KSSP in undirected graphs, since empirical studies have found its implementation to be often slower than

| Dataset | graph | Type | $\|V\|$ | $\|E\|$ | D | W | $d_{\text{AVG}}$ | $d_{\text{MAX}}$ | $\Delta$ | S |
|---|---|---|---|---|---|---|---|---|---|---|
| Linux | LIN | blogging | 913 | 4162 | ● | ○ | 4.56 | 61 | 6 | ○ |
| Ca-GrQc | CAG | collabor. | 4158 | 13422 | ○ | ○ | 6.46 | 81 | 17 | ○ |
| SlashDot | SLD | friendship | 2196 | 22773 | ● | ○ | 10.37 | 1991 | 4 | ○ |
| Caida | CAI | ethernet | 32000 | 40204 | ○ | ● | 2.51 | 203 | 61 | ○ |
| P2p-Gnutella | GNU | peer2peer | 14149 | 50916 | ● | ○ | 3.60 | 32 | 9 | ○ |
| Luxembourg | LUX | road network | 30647 | 75546 | ● | ● | 2.47 | 9 | 490 | ○ |
| Amazon | AMZ | rating | 20168 | 104513 | ○ | ○ | 10.36 | 169 | 31 | ○ |
| Email EU | EMA | email | 34203 | 151132 | ● | ○ | 4.42 | 715 | 13 | ○ |
| Brightkite | BRI | location | 56739 | 212945 | ○ | ○ | 7.51 | 1134 | 30 | ○ |
| youtube | YTB | media | 28372 | 249511 | ○ | ○ | 17.59 | 8920 | 13 | ○ |
| Barabasi-Albert | BAR | random power law | 71920 | 299480 | ● | ○ | 4.16 | 571 | 17 | ● |
| Mocnik | MOC | rand. spatial | 18848 | 318248 | ● | ● | 16.88 | 160 | 1168 | ● |
| ArXiv | ARX | citation | 34401 | 420784 | ● | ○ | 24.46 | 15 | 846 | ○ |
| cisco | CIS | distributed system | 30181 | 424232 | ● | ○ | 14.06 | 18589 | 12 | ○ |
| facebook | FAC | social network | 37165 | 494168 | ○ | ○ | 26.59 | 950 | 17 | ○ |
| Epinions | EPI | reviews | 41441 | 693507 | ● | ○ | 16.73 | 1820 | 17 | ○ |
| Bio-Markers | BIO | biological | 4858 | 982944 | ○ | ● | 404.67 | 2242 | 94555 | ○ |
| Erdős-Rényi | ERD | random uniform | 10000 | 999809 | ● | ○ | 99.98 | 138 | 3 | ● |

(a) Input graphs used in trials with $k \in \{2, 4, 8, 16\}$.

| Dataset | graph | Type | $\|V\|$ | $\|E\|$ | D | W | $d_{\text{AVG}}$ | $d_{\text{MAX}}$ | $\Delta$ | S |
|---|---|---|---|---|---|---|---|---|---|---|
| Italy | ITA | train network | 1329 | 3862 | ● | ● | 2.91 | 13 | 890 | ○ |
| Scale-Free | SCF | random scale free | 1050 | 4112 | ● | ○ | 3.92 | 201 | 14 | ● |
| bitcoin | BTC | financial transactions | 5875 | 21489 | ○ | ● | 7.32 | 795 | 34 | ○ |
| Oregon-AS | ORE | autonomous system | 10670 | 22002 | ○ | ○ | 4.12 | 2312 | 10 | ○ |
| Google | GOO | web index | 12354 | 164046 | ● | ○ | 13.28 | 120 | 6 | ○ |

(b) Input graphs used in trials with $k \in \{2^i\}_{i=1,2,\ldots,10}$.

Table 1: Overview of used input graphs: columns 1st to 3rd provide dataset name, graph acronym, type of network; columns 4th and 5th report number of vertices and arcs while columns 6th (**D**) and 7th (**W**) indicate whether the graph is directed and/or weighted (● = true, ○ = false); columns 8th ($d_{\text{AVG}}$), 9th ($d_{\text{MAX}}$) and 10th ($\Delta$) contain, respectively, the average and maximum (outgoing) degree and the diameter of the graph; the last column (**S**) shows whether the graph is synthetic (●) or real (○). Inputs are sorted by $|E|$, non-decreasing.

that of Yen's [34]. For fairness, we implemented and tested two variants of Algorithm BOUNDED-S-KSSP: one using Yen's algorithm to compute general predecessors (see line 14 of Algorithm 4 or line 20 of Algorithm 5), and another using PNC. We refer to these as Algorithms BND-YEN and BND-PNC, respectively.

All algorithms have been implemented with support for both undirected and unweighted graphs. This is achieved by removing edge orientations (i.e. the distinction between $N^-$ and $N^+$) or by assuming unit weights and replacing Dijkstra-like traversals with BFS-like ones, as in previous works on shortest paths [2,12,15,16]. For completeness, we implemented and tested algorithm PRUNED-S-KSSP as well. However, preliminary experimentation reveals that its running time is not infrequently huge even for small graphs and $k$, likely due to the presence of topological configurations resembling that in Fig. 1b. Hence, we do not consider this algorithm in our experimental framework and focus exclusively on polynomial-time algorithms.

## 5.1 Test Environment

All algorithms were implemented using NetworKit [42] and NetworkX [22], two widely adopted toolkits for testing graph algorithms. Our code is written in Python, with selected performance-critical routines reimplemented in Cython and Rust. Experiments were carried out using Python 3.13 on a Linux system (Kernel 6.8.0-47) running on a workstation equipped with an Intel Intel Core i9-11900KF© clocked at 3.50GHz and 96 GB RAM. Following previous studies on shortest-path algorithms [2,3,14,15], as input to our

| graph | $k=2$ | | | $k=4$ | | | $k=8$ | | | $k=16$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U |
| | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | |
| LIN | 0.26 | 0.02 | 14.79 | 0.65 | 0.07 | 8.75 | 1.48 | 0.22 | 6.99 | 4.28 | 1.07 | 4.08 |
| CAG | 2.00 | 0.16 | 12.26 | 7.31 | 0.42 | 19.11 | 26.92 | 1.38 | 19.53 | 43.53 | 12.15 | 3.61 |
| SLD | 3.03 | 0.22 | 14.27 | 6.21 | 0.46 | 14.23 | 19.68 | 7.64 | 2.66 | 37.88 | 14.88 | 3.70 |
| CAI | 1046.04 | 1.25 | 931.92 | 3008.32 | 2.86 | 1264.17 | 7402.08 | 31.24 | 617.47 | 12095.90 | 117.80 | 593.08 |
| GNU | 24.73 | 2.30 | 10.98 | 69.97 | 18.77 | 3.75 | 175.83 | 75.74 | 2.33 | 391.66 | 229.21 | 1.71 |
| LUX | 32510.60 | 3.27 | 9566.80 | 65648.57 | 25.88 | 3677.38 | 232525.70 | 190.17 | 1222.71 | 224704.82 | 1004.18 | 223.77 |
| AMZ | 138.63 | 1.91 | 72.50 | 357.48 | 3.74 | 95.53 | 639.92 | 15.92 | 38.78 | 1199.77 | 68.97 | 19.15 |
| EMA | 21.49 | 1.07 | 20.17 | 60.64 | 2.13 | 28.50 | 140.85 | 4.98 | 28.30 | 317.07 | 12.67 | 25.03 |
| 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 |
| YTB | 303.58 | 3.96 | 76.88 | 1392.39 | 15.02 | 94.33 | 2067.67 | 40.85 | 53.95 | 7491.01 | 765.61 | 11.54 |
| BAR | 107.78 | 5.35 | 20.15 | 359.33 | 45.33 | 7.93 | 787.97 | 148.28 | 5.31 | 1536.22 | 424.09 | 3.62 |
| MOC | 7247.56 | 2.18 | 3349.80 | 30169.19 | 5.76 | 5237.91 | 68681.53 | 17.25 | 4089.43 | 100914.23 | 113.12 | 939.80 |
| ARX | 116.29 | 7.56 | 15.34 | 252.98 | 15.37 | 16.56 | 680.67 | 51.82 | 13.64 | 1222.15 | 164.00 | 8.16 |
| CIS | 1456.39 | 3.62 | 402.07 | 5240.55 | 43.25 | 121.16 | 11972.64 | 19.28 | 620.94 | 27380.17 | 406.29 | 67.39 |
| FAC | 216.45 | 8.50 | 25.45 | 443.46 | 23.03 | 19.25 | 919.75 | 80.63 | 11.41 | 2214.98 | 316.76 | 6.99 |
| 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 |
| BIO | 920.54 | 25.10 | 36.68 | 2143.79 | 63.35 | 34.05 | 7041.90 | 313.83 | 23.79 | 16838.18 | 2500.08 | 6.80 |
| ERD | 4.76 | 3.43 | 1.39 | 13.01 | 7.11 | 1.83 | 33.00 | 18.47 | 1.83 | 83.38 | 51.90 | 1.69 |

| graph | $k=2$ | | | $k=4$ | | | $k=8$ | | | $k=16$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U |
| | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | |
| LIN | 5.22 | 0.13 | 54.73 | 5.64 | 0.57 | 10.48 | 6.09 | 1.00 | 6.43 | 7.35 | 2.04 | 3.62 |
| CAG | 152.30 | 0.63 | 286.62 | 167.91 | 2.98 | 89.33 | 165.33 | 6.92 | 26.91 | 198.91 | 53.28 | 3.79 |
| SLD | 56.20 | 0.29 | 245.53 | 60.28 | 2.54 | 31.90 | 98.33 | 34.15 | 2.94 | 134.92 | 56.26 | 3.81 |
| CAI | 19138.74 | 31.45 | 4918.95 | 19824.46 | 38.59 | 535.36 | 23463.01 | 146.96 | 242.71 | 28526.35 | 463.62 | 137.19 |
| GNU | 3135.32 | 304.80 | 10.38 | 3247.06 | 902.18 | 3.60 | 3415.10 | 1515.82 | 2.26 | 3812.58 | 2108.34 | 1.81 |
| LUX | 25634.25 | 24.14 | 3004.36 | 28130.26 | 164.41 | 180.38 | 31067.73 | 339.07 | 99.86 | 42650.01 | 671.46 | 74.32 |
| AMZ | 6700.12 | 5.87 | 1235.82 | 6985.21 | 10.96 | 645.14 | 8336.40 | 211.47 | 42.40 | 9061.01 | 668.95 | 18.59 |
| EMA | 16882.38 | 9.42 | 1999.62 | 17959.84 | 39.10 | 482.75 | 18420.20 | 101.50 | 181.75 | 21227.37 | 336.61 | 64.68 |
| BRI | 54469.45 | 97.31 | 697.01 | 58559.41 | 117.47 | 555.36 | 62385.45 | 1487.89 | 43.38 | 72495.34 | 4527.03 | 16.13 |
| YTB | 15215.78 | 13.94 | 1158.48 | 16949.68 | 67.88 | 371.37 | 18787.27 | 438.53 | 50.10 | 26439.27 | 2427.20 | 12.27 |
| BAR | 99439.21 | 3458.27 | 28.75 | 121946.11 | 14027.66 | 8.69 | 122631.33 | 28558.34 | 4.29 | 135304.94 | 55305.18 | 2.45 |
| MOC | 12973.91 | 2.55 | 5259.79 | 13945.51 | 6.39 | 2215.55 | 14379.65 | 73.76 | 195.35 | 15803.83 | 217.69 | 72.91 |
| ARX | 25407.09 | 34.45 | 1391.62 | 26276.72 | 135.41 | 217.26 | 26875.59 | 1350.72 | 29.43 | 29629.26 | 3518.56 | 10.35 |
| CIS | 19199.88 | 68.77 | 279.19 | 26491.83 | 406.70 | 65.14 | 38130.15 | 86.40 | 441.32 | 65096.57 | 1536.69 | 42.36 |
| FAC | 29367.16 | 30.18 | 973.15 | 29120.78 | 608.44 | 47.86 | 30483.41 | 1966.75 | 15.50 | 32412.54 | 3131.96 | 10.35 |
| EPI | 37512.01 | 34.67 | 1546.97 | 43685.82 | 263.12 | 169.27 | 43349.16 | 938.64 | 49.82 | 46936.12 | 1547.90 | 30.48 |
| BIO | 13237.96 | 42.40 | 342.50 | 13076.11 | 171.61 | 85.93 | 12921.53 | 599.54 | 23.25 | 13276.35 | 1220.97 | 11.54 |
| ERD | 5244.63 | 77.83 | 67.67 | 4954.62 | 319.84 | 15.50 | 5116.20 | 759.26 | 6.74 | 5202.65 | 1467.91 | 3.54 |

Table 2: Results of the executions of SS-YEN and BND-YEN (top) and of SS-PNC and BND-PNC (bottom) for $k \in \{2, 4, 8, 16\}$.

tests we employ a large dataset that includes both real-world graphs, from publicly available repositories [28, 38], and synthetic instances, generated via well-known random models (e.g., *Erdős-Rényi* or *Barabási-Albert* models) [7], with heterogeneous sizes, structures, orientations, and weight distributions. Details on used inputs are given in Table 1. For each dataset we extract the largest (simply or strongly) connected component.

## 5.2 Executed Tests

For each graph of Table 1a and $k \in \{2, 4, 8, 16\}$ we execute BND-YEN, BND-PNC, SS-YEN, and SS-PNC, measuring their running times to solve S-KSSP. Depending on whether the graph is directed or undirected, and weighted or unweighted, we use the appropriate implementation of each method to handle the specific case. The choice of $k$ is inspired by real-world applications of top-$k$ simple shortest paths, where the number of required paths rarely exceeds a few tens [2, 12, 36]. For completeness and to evaluate the scalability of the algorithms against $k$ we also extend our experimentation to significantly larger values of $k$: specifically, for a selection of graphs (see Table 1b), we run and measure the running time of the four algorithms with $k \in \{2^i\}_{i=1,2,\ldots,10}$. The selection of graphs is imposed by the large running times of the algorithms under study when $k$ increases substantially. To mitigate the bias from the choice of the root, we performed three independent runs for each combination of graph and $k$, selecting three different roots uniformly at random from the graph's vertex set: we report running times averaged over these runs. For validity, we verify the correctness of solutions to S-KSSP calculated by our implementations by comparing them with those produced by the NetworkX standard implementation of Yen's algorithm.

| graph | $k=2$ | | | $k=4$ | | | $k=8$ | | | $k=16$ | | | $k=32$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U |
| | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | |
| 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 | 426.62 | 48.13 | 9.01 |
| SCF | 0.17 | 0.02 | 11.03 | 0.48 | 0.05 | 9.61 | 1.13 | 0.20 | 6.08 | 2.89 | 0.73 | 4.13 | 7.29 | 2.35 | 3.17 |
| 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 | 1407.71 | 264.26 | 5.33 |
| ORE | 12.10 | 0.33 | 36.82 | 35.36 | 0.85 | 41.82 | 129.06 | 6.21 | 34.46 | 227.04 | 14.97 | 19.00 | 593.86 | 74.54 | 8.60 |
| 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 | 574.69 | 177.38 | 3.22 |

| graph | $k=64$ | | | $k=128$ | | | $k=256$ | | | $k=512$ | | | $k=1024$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U |
| | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | | SS-YEN | BND-YEN | |
| ITA | 1017.23 | 183.41 | 6.48 | 1947.49 | 532.18 | 3.97 | 3717.61 | 1388.11 | 2.83 | 11375.59 | 6425.41 | 1.79 | 28667.78 | 16576.38 | 1.74 |
| SCF | 19.67 | 7.14 | 2.77 | 54.41 | 23.88 | 2.29 | 158.97 | 77.15 | 2.06 | 522.59 | 304.93 | 1.71 | 1898.88 | 1232.71 | 1.54 |
| BTC | 3729.68 | 1023.93 | 3.68 | 7264.18 | 2605.65 | 2.79 | 16805.06 | 6245.23 | 2.70 | 53422.27 | 20644.78 | 2.59 | 106783.87 | 42113.28 | 2.54 |
| ORE | 1382.01 | 268.55 | 5.28 | 3763.50 | 1065.94 | 3.66 | 6079.05 | 1885.11 | 3.29 | 18373.12 | 8258.64 | 2.27 | 41508.45 | 23215.69 | 1.79 |
| GOO | 973.13 | 418.53 | 2.45 | 1863.27 | 997.38 | 1.89 | 4977.50 | 3230.20 | 1.54 | 15314.60 | 10556.06 | 1.45 | 43984.99 | 30792.52 | 1.43 |

| graph | $k=2$ | | | $k=4$ | | | $k=8$ | | | $k=16$ | | | $k=32$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U |
| | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | |
| ITA | 18.01 | 1.07 | 28.65 | 19.36 | 2.22 | 8.82 | 26.89 | 6.47 | 4.16 | 36.04 | 10.19 | 3.54 | 72.44 | 31.13 | 2.41 |
| SCF | 6.41 | 0.11 | 142.95 | 7.05 | 0.42 | 17.24 | 7.92 | 0.79 | 10.13 | 11.09 | 2.56 | 4.76 | 16.34 | 4.87 | 3.65 |
| BTC | 588.49 | 5.16 | 114.05 | 591.15 | 14.43 | 40.98 | 628.30 | 44.13 | 14.24 | 694.49 | 100.13 | 6.94 | 954.04 | 191.21 | 4.99 |
| ORE | 1081.73 | 0.37 | 3078.67 | 1199.27 | 0.83 | 1467.22 | 1358.87 | 6.36 | 478.31 | 1722.24 | 15.39 | 139.40 | 2550.65 | 76.59 | 35.29 |
| GOO | 2379.78 | 2.66 | 895.49 | 2465.44 | 9.40 | 262.32 | 2534.13 | 427.69 | 5.93 | 3135.56 | 718.90 | 4.36 | 3844.48 | 1221.88 | 3.15 |

| graph | $k=64$ | | | $k=128$ | | | $k=256$ | | | $k=512$ | | | $k=1024$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U | RUNTIME (s) | | SP-U |
| | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | | SS-PNC | BND-PNC | | SS-PNC | BND-YEN | | SS-PNC | BND-PNC | |
| ITA | 167.04 | 80.78 | 2.16 | 360.45 | 193.15 | 1.92 | 612.22 | 397.55 | 1.56 | 1089.68 | 769.16 | 1.42 | 2246.45 | 1832.29 | 1.23 |
| SCF | 31.11 | 12.19 | 2.63 | 56.95 | 27.02 | 2.16 | 104.73 | 54.44 | 1.94 | 230.52 | 142.83 | 1.62 | 474.47 | 332.20 | 1.43 |
| BTC | 1374.11 | 460.84 | 2.98 | 3873.14 | 1601.16 | 2.42 | 6353.60 | 2786.40 | 2.28 | 8467.54 | 4432.84 | 1.91 | 18418.47 | 10017.47 | 1.84 |
| ORE | 3948.19 | 270.99 | 15.56 | 8475.54 | 1082.66 | 8.14 | 13052.88 | 1948.15 | 6.79 | 36193.34 | 8397.07 | 4.31 | 76662.79 | 22372.69 | 3.40 |
| GOO | 5400.40 | 1853.90 | 2.91 | 7239.20 | 4101.81 | 1.76 | 15800.12 | 10926.51 | 1.45 | 31077.07 | 20510.12 | 1.52 | 66743.96 | 44410.96 | 1.50 |

Table 3: Results of the executions of SS-YEN and BND-YEN (top) and of SS-PNC and BND-PNC (bottom) for $k \in \{2^i\}_{i=1,2,\ldots,10}$.

## 5.3 Analysis

The results of our experiments for $k \in \{2, 4, 8, 16\}$ are summarized in Table 2. For each graph and $k$, we report the average running time (in seconds) spent by SS-YEN (SS-PNC, respectively) and BND-YEN (BND-PNC, respectively) to solve an instance of S-KSSP, along with the average *speed-up* (column SP-U), defined as the average ratio of the runtime of SS-YEN (SS-PNC, respectively) to the runtime of BND-YEN (BND-PNC, respectively). This ratio quantifies how much the latter is faster than the former at solving S-KSSP. The main conclusion that can be drawn from the data of Table 2 is that both versions of Algorithm BOUNDED-S-KSSP outperform the two baselines SS-YEN and SS-PNC in handling S-KSSP. In particular, BND-YEN is always faster than SS-YEN in all tested instances. Similarly, BND-PNC outperforms SS-PNC. In more detail, the runtime of SS-YEN is observed to be larger than that of BND-YEN by factors that range from a minimum of 1.39 (instance ERD, $k = 2$) to a maximum of several thousands (e.g, graphs LUX for $k = 2$ or CAI for $k = 4$ or MOC for $k = 8$). Similarly, the measured runtime of SS-PNC is always larger than that of BND-PNC, by factors that range from a minimum of 3.15 (instance LIN, $k = 16$) to a maximum of thousands (e.g. instance YTB for $k = 2$). In general, in the majority of our experiments Algorithms BND-YEN and BND-PNC are more than one order of magnitude faster than the respective baselines, i.e. SS-YEN and SS-PNC. Even more remarkably, the execution of the two variants of BOUNDED-S-KSSP often terminates within a few seconds and never lasts more than around an hour and a half, while SS-YEN or SS-PNC can take several hours to solve a single instance of S-KSSP even for small $k$ (see, e.g., graph LUX for $k = 2$ or CIS with $k = 8$). This suggests that BOUNDED-S-KSSP scales better w.r.t. graph size and can be successfully used in real-world applications that manage large graphs (for $k \ll n$), while SS-YEN or SS-PNC are impractical even for medium-sized graphs and small $k$. This effectiveness of BOUNDED-S-KSSP is also supported by the data in Table 3 where we report the runtime of all algorithms when applied on smaller graphs but with substantially larger values of $k$, and the speed-up provided by our new solution. This experiment was designed to evaluate how the speed-up varies with $k$. Values of such parameter here were selected by a doubling strategy to highlight observed trends [32]. From such data, we notice that the speed-up tends to decrease as $k$ increases; this is fairly expected since, as $k$ grows, the number of general predecessors that are not saturated during the execution of BOUNDED-S-KSSP (see line 14) increases, and, with it, the number of times a solution to P-KSSP has to be computed increases, tending to the upper bound of $n - 1$. This is clearly related to the fact that, topologically speaking, the

number of prefixes, shared by top-$k$ simple shortest paths from the root to different vertices, tends to decrease as $k$ increases. Hence, computing solutions together, as done by algorithms BND-YEN and BND-PNC, tends to become less effective with $k$ increasing. Still, BND-YEN and BND-PNC remain faster than SS-YEN and SS-PNC even for the largest graphs and the highest values of $k$. Other empirical insights can be derived by comparing the algorithms' runtimes for a fixed graph and value of $k$. For example, for $k \in \{2, 4, 8, 16\}$ (see Table 2), in weighted digraphs (e.g. LUX or MOC) SS-PNC runs faster than SS-YEN while BND-PNC is slightly slower than BND-YEN, hence a slightly reduced speed-up is observed. Viceversa, in all other cases, SS-YEN and BND-YEN exhibit smaller runtimes compared to SS-PNC and BND-PNC. The observed performance changes significantly for larger $k$ (see Table 3), where PNC benefits of the precomputed shortest-path tree to reduce the number of shortest-path computations necessary to complete a solution to P-KSSP. This results in shorter runtimes for both SS-PNC and BND-PNC when $k \geq 64$, suggests that these algorithms should be used for large $k$, and confirms that PNC is better than Yen's for P-KSSP when $k$ increases. [1]

# 6 Conclusion

In this paper, we studied the Single-Source Top-$k$ Simple Shortest Paths problem (S-KSSP) and introduced BOUNDED-S-KSSP, the first algorithm specifically designed to solve it, based on a careful analysis of some key structural properties of the problem's solutions. We proved that the time complexity of BOUNDED-S-KSSP matches that of the best-known existing methods. However, extensive experiments show that it significantly outperforms such methods in practice, often by orders of magnitude, making it the preferred choice to handle S-KSSP in real-world scenarios.

A promising direction for future work is to improve BOUNDED-S-KSSP either asymptotically, by reducing its time complexity to below $\mathcal{O}(n)$ times that of the best P-KSSP algorithm, or empirically, by enhancing its practical performance through heuristics. To support this, we plan to extend our experimental analysis to better identify the algorithm's computational bottlenecks. Finally, a major open question is whether the strategy underlying BOUNDED-S-KSSP can be generalized to efficiently compute the $k$ simple shortest paths for all vertex pairs. Currently, no polynomial-time algorithm is known, for general values of $k$, which needs $o(\binom{n}{2})$ individual P-KSSP computations. Addressing this challenge would mark a significant advancement in the field.

# Acknowledgments

This work was supported by: National Group for Scientific Computation of Istituto Nazionale di Alta Matematica (GNCS-INdAM); and European Union under the Italian National Recovery and Resilience Plan of NextGenerationEU, partnership on "Telecommunications of the Future" (program RESTART), project MoVeOver/SCHEDULE ("Smart interseCtions witH connEcteD and aUtonomous vehicLEs", CUP J33C22002880001).

[1] All our source codes and used inputs can be accessed at https://tinyurl.com/3uew6rx9.

# References


[1] Udit Agarwal and Vijaya Ramachandran. Finding k simple shortest paths and cycles. In Seok-Hee Hong, editor, *Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *LIPIcs*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[2] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In Blai Bonet and Sven Koenig, editors, *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 2–8. AAAI Press, 2015.

[3] Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms*, 12(7):127, 2019.

[4] Shikha Anirban, Junhu Wang, Md. Saiful Islam, Humayun Kayesh, Jianxin Li, and Mao Lin Huang. Compression techniques for 2-hop labeling for shortest distance queries. *World Wide Web*, 25(1):151–174, 2022.

[5] Claudia Archetti, Nicola Bianchessi, and Maria Grazia Speranza. Optimal solutions for routing problems with profits. *Discrete Applied Mathematics*, 161(4-5):547–557, 2013.

[6] Fritz Bökler and Markus Chimani. Approximating multiobjective shortest path in practice. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 120–133. SIAM, 2020.

[7] Béla Bollobás. *Random Graphs, Second Edition*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 2011.

[8] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Jian Pei. Efficiently computing top-k shortest path join. In Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens, editors, *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, pages 133–144. OpenProceedings.org, 2015.

[9] Bin Chen, Di Leng, Boling Chen, Xiang Huang, and Wenpeng Wu. Lightweight and secure encryption of sensitive data in power communication transmission networks. In *2025 2nd International Conference on Smart Grid and Artificial Intelligence (SGAI)*, pages 812–816. IEEE, 2025.

[10] Israel Cidon, Shay Kutten, Yishay Mansour, and David Peleg. Greedy packet scheduling. *SIAM Journal on Computing*, 24(1):148–157, 1995.

[11] David Coudert, Andrea D'Ascenzo, and Clément Rambaud. k-shortest simple paths in bounded treewidth graphs. *Theoretical Computer Science*, 1039:115182, 2025.

[12] Andrea D'Ascenzo and Mattia D'Emidio. Top-k distance queries on large time-evolving graphs. *IEEE Access*, 11:102228–102242, 2023.

[13] Andrea D'Ascenzo and Mattia D'Emidio. On mining dynamic graphs for k shortest paths. In Luca Maria Aiello, Tanmoy Chakraborty, and Sabrina Gaito, editors, *Social Networks Analysis and Mining - 16th International Conference, ASONAM 2024, Rende, Italy, September 2-5, 2024, Proceedings, Part I*, volume 15211 of *Lecture Notes in Computer Science*, pages 320–336. Springer, 2024.

[14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Robust distance queries on massive networks. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA 2014)*, volume 8737 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2014.

[15] Mattia D'Emidio, Luca Forlizzi, Daniele Frigioni, Stefano Leucci, and Guido Proietti. Hardness, approximability, and fixed-parameter tractability of the clustered shortest-path tree problem. *J. Comb. Optim.*, 38(1):165–184, 2019.

[16] David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998.

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

[18] Wang Feng, Shiyang Chen, Hang Liu, and Yuede Ji. Peek: A prune-centric approach for K shortest path computation. In Dorian Arnold, Rosa M. Badia, and Kathryn M. Mohror, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, pages 18:1–18:14. ACM, 2023.

[19] Funa Fukui, Tomotaka Kimura, Yutaka Fukuchi, and Kouji Hirata. Grouping-based dynamic routing, core, and spectrum allocation method for avoiding spectrum fragmentation and inter-core crosstalk in multi-core fiber networks. *Future Internet*, 17(6), 2025.

[20] Jun Gao, Huida Qiu, Xiao Jiang, Tengjiao Wang, and Dongqing Yang. Fast top-k simple shortest paths discovery in graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, page 509–518, New York, NY, USA, 2010. Association for Computing Machinery.

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

[22] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[23] Riko Jacob. *Shortest Paths Approaches for Timetable Information*, pages 837–838. Springer US, Boston, MA, 2008.

[24] Jerald Jariyasunant, Daniel B Work, Branko Kerkez, Raja Sengupta, Steven Glaser, and Alexandre Bayen. Mobile transit trip planning with real-time data, 2011.

[25] Bin Jiang and Christophe Claramunt. Topological analysis of urban street networks. *Environment and Planning B: Planning and Design*, 31(1):151–162, 2004.

[26] Naoki Katoh, Toshihide Ibaraki, and Hisashi Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982.

[27] Denis Kurz and Petra Mutzel. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. In Seok-Hee Hong, editor, *Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *LIPIcs*, pages 49:1–49:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[28] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, jun 2014.

[29] Jiajia Li, Xing Xiong, Lei Li, Dan He, Chuanyu Zong, and Xiaofang Zhou. Finding top-k optimal routes with collective spatial keywords on road networks. In *Proceedings of the 39th IEEE International Conference on Data Engineering (ICDE 2023)*, pages 368–380. IEEE, 2023.

[30] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. Finding top-k shortest paths with diversity. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE 2018)*, pages 1761–1762. IEEE Computer Society, 2018.

[31] Zihan Luo, Lei Li, Mengxuan Zhang, Wen Hua, Yehong Xu, and Xiaofang Zhou. Diversified top-k route planning in road network. *Proc. VLDB Endow.*, 15(11):3199–3212, July 2022.

[32] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

[33] K Nitheesh and BK Bhavathrathan. Disaster-resilient transport network design considering risk-averse evacuation traffic demand. *Transportmetrica A: Transport Science*, pages 1–31, 2025.

[34] Marta Pascoal. Implementations and empirical comparison of k shortest loopless path algorithms, 2006.

[35] Marta M.B. Pascoal and Antonio Sedeño-Noda. Enumerating k best paths in length order in dags. *European Journal of Operational Research*, 221(2):308–316, 2012.

[36] Syed Md. Mukit Rashid, Mohammed Eunus Ali, and Muhammad Aamir Cheema. DeepAltTrip: Top-K Alternative Itineraries for Trip Recommendation. *IEEE Transactions on Knowledge & Data Engineering*, 35(09):9433–9447, September 2023.

[37] Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Trans. Algorithms*, 8(4), October 2012.

[38] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, page 4292–4293. AAAI Press, 2015.

[39] Robert Sedgewick. *Algorithms in c, part 5: graph algorithms, third edition*. Addison-Wesley Professional, third edition, 2001.

[40] Guan-Rong Shih and Pei-Hsuan Tsai. Safest-path planning approach for indoor fire evacuation. *International Journal of Disaster Risk Reduction*, 93:103760, 2023.

[41] Yu-Keng Shih and Srinivasan Parthasarathy. A single source $k$-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinform.*, 28(12):49–58, 2012.

[42] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Netw. Sci.*, 4(4):508–530, 2016.

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

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

[45] Junling Yuan, Xuyang Hao, Qiang Guo, Xuhong Li, and Qikun Zhang. Service provisioning for anycast requests in semi-filterless optical networks with spectrum and it resource balancing. *Optical Fiber Technology*, 93:104263, 2025.

[46] Ali Al Zoobi, David Coudert, and Nicolas Nisse. Finding the $k$ shortest simple paths: Time and space trade-offs. *ACM J. Exp. Algorithmics*, 28:1.11:1–1.11:23, 2023.