Finding k Simple Shortest Paths and Cycles

Udit Agarwal * and Vijaya Ramachandran*

Abstract

We present algorithms and techniques for several problems related to finding multiple simple shortest paths and cycles in a graph. Our main result is a new algorithm for finding k simple shortest paths for all pairs of vertices in a weighted directed graph $G = (V, E)$. For $k = 2$ our algorithm runs in $O(mn + n^2 \log n)$ time where m and n are the number of edges and vertices in G. For $k = 3$ our algorithm runs in $O(mn^2 + n^3 \log n)$ time, which is almost a factor of n faster than the best previous algorithm.

Our approach is based on forming suitable path extensions to find simple shortest paths; this method is different from the 'detour finding' technique used in most of the prior work on simple shortest paths, replacement paths, and distance sensitivity oracles.

We present new algorithms for generating simple cycles and simple paths in G in non-decreasing order of their weight. The algorithm for generating simple paths is much faster, and uses another variant of path extensions.

1998 ACM Subject Classification **G.2.2. Graph Theory** Graph Algorithms

Keywords and phrases Graph Algorithms, Shortest Paths, k Simple Shortest Paths, Enumerating Simple Cycles, Enumerating Simple Paths

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.60

1 Introduction

We present new algorithms and fundamentally new techniques for several problems related to finding multiple simple shortest paths and cycles in a graph.

Computing shortest paths in a weighted directed graph is a very well-studied problem. Let $G = (V, E)$ be a directed graph with non-negative edge weights, with $|V| = n$, $|E| = m$. A shortest path for a single pair of vertices in G, or for a single source, can be computed in $\tilde{O}(m)$ time using Dijkstra's algorithm, and the all pairs shortest paths (APSP) can be computed in $\tilde{O}(mn)$ time [4], where $\tilde{O}$ hides $polylog(n)$ factors.

A related problem is one of computing a sequence of k shortest paths, for $k > 1$. If the paths need not be simple, the problem of generating k shortest paths is well understood, and the most efficient algorithm is due to Eppstein [8], which has the following bounds — $O(m + n \log n + k)$ for a single pair of vertices and $O(m + n \log n + kn)$ for single source.

In the *k simple shortest paths (k-SiSP)* problem, given a pair of vertices s, t, the output is a sequence of k simple paths from s to t, where the i-th path in the collection is a shortest simple path in the graph that is not identical to any of the $i - 1$ paths preceding it in the output. (Note that these k simple shortest paths need not have the same weight.) It is noted in [8] that the k-SiSP problem is more common than the version where a path can contain cycles.

In this paper we consider the problem of generating multiple simple shortest paths (SiSP) and cycles (SiSC) in a weighted directed graph under the following set-ups: the k simple

* Dept. of Computer Science, University of Texas, Austin TX 78712. Email: udit@cs.utexas.edu, vlr@cs.utexas.edu. This work was supported in part by NSF Grant CCF-1320675. The first author's research was also supported in part by a Calhoun Fellowship.

shortest paths for all pairs of vertices (k-APSiSP), k simple shortest paths in the overall graph (k-All-SiSP), and the corresponding problem of finding simple shortest cycles in the overall graph (k-All-SiSC). We obtain significantly faster algorithms for k-APSiSP for small values of k, and fast algorithms, that also appear to be the first nontrivial algorithms, for the remaining two problems for all $k \geq 1$. Implicit in our method for k-All-SiSC are new algorithms for finding k simple shortest cycles through a specified vertex (k-SiSC) and through every vertex (k-ANSiSC) in weighted directed graphs.

The techniques we use in our algorithms are of special interest: We use two *path extension* techniques, a new method for k-APSiSP, and another for k-All-SiSP that is related to a method used in [5] for fully dynamic APSP, but which is still new for the context in which we use it.

1.1 Related Work

For the case when the k shortest paths need not be simple, the all-pairs version (k-APSP) was considered in the classical papers of Lawler [15, 16] and Minieka [17]. The most efficient current algorithm for k-APSP runs the k-SSSP algorithm in [8] on each of the n vertices in turn, leading to a bound of $O(mn + n^2 \log n + kn^2)$. It was noted in Minieka [17] that the all-pairs version of k shortest paths becomes significantly harder when simple paths are required, i.e., that the problem we study here, k-APSiSP, appears to be significantly harder than k-APSP.

Even for a single source-sink pair, the problem of generating k simple shortest paths (k-SiSP) is considerably more challenging than the unrestricted version considered in [8]. Yen's algorithm [24] finds the k simple shortest paths for a specific pair of vertices in $O(k \cdot (mn + n^2 \log n))$. This time bound was improved slightly [9], using Pettie's faster APSP algorithm [18], to $O(k(mn + n^2 \log\log n))$. On the other hand, it is shown in [23] that if the *second* simple shortest path for a single source-sink pair (i.e., $k = 2$ in k-SiSP) can be found in $O(n^{3-\delta})$ time for some $\delta > 0$, then APSP can also be computed in $O(n^{3-\alpha})$ time for some $\alpha > 0$; the latter is a major open problem. Thus, for dense graphs, where $m = \Theta(n^2)$, we cannot expect to improve the $\tilde{O}(mn)$ bound, even for 2-SiSP, unless we solve a major and long-standing open problem for APSP.

The k-SiSP problem is much simpler in the undirected case and is known to be solvable in $O(k(m + n \log n))$ time [14]. For unweighted directed graphs, Roditty and Zwick [19] gave an $\tilde{O}(km\sqrt{n})$ randomized algorithm for directed k-SiSP. They also showed that k-SiSP can be solved with $O(k)$ executions of an algorithm for the 2-SiSP problem.

A problem related to 2-SiSP is the *replacement paths* problem. In the s-t version of this problem, we need to output a shortest path from s to t when an edge on the shortest path p is removed; the output is a collection of $|p|$ paths, each a shortest path from s to t when an edge on p is removed. Clearly, given a solution to the s-t replacement paths problem, the second shortest path from s to t can be computed as the path of minimum weight in this solution. This is essentially the method used in all prior algorithms for 2-SiSP (and with modifications, for k-SiSP), and thus the current fastest algorithms for 2-SiSP and replacement paths have the same time bound. For the all-pairs case that is of interest to us, the output for the replacement paths problem would be $O(n^3)$ paths, where each path is shortest for a specific vertex pair, when a specific edge in its shortest path is removed. In view of the large space needed for this output, in the all-pairs version of replacement paths, the problem of interest is *distance sensitivity oracles (DSO)*. Here, the output is a compact representation from which any specific replacement path can be found with $O(1)$ time. The first such oracle was developed in Demetrescu et. al. [7], and it has size $O(n^2 \log n)$. The

current best construction time for an oracle of this size is $O(mn \log n + n^2 \log^2 n)$ time for a randomized algorithm, and a log factor slower for a deterministic algorithm, given in Bernstein and Karger [3]. Given such an oracle, the output to 2-APSiSP can be computed with $O(n)$ queries for each source-sink pair, i.e., with $O(n^3)$ queries to the DSO.

To the best of our knowledge, for $k > 1$ the problem of generating k simple shortest cycles in the overall graph in non-decreasing order of their weights (k-All-SiSC) has not been studied before, and neither has k-SiSC (k Simple Shortest Cycles through a given node) or k-ANSiSC (k All Nodes Simple Shortest Cycles); for $k = 1$, 1-All-SiSC asks for a minimum weight cycle and 1-ANSiSC is the ANSC problem [25], both of which can be found in $\tilde{O}(mn)$ time, and 1-SiSC can be solved in $\tilde{O}(m + n)$ time. On the other hand, enumerating simple (or *elementary*) cycles in no particular order — which is thus a special case of k-All-SiSC — has been studied extensively [21, 22, 20, 11]. The first polynomial time algorithm was given by Tarjan [20], and ran in $O(kmn)$ time for k cycles. This result was improved to $O(k \cdot m + n)$ by Johnson [11]. We do not expect to match this linear time result for k-All-SiSC since it includes the minimum weight cycle problem for $k = 1$.

In this paper, we concentrate on results for truly sparse graphs with arbitrary non-negative edge weights. Hence we do not consider results for small integers weights or for dense graphs; several subcubic results for such inputs are known using fast matrix multiplication.

1.2 Our Contributions

We present several algorithmic results on finding k simple paths and cycles in a directed graph with non-negative edge weights.[1] A summary of our results is given in Table 1.

Computing k simple shortest paths for all pairs (k-APSiSP) in G. We present a new approach to the k-APSiSP problem, which computes the sets $P_k^*(x, y)$ as defined below. Our method introduces the key notion of a 'nearly k SiSP set', $Q_k(x, y)$, defined as follows.

▶ **Definition 1.1.** Let $G = (V, E)$ be a directed graph with nonnegative edge weights. For $k \geq 2$, and a vertex pair x, y, let $k^* = \min\{r, k\}$, where r is the number of simple paths from x to y in G. Then,

(i) $P_k^*(x, y)$ is the set of k^* *simple shortest paths from x to y* in G

(ii) $Q_k(x, y)$ is the set of k *nearly simple shortest paths from x to y*, defined as follows. If $k^* = k$ and the $k - 1$ simple shortest paths from x to y share the same first edge (x, a) then $Q_k(x, y)$ contains these $k-1$ simple shortest paths, together with the simple shortest path from x to y that does not start with edge (x, a), if such a path exists. Otherwise (i.e, if either the former or latter condition does not hold), $Q_k(x, y) = P_k^*(x, y)$.

Our algorithm for k-APSiSP first constructs $Q_k(x, y)$ for all pairs of vertices x, y, and then uses these sets in an efficient algorithm, Compute-APSiSP, to compute the $P_k^*(x, y)$ for all x, y. The latter algorithm runs in time $O(k \cdot n^2 + n^2 \log n)$ for any k, while our method for constructing the $Q_k(x, y)$ depends on k. For $k = 2$ we present an $O(mn + n^2 \log n)$ time method to compute the $Q_2(x, y)$ sets; this gives a 2-APSiSP algorithm that matches Yen's bound of $O(mn + n^2 \log n)$ for 2-*SiSP for a single pair of vertices*. It is also faster (by a

[1] Except for k-All-SiSP (see Section 3.1), we can also handle negative edge-weights as long as there are no negative-weight cycles, by applying Johnson's transformation [12] to obtain an equivalent input with nonnegative edge weights. If the resulting edge-weights include weight 0, we will use the pair $(wt(p), len(p))$ as the weight for path p, where $len(p)$ is the number of edges in it; this causes the weight of a proper subpath of p to be smaller than the weight of p.

polylogarithmic factor) than the best algorithm for DSO (distance sensitivity oracles) for the all-pairs replacement paths problem [3]. In fact, we also show that the $Q_2(x, y)$ sets can be computed in $O(n^2)$ time using a DSO, and hence 2-APSiSP can be computed in $O(n^2 \log n)$ time plus the time to construct the DSO.

For $k \geq 3$ our algorithm to compute the Q_k sets makes calls to an algorithm for $(k-1)$-APSiSP, so we combine the two components together in a single recursive method, APSiSP, that takes as input G and k, and outputs the P_k^* sets for all vertex pairs. The time bound for APSiSP increases with k: it is faster than Yen's method for $k = 3$ by a factor of n (and hence is faster than the current fastest method by almost a factor of n), it matches Yen for $k = 4$, and its performance degrades for larger k.

If a faster algorithm can be designed to compute the Q_k sets, then we can run Compute-APSiSP on its output and hence compute k-APSiSP in additional $O(k \cdot n^2 + n^2 \log n)$ time. Thus, a major open problem left by our results is the design of a faster algorithm to compute the Q_k sets for larger values of k.

New Approach: Computing simple shortest paths without finding detours. Our method for computing k-APSiSP (using the $Q_k(x, y)$ sets) extends an existing simple path in the data structure to create a new simple path by adding a single incoming edge. This approach differs from all previous approaches to finding k simple paths and replacement paths. All known previous algorithms for 2-SiSP compute replacement paths for every edge on the shortest path (by computing suitable 'detours'). In fact, Hershberger et al. [10] present a lower bound for k-SiSP, exclusively for the class of algorithms that use detours, by pointing out that all known algorithms for k-SiSP compute replacement paths, and all known replacement path algorithms use detours. In contrast, our method may enumerate and inspect paths that are not detours, including paths with cycles (e.g., Step 17 in algorithm Compute-APSiSP in Section 2.1). Thus our method is fundamentally new.

Generating k simple shortest cycles and paths (k-All-SiSC, k-SiSC, k-ANSiSC) and k-All-SiSP. We consider the problem of generating the k simple shortest cycles in the graph G in nonincreasing order of their weight (k-All-SiSC). In Section 3 we present an algorithm for k-All-SiSC that runs in $\tilde{O}(k \cdot mn)$ time by generating each successive simple shortest cycle in G in $\tilde{O}(mn)$ time. The same algorithm can be used to enumerate all simple cycles in G in nondecreasing order of their weights. Recall that the related problem of simply enumerating simple cycles in a graph in no particular order was a very well-studied classical problem [21, 22, 20, 11] until an algorithm that generates successive cycles in linear time was obtained [11]. Our algorithm does not match the linear time bound per successive cycle, but it is to be noted that 1-All-SiSC (i.e., the problem of generating a minimum weight cycle) is a very fundamental and well-studied problem for which the current best bound is $\tilde{O}(mn)$.

Our algorithm for k-All-SiSC creates a auxiliary graph on which suitable SiSP computation can be performed to generate the desired output. We give fast algorithms for k-SiSC and k-ANSiSC using the same auxiliary graph.

Complementing our result for k-All-SiSC, we present in Section 3.1 an algorithm for k-All-SiSP that generates each successive simple path in $\tilde{O}(k)$ time if $k < n$, and in $\tilde{O}(n)$ time if $k > n$, after an initial start-up cost of $O(m)$ to find the first path. This time bound is considerably faster than that for k-All-SiSC. Our method, All-SiSP, is again one of extending existing paths by an edge (as is Compute-APSiSP); it is, however, a different path extension method.

Problem	Known Results	New Results
2-APSiSP (Sec. 2.2.1)	$O(n^3 + mn\log^2 n)$ (using DSO [3])	$\boldsymbol{O(mn + n^2 \log n)}$
3-APSiSP (Sec. 2.2.2)	$\tilde{O}(mn^3)$ [24]	$\boldsymbol{O(mn^2 + n^3 \log n)}$
k-SiSC (Sec. 2.3)	—	$\boldsymbol{O(k \cdot (mn + n^2 \log\log n))}$
k-ANSiSC (Sec. 2.3)	—	$\boldsymbol{O(mn + n^2 \log n)}$ **if** $\boldsymbol{k = 2}$ **and** $\boldsymbol{O(k \cdot (mn^2 + n^3 \log\log n))}$ **if** $\boldsymbol{k > 2}$
k-All-SiSC (Sec. 3)	—	$\boldsymbol{\tilde{O}(kmn)}$
k-All-SiSP (Sec. 3.1)	—	$\boldsymbol{\tilde{O}(k)}$ **if** $\boldsymbol{k < n}$ **and** $\boldsymbol{\tilde{O}(n)}$ **if** $\boldsymbol{k \geq n}$ **per path amortized, after a startup cost of** $\boldsymbol{O(m)}$

Table 1 Our results for directed graphs. All algorithms are deterministic. (DSO stands for Distance Sensitivity Oracles).

Path Extensions. We use two different path extension methods, one for k-APSiSP and the other for k-All-SiSP. Path extensions have been used before in the hidden paths algorithm for APSP [13] and more recently, for fully dynamic APSP [5]. These two path extension methods differ from each other, as noted in [6]. Our path extension method for k-All-SiSP is inspired by a method in [5] to compute 'locally shortest paths' for fully dynamic APSP. Our path extension method for k-APSiSP appears to be new.

Here are the main theorems we establish for our algorithmic results. In all cases, the input is a directed graph $G = (V, E)$ with nonnegative edge weights, and $|V| = n$, $|E| = m$.

▶ **Theorem 1.2.** *Given an integer $k > 1$, and the nearly simple shortest paths sets $Q_k(x, y)$ (Definition 1.1) for all $x, y \in V$, Algorithm* Compute-APSiSP *(Section 2.1) produces the k simple shortest paths for every pair of vertices in $O(k \cdot n^2 + n^2 \log n)$ time.*

▶ **Theorem 1.3.** *(i) Algorithm* 2-APSiSP *(Section 2.2.1) correctly computes 2-APSiSP in $O(mn + n^2 \log n)$ time.*

(ii) For $k > 2$, Algorithm APSiSP *(Section 2.2.2) correctly computes k-APSiSP in $T(m, n, k)$ time, where $T(m, n, k) \leq n \cdot T(m, n, k-1) + O(mn + n^2 \cdot (k + \log n))$.*

(iii) $T(m, n, 3)$, the time bound for algorithm APSiSP *for $k = 3$, is $O(m \cdot n^2 + n^3 \cdot \log n)$.*

▶ **Theorem 1.4.** *(k-All-SiSC) After an initial start-up cost of $O(mn+n^2 \log n)$ time, we can compute each successive simple shortest cycle in $O(mn + n^2 \log\log n)$ time. This computes k-All-SiSC (Section 3).*

▶ **Theorem 1.5.** *(k-All-SiSP) After an initial start-up cost of $O(m)$ time to generate the first path, Algorithm* All-SiSP *(Section 3.1) computes each succeeding simple shortest path with the following bounds:*

(i) amortized $O(k + \log n)$ time if $k = O(n)$ and $O(n + \log k)$ time if $k = \Omega(n)$;

(ii) worst-case $O(k \cdot \log n)$ time if $k = O(n)$, and $O(n \cdot \log k)$ time if $k = \Omega(n)$.

Space Bounds. Our k-APSiSP algorithm uses $O(k^2 \cdot n^2)$ space, which is a factor of k larger than the bound on the size of the output. In contrast, the earlier path extension algorithms for APSP [13] and for fully dynamic APSP [5] use $\Omega(mn)$ space in the worst case. All of our other algorithms use space $O(kn^2)$ or better.

Only proof sketches are given here; the full proofs of most results as well as details of the algorithms for simple cycles are in the arXiv paper [1]. Table 1 lists our main results.

2 The k-APSiSP Algorithm

In this section, we present our algorithm to compute k-APSiSP on a directed graph $G = (V, E)$ with nonnegative edge-weight function wt. The algorithm has two main steps. In the first step it computes the nearly k-SiSP sets $Q_k(x, y)$ for all pairs x, y. In the second step it computes the exact k-SiSP sets $P_k^*(x, y)$ for all x, y using the $Q_k(x, y)$ sets. This second step is the same for any value of k, and we describe this step first in Section 2.1. We then present efficient algorithms to compute the Q_k sets for $k = 2$ and $k > 2$ in Section 2.2.

In all of our algorithms we will maintain the paths in each $P_k^*(x, y)$ and $Q_k(x, y)$ set in an array in nondecreasing order of edge-weights.

2.1 The Compute-APSiSP Procedure

In this section we present an algorithm, Compute-APSiSP, to compute k-APSiSP. This algorithm takes as input, the graph G, together with the nearly k-SiSP sets $Q_k(x, y)$, for each pair of distinct vertices x, y, and outputs the k^* simple shortest paths from x to y in the set $P_k^*(x, y)$ for each pair of vertices $x, y \in V$ (note that k^*, which is defined in Definition 1.1, can be different for different vertex pairs x, y). As noted above, the construction of the $Q_k(x, y)$ sets will be described in the next section.

The *right (left) subpath* of a path π is defined as the path obtained by removing the first (last) edge on π. If π is a single edge (x, y) then this path is the vertex y (x).

▶ **Lemma 2.1.** *Suppose there are k simple shortest paths from x to y, all having the same first edge (x, a). Then $\forall i$, $1 \leq i \leq k$, the right subpath of the i-th simple shortest path from x to y has weight equal to the weight of the i-th simple shortest path from a to y.*

Proof. The result is trivial for $k = 1$. If it holds for $k - 1$ and not k, then the k-th lightest path p from a to y must contain x, and then we would have a shorter path from x to y that avoids edge (x, a). ◀

Algorithm Compute-APSiSP computes the $P_k^*(x, y)$ sets by extending an existing path by an edge. In particular, if the k-SiSPs from x to y all use the same first edge (x, a), then it computes the k-th SiSP by extending the k-th SiSP from a to y (otherwise, the sets $P_k^*(x, y)$ are trivially computed from the sets $Q_k(x, y)$). The algorithm first initializes the $P_k^*(x, y)$ sets with the corresponding $Q_k(x, y)$ sets in Step 4. In Step 5, it checks whether the shortest $k - 1$ paths in $P_k^*(x, y)$ have the same first edge and if so, by definition of $Q_k(x, y)$, this $P_k^*(x, y)$ may not have been correctly initialized, and may need to update its k-th shortest path to obtain the correct output. In this case, the common first edge (x, a) is added to the set $Extensions(a, y)$ in Step 7. We explain this step below.

We define the *k-Left Extended Simple Path (k-LESiP)* $\pi_{xa,y}$ from x to y as the path $\pi_{xa,y} = (x, a) \circ \pi_{a,y}$, where the path $\pi_{a,y}$ is the k-th shortest path in $Q_k(a, y)$, and $\circ$ denotes the concatenation operation. In our algorithm we will construct k-LESiPs for those pairs x, y for which the $k - 1$ simple shortest paths all start with the edge (x, a). The algorithm also maintains a set $Extensions(a, y)$ for each pair of distinct vertices a, y; this set contains those edges (x, a) incoming to a which are the first edge on all $k-1$ SiSPs from x to y. In addition to adding the common first edge (x, a) in the $(k - 1)$ SiSPs in $P_k^*(x, y)$ to $Extensions(a, y)$ in Step 7, the algorithm creates the k-LESiP with start edge (x, a) and end vertex y using the k-th shortest path in the set $P_k^*(a, y)$, and adds it to heap H in Steps 8-10. Let $\mathcal{U}$ denote the set of $P_k^*(x, y)$ sets which may need to be updated; these are the sets for which the if condition in Step 5 holds.

In the main while loop in Steps 12-17, a min-weight path is extracted in each iteration. We establish below that this min-weight path is added to the corresponding P_k^* in Step 14 or 15 only if it is the k-th SiSP; in this case, its left extensions are created and added to the heap H in Step 17, and we note that some of these paths could be cyclic.

▶ **Lemma 2.2.** *Let $G = (V, E)$ be a directed graph with nonnegative edge weight function wt, and $\forall x, y \in V$, let the set $Q_k(x, y)$ contain the nearly k-SiSPs from x to y. Then, algorithm* Compute-APSiSP *correctly computes the sets $P_k^*(x, y)$ $\forall x, y \in V$.*

Proof. We first show that every path in $P_k^*(x, y)$ is simple. The initialization in Step 4 adds only simple paths. After that, $P_k^*(x, y)$ is updated only if it is in $\mathcal{U}$. Assume so, and let $(x, a) \in Extensions(a, y)$. As the algorithm only extends along the edges in the $Extensions$ sets, every path from x to y in H has (x, a) as first edge. Now if a cyclic path (say $\pi_{xa,y}$) is added to $P_k^*(x, y)$ from H, then it contains a subpath $\pi_{xa',y}$, but this implies that either $\pi_{xa',y}$, or a path from x to y with smaller weight but not using (x, a) as the first edge, is present in $Q_k(x, y)$. This means that the check in Step 15 will be false, and $\pi_{xa,y}$ will not be added to $P_k^*(x, y)$.

To show that $P_k^*(x, y)$ contains the k shortest simple paths from x to y at termination, we observe that it was initialized with $Q_k(x, y)$, so we only need to ensure that the path of largest weight in $P_k^*(x, y)$ is indeed π_{xy}^k, the k^*-th shortest simple path from x to y. We argue this by showing that π_{ay}, the path obtained from π_{xy}^k by removing its first edge (x, a), must be in $P_k^*(a, y)$ and must have been extended to x and added to H. ◀

Algorithm 1 Compute-APSiSP$(G = (V, E), wt, k, \{Q_k(x, y), \forall x, y\})$

1: Initialize:
2: $H \leftarrow \phi$ $\{H$ is a priority queue.$\}$
3: **for all** $x, y \in V, x \neq y$ **do**
4: $P_k^*(x, y) \leftarrow Q_k(x, y)$
5: **if** the $k-1$ shortest paths in $P_k^*(x, y)$ have the same first edge **then**
6: Let (x, a) be the common first edge in the $(k-1)$ shortest paths in $P_k^*(x, y)$
7: Add (x, a) to the set $Extensions(a, y)$
8: **if** $|Q_k(a, y)| = k$ **then**
9: $\pi \leftarrow$ the path of largest weight in $Q_k(a, y)$
10: $\pi' \leftarrow (x, a) \circ \pi$; add π' to H with weight $wt(x, a) + wt(\pi)$
11: Main Loop:
12: **while** $H \neq \phi$ **do**
13: $\pi \leftarrow$ Extract-min(H); let $\pi = (xa, y)$ and π' a path of largest weight in $P_k^*(x, y)$
14: **if** $|P_k^*(x, y)| = k - 1$ **then** add π to $P_k^*(x, y)$ and set *update* flag
15: **else if** $wt(\pi) < wt(\pi')$ **then** replace π' with π in $P_k^*(x, y)$ and set *update* flag
16: **if** *update* flag is set **then**
17: **for all** $(x', x) \in Extensions(x, y)$ **do** add $(x', x) \circ \pi$ to H with weight $wt(x', x) + wt(\pi)$

It is straightforward to see that Algorithm Compute-APSiSP runs in $O(kn^2 + n^2 \log n)$ time and uses $O(kn^2)$ space.

2.2 Computing the Q_k Sets

2.2.1 Computing Q_k for $k = 2$

We now give an $O(mn+n^2 \log n)$ time algorithm to compute $Q_2(x, y)$ for all pairs x, y. This method uses the procedure FAST-EXCLUDE from Demetrescu et al. [7], which we now briefly describe (full details of this algorithm can be found in [7]).

Given a rooted tree T, edges (u_1, v_1) and (u_2, v_2) on T are *independent*[7] if the subtree of T rooted at v_1 and the subtree of T rooted at v_2 are disjoint. Given the weighted directed graph $G = (V, E)$, the SSSP tree T_s rooted at a source vertex $s \in V$, and a set S of independent edges in T_s, algorithm FAST-EXCLUDE in [7] computes, for each edge $e \in S$, a shortest path from s to every other vertex in $G - \{e\}$. This algorithm runs in time $O(m + n \log n)$.

We will compute the second path in each $Q_2(x, y)$ set, for a given $x \in V$, by running FAST-EXCLUDE with x as source, and with the set of outgoing edges from x in the shortest path tree rooted at x, T_x, as the set S. Clearly, this set S is independent, and hence algorithm FAST-EXCLUDE will produce its specified output. Now consider any vertex $y \neq x$, and let (x, a) be the first edge on the shortest path from x to y in T_x. By its specification, FAST-EXCLUDE will compute a shortest path from x to y that avoids edge (x, a) in its output, which is the second path needed for $Q_2(x, y)$. This holds for every vertex $y \in V - \{x\}$. Thus we have:

▶ **Lemma 2.3.** *The $Q_2(x, y)$ sets for pairs x, y can be computed in $O(mn + n^2 \log n)$ time.*

This leads to the following algorithm for 2-APSiSP. Its time bound in Theorem 1.3, part (i) follows from Lemma 2.3 and the time bound for COMPUTE-APSISP given in Section 2.1.

Algorithm 2 2-APSISP$(G = (V, E); wt)$

1: **for** each $x \in V$ **do**
2: Compute a shortest path in each $Q_2(x, y)$, $y \in V - \{x\}$ (Dijkstra with source x)
3: Compute the second path in each $Q_2(x, y)$, $y \in V - \{x\}$, using FAST-EXCLUDE with source x and $S = \{(x, a) \in T_x\}$
4: COMPUTE-APSISP(G, wt, 2, $\{Q_2(x, y), \forall x, y\}$)

The space bound is $O(n^2)$ since the Q_2 sets contain $O(n^2)$ paths and the call to COMPUTE-APSISP takes $O(n^2)$ space. In the full paper [1] we give a simple alternate algorithm that computes the Q_2 sets in $\tilde{O}(mn)$ time if a DSO is available. It is not clear if we can efficiently compute 2-APSiSP directly from a DSO in $\tilde{O}(mn)$ time, without using the Q_2 sets and COMPUTE-APSISP.

2.2.2 Computing Q_k for $k \geq 3$

Our algorithm will use the following types of sets. For each vertex $x \in V$, let I_x be the set of incoming edges to x. Also, for a vertex $x \in V$, and vertices $a, y \in V - \{x\}$, let $P_k^{*x}(a, y)$ be the set of k simple shortest paths from a to y in $G - I_x$, the graph obtained after removing the incoming edges to x. Recall that we maintain all P^* and Q sets as sorted arrays.

Algorithm APSISP(G, k) first computes the sets $P_{k-1}^{*x}(a, y)$, for all vertices $a, y \in V$. Then it computes each $Q_k(x, y)$ as the set of all paths in the set $P_{k-1}^*(x, y)$, together with a shortest path in $\bigcup_{\{(x,a) \text{ outgoing from } x\}} \{(x, a) \circ p \mid p \in P_{k-1}^{*x}(a, y)\}$ (which is not present in $P_{k-1}^*(x, y)$).

Algorithm 3 APSiSP$(G = (V, E),\ wt,\ k)$

1: **if** $k = 2$ **then**
2: compute Q_2 sets using algorithm in Section 2.2.1
3: **else**
4: **for** each $x \in V$ **do**
5: $I_x \leftarrow$ set of incoming edges to x
6: Call APSiSP$(G - I_x, wt, k-1)$ to compute $P^{*x}_{k-1}(u, v)\ \forall u, v \in V$
7: **for** each $y \in V - \{x\}$ **do**
8: $Q_k(x, y) \leftarrow P^{*x}_{k-1}(x, y)$
9: **for all** $(x, a) \in E$ **do** $count_a \leftarrow$ number of paths in $Q_k(x, y)$ with (x, a) as the first edge
10: $Q_k(x, y) \leftarrow Q_k(x, y) \cup \{$ a shortest path in $\bigcup_{\{(x,a)}$ outgoing from $x\}(x, a) \circ P^{*x}_{k-1}(a, y)[count_a + 1]\}$
11: Compute-APSiSP$(G, wt, k, \{Q_k(x, y)\ \forall x, y \in V\})$

To compute the P^{*x}_{k-1} sets, APSiSP(G, wt, k) recursively calls APSiSP$(G - I_x, wt, k-1)$ n times, for each vertex $x \in V$. Once we have computed the P^{*x}_{k-1} sets, the $Q_k(x, y)$ sets are readily computed as described in steps 8 - 10. After the computation of $Q_k(x, y)$ sets, APSiSP(G, wt, k) calls Compute-APSiSP$(G, wt, k, \{Q_k(x, y)\ \forall x, y \in V\})$ to compute the P^*_k sets. This establishes part (ii) of Theorem 1.3.

Proof of Theorem 1.3, part (iii)**.** The for loop starting in Step 4 is executed n times, and for $k = 3$ the cost of each iteration is dominated by the call to Algorithm 2-APSiSP in Step 6, which takes $O(mn + n^2 \log n)$ time. This contributes $O(mn^2 + n^3 \log n)$ to the total running time. The inner for loop starting in Step 7 is executed n times per iteration of the outer for loop, and the cost of each iteration is $O(k + d_x)$. Summing over all $x \in V$, this contributes $O(kn^2 + mn)$ to the total running time. Step 11 runs in $O(n^2 \log n)$ time as shown in Section 2.1. Thus, the total running time is $O(mn^2 + n^3 \log n)$. ◀

The space bound for APSiSP is $O(k^2 \cdot n^2)$, as the P^*_{k-1} and Q_k sets contain $O(kn^2)$ paths, and each recursive call to APSiSP$(G - I_x, wt, k-1)$ needs to maintain the P^*_{r-1} and Q_r sets at each level of recursion. The call to Compute-APSiSP takes $O(kn^2)$ space as noted earlier.

The performance of Algorithm APSiSP degrades by a factor of n with each increase in k. Thus, it matches Yen's algorithm (applied to all-pairs) for $k = 4$, and for larger values of k its performance is worse than Yen.

2.3 Generating k Simple Shortest Cycles

k-SiSC. This is the problem of generating the k simple shortest cycles through a specific vertex z in G. We can reduce this problem to k-SiSP by forming G'_z, where we replace vertex z by vertices z_i and z_o in G'_z, we place a directed edge of weight 0 from z_i to z_o, and we replace each incoming edge to (outgoing edge from) z with an incoming edge to z_i (outgoing edge from z_o) in G'_z. Then the k-th simple shortest path from z_o to z_i in G'_z can been seen to correspond to the k-th simple shortest cycle through z in G. This gives an $O(k \cdot (mn + n^2 \log\log n))$ time algorithm for computing k-SiSC using [9]. We also observe that we can solve k-SiSP from s to t in G if we have an algorithm for k-SiSC: create G' by adding a new vertex x^* and zero weight edges (x^*, s), (t, x^*), and then call k-SiSC for vertex x^*. Thus k-SiSP and k-SiSC are equivalent in complexity in weighted directed graphs.

k-ANSiSC. This is the problem of generating k simple shortest cycles that pass through a given vertex x, for every vertex $x \in V$. For $k = 1$ this problem can be solved in $O(mn + n^2 \log\log n)$ time by computing APSP [25]. For $k = 2$, we can reduce this problem to k-APSiSP by forming the graph G' where for each vertex x, we replace vertex x in G by vertices x_i and x_o in G', we place a directed edge of weight 0 from x_i to x_o, and we replace each edge (u, x) in G by an edge (u_o, x_i) in G' (and hence we also replace each edge (x, v) in G by an edge (x_o, v_i) in G'). For $k > 2$, k-ANSiSC can be computed in $O(k \cdot n \cdot (mn + n^2 \log\log n))$ time by computing k-SiSC for each vertex.

3 Enumerating Simple Shortest Cycles and Paths

In this section, we first give a method to generate each successive simple shortest cycle in $G = (V, E)$ (k-All-SiSC) and then in Section 3.1 we give a faster method to generate simple paths in nondecreasing order of weight (k-All-SiSP).

Enumerating Simple Shortest Cycles (k-All-SiSC). Our algorithm for k-All-SiSC creates an auxiliary graph $G' = (V', E')$ as in the construction for k-ANSiSC in Section 2.3. Our algorithm also maintains a set $\mathcal{C}$ of candidate simple shortest cycles. Initially, our algorithm computes a shortest cycle for each vertex $j \in V$ by running Dijkstra's algorithm with source vertex j_o on the subgraph G'_j of G' induced on $V'_j = \{x_i, x_o \mid x \geq j\}$, to find a shortest path p from j_o to j_i . We store these shortest cycles in $\mathcal{C}$.

For each $k \geq 1$, we generate the k-th simple shortest cycle in G by choosing a minimum weight cycle in $\mathcal{C}$. Let this cycle corresponds to some vertex r and is the k_r-th SiSP from vertex r_o to vertex r_i in G'_r. We then replace this cycle in $\mathcal{C}$ by computing the $(k_r + 1)$-th SiSP from vertex r_o to r_i in G'_r.

The initialization takes $O(mn + n^2 \log n)$ time for the n calls to Dijkstra's algorithm. Thereafter, we generate each new cycle in $O(mn + n^2 \log\log n)$ time using the k-SiSP algorithm [9], by maintaining the relevant information from the computation of earlier cycles.

3.1 Generating Simple Shortest Paths (k-All-SiSP)

Our algorithm for k-All-SiSP is inspired by the method in [5] for fully dynamic APSP. With each path π we will associate two sets of paths $L(\pi)$ and $R(\pi)$ as described below. Similar sets are used in [5] for 'locally shortest paths' but here they have a different use.

Let $\mathcal{P}$ be a collection of simple paths. For a simple path π_{xy} from x to y in $\mathcal{P}$, its *left extension set* $L(\pi_{xy})$ is the set of simple paths $\pi' \in \mathcal{P}$ such that $\pi' = (x', x) \circ \pi_{xy}$, for some $x' \in V$. Similarly, the *right extension set* $R(\pi_{xy})$ is the set of simple paths $\pi'' = \pi_{xy} \circ (y, y')$ such that $\pi'' \in \mathcal{P}$. For a trivial path $\pi = \langle v \rangle$, $L(\pi)$ is the set of incoming edges to v, and $R(\pi)$ is the set of outgoing edges from v.

Algorithm All-SiSP initializes a priority queue H with the edges in G, and it initializes the extension sets for the vertices in G. In each iteration of the main loop, the algorithm extracts the minimum weight path π in H as the next simple path in the output sequence. It then generates suitable extensions of π to be added to H as follows. Let the first edge on π be (x, a) and the last edge (b, y). Then, All-SiSP left extends π along those edges (x', x) such that there is a path $\pi_{x'b}$ in $L(l(\pi))$; it also requires that $x' \neq y$, since extending to x' would create a cycle in the path. It forms similar extensions to the right in the for loop starting at Step 12.

Algorithm 4 ALL-SISP$(G = (V, E); wt)$

1: Initialization:
2: $H \leftarrow \phi$ $\{H$ is a priority queue.$\}$
3: **for all** $(x, y) \in E$ **do**
4: Add (x, y) to priority queue H with $wt(x, y)$ as key; add (x, y) to $L(\langle y \rangle)$ and $R(\langle x \rangle)$
5: Main loop:
6: **while** $H \neq \phi$ **do**
7: $\pi \leftarrow$ EXTRACT-MIN(H); add π to the output sequence of simple paths
8: Let $\pi_{xb} = \ell(\pi)$ and $\pi_{ay} = r(\pi)$ (so (x, a) and (b, y) are the first and last edges on π)
9: **for all** $\pi_{x'b} \in L(\pi_{xb})$ with $x' \neq y$ **do**
10: Form $\pi_{x'y} \leftarrow (x', x) \circ \pi$ and add $\pi_{x'y}$ to H with $wt(\pi_{x'y})$ as key
11: Add $\pi_{x'y}$ to $L(\pi_{xy})$ and to $R(\pi_{x'b})$
12: **for all** $\pi_{ay'} \in R(\pi_{ay})$ with $y' \neq x$ **do** perform steps complementary to Steps 10-11

To establish Theorem 1.5, we first need to show that every path added to H is simple. All edges added in Step 4 are clearly simple paths. Consider a path σ added to H in Step 10. We show that both $\ell(\sigma)$ and $r(\sigma)$ must already be in H, and hence must be simple paths. So, the only way that σ could contain a cycle is if its first and last vertices are the same. But this is explicitly forbidden in the condition in Step 9. A similar argument applies to Step 12.

To show that no simple path in G is omitted in the sequence of simple shortest paths generated, we observe that if π is a simple path of smallest weight not generated by Algorithm ALL-SISP, then $\ell(\pi)$ and $r(\pi)$ must have been generated. We can then show that π will be added to H in the iteration of Step 6 when the heavier of $\ell(\pi)$ and $r(\pi)$ is extracted.

The amortized bound in Theorem 1.5 is obtained by implementing H as a Fibonacci heap and the worst-case bound is obtained by using a binary heap.

4 Discussion

Our k-All-SiSP algorithm is nearly optimal if the paths need to be output. It is also not difficult to see that our bounds for 2-APSiSP and k-All-SiSC (for constant k) are the best possible to within a polylog factor for sparse graphs unless the long-standing $\tilde{O}(mn)$ bounds for APSP and minimum weight cycles are improved. In recent work [2] we give several fine-grained reductions that demonstrate that the minimum weight cycle problem holds a central position for a class of problems that currently have $\tilde{O}(mn)$ time bound on sparse graphs, both directed and undirected.

For undirected graphs, our k-All-SiSP result gives an algorithm with the same bound. Also, our k-APSiSP algorithm works for undirected graphs, and this gives a faster algorithm for $k = 2$ and matches the previous best bound (using [14]) for $k = 3$. However, our algorithms for the three variants of finding simple shortest cycles do not work for undirected graphs. This is addressed in our recent work in [2].

The main open question for k-APSiSP is to come up with faster algorithms to compute the $Q_k(x, y)$ sets for larger values of k. This is the key to a faster k-APSiSP algorithm using our approach, for $k > 2$.

References

1 Udit Agarwal and Vijaya Ramachandran. Finding k simple shortest paths and cycles. *arXiv preprint arXiv:1512.02157*, 2015.
2 Udit Agarwal and Vijaya Ramachandran. Fine-grained reductions and algorithms for shortest cycles. *manuscript*, 2016.
3 Aaron Bernstein and David Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proc. STOC*, pages 101–110, 2009.
4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
5 Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51:968–992, 2004.
6 Camil Demetrescu and Giuseppe F Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Trans. Alg. (TALG)*, 2:578–601, 2006.
7 Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM Jour. Comput.*, 37:1299–1318, 2008.
8 David Eppstein. Finding the k shortest paths. *SIAM Jour. Comput.*, 28:652–673, 1998.
9 Zvi Gotthilf and Moshe Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Inf. Proc. Lett.*, 109(7):352–355, 2009.
10 John Hershberger, Subhash Suri, and Amit Bhosle. On the difficulty of some shortest path problems. *ACM Trans. Alg. (TALG)*, 3(1):5, 2007.
11 Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Jour. Comput.*, 4(1):77–84, 1975.
12 Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *JACM*, 24(1):1–13, 1977.
13 David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993.
14 Naoki Katoh, Toshihide Ibaraki, and Hisashi Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982.
15 Eugene L Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.
16 Eugene L Lawler. Comment on a computing the k shortest paths in a graph. *CACM*, 20(8):603–605, 1977.
17 Edward Minieka. On computing sets of shortest paths in a graph. *CACM*, 17(6):351–353, 1974.
18 Seth Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.
19 Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Trans. Alg. (TALG)*, 8(4):33, 2012.
20 Robert Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Jour. Comput.*, 2(3):211–216, 2005.
21 J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *CACM*, 13:722–726, 1970.
22 H. Weinblatt. A new search algorithm to find the elementary circuits of a graph. *JACM*, 19:43–56, 1972.
23 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proc. IEEE FOCS*, pages 645–654. IEEE, 2010.
24 Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.
25 Raphael Yuster. A shortest cycle for each vertex of a graph. *Inf. Proc. Lett.*, 111(21):1057–1061, 2011.