Finding the k Shortest Paths

David Eppstein*

Department of Information and Computer Science
University of California, Irvine, CA 92717

Tech. Report 94-26

May 31, 1994

Abstract

We describe algorithms for finding the k shortest paths connecting a given pair of vertices in a digraph (allowing cycles). Our algorithms output an implicit representation of the paths as an unordered set in time $O(m + n \log n + k)$. The paths can be output in order by length in total time $O(m + n \log n + k \log k)$. We can also find the k paths from a given source s to each vertex in the graph, in total time $O(m + n \log n + kn)$.

*Work supported in part by NSF grant CCR-9258355.

1 Introduction

We consider a generalization of the well-known shortest path problem, in which not one but several short paths must be produced. The k *shortest paths problem*, for a given k and a given source-destination pair in a digraph, is to list the k paths in the digraph with minimum total length. In the version of the problem we study, cycles of repeated vertices are allowed. We reduce this version of the k shortest paths problem to that of finding the minimum k elements in a *heap-ordered tree* [14]. We use a recent algorithm of Frederickson [14] for this selection problem, to find an implicit representation of the k shortest paths.

1.1 Applications

The applications of shortest path computations are too numerous to cite in detail. They include situations in which an actual path is the desired output, such as robot motion planning, highway and power line engineering, and network connection routing. They include problems of scheduling such as critical path computation in PERT charts. They include many optimization problems solved by dynamic programming or more complicated matrix searching techniques, such as sequence alignment in molecular biology, construction of optimal inscribed polygons, and length-limited Huffman coding.

Methods for finding k shortest paths can be and have been applied to many of these applications, for two reasons.

- **Additional constraints.** One may wish to find a path that satisfies certain constraints beyond having a small length, but those other constraints may be ill-defined or hard to optimize. For instance, in power transmission route selection [10], a power line should connect its endpoints reasonably directly, but there may be more or less community support for one option or another. A typical solution is to compute several short paths and then choose among them by considering the other criteria. This type of application is the reason cited by Dreyfus [9] and Lawler [23] for k shortest path computations.

- **Sensitivity analysis.** By computing more than one shortest path, one can determine how sensitive the optimal solution is to variation of the problem's parameters. For instance, in biological sequence alignment, one typically wishes to see several "good" alignments rather

than one optimal alignment; by comparing these several alignments, biologists can determine which portions of an alignment are most essential [6]. This problem is exactly that of finding k shortest paths in an appropriate dynamic programming matrix.

1.2 New Results

We prove the following results. In all cases we assume we are given a digraph in which each edge has a non-negative length. In each case the paths are output in an implicit representation from which simple properties such as the length are available in constant time. We may explicitly list the edges in any path in time proportional to the number of edges.

- We find the k shortest paths (allowing cycles) connecting a given pair of vertices in a digraph, in time $O(m + n \log n + k)$.
- We describe a data structure which after $O(m + n \log n)$ preprocessing time will list the paths connecting a given pair of vertices, in order by length, producing the ith shortest path in time $O(\log i)$.
- We find the k shortest paths from a given source in a digraph to each other vertex, in time $O(m + n \log n + kn)$.

Similar results hold in digraphs with negative edges but no negative cycles, in which case the time bounds above should be modified to include the time to compute a single source shortest path tree in such networks. Similar results also hold for finding the k longest paths in acyclic networks [4]; we omit the details.

1.3 Related Work

The k shortest paths problem has been well-studied [3, 5, 7, 9, 12, 17, 18, 20, 21, 23, 24, 25, 27, 28, 29, 30, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42] and many algorithms are known. Dreyfus [9] and Yen [42] cite several additional papers on the subject going back as far as 1957.

One must distinguish several common variations of the problem. In many of the papers cited above, the paths are restricted to be *simple*, i.e. no vertex can be repeated. This has advantages in some applications, but as our results show this restriction seems to make the problem significantly harder. Several papers [3, 9, 12, 25, 26, 34, 35] explicitly consider the version

of the k shortest paths problem in which repeated vertices are allowed, and it is this version that we also study. Of course, for acyclic digraphs (as used in many of the applications described above including scheduling and dynamic programming) no path can have a repeated vertex and the two versions of the problem become equivalent.

One can also make a restriction that the paths found be edge disjoint or vertex disjoint [31], however this changes the flavor of the problem dramatically, turning it into one related to network flow.

Fox [12] gives what seems to be the best previously known bound for the k shortest path problem, $O(n^2 + kn \log n)$. Dreyfus [9] mentions the version of the problem in which we must find paths from one source to each other vertex in the graph, and describes a simple $O(kn^2)$ time dynamic programming solution to this problem. For the k shortest simple paths problem, the best known time bound is $O(k(m + n \log n))$ [21].

A similar problem is that of finding the k minimum weight spanning trees in a graph. Recent algorithms for this problem [11, 13] reduce it to finding the k minimum weight nodes in a *heap-ordered tree*, defined using the *best swap* in a sequence of graphs. Heap-ordered tree selection has also been used to find the smallest interpoint distances or the nearest neighbors in geometric point sets [8]. We apply a similar tree selection technique to the k shortest path problem, however the reduction of k shortest paths to heap ordered trees is very different from the constructions in these other problems.

2 Preliminaries

We assume throughout that our input graph G has n vertices and m edges. We allow self-loops and multiple edges so m may be larger than $\binom{n}{2}$. The *length* $\ell(e)$ of an edge e is assumed to be nonnegative. By extension we can define a length $\ell(p)$ for any path in G. The *distance* $d(s,t)$ for a given pair of vertices is the length of the shortest path starting at s and ending at t; with the assumption of no negative cycles this is well defined. Note that $d(s,t)$ may be unequal to $d(t,s)$. The two endpoints of a directed edge e are denoted $\mathit{tail}(e)$ and $\mathit{head}(e)$.

For our purposes, a *heap* is a binary tree in which vertices have weights, satisfying the restriction that the weight of any vertex is less than or equal to the minimum weight of its children. We will not always care whether the tree is balanced (and in some circumstances we will allow trees with

infinite branches). More generally, a *D-heap* is a degree-D tree with the same property; thus the usual heaps above are 2-heaps. As is well known, any set of values can be placed into a balanced heap by the *heapify* operation in linear time. In a balanced heap, any new element can be inserted in logarithmic time.

Define an *m-partial heap* to be a pair (M, H) where H is a heap and M is a set of m elements each smaller than all nodes in H. If H is empty M can have fewer than m elements and we will still call (M, H) an m-partial heap.

The following result is due to Frederickson [14].

Lemma 1 (Frederickson [14]). *We can find the k smallest weight vertices in any heap, in time $O(k)$.*

Note that the time bound does not depend in any way on the overall size of the heap. We can assume without loss of generality that the smallest weights are uniquely determined by using tie-breaking information e.g. based on position in the heap. Frederickson's result applies directly to 2-heaps, but we can easily extend it to D-heaps for any constant D. One simple method of doing this involves forming a 2-heap from the given D-heap by making $D - 1$ copies of each vertex, connected in a binary tree with the D children as leaves, and breaking ties in such a way that the Dk smallest weight vertices in the 2-heap correspond exactly to the k smallest weights in the D-heap.

Frederickson's algorithm will not list the vertices in order by weight. If we wish the output to be sorted, we have to spend more time, but in exchange we can use a much simpler algorithm (best first search):

Lemma 2. *Given any heap, there is a data structure that will output the vertices in order by weight, taking time $O(\log i)$ to output the ith vertex.*

3 Implicit Representation of Paths

As discussed earlier, our algorithm does not output each path it finds explicitly as a sequence of edges; instead it uses an implicit representation, described in this section.

The ith shortest path in a graph may have length $\Omega(n + i)$, so the best time we could hope for in an explicit listing of shortest paths would be $\Omega(kn + k^2)$. Our bounds are faster than this lower bound, so we must

use an implicit representation for the paths. However our representation is not a serious obstacle to use of our algorithm: we can list the edges of any path we output in time proportional to the number of edges, and simple properties (such as the length) are available in constant time. Similar implicit representations have previously been used for related problems such as the k minimum weight spanning trees [11, 13]. Further, previous papers on the k shortest path problem give time bounds omitting the $O(k^2)$ term in the lower bound above, and so these papers must tacitly or not be using an implicit representation.

Our representation is similar in spirit to those used for the k minimum weight spanning trees problem: for that problem, each successive tree differs from a previously listed tree by a *swap*, the insertion of one edge and removal of another edge. The implicit representation consists of a pointer to the previous tree, and a description of the swap. For the shortest path problem, each successive path will turn out to differ from a previously listed path by the inclusion of a single edge not part of a shortest path tree, and appropriate adjustments in the portion of the path that involves shortest path tree edges. Our implicit representation consists of a pointer to the previous path, and a description of the newly added edge.

Given s and t in a digraph G, let T be a single-destination shortest path tree with t as destination (this is the same as a single source shortest path tree in the graph G^R formed by reversing each edge of G). We can compute T in time $O(m + n \log n)$ [15]. We denote by $next_T(v)$ the next vertex reached after v on the path from v to t in T.

Given an edge e in G, define

$$\delta(e) = \ell(e) + d(head(e), t) - d(tail(e), t).$$

Intuitively, $\delta(e)$ measures how much distance is lost by being "sidetracked" along e instead of taking a shortest path to t.

Lemma 3. *For any $e \in G$, $\delta(e) \geq 0$. For any $e \in T$, $\delta(e) = 0$.*

Proof: The quantity $\ell(e)+d(head(e), t)$ measures the length of a path from $tail(e)$ to t formed by concatenating e with a shortest path from $head(e)$ to t. This path cannot be shorter than the actual shortest path from $tail(e)$ to t, so $\delta(e) \geq 0$. But by definition of T, if $e \in T$ the path from $tail(e)$ to t in T is formed by concatenating e with a path from $head(e)$ to t, so in this case the length of the path equals $d(tail(e), t)$ and $\delta(e) = 0$. ∎

For any path p in G, formed by a sequence of edges, some edges of p may be in T, and some others may be in $G - T$. Any path p from s to t is uniquely determined solely by the subsequence $\mathit{sidetracks}(p)$ of its edges in $G - T$. For, given a pair of edges in the subsequence, there is a uniquely determined way of inserting edges from T so that the head of the first edge is connected to the tail of the second edge. As an example, the shortest path in T from s to t is represented by the empty sequence. A sequence of edges in $G - T$ may not correspond to any s-t path, if it includes a pair of edges that cannot be connected by a path in T. If $S = \mathit{sidetracks}(p)$, we define $\mathit{path}(S)$ to be the path p.

Our implicit representation will involve these sequences of edges in $G-T$. We next show how to recover $\ell(p)$ from information in $\mathit{sidetracks}(p)$.

For any nonempty sequence S of edges in $G - T$, let $\mathit{prefix}(S)$ be the sequence formed by the removal of the last edge in S. If $S = \mathit{sidetracks}(p)$, then $\mathit{prefix}(S)$ will define a path $\mathit{prefpath}(p) = \mathit{path}(\mathit{prefix}(S))$.

Lemma 4. *For any path p from s to t,*

$$\ell(p) = d(s,t) + \sum_{e \in \mathit{sidetracks}(p)} \delta(e) = d(s,t) + \sum_{e \in p} \delta(e).$$

Proof: The second equality follows immediately from Lemma 3. We prove the first by induction on the length of $\mathit{sidetracks}(p)$. Let $\mathit{sidetracks}(p)$ differ from $\mathit{prefix}(\mathit{sidetracks}(p))$ by the addition of edge $\mathit{lastedge}(p)$ connecting $\mathit{tail}(\mathit{lastedge}(p))$ to $\mathit{head}(\mathit{lastedge}(p))$. Then p differs from $\mathit{prefpath}(p)$ by the replacement of the path in T from $\mathit{tail}(\mathit{lastedge}(p))$ to t (length $d(u,t)$) by a different path formed by concatenating edge $\mathit{lastedge}(p)$ (length $\ell(e)$) with the path in T from $\mathit{head}(\mathit{lastedge}(p))$ to t (length $d(v,t)$). The difference in length caused by these replacements is

$$\ell(\mathit{lastedge}(p)) + d(\mathit{head}(\mathit{lastedge}(p)), t) - d(\mathit{tail}(\mathit{lastedge}(p)), t)$$

which is exactly the definition of $\delta(\mathit{lastedge}(p))$. ∎

Lemma 5. *For any path p from s to t in G, for which $\mathit{sidetracks}(p)$ is nonempty, $\ell(p) \geq \ell(\mathit{prefpath}(p))$.*

Proof: By Lemma 4 $\ell(p) - \ell(\mathit{prefpath}(p)) = \delta(\mathit{lastedge}(p))$ which by Lemma 3 is nonnegative. ∎

Our representation of a path p in the list of paths produced by our algorithm will then consist of two components:

- The position in the list of *prefpath*(p).
- Edge *lastedge*(p).

Although our algorithm, which uses Lemma 1, does not necessarily output paths in sorted order, we will nevertheless be able to guarantee that *prefpath*(p) would be output by our algorithm earlier than p. From this representation one can easily recover p itself, in time proportional to the number of edges in p. The length $\ell(p)$ for each path can easily be computed as $\delta(\textit{lastedge}(p)) + \ell(\textit{prefpath}(p))$. We will see later that along with this information we can keep track of many other simple properties of the paths, in constant time per path.

4 Representing Paths by a Heap

The representation of s-t paths discussed in the previous section gives a natural tree of paths, in which the parent of any path p is *prefpath*(p). The degree of any node in this *path tree* is at most m, since there can be at most one child for each possible value of *lastedge*(p). The possible values of *lastedge*(q) for paths q that are children of p are exactly those edges in $G-T$ that have tails on the path from *head*(*lastedge*(p)) to t in the shortest path tree T.

If G contains cycles, the path tree is infinite. By Lemma 5, the path tree is heap-ordered. However since its degree is not constant, we cannot directly apply Lemma 1 to find its k minimum values. Instead we form a heap by replacing each node p of the path tree with an equivalent bounded-degree subtree (essentially, a heap of the edges with tails on the path from *head*(*lastedge*(p)) to t, ordered by $\delta(e)$). We must also take care that we do this in such a way that the portion of the path tree explored by our algorithm can be easily constructed.

4.1 Heaps of Edges at Vertices

For each vertex v we wish to form a heap $H_G(v)$ for all edges with tails on the path from v to t, ordered by $\delta(e)$. We will later use this heap to modify the path tree by replacing each node p with a copy of $H_G(\textit{head}(\textit{lastedge}(p)))$.

Let $out(v)$ denote the edges in $G - T$ with tails at v. We first build a heap $H_{out}(v)$, for each vertex v, of the edges in $out(v)$. The weights used for the heap are simply the values $\delta(e)$ defined earlier. $H_{out}(v)$ will be a 2-heap with the added restriction that the root of the heap only has one child. It can be built for each v in time $O(|out(v)|)$ by letting the root $outroot(v)$ be the edge minimizing $\delta(e)$ in $out(v)$, and letting its child be a heap formed by heapification of the rest of the edges in $out(v)$. The total time for this process is $\sum O(|out(v)|) = O(m)$.

We next form the heap $H_G(v)$ by merging all heaps $H_{out}(w)$ for w on the path in T from v to t. More specifically, for each vertex v we merge $H_{out}(v)$ into $H_G(next_T(v))$ to form $H_G(v)$. We will continue to need $H_G(next_T(v))$, so this merger should be done in a nondestructive fashion.

We guide this merger of heaps using a balanced heap $H_T(v)$ for each vertex v, containing only the roots $outroot(w)$ of the heaps $H_{out}(w)$, for each w on the path from v to t. $H_T(v)$ is formed by inserting $outroot(v)$ into $H_T(next_T(v))$. Since insertion into a balanced heap can be performed with $O(\log n)$ changes of pointers on a path from the root of the heap we can store $H_T(v)$ without changing $H_T(next_T(v))$ by using an additional $O(\log n)$ words of memory to store only the nodes on that path.

$H_G(v)$ is now formed by making each node $outroot(w)$ in $H_T(v)$ point to an additional subtree beyond the two it points two in $H_T(v)$, namely to the rest of heap $H_{out}(w)$. $H_G(v)$ can be constructed at the same time as we construct $H_T(v)$, with a similar amount of work. $H_G(v)$ is thus a 3-heap as each node includes at most either two edges from $H_T(v)$ and one edge from $H_{out}(w)$, or no edges from $H_T(v)$ and two edges from $H_{out}(w)$.

We summarize the construction so far, in a form that emphasizes the shared structure in the various heaps $H_G(v)$.

Lemma 6. *In time $O(m + n \log n)$ we can construct a directed acyclic graph $D(G)$, and a mapping from vertices $v \in G$ to $h(v) \in D(G)$, with the following properties:*

- *$D(G)$ has $O(m + n \log n)$ vertices.*
- *Each vertex in $D(G)$ corresponds to an edge in $G - T$.*
- *Each vertex in $D(G)$ has out-degree at most 3.*
- *The vertices reachable in $D(G)$ from $h(v)$ form a 3-heap $H_G(v)$ in which the vertices of the heap correspond to edges of $G - T$ with tails on the path in T from v to t, in heap order by the values of $\delta(e)$.*

For any vertex v in $D(G)$, let $\delta(v)$ be a shorthand for $\delta(e)$ where e is the edge in G corresponding to v.

4.2 The Path Graph

We have constructed a graph $D(G)$, which in particular provides a structure $H(s)$ representing the paths differing from the original shortest path by the addition of a single edge in $G - T$.

We now describe how to augment $D(G)$ with additional edges to produce a graph which can represent all s-t paths, not just those paths with a single edge in $G - T$.

We define the *path graph* $P(G)$ as follows. The vertices of $P(G)$ are those of $D(G)$, with one additional vertex, the *root* $r = r(s)$. The vertices of $P(G)$ are unweighted, but the edges are given lengths. For each directed edge (u, v) in $P(G)$, we create the edge between the corresponding vertices in $D(G)$, with length $\delta(v) - \delta(u)$. We call such edges *heap edges*. For each vertex v in $P(G)$, corresponding to an edge in G connecting some pair of vertices u and w, we create a new edge from v to $h(w)$ in $P(G)$, having as its length $\delta(h(w))$. We call such edges *cross edges*. We also create an *initial edge* between r and $h(s)$, having as its length $\delta(h(s))$.

$P(G)$ has $O(m + n \log n)$ vertices, each with out-degree at most four. It can be constructed in time $O(m + n \log n)$.

We next show that there is a one-to-one correspondence between s-t paths in G, and paths starting from r in $P(G)$.

We first show how to go from a path p' in $P(G)$ to another path p in G. Recall that p is uniquely defined by *sidetracks*(p), the sequence of edges from p in $G - T$. We construct from p' a sequence *pathseq*(p') and show that it corresponds to a path in G. If p' is empty, *pathseq*(p') is also empty. Otherwise *pathseq*(p') is formed by taking in sequence the edges in G corresponding to tails of cross edges in p', and adding at the end of the sequence the edge in G corresponding to the final vertex of p'.

Lemma 7. *If two paths p and p' in $P(G)$ are different, the sequences pathseq(p) and pathseq(p') will also differ.*

Proof: Let (u, v) be the last cross edge or initial edge shared by both paths before they differ. Then $v = h(v')$ for some $v' \in G$. Since any vertex has a single outgoing cross edge, the first difference in the paths must occur at a heap edge. But the heap edges reachable from v form a tree $H_G(v')$, so once

the paths have differed they cannot rejoin without going across different cross edges, which would cause their sequences to differ. If only one path contained a further cross edge, it would give a longer sequence than the other path. And in the final case, if both paths ended at different nodes of $H_G(v')$, the final elements of the sequences would differ. ∎

Lemma 8. *If p is a path from r in $P(G)$, $pathseq(p) = sidetracks(p')$ for some s-t path p' in G.*

Proof: Let q be formed as a prefix of p, up to but not including the final cross or initial edge (u, v) in p. Note that v must be $h(v')$ for some $v' \in G$. Then $pathseq(q)$ differs from $pathseq(p)$ by the removal of one edge, corresponding to the endpoint w of p. By induction on length $pathseq(q) = sidetracks(q')$ for some path q' in G. The final edge in $sidetracks(q')$ has v' as its head, after which q' follows the shortest path tree T from v' to t. Since w is in $H_G(v')$, the tail of the edge e in $G - T$ corresponding to w can be reached along this path from v' to t, and therefore we can find an s-t path p' in G corresponding to $pathseq(p)$ by changing the final section of q' from its course in T from v' to t, to instead run from v' to e, across e, and from the head of e to t. ∎

Lemma 9. *If p is an s-t path in G, $sidetracks(p) = pathseq(p')$ for some path p' starting from r in $P(G)$.*

Proof: By induction $sidetracks(prefpath(p)) = pathseq(q')$ for some path q' which must end at a node of $P(G)$ corresponding to $lastedge(prefpath(p))$. Let $v = head(lastedge(prefpath(p)))$, then $lastedge(p)$ must be reachable in G from v via a path in T. Therefore a node corresponding to $lastedge(p)$ must be in $H_G(v)$, and we can reach that node in $P(G)$ by taking a cross edge from the end of q' to $h(v)$ and following heap edges to the corresponding node. This produces a path p' in $P(G)$ for which $pathseq(p')$ differs from $pathseq(f')$ by the addition of the single edge $lastedge(p)$. ∎

Lemma 10. *Each edge of $P(G)$ has nonnegative weight.*

Proof: For cross and initial edges this follows from Lemma 3. For heap edges this follows from the heap ordering of each $H(v)$. ∎

Lemma 11. *Let path p in G correspond to nonempty path p' in $P(G)$ by the correspondence indicated in the previous lemmas. Then $\ell(p) = \ell(p') + d(s,t)$.*

Proof: By Lemmas 3 and 4,

$$\begin{aligned}\ell(p) &= d(s,t) + \sum_{e \in p} \delta(e) \\ &= d(s,t) + \sum_{e \in sidetracks(P)} \delta(e) \\ &= d(s,t) + \sum_{e \in pathseq(p')} \delta(e).\end{aligned}$$

Each $e \in pathseq(p')$ corresponds to a node in p'. If we add up the lengths of the heap edges in p' preceding that node, together with the most recent cross edge or initial edge, we get a telescoping series adding to $\delta(e)$. Thus the sum in the final term above is equal to the sum over edges in p' of the lengths of those edges. ∎

We summarize the results of this section.

Lemma 12. *In $O(m + n \log n)$ time we can construct a graph $P(G)$ with a distinguished vertex r, having the following properties.*

- *$P(G)$ has $O(m + n \log n)$ vertices.*
- *Each vertex of $P(G)$ has outdegree at most four.*
- *Each edge of $P(G)$ has nonnegative weight.*
- *There is a one-to-one correspondence between s-t paths in G and paths starting from r in $P(G)$.*
- *The correspondence preserves lengths of paths in that length ℓ in $P(G)$ corresponds to length $d(s,t) + \ell$ in G.*

4.3 The Path Heap

To complete our construction, we find from the path graph $P(G)$ a 4-heap $H(G)$, so that the nodes in $H(G)$ represent paths in G.

Each node in $H(G)$ corresponds to a path in $P(G)$ rooted at r. The parent of a node is the path with one fewer edge. Since $P(G)$ has out-degree four, each node has at most four children. The weight of a node is the length of the corresponding path. Weights are heap-ordered by Lemma 10.

Lemma 13. *$H(G)$ is a 4-heap in which there is a one-to-one correspondence between nodes and s-t paths in G, and in which the length of a path in G is $d(s,t)$ plus the weight of the corresponding node in $H(G)$.*

We note that, if an algorithm explores a connected region of $O(k)$ nodes in $H(G)$, it can represent the nodes in constant space each by assigning them numbers and indicating for each node its parent and the additional edge in the corresponding path of $P(G)$. The children of a node are easy to find simply by following appropriate out-edges in $P(G)$, and the weight of a node is easy to compute from the weight of its parent. It is also easy to maintain along with this representation the corresponding implicit representation of s-t paths in G.

5 Algorithms for k shortest paths

By using the construction of the heap $H(G)$ described above, we can show the following results.

Theorem 1. *We can find an implicit representation of the k shortest s-t paths in a digraph G, in time $O(m+n\log n+k)$. In time $O(m+n\log n)$ we can construct a data structure that will output the shortest paths in order by weight, taking time $O(\log i)$ to output the ith path.*

Proof: We apply the algorithms of Lemmas 1 and 2 to $H(G)$, finding the k shortest s-t paths in G in time $O(k)$ once we have constructed $P(G)$. ∎

We next describe how to compute paths from s to all n vertices of the graph. In fact our construction solves more easily the reverse problem, of finding paths from each vertex to the destination t. The construction of $P(G)$ is as above, except that instead of adding a single root $r(s)$ connected to $h(s)$, we add a root $r(v)$ for each vertex $v \in G$. The modification to $P(G)$ takes $O(n)$ time. Using the modified $P(G)$, we can compute a heap $H_v(G)$ of paths from each v to t, and compute the k smallest such paths in time $O(k)$.

Theorem 2. *Given a source vertex s in a digraph G, we can find in time $O(m+n\log n+kn)$ an implicit representation of the k shortest paths from s to each other vertex in G.*

Proof: We apply the construction above to G^R, with s as destination. We form the modified path graph $P(G^R)$, find for each vertex v a heap $H_v(G^R)$ of paths in G^R from v to s, and apply Lemma 1 to this heap. Each resulting path corresponds to a path from s to v in G. ∎

6 Improved Space and Time

The only non-optimal part of our time bound is the $O(n \log n)$ term. For certain graphs, or with certain assumptions about edge lengths, shortest paths can be computed more quickly than $O(m + n \log n)$ [2, 16, 19, 22]. However the $O(n \log n)$ term in the bounds above comes both from a single-source shortest path computation, and from a sequence of heap operations performed in our algorithm. In this section we show how to reduce the part of the time bound coming from the heap operations. As a consequence we can also improve the space used by our algorithm.

Recall that the bottleneck of our algorithm is the construction of $H_T(v)$, a heap for each vertex v in G of those vertices on the path from v to t in the shortest path tree T. The vertices in $H_T(v)$ are in heap order by $\delta(\mathit{outroot}(u))$. In this section we consider the abstract problem, given a tree T with weighted nodes, of constructing a heap $H_T(v)$ for each vertex v of the other nodes on the path from v to the root of the tree. The construction of Lemma 6 solves this problem in time and space $O(n \log n)$; here we give a more efficient but also more complicated solution.

By introducing dummy nodes with large weights, we can assume without loss of generality that T is binary. We use the following technique of Frederickson [13].

Definition 1. *A restricted partition of order z with respect to a binary tree T is a partition of the vertices of V such that:*

1. *Each set in the partition contains at most z vertices.*
2. *Each set in the partition induces a connected subtree of T.*
3. *For each set S in the partition, if S contains more than one vertex, then there are at most two tree edges having one endpoint in S.*
4. *No two sets can be combined and still satisfy the other conditions.*

For our application, z will always be 2. In general such a partition can easily be found in linear time by merging sets until we get stuck. However by working bottom up we can find an optimal partition in linear time.

Lemma 14. *In linear time we can find a restricted partition of a binary tree T for which there are at most $5n/6$ sets in the partition.*

Proof: We assume without loss of generality that no leaf of T has a parent with degree one. Otherwise we could add another child to the parent and produce a tree with a worse ratio of partition size to tree size.

We partition the nodes of the tree into classes. C_0 consists of the leaves. C_1 consists of the degree one nodes C_2 consists of the remaining degree two nodes. As in any binary tree we have the relation $|C_0| = |C_2| - 1$.

Then by working bottom up, we find a partition with the following properties: Every parent of a node in C_0 is paired with a leaf child, so at least half the nodes in C_0 are paired. For each chain of k nodes in C_1 we can form $\lfloor k/2 \rfloor$ pairs.

We count the total number of pairs formed by charging 1/2 to each node in C_0, and 1/2 to each node in a chain of nodes in C_1 other than the bottommost element in the chain. The uncharged nodes consist of those in C_2 together with one node per chain of nodes in C_1. If $|C_1| \leq |C_2|$, $n = |C_0| + |C_1| + |C_2| < 3|C_0|$ and the total charge is $1/2|C_0| > n/6$. Otherwise, $|C_1| > |C_2|$, and since each chain of nodes in C_1 has at its bottom a node in C_2, the number of uncharged nodes in C_1 is at most $|C_2|$. So the total charge is $1/2(|C_0| + |C_1| - |C_2|) > 1/2|C_1| > 1/2(1/3|C_1 + 2/3|C_2|) > 1/2(1/3|C_1| + 1/3|C_2| + 1/3|C_0|) = n/6$. ∎

Contracting each set in a restricted partition gives again a binary tree. We form a *multi-level* partition [13] by recursively partitioning this contracted binary tree. We define a sequence of trees T_i as follows. Let $T_0 = T$. For any $i > 0$, let T_i be formed from T_{i-1} by performing a restricted partition as above and contracting the resulting sets. Then $|T_i| = O((5/6)^i n)$.

For any set S of vertices in T_{i-1} contracted to form a vertex v in T_i, define *nextlevel*(S) to be the set in the partition of T_i containing S. We say that S is an *interior* set if it is contracted to a degree two vertex. Since T_i is a contraction of T, each edge in T_i corresponds to an edge in T. Let e be the outgoing edge from v in T_i; then we define *rootpath*(S) to be the path in T from *head*(e) to t. If S is an interior set, with outgoing edge e', we let *inpath*(S) to be the path in T from *head*(e') to *tail*(e).

We first construct a partial heap $(M_1(S), H_1(S))$ for the vertices in each path *inpath*(S). We will make sure $|M_1(S)| > i$, so (M_1, H_1) will generally form an $(i+1)$-partial heap, but in some cases there will be an even larger number of elements in M_1. In particular, let $M_2(S)$ denote those elements in $M_1(S')$ for those S' containing S at higher levels of the multi-level partition; then we make $|M_1(S)| = \max(i+2, |M_2(S)|+1)$. Let m_i denote the sum of $|M_1(S)|$ over sets S contracted in T_i.

Lemma 15. *For each i, $m_i = O(i|T_i|)$.*

Proof: By the definition of M_1, $m_i \le m_{i+1} + \sum_S (i+2)$. Define $m'_j = \sum_{S \in T_j} (j+2) = O(j|T_j|/(5/6)^{j-i})$. Then

$$m_i \le \sum_{j \ge i} m'_j = O(|T_i| \sum_{j \ge i} j/(5/6)^{j-i}) = O(i|T_i|).$$

■

We use the following data structure to compute the sets $M_1(S)$. For each interior set S. we form a priority queue $Q(S)$, consisting of the heads of the priority queues for those subsets contracted at the next lower level that contain vertices of *inpath*(S). Since each set S has at most two elements these priority queues are trivial to maintain in constant time per operation.

We compute the sets $M_1(S)$ starting from the top of the multi-level partition and working down. We add points one at a time to each set $M_1(S)$, until there are enough to satisfy the definition above of $|M_1(S)|$. Whenever we add a point to $M_1(S)$ we add the same point to $M_1(S')$ for each lower level subset S' to which it also belongs. A point is added by removing it from $Q(S)$ and from the priority queues of the sets at each level. We then update the queues bottom up, recomputing the head of each queue and inserting it in the queue at the next level.

Lemma 16. *The amount of time to compute $M_1(S)$ for all sets S in the multi-level partition, as described above, is $O(n)$.*

Proof: By Lemma 15, the number of operations in priority queues for subsets of T_i is $O(i|T_i|)$. So the total time is $\sum O(i|T_i|) = O(n \sum i/(5/6)^i) = O(n)$. ■

We next describe how to compute the heaps $H_1(S)$ for the points on *inpath*(S) that have not been chosen as part of $M_1(S)$. For this stage we

work bottom up. Recall that S corresponds to one or two vertices of T_i; each vertex corresponds to a set S' contracted at a previous level of the multi-level partition. For each such S' along the path in S we will have already formed the partial heap $(M_1(S'), H_1(S'))$. We let $H_2(S')$ be a heap formed by adding the vertices in $M_1(S') - M_1(S)$ to $H_1(S')$. Since $M_1(S') - M_1(S)$ consists of at least one vertex (because of the requirement that $|M_1(S') \geq |M_1(S)| + 1$), we can form $H_2(S')$ as a 2-heap in which the root has degree one.

If S consists of a single vertex we then let $H_1(S) = H_2(S')$; otherwise we form $H_1(S)$ by combining the two heaps $H_2(S')$ for its two children. The time is constant per set S or linear overall.

We next compute another collection of partial heaps $(M_3(S), H_3(S))$ of vertices in $\mathit{rootpath}(S)$ for each set S contracted at some level of the tree. If S is a set contracted to a vertex in T_i, we let $(M_3(S), H_3(S))$ be an $i+1$-partial heap. In this phase of the algorithm, we work top down. For each set S, consisting of a collection of vertices in T_{i-1}, we use $(M_3(S), H_3(S))$ to compute for each vertex S' the partial heap $(M_3(S'), H_3(S'))$.

If S consists of a single set S', or if S' is the parent of the two vertices in S, we let $M_3(S')$ be formed by removing the least element from $M_3(S)$ and we let $H_3(S')$ be formed by adding that least element as a new root to $H_3(S)$.

In the remaining case, if S' and $\mathit{parent}(S')$ are both in S, we form $M_3(S')$ by taking the $i+1$ minimum values in $M_1(\mathit{parent}(S')) \cup M_3(\mathit{parent}(S'))$. The remaining values in $M_1(\mathit{parent}(S')) \cup M_3(\mathit{parent}(S')) - M_3(S')$ must include at least one value v greater than everything in $H_1(\mathit{parent}(S'))$. We form $H_3(S')$ by sorting those remaining values into a chain, together with the root of heap $H_3(\mathit{parent}(S')$, and connecting v to $H_1(\mathit{parent}(S'))$.

To complete the process, we compute the heaps $H_T(v)$ for each vertex v. Each such vertex is in T_0, so the construction above has already produced a 1-partial heap $(M_3(v), H_3(v))$. We must add the value for v itself and produce a true heap, both of which are easy. This completes the proof of the following lemma.

Lemma 17. *Given a tree T with weighted nodes, we can construct for each vertex v a 2-heap $H_T(v)$ of all nodes on the path from v to the root of the tree, in total time and space $O(n)$.*

Proof: The time for constructing (M_1, H_1) has already been analyzed. The only remaining part of the algorithm that does not take constant time per set is the time for sorting remaining values into a chain, in time $O(i \log i)$

for a set at level i of the construction. The total time at level i is thus $O(|T_i| i \log i)$ which, summed over all i, gives $O(n)$. ∎

Applying this technique in place of Lemma 6 gives the following result.

Theorem 3. *Given a digraph G and a shortest path tree from a vertex s, we can find an implicit representation of the k shortest s-t paths in G, in time and space $O(m + n + k)$.*

7 Maintaining Path Properties

In this section we show that our algorithm can maintain along with the other information in $H(G)$ various forms of simple information about the corresponding s-t paths in G.

We have already seen that $H(G)$ allows us to recover the lengths of paths. However lengths are not as difficult as some other information might be to maintain, since they form an additive group. We used this group property in defining $\delta(e)$ to be a difference of path lengths, and in defining edges of $P(G)$ to have weights that were differences of quantities $\delta(e)$.

We now show that we can in fact keep track of any quantity formed by combining information from the edges of the path using any monoid. We assume that there is some given function taking each edge e to an element $\mathit{value}(e)$ of a monoid, and that we can compute the composite value $\mathit{value}(e) \cdot \mathit{value}(f)$ in constant time. By associativity of monoids, the value $\mathit{value}(p)$ of a path p is well defined. Examples of such values include the path length and number of edges in a path (for which composition is real or integer addition) and the longest or shortest edge in a path (for which composition is minimization or maximization).

Recall that for each vertex we compute a heap $H_G(v)$ representing the possible sidetracks possible on the shortest path from v to t. For each node x in $H_G(v)$ we maintain two values: $\mathit{pathstart}(x)$ pointing to a vertex on the path from v to t, and $\mathit{value}(x)$ representing the value of the path from $\mathit{pathstart}(x)$ to the head of the sidetrack edge represented by x. We require that *pathstart* of the root of the tree is v itself, that $\mathit{pathstart}(x)$ be a vertex between v and the head of the sidetrack edge representing x, and that all descendents of x have *pathstart* values on the path from $\mathit{pathstart}(x)$ to t. For each edge in $H_G(v)$ connecting nodes x and y we store a further value, representing the value of the path from $\mathit{pathstart}(x)$ to $\mathit{pathstart}(y)$. We also store for each vertex in G the value of the shortest path from v to t.

Then as we compute paths from the root in the heap $H(G)$, representing s-t paths in G, we can keep track of the value of each path merely by composing the stored values of appropriate paths and nodes in the path in $H(G)$. Specifically, when we follow an edge in a heap $H_G(v)$ we include the value stored at that edge, and when we take a sidetrack edge e from a node x in $H_G(v)$ we include $\mathit{value}(x)$ and $\mathit{value}(e)$. Finally we include the value of the shortest path to t from the tail of the last sidetrack edge to t. The portion of the value except for the final shortest path can be updated in constant time from the same information for a shorter path in $H(G)$, and the remaining shortest path value can be included again in constant time, so this computation takes $O(1)$ time per path found.

The remaining difficulty is computing the values $\mathit{value}(x)$, $\mathit{pathstart}(x)$, and also the values of edges in $H_G(v)$.

In the construction of Lemma 6, we need only compute these values for the $O(\log n)$ nodes by which $H_G(v)$ differs from $H_G(\mathit{parent}(v))$, and we can compute each such value as we update the heap in constant time per value. Thus the construction here goes through with unchanged complexity.

In the construction of Lemma 17, each partial heap at each level of the construction corresponds to all sidetracks with heads taken from some path in the shortest path tree. As each partial heap is formed the corresponding path is formed by concatenating two shorter paths. We let $\mathit{pathstart}(x)$ for each root of a heap be equal to the endpoint of this path farthest from t. We also store for each partial heap the near endpoint of the path, and the value of the path. Then these values can all be updated in constant time when we merge heaps.

Theorem 4. *Given a digraph G and a shortest path tree from a vertex s, and given a monoid with values $\mathit{value}(e)$ for each edge $e \in G$, we can compute $\mathit{value}(p)$ for each of the k shortest s-t paths in G, in time and space $O(m + n + k)$.*

8 Conclusions

We have described algorithms for the k shortest paths problem, improving by an order of magnitude previously known bounds.

We list the following as open problems.

- Are there properties of paths not described by monoids which we can nevertheless compute efficiently from our representation? In particular

how quickly can we test whether each path generated is simple?

- The linear time construction detailed in Lemma 17 is very complicated. Is there a simpler method for achieving the same result? How quickly can we maintain heaps $H_T(v)$ if new leaves are added to the tree? (Lemma 6 solves this in $O(\log n)$ time per vertex but it seems that at least $O(\log \log n)$ should be possible.)

- Certain applications mentioned in the introduction can be solved as shortest path problems, but have more efficient alternate solutions. For instance, consider the maximum area or perimeter convex r-gon inscribed in a convex n-gon. Once we fix a vertex on the r-gon, the problem is one of computing a shortest path in an r-level acyclic graph with n vertices and $O(n^2)$ edges per level. The overall problem can then be solved as a shortest path in a single graph formed by combining n such subgraphs. However the problem can be solved much more efficiently, in time $O(n \log n + n\sqrt{r \log n})$ [1]. Our algorithms give an $O(rn^3 + k)$ time solution to finding the k best r-gons, which is efficient only when $k = \Omega(n^3)$. Can we improve the $O(rn^3)$ term in this bound?

References

[1] A. Aggarwal, B. Schieber, and T. Tokuyama. Finding a minimum weight K-link path in graphs with Monge property and applications. In *Proc. 9th ACM Symp. Comput. Geom.*, pages 189–197, 1993.

[2] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *J. Assoc. Comput. Mach.*, 37:213–223, 1990.

[3] J. A. Azevedo, M. E. O. Santos Costa, J. J. E. R. Silvestre Madeira, and E. Q. V. Martins. An algorithm for the ranking of shortest paths. *European J. Operational Research*, 69:97–106, 1993.

[4] A. Bako. All paths in an activity network. *Mathematische Operationsforschung und Statistik*, 7:851–858, 1976.

[5] A. Bako and P. Kas. Determining the k-th shortest path by matrix method. *Szigma*, 10:61–66, 1977. In Hungarian.

[6] T. H. Byers and M. S. Waterman. Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. *Operations Research*, 32:1381–1384, 1984.

[7] P. Carraresi and C. Sodini. A binary enumeration tree to find k shortest paths. In *7th Symp. Operations Research*, pages 177–188. Methods of Operations Research, 1983.

[8] M. T. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Comp. Geom. Theory & Appl.* To appear.

[9] S. E. Dreyfus. An appraisal of some shortest path algorithms. *Operations Research*, 17:395–412, 1969. Romanovsky cites this as being pp. 345-412. I haven't doublechecked which is correct.

[10] El-Amin and Al-Ghamdi. An expert system for transmission line route selection. In *Int. Power Engineering Conf*, volume 2, pages 697–702. Nanyang Technol. Univ, Singapore, 1993.

[11] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Symp. Foundations of Computer Science*, pages 60–69, 1992.

[12] B. L. Fox. k-th shortest paths and applications to the probabilistic networks. In *ORSA/TIMS National Mtg*, volume 23, page B263. Bull. Operations Research Soc. of America, 1975.

[13] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *Proc. 32nd IEEE Symp. Foundations of Computer Science*, pages 632–641, 1991.

[14] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104:197–214, 1993.

[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[16] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. 31st IEEE Symp. Foundations of Computer Science*, pages 719–725, 1990.

[17] G. J. Horne. Finding the k least cost paths in an acyclic activity network. *J. Operational Research Society*, 31:443–448, 1980.

[18] L.-M. Jin and S.-P. Chan. An electrical method for finding suboptimal routes. In *Proc. IEEE Int. Symp. Circuits and Systems*, volume 2, pages 935–938, 1989.

[19] D. B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.

[20] N. Katoh, T. Ibaraki, and H. Mine. An $O(Kn^2)$ algorithm for K shortest simple paths in an undirected graph with nonnegative arc length. *Trans. Inst. Electronics and Communication Engineers of Japan*, E61:971–972, 1978.

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

[22] P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *26th ACM Symp. Theory of Computing*, pages 27–37, 1994.

[23] E. L. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18:401–405, 1972.

[24] E. L. Lawler. Comment on computing the k shortest paths in a graph. *Commun. Assoc. Comput. Mach.*, 20:603–604, 1977.

[25] E. Q. V. Martins. An algorithm for ranking paths that may contain cycles. *European J. Operational Research*, 18:123–130, 1984.

[26] S.-P. Miaou and S.-M. Chin. Computing k-shortest path for nuclear spent fuel highway transportation. *European J. Operational Research*, 53:64–80, 1991.

[27] E. Minieka. On computing sets of shortest paths in a graph. *Commun. Assoc. Comput. Mach.*, 17:351–353, 1974.

[28] E. Minieka. The K-th shortest path problem. In *ORSA/TIMS National Mtg.*, volume 23, page B/116. Bull. Operations Research Soc. of America, 1975.

[29] E. Minieka and D. R. Shier. A note on an algebra for the k best routes in a network. *J. Inst. Mathematics and Its Applications*, 11:145–149, 1973.

[30] A. Perko. Implementation of algorithms for k shortest loopless paths. *Networks*, 16:149–160, 1986.

[31] D. R. Shier. Computational experience with an algorithm for finding the K shortest paths in a network. *J. Res. Nat. Bur. Standards Sec. B*, 78B:139–165, 1974.

[32] D. R. Shier. Algorithms for finding the k shortest paths in a network. In *ORSA/TIMS Joint National Mtg.*, page 115. TIMS/ORSA Bulletin, 1976.

[33] D. R. Shier. Iterative methods for determining the k shortest paths in a network. *Networks*, 6:205–229, 1976.

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

[35] C. C. Skicism and B. L. Golden. Solving k-shortest and constrained shortest path problems efficiently. *Ann. Operations Research*, 20:249–282, 1989.

[36] K. Sugimoto and M. Katoh. An algorithm for finding k shortest loopless paths in a directed network. *Trans. Information Processing Soc. Japan*, 26:356–364, 1985. In Japanese.

[37] R. Thumer. A method for selecting the shortest path of a network. *Zeitschrift fur Operations Research, Serie B (Praxis)*, 19:B149–153, 1975. In German.

[38] M. M. Weigand. A new algorithm for the solution of the k-th best route problem. *Computing*, 16:139–151, 1976.

[39] A. Wongseelashote. An algebra for determining all path-values in a network with application to k-shortest-paths problems. *Networks*, 6:307–334, 1976.

[40] A. Wongseelashote. Semirings and path spaces. *Discrete Mathematics*, 26:55–78, 1979.

[41] J. Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

[42] J. Y. Yen. Another algorithm for finding the k shortest-loopless network paths. In *41st Mtg. Operations Research Society of America*, volume 20, page B/185. Bull. Operations Research Soc. of America, 1972.