Finding the k Shortest Paths

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

Abstract

We give algorithms for finding the k shortest paths (not required to be simple) connecting a pair of vertices in a digraph. Our algorithms output an implicit representation of these paths in a digraph with n vertices and m edges, in time $O(m + n \log n + k)$. We can also find the k shortest paths from a given source s to each vertex in the graph, in total time $O(m + n \log n + kn)$. We describe applications to dynamic programming problems including the knapsack problem, sequence alignment, and maximum inscribed polygons.

1 Introduction

We consider a long-studied generalization of the shortest path problem, in which not one but several short paths must be produced. The *k shortest paths problem* is to list the k paths connecting a given source-destination pair in the digraph with minimum total length. In the version of the problem we study, cycles of repeated vertices are allowed. We first present a basic version of our algorithm, which is simple enough to be suitable for practical implementation while losing only a logarithmic factor in time complexity. We then show how to achieve optimal time (linear once a shortest path tree has been computed) by applying Frederickson's [18] algorithm for finding the minimum k elements in a *heap-ordered tree.*

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. Many optimization problems solved by dynamic programming or more complicated matrix searching techniques, such as the knapsack problem, sequence alignment in molecular biology, construction of optimal inscribed polygons, and length-limited Huffman coding, can be expressed as shortest path problems.

Methods for finding k shortest paths 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 [14], 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 [13] and Lawler [27] 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, 44]. This problem can be reduced to finding several shortest paths in a grid graph.

We later discuss in more detail some of the dynamic programming applications listed above, and show how to find the k best solutions to these problems by using our shortest path algorithms. Our results improve previous algorithms for finding length-bounded paths in the grid graphs arising in sequence alignment [6] and for finding the k best solutions to the knapsack problem [11].

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 per path. 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 find the k shortest paths from a given source in a digraph to each other vertex, in time $O(m + n \log n + kn)$.

We can also solve the similar problem of finding all paths shorter than a given length, with the same time bounds. The same techniques apply to digraphs with negative edges but no negative cycles, but the time bounds above should be modified to include the time to compute a single source shortest path tree in such networks. For a DAG (with or without negative edge lengths) shortest path trees can be constructed in linear time and the $O(n \log n)$ term above can be omitted. The related problem of finding the k longest paths in a DAG [4] can be transformed to a shortest path problem simply by negating all edge lengths; we can therefore also solve it in the same time bounds.

1.3 Related Work

Many papers study algorithms for k shortest paths [3, 5, 7, 13, 16, 21, 22, 24, 25, 27, 28, 29, 31, 32, 33, 35, 37, 38, 39, 40, 41, 43, 45, 46, 47, 48, 49]. Dreyfus [13] and Yen [49] 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, 13, 16, 29, 30, 39, 40] 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 the acyclic digraphs that arise 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 [42], or include capacities on the edges [8, 9, 10, 36], however such changes turn the problem into one more closely related to network flow.

Fox [16] gives what seems to be the best previously known bound for the k shortest path problem, $O(n^2 + kn \log n)$. Dreyfus [13] 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 bound is $O(k(m + n \log n))$ [25]. Thus all previous algorithms took time $O(n \log n)$ or more per path. We improve this to constant time per path.

A similar problem to the one studied here is that of finding the k minimum weight spanning trees in a graph. Recent algorithms for this problem [15, 17] 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 [12]. 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 The Basic Algorithm

Finding the k shortest paths between two terminals s and t has been a difficult enough problem to warrant much research. In contrast, the similar problem of finding paths with only one terminal s, ending anywhere else in the graph, is much easier: one can simply use breadth first search. If the graph has bounded degree d, a breadth first search from s until k paths are found takes time $O(dk + k \log k)$; note that this bound does not depend in any way on the overall size of the graph. If the paths need not be output in order by length, Frederickson's heap selection algorithm [18] can be used to speed this up to $O(dk)$.

The main idea of our k shortest paths algorithm, then, is to translate the problem from one with two terminals, s and t, to a problem with only one terminal. One can find paths from s to t simply by finding paths from s to any other vertex and concatenating the shortest path from that vertex to t. However we cannot simply apply this idea directly, for several reasons: (1) There is no obvious relation between the ordering of the paths from s to other vertices, and of the corresponding paths from s to t. (2) Each path from s to t may be represented in many ways as a path from s to some vertex followed by a shortest path from that vertex to t. (3) Our input graph may not have bounded degree.

In outline, we deal with problem (1) by using a potential function to modify the edge lengths in the graph so that the length of any shortest path to t is zero; therefore concatenating such paths to paths from s will preserve the ordering of the path lengths. We deal with problem (2) by only considering paths from s in which the last edge is not on a shortest path to t; this leads to the implicit representation we use to represent each path in constant space. (Similar ideas to these appear also in [34].) However this solution gives rise to a fourth problem: (4) We do not wish to spend much time searching edges in shortest paths to t, as this time can not be charged against newly found s-t paths.

The heart of our algorithm is the solution to problems (3) and (4). Our idea is to construct a binary heap for each vertex, listing the edges that are not on shortest paths to t and that can be reached from that vertex by shortest-path edges. In order to save time and space, we use persistence techniques to allow these heaps to share common structures with each other. In the basic version of the algorithm, this collection of heaps forms a bounded-degree graph having $O(m + n \log n)$ vertices. Later we show how to improve the time and space bounds of this part of the algorithm using tree decomposition techniques of Frederickson [17].

2.1 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* of an edge e is denoted $\ell(e)$. By extension we can define the length $\ell(p)$ for any path in G to be the sum of its edge lengths. 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 $tail(e)$ and $head(e)$; the edge is directed from $tail(e)$ to $head(e)$.

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 depth). More generally, a *D-heap* is a degree-D tree with the same weight-ordering 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. We can list the elements of a heap in order by weight, taking logarithmic time to generate each element, simply by using breadth first search.

2.2 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 digraph may have length $\Omega(ni)$, so the best time we could hope for in an explicit listing of shortest paths would be $O(k^2 n)$. Our time bounds are faster than this, 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 [15, 17]. Further, previous papers on the k shortest path problem give time bounds omitting the $O(k^2 n)$ term 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 $O(m + n \log n)$ [19]. 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 1. *For any $e \in G$, $\delta(e) \geq 0$. For any $e \in T$, $\delta(e) = 0$.*

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 $sidetracks(p)$ 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 $S = sidetracks(p)$, we define $path(S)$ 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 $sidetracks(p)$.

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

Lemma 2. *For any path p from s to t,*

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

Lemma 3. *For any path p from s to t in G, for which $sidetracks(p)$ is nonempty, $\ell(p) \geq \ell(prefpath(p))$.*

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 the final version of our algorithm, which uses Frederickson's heap selection technique, does not necessarily output paths in sorted order, we will nevertheless be able to guarantee that $prefpath(p)$ is output before p. One can easily recover p itself from our representation in time proportional to the number of edges in p. The length $\ell(p)$ for each path can easily be computed as $\delta(lastedge(p)) + \ell(prefpath(p))$. We will see later that we can also compute many other simple properties of the paths, in constant time per path.

2.3 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 3, the path tree is heap-ordered. However since its degree is not constant, we cannot directly apply breadth first search (nor Frederickson's heap selection technique, described later in Lemma 7) 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.

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 $H_G(head(lastedge(p)))$.

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 persistent (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.

We now form $H_G(v)$ by connecting each node *outroot*(w) in $H_T(v)$ to an additional subtree beyond the two it points to 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 4. *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.

At this point 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 $D(G)$, we create the edge between the corresponding vertices in $P(G)$, 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 outdegree 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 $P(G)$. Recall that an s-t path p in G is uniquely defined by *sidetracks*(p), the sequence of edges from p in $G - T$. We construct from a path p' in $P(G)$ 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 5. *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.*

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. $H(G)$ is constructed by forming a node for each 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. Weights are heap-ordered, and the weight of a node is the length of the corresponding path.

Lemma 6. *$H(G)$ 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.

2.4 Finding the k Shortest Paths

Theorem 1. *In time $O(m + n \log n)$ we can construct a data structure that will output the shortest paths from s to t in a graph in order by weight, taking time $O(\log i)$ to output the ith path.*

Proof: We apply breadth first search to $H(G)$, and translate the search results to paths using the correspondence described above. ∎

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 $O(m + n \log n + kn \log k)$ 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 breadth first search to this heap. Each resulting path corresponds to a path from s to v in G. ∎

3 Improved Space and Time

The basic algorithm described above takes time $O(m + n \log n + k \log k)$, even if a shortest path tree has been given. For certain graphs, or with certain assumptions about edge lengths, shortest paths can be computed more quickly than $O(m + n \log n)$ [2, 20, 23, 26]. Further k may be large enough that the $k \log k$ term dominates the time bound. In this section we show how to reduce the time for our algorithm to $O(m + n + k)$, assuming a shortest path tree is given in the input. As a consequence we can also improve the space used by our algorithm.

3.1 Faster Heap Selection

The following result is due to Frederickson [18].

Lemma 7. *We can find the k smallest weight vertices in any heap, in time $O(k)$.*

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.

By using this algorithm in place of breadth first search, we can reduce the $O(k \log k)$ term in our time bounds to $O(k)$.

3.2 Faster Path Heap Construction

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 $\delta(outroot(u))$. 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 4 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 [17].

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.*

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

Lemma 8 (Frederickson [17]). *In linear time we can find an order-2 partition of a binary tree T for which there are at most $5n/6$ sets in the partition.*

Contracting each set in a restricted partition gives again a binary tree. We form a *multi-level* partition [17] 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) be the path in T from *head*(e') to *tail*(e).

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.

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 9. *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 10. *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 9, 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 $|M_1(S') \ge |M_1(S)|+1$), 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 *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 $parent(S')$ are both in S, we form $M_3(S')$ by taking the $i+1$ minimum values in $M_1(parent(S')) \cup M_3(parent(S'))$. The remaining values in $M_1(parent(S')) \cup M_3(parent(S')) - M_3(S')$ must include at least one value v greater than everything in $H_1(parent(S'))$. We form $H_3(S')$ by sorting those remaining values into a chain, together with the root of heap $H_3(parent(S')$, and connecting v to $H_1(parent(S'))$.

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 11. *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 4 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 $O(m+n+k)$.*

4 Maintaining Path Properties

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 $\delta(e)$.

In fact we can 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 $value(e)$ of a monoid, and that we can compute the composite value $value(e) \cdot value(f)$ in constant time. By associativity of monoids, the value $value(p)$ 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).

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

We omit the details from this extended abstract.

5 Dynamic Programming Applications

Many optimization problems solved by dynamic programming or more complicated matrix searching techniques can be expressed as shortest path problems. Since the graphs arising from dynamic programs are typically acyclic, we can use our algorithm to find longest as well as shortest paths. We demonstrate this approach by a few selected examples.

5.1 The Knapsack Problem

The *optimization 0-1 knapsack problem* (or *knapsack problem* for short) consists of placing "objects" into a "knapsack" that only has room for a subset of the objects, and maximizing the total value of the included objects. Formally, one is given integers L, c_i, and w_i $(0 \le i < n)$ and one must find $x_i \in \{0,1\}$ satisfying $\sum x_i c_i \le L$ and maximizing $\sum x_i w_i$. Dynamic programming solves the problem in time $O(nL)$; Dai et al. [11] show how to find the k best solutions in time $O(knL)$. We now show how to improve this to $O(nL+k)$ using longest paths in a DAG.

Let directed acyclic graph G have $nL+L+2$ vertices: two terminals s and t, and $(n+1)L$ other vertices with labels (i,j), $0 \le i \le n$ and $0 \le j \le L$. Draw an edge from s to each $(0,j)$ and from each (n,j) to t, each having length 0. From each (i,j) with $i < n$, draw two edges: one to $(i+1,j)$ with length 0, and one to $(i+1, j+c_i)$ with length w_i (omit this last edge if $j+c_i > L$).

There is a simple one-to-one correspondence between s-t paths and solutions to the knapsack problem: given a path, define x_i to be 1 if the path includes an edge from (i, j) to $(i+1, j+c_i)$; instead let x_i be 0 if the path includes an edge from (i, j) to $(i+1, j)$. The length of the path is equal to the corresponding value of $\sum x_i w_i$, so we can find the k best solutions simply by finding the k longest paths in the graph.

Theorem 5. *We can find the k best solutions to the knapsack problem as defined above, in time $O(nL+k)$.*

5.2 Sequence Alignment

The *sequence alignment* or *edit distance* problem is that of matching the characters in one sequence against those of another, obtaining a matching of minimum *cost* where the cost combines terms for mismatched and unmatched characters. This problem and many of its variations can be solved in time $O(xy)$ (where x and y denote the lengths of the two sequences) by a dynamic programming algorithm that takes the form of a shortest path computation in a grid graph.

Byers and Waterman [6, 44] describe a problem of finding all near-optimal solutions to sequence alignment and similar dynamic programming problems. Essentially their problem is that of finding all s-t paths with length less than a given bound L. They describe a simple depth first search algorithm for this problem, which is especially suited for grid graphs although it will work in any graph and although the authors discuss it in terms of general DAGs. In a general digraph their algorithm would use time $O(k^2 m)$ and space $O(km)$. In the acyclic case discussed in the paper, these bounds can be reduced to $O(km)$ and $O(m)$. In grid graphs its performance is even better: time $O(xy + k(x+y))$ and space $O(xy)$. Naor and Brutlag [34] discuss improvements to this technique that among other results include a similar time bound for k shortest paths in grid graphs.

We now discuss the performance of our algorithm for the same length-limited path problem. In general one could apply any k shortest paths algorithm together with a doubling search to find the value of k corresponding to the length limit, but in our case the problem can be solved more simply: simply replace the breadth first search in $H(G)$ with a length-limited depth first search.

Theorem 6. *We can find the k s-t paths in a graph G that are shorter than a given length limit L, in time $O(m+n+k)$ once a shortest path tree in G is computed.*

Even for the grid graphs arising in sequence analysis, our $O(xy+k)$ bound improves by a factor of $O(x+y)$ the times of the algorithms of Byers and Waterman [6] and Naor and Brutlag [34].

5.3 Inscribed Polygons

We next discuss the problem of, given an n-vertex convex polygon, finding the "best" approximation to it by an r-vertex polygon, $r < n$. This arises e.g. in computer graphics, in which significant speedups are possible by simplifying the shapes of faraway objects. To our knowledge the "k best solution" version of the problem has not been studied before. We include it as an example in which the best known algorithms for the single solution case do not appear to be of the form needed by our techniques; however one can transform an inefficient algorithm for the original problem into a shortest path problem that with our techniques gives an efficient solution for large enough k.

We formalize the problem as that of finding the maximum area or perimeter convex r-gon inscribed in a convex n-gon. The best known solution takes time $O(n \log n + n\sqrt{r \log n})$ [1]. However this algorithm does not appear to be in the form of a shortest path problem, as needed by our techniques.

Instead we describe a less efficient technique for solving the problem by using shortest paths. Number the n-gon vertices v_1, v_2, etc. Suppose we know that v_i is the lowest numbered vertex to be part of the optimal r-gon. We then form a DAG G_i with $O(rn)$ vertices and $O(rn^2)$ edges, in r levels. In each level we place a copy of each vertex v_j, connected to all vertices with lower numbers in the previous level. Each path from the copy of v_i in the first level of the graph to a vertex in the last level of the graph has r vertices with numbers in ascending order from v_i, and thus corresponds to an inscribed r-gon. We connect one such graph for each initial vertex v_i into one large graph, by adding two vertices s and t, edges from s to each copy of a vertex v_i at the first level of G_i, and edges from each vertex on level r of each G_i to t. Paths in the overall graph G thus correspond to inscribed r-gons with any starting vertex.

It remains to describe the edge lengths in this graph. Edges from s to each v_i will have length zero for either definition of the problem. Edges from a copy of v_i at one level to a copy of v_j at the next level will have length equal to the Euclidean distance from v_i

to v_j, for the maximum perimeter version of the problem, and edges connecting a copy of v_j at the last level to t will have length equal to the distance between v_j and the initial vertex v_i. Thus the length of a path becomes exactly the perimeter of the corresponding polygon, and we can find the k best r-gons by finding the k longest paths.

For the maximum area problem, we instead let the distance from v_i to v_j be measured by the area of the n-gon cut off by a line segment from v_i to v_j. Thus the total length of a path is equal to the total area outside the corresponding r-gon. Since we want to maximize the area inside the r-gon, we can find the k best r-gons by finding the k shortest paths.

Theorem 7. *We can find the k maximum area or perimeter r-gons inscribed in an n-gon, in time $O(rn^3 + k)$.*

6 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.

- The linear time construction when the shortest path tree is known is rather 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 4 solves this in $O(\log n)$ time per vertex but it seems that at least $O(\log\log n)$ should be possible.)
- As described above, we can find the k best inscribed r-gons in an n-gon, in time $O(rn^3 + k)$. However the best single-optimum solution has the much faster time bound $O(n \log n + n\sqrt{r \log n})$ [1]. Our algorithms for the k best r-gons are efficient (in the sense that we use constant time per r-gon) only when $k = \Omega(rn^3)$. Can we improve this bound?
- 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?

Acknowledgements

This work was supported in part by NSF grant CCR-9258355. I thank Greg Frederickson, Sandy Irani and George Lueker for helpful comments on drafts of this paper.

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] G.-H. Chen and Y.-C. Hung. Algorithms for the constrained quickest path problem and the enumeration of quickest paths. *Computers & Operations Research*, 21:113–118, 1994.

[9] Y. L. Chen. An algorithm for finding the k quickest paths in a network. *Computers & Operations Research*, 20:59–65, 1993.

[10] Y. L. Chen. Finding the k quickest simple paths in a network. *Information Processing Letters*, 50:89–92, 1994.

[11] Y. Dai, H. Imai, K. Iwano, and N. Katoh. How to treat delete requests in semi-online problems.

In *Proc. 4th Int. Symp. Algorithms and Computation*, pages 48–57. Springer-Verlag, LNCS 762, 1993.

[12] M. T. Dickerson and D. Eppstein. Fast and simple algorithms for proximity problems in higher dimensions. Manuscript, 1994.

[13] S. E. Dreyfus. An appraisal of some shortest path algorithms. *Operations Research*, 17:395–412, 1969.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

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

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

[30] 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.

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

[32] 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.

[33] 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.

[34] D. Naor and D. Brutlag. On suboptimal alignments of biological sequences. In *Proc. 4th Symp. Combinatorial Pattern Matching*, pages 179–196. Springer-Verlag LNCS 684, 1993.

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

[36] J. B. Rosen, S.-Z. Sun, and G.-L. Xue. Algorithms for the quickest path problem and the enumeration of quickest paths. *Computers & Operations Research*, 18:579–584, 1991.

[37] 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.

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

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

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

[41] 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.

[42] J. W. Suurballe. Disjoint paths in a network. *Networks*, 4:125–145, 1974.

[43] 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.

[44] M. S. Waterman. Sequence alignments in the neighborhood of the optimum. *Proc. Natl. Acad. Sci. USA*, 80:3123–3124, 1983.

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

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

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

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

[49] 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.