2003-06-02

Near-Shortest and K-Shortest Simple Paths, Draft

Carlyle, W. Matthew

þÿCarlyle, W.M., and Wood, R.K., K-Shortest and Near-Shortest Simple Pa
Engineering Science Seminar, University of Auckland, Auckland, New Zealand, 29 March 2003.

Near-Shortest and K-Shortest Simple Paths

W. Matthew Carlyle
R. Kevin Wood
Operations Research Dept.
Naval Postgraduate School
Monterey, CA 93943
2 June 2003

Abstract

We describe a new algorithm for solving the problem of enumerating all near-shortest simple (loopless) s-t paths in a graph $G = (V, E)$ with non-negative edge lengths. Letting $n = |V|$ and $m = |E|$, the time per path enumerated is $O(nS(n, m))$ given a user-selected shortest-path subroutine with complexity $O(S(n, m))$. When coupled with binary search, this algorithm solves the corresponding K-shortest simple paths problem (KSPR) in $O(KnS(n, m)(\log n + \log c_{\max}))$ time, where $c_{\max}$ is the largest edge length. This time complexity is inferior to some other algorithms, but the space complexity is the best available at $O(m)$. Both algorithms are easy to describe, to implement and to extend to more general classes of graphs. In computational tests on grid and road networks, our best polynomial-time algorithm for KSPR appears to be at least an order of magnitude faster than the best algorithm from the literature. However, we devise a simpler algorithm with exponential worst-case complexity, which is orders of magnitude faster yet on the test problems. A minor variant on this algorithm also solves "KSPU," which is analogous to KSPR but with loops allowed.

1 Introduction

The problem of enumerating the K shortest paths in a graph, denoted KSP here, has a long history in operations research and computer science. Eppstein (1998) and Hadjiconstantinou and Christofides (1999) provide excellent reviews, and Eppstein maintains an online bibliography at http://liinwww.ira.uka.de/bibliography/Theory/k-path.html. This paper describes a new algorithm to solve a version of KSP efficiently, in both time and space, through the solution of another problem that has received less attention, the "near-shortest-paths problem" (NSP); see Byers and Waterman (1984). NSP requires enumeration of each path whose length is within a factor of $1 + \epsilon$ of the shortest-path length for some user-specified $\epsilon \geq 0$.

For simplicity, we focus on directed graphs $G = (V, E)$ with integer edge lengths $c_e \geq 0$ for all $e \in E$, and make various extensions later. Throughout, we let $n = |V|$ and $m = |E|$

and assume that $m \geq n$. We also focus on enumerating simple paths, i.e., paths that contain no loops. NSPR and KSPR will denote the restrictions of NSP and KSP, respectively, to enumerating only simple paths. For clarity, we let NSPU and KSPU denote the respective unrestricted problems, i.e., with loops allowed.

KSPR may be more difficult than KSPU (Hadjicontantinou and Christofides 1999), but our applications and thus interests lie with simple paths (e.g., Wevley 1999, Israeli and Wood 2002). Nonetheless, a simple variation of our algorithm for KSPR leads to a new algorithm for KSPU, so we do briefly cover the latter problem.

Byers and Waterman (1984) use dynamic-programming ideas to solve NSPU, with what we shall call the "B&W algorithm." The advantages of solving NSPU with the B&W algorithm, compared to solving KSPU, are that (*i*) it is much simpler to implement than algorithms for KSPU, (*ii*) it requires only $O(m)$ work per path enumerated if the number of loops in any path is bounded by a constant, and (*iii*) it requires only $O(m)$ space to implement under the same conditions. (We ignore the work associated with a single shortest-path problem solved at the beginning of this algorithm and, for all algorithms, we ignore the space required to write out the enumerated paths.) We create a new algorithm, **ANSPR1**, which extends the B&W approach to NSPR while giving up only a little in simplicity and efficiency. To our knowledge, **ANSPR1** is the first algorithm specifically designed to solve NSPR. Importantly, this algorithm maintains $O(m)$ space complexity, and its time complexity increases only to $O(nS(n,m))$ per path enumerated, where $O(S(n,m))$ is the worst-case complexity of the user's shortest-path subroutine.

Of course, there is a disadvantage to solving NSPR rather than KSPR: A user might prefer to prespecify K and solve KSPR, rather than guessing an appropriate value of ϵ and solving NSPR, perhaps obtaining too few or too many paths for the ultimate application. But one pays a price to use the best known algorithm for KSPR: (*i*) the algorithm requires $O(K)$ space, (*ii*) it requires special data structures, and (*iii*) it is difficult to implement, requiring complicated operations to join paths together; however, it does require only $O(n^2)$

work per path enumerated (Hadjiconstantinou and Christofides 1999).

To overcome the difficulties above, we show how our extension of the B&W algorithm can be coupled with a binary search on ϵ to solve KSPR. The amount of work per path enumerated increases over **ANSPR1** by only a factor of $\log c_{\max} + \log n$, and space requirements remain $O(m)$. The simplicity of the approach remains, and computational results are excellent.

Section 2 describes the B&W algorithm for NSPU and provides two modifications for solving NSPR: The first modification is very simple but does not yield polynomial complexity (per path enumerated); the second is only slightly more complicated and does yield polynomial complexity. Section 3 describes some practical improvements to the second, theoretically efficient algorithm. Section 4 then describes how to use any efficient algorithm for NSPR as a subroutine in an algorithm to solve KSPR efficiently. Section 5 gives computational results for implementations of the basic algorithms and their variants. We also describe modifications of one of our algorithms to solve KSPU and give some brief computational results. Section 6 provides conclusions.

2 Solving the Near-Shortest-Paths Problem

We are given a directed graph $G = (V, E)$ with vertex set V and edge set $E \subseteq V \times V$. Each edge $e = (u, v) \in E$ has an integer edge length $c_e \geq 0$; equivalent notation is $c(u, v) \geq 0$. A source vertex s and sink vertex $t \neq s$ are specified, along with a parameter $\epsilon \geq 0$. The (unrestricted) near-shortest-paths problem (NSPU) requires enumeration of all (simple and non-simple) s-t paths that are no longer than $(1 + \epsilon)L_{\min}$, where $L_{\min}$ denotes the length of a shortest s-t path; $L_{\min} > 0$ is assumed.

Byers and Waterman (1984) give a simple algorithm for solving NSP, which can be summarized as follows:

1. For all $v \in V$, find the shortest-path length from v to t, which we denote as $d'(v)$ in this paper. All of these values can be computed by solving a single shortest-path problem

starting at t and traversing edges backwards.

2. Run a straightforward s-t path-enumeration algorithm, but allow loops and extend an s-u subpath to v along the edge $e = (u, v)$ if and only if $L(u) + c(u, v) + d'(v) \leq (1 + \epsilon)L_{\text{min}}$, where $L(u)$ is the length of the current s-u subpath.

3. Whenever an s-t path is found using the above rule, print (output) it.

The correctness of the approach follows from the obvious dynamic-programming interpretation of Step 2.

The B&W algorithm for NSPU is specified below with several dummy statements and other features that facilitate discussion of modifications to solve NSPR. We assume that no vertex will every appear on a given path more than $\mathcal{T}$ times. For simplicity, the algorithm just outputs vertices on the enumerated paths. This would be inappropriate in graphs with parallel edges, but modifications for such instances are straightforward.

B&W Algorithm (Byers and Waterman 1984)
DESCRIPTION: An algorithm to solve NSPU.
INPUT: A directed graph $G = (V, E)$ in adjacency list format, s, t, $\mathbf{c} \geq \mathbf{0}$, and $\epsilon \geq 0$.
    "firstEdge(v)" points to the first edge in a linked list of edges directed out of v.
OUTPUT: All s-t paths (may include loops), whose lengths are within
    a factor of $1 + \epsilon$ of being shortest.

```
{
  /* The following requires the solution of only a single shortest-path problem */
  for( all $v \in V$ ){ $d'(v) \leftarrow$ shortest-path distance from $v$ to $t$; }
  $\bar{d} \leftarrow (1 + \epsilon)d'(s)$;
  for( all $v \in V$ and $\tau = 1, \ldots, \mathcal{T}$ ) { nextEdge($v, t$) $\leftarrow$ firstEdge($v$); }
  theStack $\leftarrow s$; $L(s) \leftarrow 0$;
  /* $\tau(v)$ denotes the number of times vertex $v$ appears on the current subpath */
  /* It is not used in the basic version of this algorithm */
  $\tau(s) \leftarrow 1$; for( all $v \in V - s$ ){ $\tau(v) \leftarrow 0$; }
  while( theStack is not empty ){
    $u \leftarrow$ vertex at the top of theStack;
    if( nextEdge($u, \tau(v)$) $\neq \emptyset$ ) {
      $(u, v) \leftarrow$ the edge pointed to by nextEdge($u$);
      increment nextEdge($u, \tau(u)$);
```

```
            **if**( $L(u) + c(u,v) + d'(v) \le \bar{d}$ ){   /* Step ($i$) */
                **if**( $v = t$ ){
                    print( theStack $\cup\ t$ );
                } **else** {
                    push $v$ on theStack;
                    $\tau(v) \leftarrow \tau(v)+1$;
                    $L(v) \leftarrow L(u) + c(u,v)$;
                    Dummy Step ($ii$);
                }
            }
        } **else** {
            Pop $u$ from theStack;
            $\tau(u) \leftarrow \tau(u)-1$;
            nextEdge($u, \tau(u)$) $\leftarrow$ firstEdge($u$);
            Dummy Step ($iii$);
        }
    }
}
```

A straightforward modification of the B&W algorithm stops it from enumerating paths with loops so that it solves NSPR: If a vertex is already on the "current subpath," i.e., on the s-u path currently represented by the stack, do not allow it to go onto that subpath again. This can be accomplished by replacing the if-statement at Step (i) with

if($\tau(v) = 0$ and $L(u) + c(u,v) + d'(v) \le \bar{d}$) { /* Step (i) modified */

Also, nextEdge(u) replaces nextEdge($u, \tau(u)$) because τ can only equal 1 when examining edges directed out of u in the modified algorithm. We denote this modification of the B&W algorithm as **ANSPR0**.

Unfortunately, **ANSPR0** has exponential complexity because it may waste time extending subpaths that have no feasible completions. This is true because $d'(v)$ gives the length of shortest path from v to t, but such a shortest path might require the traversal of vertices already in the subpath represented by the stack; see Figure 1. However, **ANSPR0** can be faster than our polynomial-time algorithms, in practice, because only a single shortest-path problem is solved, and because it turns out that not much effort is wasted in "going down blind alleys." We investigate this version of the algorithm in more detail, later.

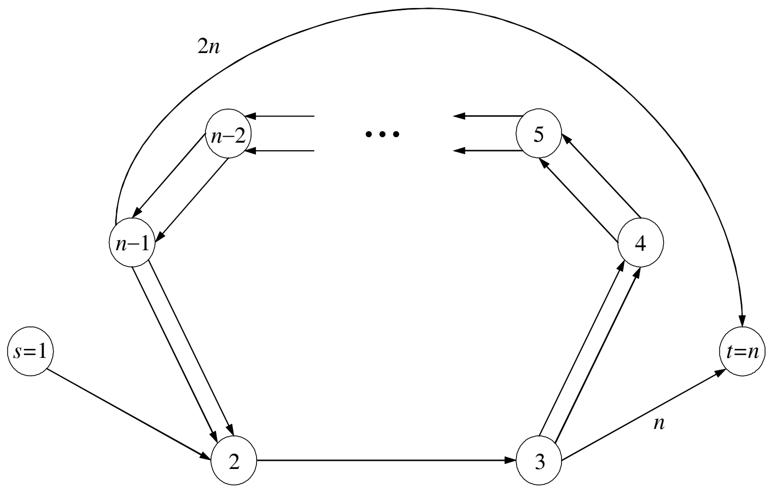

Figure 1: A graph that demonstrates algorithm **ANSPR0**'s exponential worst-case complexity. Suppose $\epsilon = 1.0$, $c_e = 1$ for all unmarked edges, and $c(n-1,t) = 2n$ and $c(3,t) = n$, as marked. Then, there is only one near-shortest path ($s \to 2 \to 3 \to t$), but the algorithm will investigate all 2^{n-3} partial paths starting at vertex 3 and ending at vertex 2, in addition to the 2^{n-4} partial paths traversing edge $(n-1,t)$.

The exponential complexity of **ANSPR0** can be remedied by redefining $d'(v)$ to be the shortest-path distance from v to t that does not use any vertex of the current subpath. This definition requires us, in the worst case, to solve a shortest-path problem each time the current subpath is extended or retracted by one vertex. (We discuss simplifications later.) Therefore, the new algorithm, denoted **ANSPR1**,

1. Uses Step (i) modified, and

2. At Steps (ii) and (iii) uses:

   for(all $v \in V$){ $d'(v) \leftarrow$ shortest-path distance from v to t where no vertices v' with $\tau(v') = 1$ may be traversed; }.

Of course, each for-loop above represents a single call to a shortest-path subroutine which is modified trivially to avoid vertices on the stack.

The argument for **ANSPR1**'s correctness is nearly identical to the argument for the B&W algorithm: **ANSPR1** is a modified path-enumeration algorithm for simple s-t paths which adds an edge to the end of the current subpath if and only if the resulting extended subpath has a feasible completion using only vertices that are not already on the current path, and whose total length does not exceed the cutoff value of $\bar{d} = (1+\epsilon)L_{\min}$. Specifically, if a vertex v is added to the current s-u subpath, then it must have received a distance label $d'(v) \leq \bar{d} - L(u) - c(u,v)$. This label came from a v-t path containing only "unused" vertices, and, therefore, appending this entire path to u, including edge (u,v), is a feasible completion with the desired properties. Conversely, if a vertex v has distance label $d'(v) > \bar{d} - L(u) - c(u,v)$, then clearly there can be no feasible completion of the path once vertex v is added to the stack.

Theorem 1 *The amount of work per enumerated path in* **ANSPR1** *is $O(nS(n,m))$ if the algorithm uses a shortest-path subroutine with worst-case complexity $O(S(n,m))$.*

Proof: Each path has at most $n-1$ edges, so at most $O(nS(n,m))$ work is involved in solving shortest-path problems before generating the first path. The path-enumeration process, apart from solving shortest-path problems, scans at most all of the edges in G before finding that first path, so the work there is at most $O(m)$ and can be ignored. Now, the number of shortest-path problems to be solved before the next path is enumerated (or until the algorithm backtracks all the way to s and discovers that there are no more paths to enumerate) is at most $2n$: At most $n-1$ problems arise while backtracking (perhaps as far as s), and at most $n-1$ problems arise while extending the path back to t. Again, overhead is at most $O(m)$ and can be ignored. The argument holds for all subsequent paths enumerated. ∎

Shortest-path algorithms are covered extensively in Ahuja et al. (1993, pp. 93-165). For graphs with non-negative edge lengths, Dijkstra's "label-setting algorithm" is the classic approach (Dijkstra 1959). For fully dense graphs, where $m = \Omega(n^2)$, the basic implementation of Dijkstra's algorithm has the best worst-case complexity of $O(n^2)$. For less dense graphs, Dijkstra's algorithm implemented with a binary heap (Johnson 1977) is one of the simplest efficient algorithms, with a run time of $O(m \log n)$. Other modifications to Dijkstra's algorithm reduce this to $O(m + n \log n)$ (Fredman and Tarjan 1984) and, theoretically, to $O(m)$ for dense graphs (Johnson 1977).

For shortest-path problems on graphs with arbitrary edge lengths but no negative-length cycles, the algorithm with the best, provable, worst-case performance, $O(mn)$, is a "label-correcting algorithm" implemented with a first-in first-out queue (Bellman 1958). Both polynomial-time and exponential-time versions of label-correcting algorithms can be very fast in practice (Cherkassky et al. 1994).

3 Practical Improvements in Efficiency

Here we investigate four modifications, denoted A, B, C and D, for improving the practical efficiency of **ANSPR1**. The first three modifications maintain the polynomial complexity of the algorithm, although modification D will not except under certain conditions.

3.1 Modification A

The algorithm never extends a path to include a vertex v with distance label $d'(v) > (1+\epsilon)L_{\min} - L_P$, where L_P is the length of the current subpath P, represented by the stack. In the shortest-path calculations, since all edge lengths are non-negative, we can obtain a significant speedup by not updating a label on a vertex if that label exceeds the cutoff. This is "modification A." For typical values of ϵ and in typical, sparse graphs, this modification can prevent the shortest-path algorithm from investigating a huge number of vertices. In fact, as L_P approaches $L_{\min}$, the number of vertices not investigated approaches n.

3.2 Modification B

Suppose that when a path is extended to a new vertex v, we push all of the newly computed $d'()$ values onto a stack: We always use the n topmost values in computations. Then, when the algorithm backtracks at Step (iii), it need not solve another shortest-path problem;

instead, it simply pops the n values of $d'()$ from the top of the stack and makes the now-top n values current again. This "modification B" replaces $O(S(n,m))$ work with $O(n)$ work which is undoubtedly a savings, although worst-case storage requirements increase to $O(n^2)$. (Note: It is also necessary to maintain a parallel stack of the same size containing the predecessors of each vertex in the corresponding shortest-path tree. We assume the reader is familiar with the concept of a "shortest-path tree;" otherwise, see Ahuja, Magnanti and Orlin 1993, pp. 106-107. Of course, our tree is rooted at t and corresponds to traversing edges backwards, rather than starting at s and traversing edges in their nominal directions.)

3.3 Modification C

Suppose that **ANSPR1** is about to extend the current path from u to v, and suppose that v is a leaf of the shortest-path tree computed (or implied) at Step (ii) when u was added to the subpath. Then, none of the values $d'()$ change, except that $d'(v)$ is, essentially, no longer defined. The total amount of work involved in handling such a case is at most $O(n)$. Thus, we would again exchange $O(S(n,m))$ work for the undoubtedly smaller quantity $O(n)$.

This idea generalizes. Suppose we have just computed the current shortest-path tree T, then immediately extend the current subpath to some vertex v, and then discover that all of the vertices u that have v as a predecessor in T are now "too far" from t, in that $d'(u)+L(v) > (1+\epsilon)L_{\min}$. The distance labels $d'(u)$ cannot decrease in subsequent shortest-path calculations, and therefore none of these vertices u can ever join the current subpath. In fact, because edge lengths are non-negative, none of the vertices in the subtree of T rooted at v can ever join the current subpath. Thus, we can avoid recomputing shortest paths entirely in the upcoming iteration. We implement this "modification C" in a single for-loop over the edges directed into v. Note that when there is no vertex u having v as its predecessor, this modification specializes to the "leaf-checking modification" described in the previous paragraph.

When the algorithm extends the current subpath to a vertex v that is not a leaf, it would be possible to recompute shortest-path distances only to those vertices in the shortest-path tree that are "cut off" by adding v, i.e., to only those vertices u that had v as a direct or indirect predecessor. (Of course, no calculations would be necessary for vertices in a subtree rooted at u which is "too far" from t, as in modification C.) A label-correcting algorithm could be arranged to accomplish this task. However, the complexity of the algorithm would increase substantially and new data structures would be necessitated. Our purpose is to develop a simple algorithm, so we have not implemented this modification.

3.4 Modification D

If we modify **ANSPR1** to never recompute shortest-path lengths to t, we end up with **ANSPR0** which has exponential complexity. However, if the algorithm recomputes those path lengths only after it extends the current path by a fixed number of edges ℓ, the work saved may offset the extra work involved in following paths in error. Furthermore, if the maximum degree in G is bounded by a constant d, then the maximum number of edges the algorithm can follow in error is bounded by ℓd^ℓ which may not be too large. Of course, we only claim this "modification D" has polynomial complexity for graphs with bounded degree. "D_ℓ" denotes this modification when shortest-paths are recalculated after every ℓ extensions of the subpath. ABCD_ℓ denotes all four modifications. "D_∞" appears by itself because none of the other modifications are of any consequence when $\ell = \infty$. Indeed, D_∞ simply represents **ANSPR0**.

Results for the basic algorithm, with and without the modifications described above, are presented in Section 5. Before presenting those results, however, we show how to solve the K-shortest paths problem (KSPR) using an algorithm for NSPR as a subroutine.

4 Solving the K-Shortest-Paths Problem

Our approach to solving KSPR uses **ANSPR1** as a subroutine to solve NSPR inside of a binary search on the value of ϵ. For this section, we modify the notation slightly to simplify the presentation: Since edge lengths are integral, all path lengths will be integral; therefore, requiring that path lengths be less than $(1+\epsilon)L_{\min}$ is equivalent to requiring that path lengths be less than $L_{\min}+\delta$, where $\delta = \lfloor \epsilon L_{\min} \rfloor$. As a consequence, we can and do perform binary search on (integer) values of δ (and use $\epsilon = \delta/L_{\min}$ in **ANSPR1**).

4.1 Directed Graphs with Non-negative Edge Lengths

When δ is set to a particular value, it leads to the enumeration of some number of paths denoted here by $\kappa(\delta)$; we assume without loss of generality that $\kappa(0) < K$. The longest (simple) path length is bounded by $nc_{\max}$, so the only relevant values of δ are contained in $\{0, 1, \ldots, nc_{\max}\}$. (The largest value can be shrunk somewhat, but this bound is sufficient for our purposes.) Since $\kappa(\delta)$ is non-decreasing, we can solve KSPR by using binary search on δ, starting with an interval of uncertainty of $[0, nc_{\max}]$ and ending with $[\delta', \delta'']$ such that $\delta'' - \delta' = 1$, $\kappa(\delta') \leq K$ and $\kappa(\delta'') > K$. (We abuse the phrase "interval of uncertainty" slightly.) After identifying δ' and δ'', the solution to KSPR is simple:

1. Enumerate $\kappa(\delta')$ paths using $\epsilon = \delta'/L_{\min}$ in **ANSPR1**. Let the set enumerated be $\mathcal{P}$. If $\kappa(\delta') = K$, go to Step 3.

2. Otherwise, begin **ANSPR1** with $\epsilon = \delta''/L_{\min}$, adding any path with length $L_{\min} + \delta''$ to $\mathcal{P}$ until $|\mathcal{P}| = K$; then halt.

3. The set $\mathcal{P}$ solves KSPR for the given K.

Applying Theorem 1, we can see that the amount of work involved above (i.e., given δ'' and δ' such that $\delta' = \delta'' - 1$) is $O(KnS(n,m))$.

The binary search algorithm will require $O(\log n + \log c_{max})$ iterations to reduce the interval of uncertainty on δ to 1. If we modify **ANSPR1** to always halt after it generates at most $K+1$ paths, except when executing step 2 above, then the total work involved in the binary search is $O(KnS(n,m)(\log n + \log c_{max}))$; the work involved in steps 1 and 2 is therefore dominated. Since we are using **ANSPR1** as a subroutine, the total amount of storage required never exceeds $O(m)$. Hence, we have proven:

Theorem 2 **ANSPR1** *coupled with binary search solves KSPR in* $O(KnS(n,m)(\log n + \log c_{max}))$ *time and* $O(m)$ *space.*

4.2 Extensions

The algorithms described extend trivially to undirected graphs with non-negative edge lengths: Simply replace each undirected edge with two directed, anti-parallel edges both having the undirected edge's length. The algorithms also extend to directed graphs with negative edge lengths as long as there are no negative-length cycles: Solve all shortest-path problems with a label-correcting shortest-path algorithm that handles such situations (e.g., Ahuja, et al. 1993, pp. 136-144). It clear, however, that modification A is inapplicable to such problems.

The B&W algorithm, without modification, solves NSPR in directed acyclic graphs because paths in such graphs can never contain loops. Hence, the B&W algorithm can be used within the binary-search procedure to solve KSPU (equivalently, KSPR) in such graphs. Since the initial shortest-path computation of $d'(v)$ does not need to be repeated at each iteration, the overall complexity of the algorithm will be $O(Km(\log n + \log c_{max}))$. The multiplicative term m arises here instead of n, because the work associated with scanning edges while searching for a path is not dominated by repeated shortest-path calculations, and because $m \geq n$ is assumed; refer to the proof of Theorem 1.

When G contains parallel edges, we may wish to enumerate paths that contain repeated vertices but no repeated edges. The new algorithms for NSPR and KSPR easily extend to handle this situation:

1. Keep track of which edges are on the current s-u path and ignore $\tau()$,
2. Do not allow edge $e = (u, v)$ to extend the current subpath at Step (i) if it is already on that subpath, and

3. Modify the shortest-path subroutine that computes $d'(v)$ so that it does not traverse any edges that are on the current s-u subpath.

5 Computational Results

We have implemented algorithm **ANSPR1** in the C programming language, along with certain combinations of modifications A-D. This section describes tests of the basic algorithm and its variants for directly solving NSPR and for solving KSPR when used as a subroutine within a binary search. We refer to the latter algorithm as **AKSPR1**. All shortest-path problems are solved with the label-correcting algorithm described by Pape (1974). This algorithm is typically very fast, but it has exponential worst-case complexity, and its run times can vary widely between different classes of problems (Cherkassky et al. 1994). However, we like this algorithm's ease of implementation and it behaves adequately for this paper's test problems.

All computations are carried out on a personal computer with an Intel 2.5 GHz Pentium IV processor, 1 GB of RAM, the Microsoft Windows 2000 operating system, and with programs written and compiled using the Microsoft Visual C++ Version 6.0. Run times do not include the time required to write the paths to a text file. This time is roughly proportional to $\bar{e}K$, where $\bar{e}$ represents the average number of edges in the paths enumerated. As an example, about 100 seconds extra are required to write out the paths in **AKSPR** when $K = 10^6$ and when paths average 100–200 edges.

5.1 Test Problems and Environment

We test our algorithms on four different directed graphs:

- "Grid 40×25" is based on a rectangular grid, 25 vertices tall and 40 vertices wide, with a separate source vertex s and sink vertex t external to the grid. The source s is connected to all 25 vertices in the leftmost column of the grid, and all 25 vertices in the rightmost column are connect to t. Each vertex u within the grid has (up to) four edges (u, v) directed out of it, up, down, to the left and to the right, as long as the vertex v exists in the grid. Edge lengths are independent integers drawn from the discrete uniform distribution on [1,10]. This graph has $n = 1,002$ and $m = 3,920$. Cherkassky et al. (1994) use similar graphs for some of their tests on shortest-path algorithms.

- "Grid 100 $\times$ 50" has the same basic structure as Grid 40 $\times$ 25, but uses a 100 $\times$ 50 graph of vertices. This graph has $n = 5,002$ and $m = 19,800$.
- "Road 1" represents the major highways and thoroughfares in the road network of a metropolitan area in the northeastern United States. It covers an area of about 500 square miles, and its integer edge lengths measure 100ths of miles. This graph has $n = 3,670$ and $m = 9,876$.
- "Road 2" depicts the same road network as Road 1, but represents a much higher level of resolution, containing many smaller streets and intersections. This graph has $n = 112,556$ and $m = 274,510$.

The largest graph on which Hadjiconstantinou and Christofides (1999) test their algorithm for KSPR has $n = 1,000$ and $m = 10,000$. In terms of $n + m$, this is about twice as large as our smallest test problem, but about 40 times smaller than our largest. The largest value of K they test is 10^3; the largest we test is 10^7. Their computer is a 100 MHz Silicon Graphics Indigo workstation.

We implement these variants of **ANSPR1**, and corresponding variants of **AKSPR1**:

- The unmodified algorithm denoted "basic,"
- Modification A (denoted "A") which modifies shortest-path calculations to avoid updating distance labels on vertices that are too far from t to be included in any near-shortest path,

- Modifications A and B (“AB”), where B maintains a stack of distance labels to make backtracking in the path enumeration more efficient,
- Modifications A, B and C (“ABC”), where C checks the next vertex to be added to the stack and does not recompute $d'()$ if no relevant changes can occur,
- Modifications A, B, C and D with shortest paths recalculated after every ℓth edge is added to the current subpath (denoted “ABCD$_\ell$ and tested for $\ell = 10$ and $\ell = 50$), and
- Modification D alone in which shortest paths are never recalculated (“D$_\infty$”). This is essentially **ANSPR0**.

We have tested other combinations of these modifications, but believe that the clear impact that each successive modification has on run times shows clearly enough the potential value of each modification.

5.2 Near-shortest Simple Paths

Table 1 displays results obtained by solving NSPR with **ANSPR1**. For all test problems, each added modification significantly improves run times, except in the case of Road 2, in which the combination ABC is slightly slower than AB. Road 2 has a different general structure than the others tested in that it contains many degree-2 vertices. (Small streets have been eliminated in this network’s level of resolution, but the intersections where these small streets were connected to larger streets remain.) This may account for the difference here. However, the difference in run times is insignificant in these cases, and therefore we can probably use the ABC variant with reasonable confidence in most applications.

ANSPR0, labeled “D$_\infty$,” is clearly superior in all instances. However, as noted in Section 2, there are examples in which its run time can be exponential in n, and Figure 1 provides such an example. Table 2 clearly demonstrates this exponential behavior for the graph topology indicated in that figure. However, the ABCD$_{10}$ variant of **ANSPR1** does

			Run Times for Variants of **ANSPR1**						
graph	δ	paths	basic	A	AB	ABC	ABCD$_{10}$	ABCD$_{50}$	D$_\infty$
		(no.)	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)
Grid	0	2	0.02	0.00	0.02	0.00	0.00	0.00	0.00
40×25	1	16	0.06	0.02	0.02	0.00	0.00	0.00	0.00
	2	56	0.14	0.06	0.03	0.02	0.00	0.00	0.00
	3	139	0.28	0.13	0.06	0.03	0.00	0.00	0.00
	4	334	0.59	0.23	0.13	0.05	0.02	0.00	0.00
	5	770	1.27	0.50	0.27	0.09	0.02	0.00	0.00
	6	1633	2.45	0.95	0.50	0.19	0.05	0.02	0.00
Grid	0	6	0.20	0.11	0.08	0.03	0.02	0.00	0.00
100×50	1	44	0.94	0.38	0.22	0.09	0.03	0.00	0.00
	2	218	3.69	1.23	0.78	0.36	0.08	0.02	0.00
	3	894	12.77	3.88	2.45	1.23	0.23	0.06	0.00
	4	3210	39.83	11.13	7.13	3.81	0.67	0.16	0.00
	5	10320	113.52	30.02	19.25	10.77	1.72	0.39	0.02
	6	30632	303.50	76.39	48.89	29.36	4.34	0.95	0.05
Road 1	0	1	0.05	0.02	0.03	0.00	0.00	0.00	0.00
	10	21	0.36	0.11	0.06	0.02	0.00	0.00	0.00
	20	42	0.52	0.11	0.08	0.03	0.02	0.00	0.00
	30	98	1.42	0.31	0.19	0.08	0.02	0.00	0.00
	40	210	2.91	0.64	0.39	0.16	0.03	0.00	0.00
	50	554	7.95	1.78	1.08	0.42	0.09	0.02	0.00
	60	1229	15.97	3.25	1.97	0.86	0.17	0.03	0.02
Road 2	0	1	32.33	1.30	0.84	0.55	0.25	0.20	0.00
	10	22	176.64	4.74	2.55	1.89	0.44	0.22	0.00
	20	65	347.74	7.94	4.19	3.73	0.63	0.24	0.00
	30	179		18.27	9.48	9.17	1.20	0.30	0.00
	40	484		42.61	21.94	22.44	2.47	0.47	0.00
	50	1126		88.22	44.84	47.31	4.94	0.80	0.00
	60	2437		180.81	92.59	97.91	9.81	1.67	0.20

Table 1: Run times, in CPU seconds, for variants of the algorithm **ANSPR1** which solves NSPR. The last variant, which is essentially the exponential algorithm **ANSPR0**, is evidently superior for these problems. Note that "paths" specifies the number of paths found for the given value of δ.

n	δ	D_∞ (sec.)
20	22	0.00
25	27	0.27
26	28	0.55
27	29	1.09
28	30	2.19
29	31	4.41
30	32	8.75

Table 2: A computational example demonstrating algorithm **ANSPR0**'s exponential worst-case complexity. Figure 1 displays the structure of the test graphs for any $n > 5$. The table here reports, for various values of n, the value of δ that corresponds to $\epsilon = 1.0$, and the run time for the D_∞ variant of **ANSPR1**, which is **ANSPR0**: The exponential behavior of the algorithm, as a function of n, is apparent. The run times for the ABCD_{10} variant of **ANSPR1** are all indistinguishable from zero on these problems.

solve each of the instances in Table 2 in time that registers as 0.00 seconds, so this variant may provide a good backup for **ANSPR0** if exponential behavior should ever become a problem. (For K sufficiently large, this exponential behavior would begin to appear even in a network like Road 2. However, $K = 10^7$ is apparently too small to cause difficulties.)

5.3 K Shortest Simple Paths

Table 3 displays results for the different variants of our new algorithm **AKSPR1** used to solve KSPR. We can call the D_∞ variant "**AKSPR0**" since it is really using **ANSPR0** as a subroutine. Tests are performed on the same four networks used for testing **ANSPR1**. For each test graph and for each algorithmic variant, we solved KSPR for various values of K between 10^2 and 10^7. The column labeled δ'' represents the final interval of uncertainty on δ such that $[\delta', \delta''] = [\delta'' - 1, \delta'']$ (and where $\kappa(\delta') \le K$ and $\kappa(\delta'') > K$). Empty entries in the table represent problems that were too difficult to solve in a reasonable amount of time ($< 1,000$ seconds) for the given algorithm.

Hadjiconstantinou and Christofides (1999, Figure 6(c)) indicate a run time for their KSPR algorithm of over 1,400 seconds when $K = 10^3$ for a test graph having 1,000 vertices and an edge-to-vertex density of four. This graph is of roughly the same size and structure as

			Run Times for Variants of **AKSPR1**						
graph	K	δ''	basic (sec.)	A (sec.)	AB (sec.)	ABC (sec.)	ABCD$_{10}$ (sec.)	ABCD$_{50}$ (sec.)	D$_\infty$ (sec.)
Grid	10^2	3	0.81	0.36	0.22	0.08	0.03	0.02	0.00
40×25	10^3	6	6.39	2.52	1.42	0.59	0.14	0.05	0.00
	10^4	9	67.61	23.92	13.22	5.67	1.27	0.50	0.06
	10^5	13		148.77	79.73	38.41	7.63	3.66	0.36
	10^6	17				430.63	80.531	42.00	4.03
	10^7	21							34.00
Grid	10^2	2	5.25	1.89	1.23	0.52	0.11	0.02	0.00
100×50	10^3	4	46.44	13.70	8.72	4.44	0.80	0.19	0.02
	10^4	5		132.53	84.78	48.98	7.50	1.67	0.08
	10^5	8			462.07	284.45	41.48	8.31	0.45
	10^6	10						90.70	5.83
	10^7	13							47.73
Road 1	10^2	31	9.77	2.19	1.36	0.58	0.13	0.03	0.00
	10^3	57	99.77	20.11	12.20	5.41	1.09	0.20	0.03
	10^4	90		153.36	94.09	48.45	8.16	1.59	0.09
	10^5	128			685.07	379.55	59.14	11.30	0.45
	10^6	171						132.27	10.75
	10^7	219							94.08
Road 2	10^2	25	3723.51	82.39	46.11	41.45	6.99	3.20	2.80
	10^3	49			315.11	323.61	35.31	7.44	3.19
	10^4	81			2861.55	2971.93	294.03	51.89	3.73
	10^5	120						424.15	4.69
	10^6	167							16.91
	10^7	222							111.20

Table 3: Run times, in CPU seconds, for variants of **AKSPR1** which solve KSPR. K is the number of paths generated, and δ'' is the value of δ such that $\kappa(\delta - 1) \leq K$ and $\kappa(\delta) > K$. The variants of **AKSPR1** depend on which version of the **ANSPR1** subroutine is used, just as in Table 1.

our Grid 40×25 which we solve in 0.59 seconds (Table 3) with the polynomial-time variant, ABC, of **AKSPR1**; and the D_∞ variant (**AKSPR0**) solves this problem so quickly that it does not register with the C library time functions. Indeed, our best algorithm produces 10^7 paths in about one 40th of the time in which their algorithm produces 10^3 paths. Even taking our faster computer into account, it is safe to conclude that our best algorithms are several orders of magnitude more efficient than theirs.

5.4 K Shortest Paths with Loops Allowed

It is a simple matter to convert **AKSPR0** to "**AKSPU0**," an algorithm to solve KSPU (which allows loops): Simply use the original Byers and Waterman algorithm for NSPU in lieu of the **ANSPR1** subroutine inside of **AKSPR1**. That is, we perform a binary search on δ as in **AKSPR0**, but for each value of $\epsilon = \delta / L_{\min}$, we solve a near-shortest paths problem with loops allowed instead of being explicitly disallowed. Apart from instances in which **AKSPR0** demonstrates its exponential worst-case complexity, we expect the run times for **AKSPU0** to be similar to those for **AKSPR0**, because only one shortest-path problem need be solved in **AKSPU0** for each value of δ. This appears to be the case, as demonstrated by Table 4, which compares the two algorithms on the two largest test problems, Grid 100×50 and Road 2.

Under the assumption that a path visits any single vertex a bounded number of times, the worst-case complexity of **AKSPU0** is $O(Km(\log c_{\max} + \log n)$. This follows from an argument that is similar to the one used in Section 4.2 for establishing the complexity of solving KSPU (equivalently, KSPR) in directed acyclic graphs. The short run times and simplicity of the algorithm certainly make it attractive for practical use, but KSPU is a peripheral issue in this paper and we make no computational comparisons with alternative algorithms.

graph	K	δ''	**AKSPR0** (sec.)	δ''	**AKSPU0** (sec.)
Grid	10^2	3	0.02	2	0.02
100×50	10^3	5	0.02	4	0.02
	10^4	8	0.05	6	0.07
	10^5	11	0.41	9	0.62
	10^6	15	3.11	11	5.16
	10^7	18	40.04	14	41.42
Road 2	10^2	24	3.22	11	2.75
	10^3	49	3.70	17	3.22
	10^4	81	4.34	25	3.50
	10^5	120	5.81	33	7.41
	10^6	167	24.04	43	31.84
	10^7	222	167.74	53	256.10

Table 4: Run times for **AKSPU0** solving KSPU, compared to run times for **AKSPR0** solving KSPR. Note that **AKSPR0** is the D_∞ variant of **AKSPR1** whose times are also presented in Table 3.

6 Conclusions

We have described a theoretically efficient and easily implemented algorithm, **ANSPR1**, for enumerating all near-shortest, simple s-t paths in a directed graph $G = (V, E)$. Near-shortest paths are those that are no longer than $(1+\epsilon)L_{\min}$ where $\epsilon \geq 0$ is a user-specified parameter and $L_{\min}$ is the shortest s-t path length (assumed to be positive). Letting $n = |V|$ and $m = |E|$, the amount of work per path enumerated is $O(nS(n,m))$, where $S(n,m)$ corresponds to the worst-case complexity of the user-selected shortest-path subroutine. The basic algorithm is described for directed graphs with non-negative edge lengths, but easily extends to (i) undirected graphs, (ii) directed graphs with negative-length edges but no negative-length cycles, and (iii) paths with repeated vertices but no repeated edges. **ANSPR1** is also used as a subroutine, combined with binary search in an algorithm denoted **AKSPR1**, to solve the K-shortest-path problem restricted to simple paths (KSPR). All polynomial variants of this algorithm have worst-case complexities of $O(KnS(n,m)(\log n + \log c_{\max}))$, where $c_{\max}$ is the largest edge length.

Several different modifications of **ANSPR1** achieve significant reductions in run time and maintain polynomial complexity. Interestingly, the last modification studied has exponential complexity, yet provides the fastest run times in practice. **ANSPR1** may be viewed as a path-enumeration algorithm that checks whether or not the currently enumerated subpath can be extended to a path of acceptable length. For this check to be accurate, the algorithm may need to run a shortest-path algorithm after each edge is added to or removed from the current subpath. The exponential algorithm performs the shortest-path calculation only once and hence may extend a subpath incorrectly, a mistake that must be corrected later after wasting computational effort. Evidently, in practice, not much effort is wasted.

We finish by reiterating several main points: (*i*) **ANSPR1** and its variants are the first algorithms ever described for NSPR, (*ii*) several variants of **AKSPR1** appear to be the fastest algorithms available for KSPR, by a wide margin, and (*iii*) all of the algorithms are easy to implement. Our fastest algorithms for NSPR and KSPR have exponential worst-case complexity, but exponential behavior may only arise only with pathological data. The polynomial-time versions of these algorithms are fast enough to back these algorithms up if such behavior should ever arise.

7 References

Ahuja, R.K., Magnanti, T.L. and Orlin, J.B., (1993), *Network Flows*, Prentice Hall, Englewood Cliffs, New Jersey.

Bellman, R., (1958), "On a routing problem," *Quarterly of Applied Mathematics*, **16**, pp. 87–90.

Byers, T.H. and Waterman, M.S., (1984), "Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming," *Operations Research*, **32**, pp. 1381–1384.

Cherkassky, B.V., Goldberg, A.V., and Radzik, T., (1994), "Shortest path algorithms: theory and experimental evaluation," *Proceedings of the Fifth Annual ACM-SIAM*

Symposium on Discrete Algorithms, ACM Press, New York, New York, pp. 516-525.

Dijkstra, E., (1959), "A note on two problems in connexion with graphs," *Numerische Mathematik*, **1**, pp. 269–271.

Eppstein, D., (1998), "Finding the K Shortest Paths," *SIAM Journal on Computing*, **28**, pp. 652–673.

Fredman, M. L., and Tarjan, R. E., (1984), "Fibonacci heaps and their uses in improved network optimization algorithms," *Proceedings, 25th Annual IEEE Symposium on Foundations of Computer Science*, pp. 338–346.

Hadjiconstantinou, E. and Christofides, N., (1999), "An efficient implementation of an algorithm for finding K shortest simple paths," *Networks*, **34**, pp. 88–101.

Israeli, E. and Wood, R.K., (2002), "Shortest-path network interdiction," *Networks*, **40**, pp. 97–111.

Johnson, D. S., (1977), "Efficient shortest path algorithms," *Journal of the ACM*, **24**, pp. 1–13.

Naor, D. and Brutlag, D.L., (1994), "On near-optimal alignments of biological sequences," *Journal of Computational Biology*, **1**(4), pp. 349–366.

Pape, U., (1974), "Implementation and efficiency of Moore-algorithms for the shortest route problem," *Mathematical Programming*, **7**, pp. 212-222.

Wevley, C. (1999), "The Quickest Path Network Interdiction Problem," Masters Thesis, Operations Research Department, Naval Postgraduate School, Monterey, California, March.