Near-Shortest and K-Shortest Simple Paths

W. Matthew Carlyle, R. Kevin Wood
Department of Operations Research, Naval Postgraduate School, Monterey, California 93943

We present a new algorithm for enumerating all near-shortest simple (loopless) *s-t* paths in a graph $G = (V, E)$ with nonnegative 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 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, that is several orders of magnitude faster yet on those test problems. A minor variant on this algorithm also solves "KSPU," which is analogous to KSPR but with loops allowed. **NETWORKS, Vol. 46(2), 98–109 2005**

Keywords: K-shortest paths; near-shortest paths; path enumeration

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 [8] and Hadjiconstantinou and Christofides [11] provide excellent reviews, and Eppstein maintains an online bibliography at http://liinwww.ira.uka.de/bibliography/Theory/k-path.html. This article 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 [5]. 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 focus on enumerating simple (loopless) paths, and use "NSPR" and "KSPR" to denote NSP and KSP, respectively, when restricted to enumerating only simple paths. For clarity, "NSPU" and "KSPU" will denote the respective unrestricted problems.

KSPR may be more difficult than KSPU [11], but our applications and thus interests lie with simple paths (e.g., [14, 17]). 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 [5] employ dynamic-programming ideas to solve NSPU, using 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 enumeration 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 $O(S(n, m))$ represents 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*)

Received February 2003; accepted May 2005
Correspondence to: W.M. Carlyle; e-mail: mcarlyle@nps.edu
DOI 10.1002/net.20077
Published online in Wiley InterScience (www.interscience.wiley.com).

it is difficult to implement, requiring complicated operations to join paths together; however, it does require only $O(n^2)$ work per path enumerated [11].

To overcome the difficulties listed 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** only by 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 extremely 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 presents computational results for implementations of the basic algorithms and their variants. We also describe modifications of one algorithm to solve KSPU and present brief computational results. Section 6 provides conclusions.

2. SOLVING THE NEAR-SHORTEST-PATHS PROBLEM

Let $G = (V, E)$ denote a directed graph 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 nonsimple) 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 [5] give a simple algorithm for solving NSP, which can be summarized as follows:

1. For all $v \in V$, find $d'(v)$, the shortest-path length from v to t. 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 $L(u) + c(u, v) + d'(v) \leq (1 + \epsilon)L_{\min}$, 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 ever 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 [5]
DESCRIPTION: An algorithm to solve NSPU.
INPUT: A directed graph $G = (V, E)$ in adjacency list format, $\mathcal{T}$, 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.

```
{
  /* A single shortest-path calculation evaluates all d'(v) in
     the next step. */
  for( all v ∈ V )
     { d'(v) ← shortest-path distance from v to t; }
  d̄ ← (1 + ε)d'(s);
  for( all v ∈ V and τ = 1, ..., T )
     { nextEdge(v, t) ← firstEdge(v); }
  theStack ← s; L(s) ← 0;
  /* τ(v) denotes the number of times vertex v appears on
     the current subpath. */
  /* It is not used in the basic version of this algorithm. */
  τ(s) ← 1; for( all v ∈ V − s ){ τ(v) ← 0; }
  while( theStack is not empty ){
     u ← vertex at the top of theStack;
     if( nextEdge(u, τ(v)) ≠ ∅ ) {
        (u, v) ← the edge pointed to by nextEdge(u);
        increment nextEdge(u, τ(u));
        if( L(u) + c(u, v) + d'(v) ≤ d̄ ){   /* Step (i) */
           if( v = t ){
              print( theStack ∪ t );
           } else {
              push v on theStack;
              τ(v) ← τ(v)+1;
              L(v) ← L(u) + c(u, v);
              Dummy Step (ii);
           }
        }
     } else {
        Pop u from theStack;
        τ(u) ← τ(u)−1;
        nextEdge(u, τ(u)) ← 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," that is, on the s-u path currently represented by the stack, do not allow

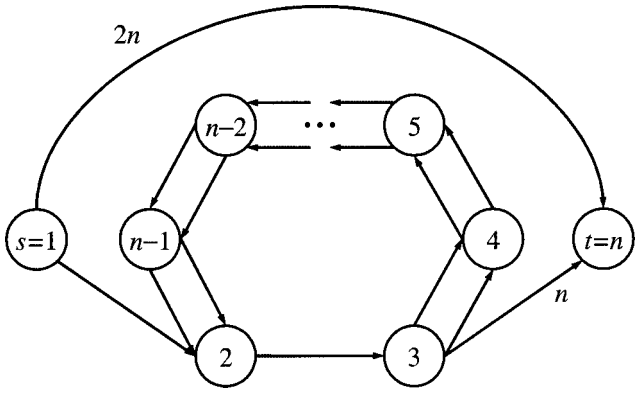

FIG. 1. A graph that demonstrates **ANSPR0**'s exponential worst-case complexity. Suppose $\epsilon = 1.0, c_e = 1$ for all unmarked edges, but $c(n-1,t) = 2n$ and $c(3,t) = n$, as marked. Then, there is only one near-shortest path $(s \to 2 \to 3 \to t)$, but the algorithm will investigate all 2^{n-3}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

it to go onto that subpath again. This can be accomplished by replacing the if-statement at Step (i) with

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

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 a shortest path from v to t, but every 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 little effort seems to be wasted in "going down blind alleys." We investigate this version of the algorithm in more detail later.

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 in which no vertices v' with $\tau(v') = 1$ are 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, which represent the current subpath.

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 the resulting subpath is, or can be extended to, a simple s-t path whose 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 $d'(v) \le \bar{d} - L(u) - c(u,v)$. 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, no feasible completion of the path can result if v were added to the stack when $d'(v) > \bar{d} - L(u) - c(u,v)$.

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. ∎

For a detailed description of what $O(S(n,m))$ can be, we refer the reader to the extensive coverage of shortest paths in ([1], pp. 93–165). For graphs with nonnegative edge lengths, Dijkstra's "label-setting algorithm" is the classic approach [7]. 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 [15] 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 $O(m + n \log n)$ [10], and, theoretically, to $O(m)$ for dense graphs [15].

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 [4]. Both polynomial-time and exponential-time versions of label-correcting algorithms can be very fast in practice [6].

3. PRACTICAL IMPROVEMENTS IN EFFICIENCY

Here we investigate four modifications of **ANSPR1**, denoted A, B, C, and D, for improving the basic algorithm's

practical efficiency. The first three modifications maintain the polynomial complexity of the algorithm; modification D does not, however, except under certain conditions.

3.1. Modification A

ANSPR1 never extends a path to 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. Also, distance labels cannot decrease as the algorithm extends that subpath. Consequently, we can simplify the shortest-path calculations by never updating a vertex label that exceed the current "cut-off" of $(1+\epsilon)L_{\min} - L_P$. 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 update the n $d'()$ values as usual, push them onto a stack, and require the algorithm to use the n topmost values in its 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, [1], pp. 106–107.)

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 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 $d'(u)$ are nondecreasing as a subpath is extended, and therefore none of these vertices u can ever join the current subpath. In fact, because edge lengths are nonnegative, 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, that is, 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 shortest-path algorithm could be arranged to accomplish this task. However, the complexity of **ANSPR1** would increase substantially and new data structures would be necessitated. We wish to keep **ANSPR1** simple, 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 worst-case complexity. However, if the algorithm recomputes those path lengths only after it (*i*) extends the current path by a fixed number of edges ℓ to some vertex v, or (*ii*) it backtracks from v, the work saved may offset the extra work incurred by following paths in error. Rather than recomputing the path lengths when backtracking from v, however, we maintain valid distance labels by using the distance-label stack of Modification B, described above.

If the maximum degree in G is bounded by a constant d, then the maximum number of edges the modified algorithm can follow in error is bounded by ℓd^{ℓ}, which may not be too large. Of course, we only claim that 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. $ABCD_\ell$ 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 KSPR using an algorithm for NSPR as a subroutine.

4. SOLVING THE K-SHORTEST-PATHS PROBLEM

Our approach to solving KSPR solves NSPR, using **ANSPR1** as a subroutine, within a binary search on the value of ϵ. To simplify the presentiation here, we modify the notation slightly: because 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 $\delta = \lfloor \epsilon L_{\min} \rfloor$. Consequently, 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 Nonnegative 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 suffices for our purposes.) Because $\kappa(\delta)$ is nondecreasing, 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 $\epsilon = \delta''/L_{\min}$, add any path with (exact) length $L_{\min} + \delta''$ to $\mathcal{P}$, and halt the enumeration when $|\mathcal{P}| = K$;
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, then the total work involved in the binary search becomes $O(KnS(n, m)(\log n + \log c_{\max}))$; the work performed in steps 1 and 2 is therefore dominated. Because 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 above extend trivially to undirected graphs with nonnegative edge lengths. Simply replace each undirected edge with two directed, antiparallel 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. [1], pp. 136–144). Of course, modification A does not apply 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 KSPR (as well as KSPU) in such graphs. Because the initial shortest-path computation of $d'(v)$ need not be repeated at each iteration, the overall complexity of the algorithm will be $O(Km(\log n + \log c_{\max}))$. The multiplicative term m appears 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 [16]. This algorithm is typically very fast, but does have exponential worst-case complexity, and its run times can vary widely across different problem classes [6]. However, we like this algorithm's ease of implementation, and it behaves adequately for this article's test problems.

Initial 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 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, our computer requires about 100 seconds to write $K = 10^6$ paths from **AKSPR1** 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 integers drawn independently from the discrete uniform distribution on [1,10]. This graph has $n = 1002$ and $m = 3920$. Cherkassky et al. [6] use similar graphs for some of their tests on shortest-path algorithms.

TABLE 1. Run times, in CPU seconds, for variants of **ANSPR1** solving NSPR.

Graph	δ	Paths (no.)	Run Times for Variants of **ANSPR1**						
			Basic (seconds)	A (seconds)	AB (seconds)	ABC (seconds)	$ABCD_{10}$ (seconds)	$ABCD_{50}$ (seconds)	D_∞ (seconds)
Grid	0	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40×25	1	16	0.1	0.0	0.0	0.0	0.0	0.0	0.0
	2	56	0.1	0.1	0.0	0.0	0.0	0.0	0.0
	3	139	0.3	0.1	0.1	0.0	0.0	0.0	0.0
	4	334	0.6	0.2	0.1	0.1	0.0	0.0	0.0
	5	770	1.3	0.5	0.3	0.1	0.0	0.0	0.0
	6	1,633	2.5	1.0	0.5	0.2	0.1	0.0	0.0
Grid	0	6	0.2	0.1	0.1	0.0	0.0	0.0	0.0
100×50	1	44	0.9	0.4	0.2	0.1	0.0	0.0	0.0
	2	218	3.7	1.2	0.8	0.4	0.1	0.0	0.0
	3	894	12.8	3.9	2.5	1.2	0.2	0.1	0.0
	4	3,210	39.8	11.1	7.1	3.8	0.7	0.2	0.0
	5	10,320	113.5	30.0	19.3	10.8	1.7	0.4	0.0
	6	30,632	303.5	76.4	48.9	29.4	4.3	0.9	0.1
Road 1	0	1	0.1	0.0	0.0	0.0	0.0	0.0	0.0
	10	21	0.4	0.1	0.1	0.0	0.0	0.0	0.0
	20	42	0.5	0.1	0.1	0.0	0.0	0.0	0.0
	30	98	1.4	0.3	0.2	0.1	0.0	0.0	0.0
	40	210	2.9	0.6	0.4	0.2	0.0	0.0	0.0
	50	554	8.0	1.8	1.1	0.4	0.1	0.0	0.0
	60	1,229	16.0	3.3	2.0	0.9	0.2	0.0	0.0
Road 2	0	1	32.3	1.3	0.8	0.6	0.3	0.2	0.0
	10	22	176.6	4.7	2.6	1.9	0.4	0.2	0.0
	20	65	347.7	7.9	4.2	3.7	0.6	0.2	0.0
	30	179		18.3	9.5	9.2	1.2	0.3	0.0
	40	484		42.6	21.9	22.4	2.5	0.6	0.0
	50	1,126		88.2	44.8	47.3	4.9	0.8	0.0
	60	2,437		180.8	92.6	97.9	9.8	1.7	0.2

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. (Computations performed on a 2.5-GHz laptop computer.)

- "Grid 100×50" has the same basic structure as Grid 40×25, but uses a 100×50 graph of vertices. This graph has $n = 5002$ 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 $n = 3670$ and $m = 9876$.
- "Road 2" depicts the same 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 [11] test their algorithm for KSPR has $n = 1000$ 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_∞"). This is essentially **ANSPR0**.

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

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.

TABLE 2. A computational example demonstrating **ANSPR0**'s exponential worst-case complexity.

n	δ	D_∞ (seconds)
20	22	0.0
25	27	0.3
26	28	0.6
27	29	1.1
28	30	2.2
29	31	4.4
30	32	8.8

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 $ABCD_{10}$ variant of **ANSPR1** provides an good alternative algorithm, however, as all its run times are indistinguishable from zero on these problems. (Computations performed on a 2.5-GHz laptop computer.)

Road 2 has a different general structure than the others tested in that it contains many degree-2 vertices. (This network's level of resolution eliminates many small streets that intersect large streets, but it does not eliminate the vertices that represent the corresponding intersections.) This may account for the difference here. However, the difference in run times is insignificant in these cases, and it seems likely that we can use the ABC variant with reasonable confidence in most applications.

ANSPR0, labeled "D_∞," 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 solve each of the instances in Table 2 in time that registers as 0.00 seconds. Thus, 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**" because 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 solve 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 $\kappa(\delta') \leq K$ and $\kappa(\delta'') > K$). Empty entries in the table represent problems that cannot be solved in 1000 seconds by the given algorithm.

Hadjiconstantinou and Christofides ([11], Fig. 6c) indicate a run time for their KSPR algorithm of over 1400 seconds when $K = 10^3$ for a test graph having 1000 vertices and an edge-to-vertex density of 4. This graph is of roughly the same size and structure as 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 time functions in the standard C library. 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.

The next two subsections provide additional computational tests, further investigating the potential exponential behavior of **ANSPR0**, and looking at different class of topologies.

5.4. On the Potential Exponential Behavior of ANSPR0

This subsection further explores the computational behavior **ANSPR0**, and implicitly, **AKSPR0**. **ANSPR0** is attractive because of its simplicity and empirical speed, but its worst-case exponential complexity might be of concern to a user. We investigate this issue by evaluating solution statistics for different edge-length functions, and different values of δ, on "Grid 100×50" described in Section 5.1. A 3-GHz Pentium IV computer carries out the computations in this subsection and the next, but RAM size, operating system and software remain the same.

The example of Figure 1 leads one to believe that a high variance in edge lengths might induce exponential behavior, that is, cause searches to follow "blind alleys" and backtrack excessively from "deadends" because $d'(v)$ is never updated. Following this idea, we randomly set each edge's length to 0 with probability q_0, and let q_0 vary between 0.00 and 0.30. The key statistic we evaluate is the average number of backtracks per enumerated path, divided by the average number of edges, or "hops," in a path. We normalize by hops because even if the algorithm hits no deadends, the amount of backtracking must increase as a function of the number of edges in the paths it enumerates. (We could evaluate deadends directly, but that would ignore the extra work that arises when multiple backtracks follow the discovery of a deadend.) We also focus on another statistic, the "longest backtrack sequence" (LBTS); this specifies the maximum number of backtracks observed between identification of two consecutive paths. This value must become large if the algorithm hits large numbers of deadends. Table 4 displays results for these experiments.

Table 4 shows only modest changes in the two main statistics except for some potentially significant increases when $(\delta, q_0) = (5, 0.30), (7, 0.20)$, and $(9, 0.10)$. However, the average time spent enumerating each path does not increase commensurately, so we conclude that **ANSPR0** can remain empirically efficient in grid graphs even as edge-length

TABLE 3. Run times, in CPU seconds, for variants of **AKSPR1** solving KSPR.

			Run Times for Variants of **AKSPR1**						
Graph	K	δ''	Basic (seconds)	A (seconds)	AB (seconds)	ABC (seconds)	$ABCD_{10}$ (seconds)	$ABCD_{50}$ (seconds)	D_∞ (seconds)
Grid	10^2	3	0.8	0.4	0.2	0.1	0.0	0.0	0.0
40×25	10^3	6	6.4	2.5	1.4	0.6	0.1	0.1	0.0
	10^4	9	67.6	23.9	13.2	5.7	1.3	0.5	0.1
	10^5	13		148.8	79.7	38.4	7.6	3.7	0.4
	10^6	17				430.6	80.5	42.0	4.0
	10^7	21							34.0
Grid	10^2	2	5.3	1.9	1.2	0.5	0.1	0.0	0.0
100×50	10^3	4	46.4	13.7	8.7	4.4	0.8	0.2	0.0
	10^4	5		132.5	84.8	49.0	7.5	1.7	0.1
	10^5	8			462.1	284.5	41.5	8.3	0.6
	10^6	10						90.7	5.8
	10^7	13							47.7
Road 1	10^2	31	9.8	2.2	1.4	0.6	0.1	0.0	0.0
	10^3	57	99.8	20.1	12.2	5.4	1.1	0.2	0.0
	10^4	90		153.4	94.1	48.6	8.2	1.6	0.1
	10^5	128			685.1	379.6	59.1	11.3	0.5
	10^6	171						132.3	10.8
	10^7	219							94.1
Road 2	10^2	25	3,723.5	82.4	46.1	41.5	7.0	3.2	2.8
	10^3	49			315.1	323.6	35.3	7.4	3.2
	10^4	81			2861.6	2971.9	294.0	51.9	3.7
	10^5	120						424.2	4.7
	10^6	167							16.9
	10^7	222							111.2

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. (Computations performed on a 2.5-GHz laptop computer.)

variance and δ increase. The next section examines the efficiency of **ANSPR0** in a network-routing application, and further investigates the potential for exponential behavior. (We shall also obtain indirect evidence that the largest increase observed above in "longest backtrack sequence," from 144 to 330, is, indeed, insignificant.)

5.5. *Hop-Limited Paths*

A hop in a communication network corresponds to a signal, or packet, traversing a single link along a source-destination path. Hops should be limited to reduce transmission delay and network congestion. Consequently, a number of authors have employed "hop-limited paths" in their procedures for solving capacity-expansion problems in communication networks (e.g., Herzberg et al. [12], Iraschko et al. [13], Yurcik et al. [18]). (Actually, some authors do not limit the number of hops between an origin-destination pair, per se, but rather the number of hops exceeding the minimum, a concept analogous to our δ.) The methods developed in the cited papers all require explicit generation of hop-limited paths, for which **ANSPR0** certainly applies, so this section investigates the empirical efficiency of **ANSPR0** for that purpose. Perhaps **ANSPR0** will prove to be a useful tool in this arena.

For testing, we employ randomly generated, scale-free, undirected graphs [3] as surrogates for communication networks. Although these scale-free graphs must lack some of the structure found in real communication networks [2], they match certain characteristics well, such as the distribution of vertex degrees, and, importantly, the distribution of hop distances between vertices [9]. We note that Herzberg et al. [12] analyze realistically dimensioned test networks with $n = 100$ and $m = 200$, and with a hop limit that corresponds to $\delta \leq 6$. These parameter values fall within the boundaries of the smallest test problems we investigate here.

We follow Barábasi and Albert in generating these graphs: for given integer parameters a and n, we create a complete graph on a vertices, and then sequentially add $n - a$ vertices to the graph, connecting each new vertex to the previous ones with a edges. The edges are connected to the previously generated vertices randomly, but with a "preference" for vertices with higher degree. In particular, a new vertex u is connected to existing vertex $v \in V'$ with a probability $k_v / \sum_{v' \in V'} k_{v'}$, where k_v denotes the degree of vertex v. For simplicity, we use $a = 3$ in all tests. (A similar pattern of results appears when using other reasonable parameters values $a = 2$ and $a = 4$, or when specified fractions of the vertices are created with $a = 2$, $a = 3$, and $a = 4$.)

Table 5 displays results for three different graph sizes, and two choices of the source-sink pair. We select two "extreme" source-sink pairs, $(s, t) = (1, n)$, and $(s, t) = (n - 1, n)$, i.e., the first and last vertices created while generating the graph,

TABLE 4. Statistics for **ANSPR0** when enumerating paths in Grid 100 × 50 as q_0 and δ vary.

		Solution Statistics for **ANSPR0**					
δ	q_0	Avg. paths (no.)	Avg. hops (no.)	Normalized backtracks (no.)	LBTS (no.)	Avg. run time (seconds)	Avg. time per path (microseconds)
1	0.00	214	132.9	0.31	144	0.1	*
	0.10	550	143.3	0.36	160	0.1	*
	0.20	1502	164.9	0.28	191	0.1	*
	0.30	138,391	190.5	0.16	267	0.4	*
3	0.00	6019	133.2	0.22	143	0.1	*
	0.10	15,240	144.3	0.25	160	0.1	*
	0.20	76,691	165.3	0.19	198	0.3	*
	0.30	10,838,944	157.4	0.28	299	18.4	2.6
5	0.00	80,373	133.4	0.18	144	0.2	*
	0.10	211,249	145.0	0.19	162	0.4	*
	0.20	1,770,896	165.8	0.16	206	3.4	2.5
	0.30	361,783,736	130.9	11.68	330	572.1	1.9
7	0.00	749,961	133.6	0.16	145	1.3	3.2
	0.10	2,059,141	145.7	0.16	164	3.1	2.3
	0.20	26,712,288	137.1	0.93	206	44.8	1.9
9	0.00	5,627,716	133.8	0.14	146	7.8	1.8
	0.10	36,012,890	120.5	0.93	183	54.3	2.0

Legend:

q_0	Fraction of edge lengths set to 0.
Normalized backtracks:	Avg. backtracks per path enumerated, normalized by avg. path hops.
LBTS:	Longest backtrack sequence between two consecutive paths, observed across all ten problem instances.
avg. time per path:	Avg. microseconds to enumerate a path; to avoid bias from computational overhead, this is computed only if run time exceeds 0.5 seconds.

Nominal edge lengths are uniformly distributed integers on [1, 10]. Each row in the table represents 10 randomly generated instances. Normalized backtracks and longest backtrack sequences remain reasonably small, indicating that the algorithm does not spend much time backtracking from dead ends. Potentially significant increases appear for $(\delta, q_0) = (5, 0.30)$, $(7, 0.20)$, and $(9, 0.10)$, but the average time required to enumerate a path in those cases does not increase. We conclude that **ANSPR0** exhibits good empirical efficiency for this grid topology, over these parameter values. (Computations performed on a 3-GHz laptop computer.)

or the last two. Each row corresponds to 10 different randomly generated instances having the same parameter values. For both (s, t) options, and for differing values of δ, we list the minimum and maximum number of paths identified, and the minimum and maximum run times observed. The run times for each graph seem to be proportional to the number of paths generated, and, on average, the algorithm requires only 0.5 to 2.1 microseconds to generate each path. Thus, **ANSPR0** appears to be an excellent algorithm for enumerating hop-limited paths in communication networks.

The scale-free graphs present another opportunity to test the potentially exponential behavior of **ANSPR0** as edge-length variance increases. For these tests, the nominal edge lengths remains 1, but each edge has its length set to 0, independently, with probabiilty q_0. The value of q_0 ranges from 0.00 to 0.25 while δ ranges from 0 to 5. The number of near-shortest paths explodes in these graphs as q_0 and δ increase because no vertex is many hops away from any other [9], and, presumably, because of the "coarseness" in edge lengths. Consequently, we run tests only on a small graph with $n = 100$. Also, for simplicity, $s = 99$ and $t = 100$ in all instances.

Table 6 lists results for these tests. Each row in the table derives from ten randomly generated instances. The statistics do show that as δ and q_0 increase, **ANSPR0** can traverse huge numbers of edges to reach deadend after deadend. In one case, the number of backtracks recorded between the identification of two consecutive paths ("max LBTS") exceeds 1 million. It seems that the exponential worst-case complexity of the algorithm may be making itself known. However, the average amount of work to generate a path does not increase greatly, so encountering large backtrack sequences between consecutive paths is a rare event. Further evidence of this rarity, not shown in the table, comes from the instances with $\delta = 5$ and $q_0 = 0.20$. There, the instance with over 1 million for "max LBTS" exhibits a total value for normalized backtracks of 3.2, while the instance that generates the "min LBTS" value of 992 exhibits only a modestly lower value for this statistic, 1.4. (The smallest value among the 10 instances is 0.9.)

In summary, we see that not recomputing $d'(v)$ after extending or retracting a path can cause **ANSPR0** to execute huge sequences of "unnecessary" backtracks. Such events are rare, however, and do not greatly affect the run time of the algorithm in these tests. The algorithm remains empirically efficient for the scale-free topology over the range of parameter values tested. If excessive backtracking does become a concern, $d'(v)$ can be recomputed periodically as described by "Modification D" in Section 3.4. However,

TABLE 5. Statistics for **ANSPR0** when enumerating hop-limited paths in scale-free graphs.

		Solution Statistics for **ANSPR0**							
		$s = 1, t = n$				$s = n - 1, t = n$			
Graph	δ	Min paths (no.)	Max paths (no.)	Min time (sec.)	Max time (sec.)	Min paths (no.)	Max paths (no.)	Min time (sec.)	Max time (sec.)
$n = 100$	0	1	11	0.0	0.0	1	5	0.0	0.0
$m = 588$	1	3	87	0.0	0.0	2	51	0.0	0.0
	2	31	524	0.0	0.0	8	336	0.0	0.0
	3	201	3042	0.0	0.0	59	2380	0.0	0.0
	4	1317	17,103	0.0	0.0	467	16,185	0.0	0.0
	5	7988	92,180	0.0	0.1	3353	103,900	0.0	0.0
	6	47,386	532,799	0.0	0.3	22,492	642,940	0.0	0.3
	7	266,566	3,063,571	0.1	1.5	142,472	3,815,302	0.1	1.8
	8	1,433,451	16,832,065	0.6	8.5	868,070	21,727,107	0.4	10.3
	9	7,400,343	88,504,052	3.2	44.9	5,066,299	118,863,848	2.6	57.2
$n = 1000$	0	1	4	0.0	0.0	1	10	0.0	0.0
$m = 5988$	1	2	44	0.0	0.0	1	76	0.0	0.0
	2	19	578	0.0	0.0	9	990	0.0	0.0
	3	212	5326	0.0	0.0	82	10,017	0.0	0.0
	4	2086	52,455	0.0	0.1	769	99,455	0.0	0.1
	5	20,795	495,350	0.0	0.5	7,580	966,532	0.0	1.0
	6	201,230	4,559,950	0.3	4.5	74,024	9,199,468	0.1	9.5
	7	1,926,665	41,032,044	2.0	39.6	713,747	86,158,514	0.7	96.3
$n = 10{,}000$	0	1	12	0.0	0.0	1	12	0.0	0.0
$m = 59{,}988$	1	4	207	0.0	0.0	3	163	0.0	0.0
	2	35	2890	0.0	0.0	61	2500	0.0	0.0
	3	555	39,721	0.0	0.1	905	36,088	0.0	0.1
	4	8064	553,387	0.0	1.3	12,582	500,639	0.0	1.3
	5	109,859	7,584,949	0.3	17.6	170,211	6,860,350	0.4	16.0
	6	1,493,710	102,849,075	3.4	238.3	2,296,49	92,734,583	5.2	197.3

Each row corresponds to 10 randomly generated instances. "Min (max) paths" indicates the minimum (maximum) number paths among the 10 instances, and "min (max) time" indicates the minimum (maximum) run time in CPU seconds. It turns out that, in all cases, the best run time corresponds to min paths and the worst corresponds to max paths. Run times for each graph are all roughly proportional to the number paths generated—this is the best one could hope for with a true enumeration procedure—and, at most 2.1 microseconds are required to generate any path, on average. (Computations performed on a 3-GHz laptop computer.)

the rarity of long backtrack sequences suggests another approach. Recompute $d'(v)$ whenever a counter for the current backtrack sequence reaches an empirically determined threshold.

5.6. *K-Shortest Paths with Loops Allowed*

Our final computational tests investigate the enumeration of paths with loops. 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 residing within **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 7, which compares the two algorithms on the two largest test problems, Grid 100×50 and Road 2. (We return to the 2.5-GHz personal computer for these computations.)

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 article, and we make no computational comparisons with alternative algorithms.

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

TABLE 6. Statistics for **ANSPR0** applied to a scale-free graph as q_0 and δ vary.

		Solution Statistics for **ANSPR0**						
δ	q_0	Avg. paths (no.)	Avg. hops (no.)	Normalized backtracks (no.)	Max LBTS (no.)	Min LBTS (no.)	Avg. run time (seconds)	Avg. time per path (microseconds)
1	0.00	14	3.2	1.0	6	2	0.1	*
	0.05	29	3.8	1.1	11	2	0.1	*
	0.10	91	5.3	1.1	15	2	0.1	*
	0.15	205	8.6	1.5	232	13	0.1	*
	0.20	13,156	10.6	2.8	2624	13	0.1	*
	0.25	15,861	13.4	3.2	14,540	2	0.1	*
3	0.00	692	5.4	0.6	29	4	0.1	*
	0.05	2484	6.5	0.7	43	7	0.1	*
	0.10	13,318	9.1	0.8	284	19	0.1	*
	0.15	45,928	12.1	1.0	1424	73	0.2	*
	0.20	413,464	14.6	1.8	10,661	136	0.9	4.6
	0.25	6,866,719	18.0	2.1	326,982	46	7.4	4.8
5	0.00	28,247	7.4	0.5	53	16	0.1	*
	0.05	141,769	8.9	0.6	325	19	0.2	*
	0.10	1,045,342	11.9	0.6	3006	33	1.0	1.6
	0.15	5,190,267	14.5	0.9	107,534	317	7.6	1.6
	0.20	54,907,762	16.2	1.5	1,088,798	992	95.2	3.0
	0.25	—	—	—	—	—	—	—

Legend:

q_0	Fraction of edge lengths set to 0.
Normalized backtracks:	Avg. backtracks per path identified, normalized by avg. hops.
Max longest bt sequence:	Maximum value of LBTS across all 10 instances.
Min longest bt sequence:	Minimum value of LBTS across all 10 instances.
Avg. time per path:	Avg. microseconds per enumerated path, computed only if average run time exceeds 0.5 seconds.

Nominal edge lengths are 1, and $n = 100$, $s = 99$, and $t = 100$ for all problem instances. Each row corresponds to 10 randomly generated instances, except "-" indicates the total run time exceeded 1 hour and the run was terminated. "Max LBTS" does increase dramatically, indicating that large sequences of deadends can arise because $d'(v)$ is not updated. Also, the large differences between that value and "min LBTS" show great variance in the algorithm's behavior depending on the data. Normalized backtracks and "average time per path" remain small, however, so large values for backtrack sequences must be relatively rare events. We conclude that **ANSPR0** remains empirically efficient for this scale-free graph as edge-length variances increases, at least when generating 50 million or fewer paths. (Computations performed on a 3-GHz laptop computer.)

TABLE 7. Run times for **AKSPU0** solving KSPU (K shortest paths with loops allowed), compared to run times for **AKSPR0** solving KSPR.

Graph	K	δ''	**AKSPR0** (seconds)	δ''	**AKSPU0** (seconds)
Grid	10^2	3	0.0	2	0.0
100×50	10^3	5	0.0	4	0.0
	10^4	8	0.1	6	0.1
	10^5	11	0.4	9	0.6
	10^6	15	3.1	11	5.2
	10^7	18	40.0	14	41.4
Road 2	10^2	24	3.2	11	2.8
	10^3	49	3.7	17	3.2
	10^4	81	4.3	25	3.5
	10^5	120	5.8	33	7.4
	10^6	167	24.0	43	31.8
	10^7	222	167.7	53	256.1

Note that **AKSPR0** is the D_∞ variant of **AKSPR1** whose times are also presented in Table 3. (Computations performed on a 2.5-GHz laptop computer.)

enumerated is $O(nS(n,m))$, where $S(n,m)$ corresponds to the worst-case complexity of the user-selected shortest-path subroutine. We describe the basic algorithms for directed graphs with nonnegative edge lengths, but it 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. We also combine **ANSPR1** with a binary search, in an algorithm denoted **AKSPR1**, to solve the K-shortest-paths 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 across a wide range of problems. **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, that algorithm wastes little effort in practice.

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 arise only with pathological data. Certain polynomial-time versions of these algorithms are fast enough to use as backups should such data be encountered.

REFERENCES

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, Network flows, Prentice Hall, Englewood Cliffs, NJ.

[2] D. Alderson, J. Doyle, R. Govindan, and W. Willinger, Toward an optimization-driven framework for designing and generating realistic internet topologies, ACM SIGCOMM Comput Commun Rev 33 (2003), 41–46.

[3] A.-L. Barabási and R. Albert, Emergence of scaling in random networks, Science 286 (1999), 509–512.

[4] R. Bellman, On a routing problem, Q Appl Math 16 (1958), 87–90.

[5] T.H. Byers and M.S. Waterman, Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming, Operat Res 32 (1984), 1381–1384.

[6] B.V. Cherkassky, A.V. Goldberg, and T. Radzik, "Shortest path algorithms: Theory and experimental evaluation," Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, Arlington, Virgina, 23–25 January, 1994, pp. 516–525.

[7] E. Dijkstra, A note on two problems in connexion with graphs, Num Math 1 (1959), 269–271.

[8] D. Eppstein, Finding the *K* shortest paths, SIAM J Comput 28 (1998), 652–673.

[9] M. Faloutsos, P. Faloutsos, and C. Faloutsos, On power-law relationships of the Internet topology, ACM SIGCOMM Comput Commun Rev 29 (1999), 251–262.

[10] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J ACM 34 (1987), 338–346.

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

[12] M. Herzberg, S.J. Bye, and A. Utano, The hop-limit approach for spare-capacity assignment in survivable networks, IEEE/ACM Trans Network 3 (1995), 775–784.

[13] R.R. Iraschko, M.H. MacGregor, and W.D. Grover, Optimal capacity placement for path restoration in STM or ATM mesh-survivable networks, IEEE/ACM Trans Network 6 (1998), 325–336.

[14] E. Israeli and R.K. Wood, Shortest-path network interdiction, Networks 40 (2002), 97–111.

[15] D.S. Johnson, Efficient shortest path algorithms, J ACM 24 (1977), 1–13.

[16] U. Pape, Implementation and efficiency of Moore-algorithms for the shortest route problem, Math Program 7 (1974), 212–222.

[17] C. Wevley, The quickest path network interdiction problem, Masters Thesis, Operations Research Department, Naval Postgraduate School, Monterey, California, 1999.

[18] W. Yurcik, D. Tipper, and D. Medhi, "The use of hop-limits to provide survivable ATM group communications," Proceedings of NGC 2000 on Networked Group Communication, Palo Alto, California, 8–10 November, 2000, pp. 131–140.