Finding the k Shortest Simple Paths: A New Algorithm and its Implementation

John Hershberger* Matthew Maxel† Subhash Suri‡

Abstract

We describe a new algorithm to enumerate the k shortest simple (loopless) paths in a directed graph and report on its implementation. Our algorithm is based on a *replacement paths* algorithm proposed recently by Hershberger and Suri [7], and can yield a factor $\Theta(n)$ improvement for this problem. But there is a caveat: the fast replacement paths subroutine is known to fail for some directed graphs. However, the failure is easily detected, and so our k shortest paths algorithm optimistically uses the fast subroutine, then switches to a slower but correct algorithm if a failure is detected. Thus the algorithm achieves its $\Theta(n)$ speed advantage only when the optimism is justified. Our empirical results show that the replacement paths failure is a rare phenomenon, and the new algorithm outperforms the current best algorithms; the improvement can be substantial in large graphs. For instance, on GIS map data with about 5000 nodes and 12000 edges, our algorithm is 4-8 times faster. In synthetic graphs modeling wireless ad hoc networks, our algorithm is about 20 times faster.

1 Introduction

The k *shortest paths problem* is a natural and long-studied generalization of the shortest path problem, in which not one but several paths in increasing order of length are sought. Given a directed graph G with non-negative edge weights, a positive integer k, and two vertices s and t, the problem asks for the k shortest paths from s to t in increasing order of length. We require that the paths be *simple* (loop free). See Figure 1 for an example illustrating the difference between the k shortest paths problem with and without the simplicity constraint. (As the figure shows, even in graphs with non-negative weights, although *the* shortest path is always simple, the subsequent paths can have cycles.) The k shortest paths problem in which paths are not required to be simple turns out to be significantly easier. An $O(m + kn \log n)$ time algorithm for this problem has been known since 1975 [3]; a recent improvement by Eppstein essentially achieves the optimal time of $O(m + n \log n + k)$—the algorithm computes an implicit representation of the paths, from which each path can be output in $O(n)$ additional time [2].

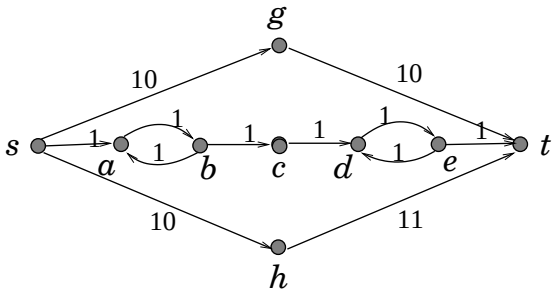

Figure 1: The difference between simple and non-simple k shortest paths. The three *simple* shortest paths have lengths $6, 20$ and 21, respectively. Without the simplicity constraint, paths may use the cycles (a, b, a) and (d, e, d), giving shortest paths of lengths $6, 8, 10$.

The problem of determining the k shortest *simple* paths has proved to be more challenging. The problem was originally examined by Hoffman and Pavley [10], but nearly all early attempts to solve it led to exponential time algorithms [16]. The best result known to date is an algorithm by Yen [18, 19] (generalized by Lawler [12]), which using modern data structures can be implemented in $O(kn(m+n \log n))$ worst-case time. This algorithm essentially performs $O(n)$

*Mentor Graphics Corp, 8005 SW Boeckman Road, Wilsonville, OR 97070, john_hershberger@mentor.com.

†Department of Computer Science, University of California, Santa Barbara, CA 93106.

‡Department of Computer Science, University of California, Santa Barbara, CA 93106, suri@cs.ucsb.edu. Supported in part by National Science Foundation grants CCR-9901958 and IIS-0121562.

single-source shortest path computations for each output path. In the case of *undirected* graphs, Katoh, Ibaraki, and Mine [11] improve Yen's algorithm to $O(k(m+n\log n))$ time. While Yen's asymptotic worst-case bound for enumerating k simple shortest paths in a directed graph remains unbeaten, several heuristic improvements to his algorithm have been proposed and implemented, as have other algorithms with the same worst-case bound. See for example [1, 6, 13, 14, 15].

In this paper we propose a new algorithm to find the k shortest simple paths in a directed graph. Our algorithm is based on the efficient *replacement paths* algorithm of Hershberger and Suri [7], which gives a $\Theta(n)$ speedup over the naïve algorithm for replacement paths. The efficient algorithm is known to fail for some directed graphs [8]. However, the failure is easy to detect, and so our k shortest paths algorithm optimistically tries the fast replacement paths subroutine, then switches to a slower but correct algorithm if a failure occurs.

We have implemented our optimism-based k shortest paths algorithm, and our empirical results show that the replacement paths algorithm almost always succeeds—in our experiments, the replacement paths subroutine failed less than 1% of the time. Thus, we obtain a speedup over Yen's algorithm that is close to the best-case linear speedup predicted by theory. The algorithm is never much worse than Yen's, and on graphs where shortest paths have many edges, such as those derived from GIS data or locally-connected radio networks, the improvement is substantial. For instance, on GIS data representing road networks in the United States, our algorithm is about 20% faster for paths from Washington, D.C., to New York (relatively close cities), and about 8 *times* faster for paths from San Diego to Piercy, CA. Similarly, on synthetic models of wireless networks, our algorithm is 10 to 20 times faster. Random graphs tend to have small diameters, but even on those graphs, our algorithm usually outperforms Yen's algorithm.

The fact that Yen's worst-case bound for directed graphs remains unbeaten after 30 years raises the possibility that no better algorithm exists. Indeed, Hershberger, Suri, and Bhosle recently have shown that in a slightly restricted version of the path comparison model, the replacement paths problem in a directed graph has complexity $\Omega(m\sqrt{n})$, if $m = O(n\sqrt{n})$ [9]. (Most known shortest path algorithms, including those by Dijkstra, Bellman-Ford and Floyd-Warshall, satisfy this model. The exceptions to this model are the matrix multiplication based algorithms by Fredman and others [4, 17, 20].) The same lower bound construction shows that any k simple shortest paths algorithm that finds the best candidate path for each possible branch point off previously chosen paths is also subject to this lower bound, even for $k = 2$. All known algorithms for the k simple shortest paths fall into this category.

2 Path Branching and Equivalence Classes

In the k shortest paths problem we are given a directed graph $G = (V, E)$, with n vertices and m edges. Each edge $e \in E$ has an associated non-negative weight $c(e)$. A path in G is a sequence of edges, with the head of each edge connected to the tail of its successor at a common vertex. A path is *simple* if all its vertices are distinct. The total weight of a path in G is the sum of the weights of the edges on the path. The *shortest path* between two vertices s and t, denoted by $path(s,t)$, is the path joining s to t, assuming one exists, that has minimum weight. The weight of $path(s,t)$ is denoted by $d(s,t)$.

We begin with an informal description of the algorithm. Our algorithm generates k shortest paths in order of increasing length. Suppose we have generated the first i shortest paths, namely, the set $\Pi_i = \{P_1, P_2, \ldots, P_i\}$. Let R_i denote the set of remaining paths that join s to t; this is the set of candidate paths for the remaining $k-i$ shortest paths. In order to find the next shortest path in R_i efficiently, we partition this set into $O(i)$ equivalence classes. These classes are intimately related to the branching structure of the first i paths, which we call the *path branching structure* $\mathcal{T}_i$. This is a rooted tree that compactly encodes how the first i paths branch off from each other topologically, and it can be defined procedurally as follows. (N.B. The procedural definition is not part of our algorithm; it is just a convenient way to describe the path branching structure.)

The subroutine to construct $\mathcal{T}_i$ is called with parameters (s, Π_i), where s is the fixed source and $\Pi_i = \{P_1, P_2, \ldots, P_i\}$ is the set of the first i shortest paths. We initialize $\mathcal{T}_i$ to the singleton root node s. Let $(s, a, b, \ldots, u)$ be the longest subpath that is a *common prefix* of all the paths in Π_i. We expand $\mathcal{T}_i$ by adding a node labeled u, and creating the branch (s, u). The node u is the child of s. The set of paths $\{P_1, P_2, \ldots, P_i\}$ is defined to be the *path bundle* of (s, u) and denoted by $B(s, u)$.[1] We now partition the path bundle $B(s, u)$ into sets $S_1, S_2, \ldots, S_\alpha$ such that

[1] Strictly speaking, the bundle notation should have the subscript i to indicate that it refers to a branch in tree $\mathcal{T}_i$. We drop this subscript to keep the notation simple, since the choice of branching structure will always be clear from the context.

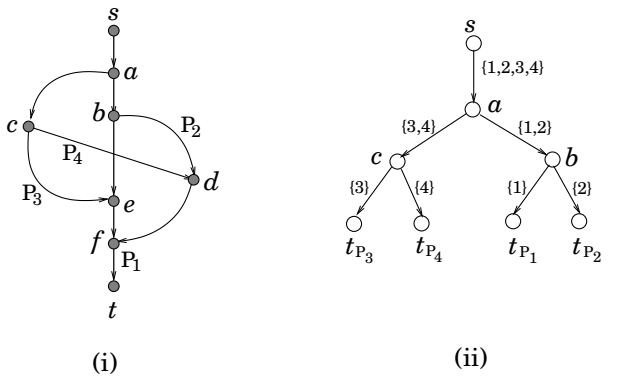

Figure 2: (i) Four shortest paths, $P_1, \ldots, P_4$. (ii) The associated path branching structure.

all paths in S_j follow the same edge after u, and paths in different groups, S_j, S_l, follow distinct edges. We then make recursive calls (u, S_j), for $j = 1, 2, \ldots, \alpha$, to complete the construction of $\mathcal{T}_i$. The recursion stops when $|S_j| = 1$, i.e., S_j contains a single path P. In this case we create a leaf node labeled t_P, representing the target vertex parameterized by the path that reaches it, and associate the bundle $\{P\}$ with the branch (u, t_P). See Figure 2 for an example of a path branching structure.

Remark: It is possible that the paths in the bundle $B(s, u)$ have no overlap, meaning $u = s$. In this case, the branch (s, u) is not created, and we go directly to the recursive calls.

For any given branch (u, v), $B(u, v)$ is exactly the set of paths that terminate at leaf descendants of v. The bundles are not maintained explicitly; rather, they are encoded in the path branching structure. Each branch (u, v) has a path in G associated with it. This path, denoted *branchPath*(u, v), is shared by all the paths in $B(u, v)$. Its endpoints are the vertices of G corresponding to nodes u and v. The first edge of *branchPath*(u, v) is the *lead edge* of the path, denoted *lead*(u, v). A node u in $\mathcal{T}_i$ has a *prefix path* associated with it, denoted *prefixPath*(u). The prefix path is the concatenation of the branch paths for all the branches from s to u in $\mathcal{T}_i$. Thus a shortest path P is equal to *prefixPath*(t_P).

It is important to note the distinction between nodes in $\mathcal{T}_i$ and their corresponding vertices in G. A single vertex $u \in V$ may have up to k corresponding nodes in $\mathcal{T}_i$, depending on how many different paths from s reach it. For example, vertex t has one node in $\mathcal{T}_i$ for each of the k shortest paths, which is why we distinguish the t nodes by their prefix paths (e.g., t_P). Likewise, a pair of vertices u, v may correspond to multiple branches, each with its own branch path. The following lemma is straightforward.

LEMMA 2.1. *The path branching structure $\mathcal{T}_i$ has $O(i)$ nodes and exactly i leaves.*

The preceding lemma would apply to a branching structure built from any i simple paths from s to t—it does not depend on the paths of Π_i being shortest paths. The following lemma characterizes the branches of $\mathcal{T}_i$ more precisely, and it *does* depend on the shortness of the paths.

LEMMA 2.2. *Let (u, v) be a branch in $\mathcal{T}_i$. Let P be the shortest path from vertex u to vertex t in G that starts with lead(u, v) and is vertex-disjoint from prefixPath(u). Then branchPath$(u, v) \subseteq P$.*

Proof. Recall that the ith path P_i is the shortest path that is not an element of Π_{i-1}. Therefore, P_i must branch off from each path in Π_{i-1}. Let x be the vertex of P_i farthest from s where P_i branches off from a path in Π_{i-1}. Since $\mathcal{T}_{i-1}$ contains exactly the paths in Π_{i-1}, P_i branches off from some *branchPath*$(\cdot, \cdot)$ of $\mathcal{T}_{i-1}$ at x. Let e be the edge on P_i that follows x. Because P_i is the shortest simple path not in Π_{i-1}, it must *a fortiori* be the shortest simple path that branches off from $\mathcal{T}_{i-1}$ at x and follows e. The simplicity requirement means that the part of P_i after x must be disjoint from all the vertices on the prefix path in $\mathcal{T}_{i-1}$ from s to x. However, the only other requirement on the part of P_i after e is that it be as short as possible; this implies that the part of the path after x contains no loops. More generally, for every edge e' on P_i after x, the subpath optimality property of shortest paths implies that the part of P_i after e' is the shortest path from the head vertex of e' to t that avoids the vertices of the prefix of P_i up through the tail vertex of e'.

Any branch (u, v) in $\mathcal{T}_i$ is a subset of a branch (x, t_{P_j}) that was created as a branch off $\mathcal{T}_{j-1}$ for some $j \leq i$. (Branches are not moved by later branches, only subdivided.) The edge *lead*(u, v) belongs to *branchPath*(x, t_{P_j}), and hence as noted in the preceding paragraph, the part of P_j after u, which contains *branchPath*(u, v), is the shortest path that starts with *lead*(u, v) and is vertex-disjoint from *prefixPath*(u). ∎

We associate the equivalence classes of candidate paths with the nodes and branches of $\mathcal{T}_i$, as follows. Consider a non-leaf node $u \in \mathcal{T}_i$, and suppose that the

children of u are labeled $v_1, v_2, \ldots, v_\alpha$. We associate one equivalence class with each branch out of u, denoted $C(u, v_j)$, and one with node u itself, denoted $C(u)$. The class $C(u, v_j)$ consists of those paths of R_i that overlap with the paths in *prefixPath*(v_j) *up to and including* the lead edge *lead*(u, v_j), but this overlap does not extend to v_j. That is, each path of $C(u, v_j)$ *diverges* from *prefixPath*(v_j) somewhere strictly between u and v_j. The final set $C(u)$ consists of those paths that overlap with each *prefixPath*(v_j) up to u, but no farther. That is, these paths branch off at u, using an edge that is distinct from any of the lead edges *lead*(u, v_j), for $j = 1, 2, \ldots, \alpha$. The equivalence partition associated with the path branching structure $\mathcal{T}_i$ is the collection of these sets over all branches and non-leaf nodes of $\mathcal{T}_i$.

For instance, consider node a in Figure 2. There are three equivalence classes associated with a: class $C(a, c)$ includes those paths that share the subpath from s to a with P_3, P_4, and branch off somewhere strictly between a and c (assume that the subpath from a to c contains more than one edge). Similarly, the class $C(a, b)$ contains paths that coincide with P_1, P_2 until a, then branch off before b. Finally, the class $C(a)$ contains paths that coincide with $P_1, \ldots, P_4$ up to a, then branch off at a.

LEMMA 2.3. *Every path from s to t that is not among the i shortest paths belongs to one of the equivalence classes associated with the nodes and branches of $\mathcal{T}_i$. The number of equivalence classes associated with the path branching structure $\mathcal{T}_i$ is $O(i)$.*

Proof. Consider a path P different from the first i shortest paths. Suppose P_j, where $1 \leq j \leq i$, is the path that shares the longest common prefix subpath with P. In $\mathcal{T}_i$, let (u, v) be the first branch where P and P_j diverge. Then P belongs in the equivalence class associated with either u or (u, v). Finally, the total number of equivalence classes is $O(i)$ because each node and branch of $\mathcal{T}_i$ has only one equivalence class associated with it, and there are $O(i)$ nodes and branches in $\mathcal{T}_i$. ∎

We are now ready to describe our algorithm for enumerating the k shortest simple paths.

3 Computing the k Shortest Paths

We maintain a heap, which records the minimum path length from each of the $O(i)$ equivalence classes. Clearly, the next shortest path from s to t is the smallest of these $O(i)$ heap entries. Once this path is chosen (and deleted from the heap), the path branching structure is modified, and so is the equivalence class partition. Computationally, this involves refining one equivalence class into at most four classes. For each class, we determine the minimum element, and then insert it into the heap. See Figure 3.

ALGORITHM k-SHORTESTPATHS

- Initialize the path branching structure $\mathcal{T}$ to contain a single node s, and put *path*(s, t) in the heap. There is one equivalence class $C(s)$ initially, which corresponds to all the s–t paths.
- Repeat the following steps k times.
  1. Extract the minimum key from the heap. The key corresponds to some path P.
  2. If P belongs to an equivalence class $C(u)$ for some node u then
     (a) Add a new branch (u, t_P) to $\mathcal{T}$ that represents the suffix of P after u.
     (b) Remove from $C(u)$ the paths that share at least one edge with P after u and put all of them except P into the newly created equivalence class $C(u, t_P)$.
  3. Else (P belongs to the equivalence class $C(u, v)$ for some branch (u, v))
     (a) Let w be the vertex where P branches off from *branchPath*(u, v).
     (b) Insert a new node labeled w, and split the branch (u, v) into two new branches (u, w) and (w, v). Add a second branch (w, t_P) that represents the suffix of P after w.
     (c) Redistribute the paths of $C(u, v) \setminus P$ among the four new equivalence classes $C(u, w)$, $C(w, v)$, $C(w, t_P)$, and $C(w)$, depending on where they branch from *branchPath*(u, v) and/or P.
        i. Paths branching off *branchPath*(u, v) before node w belong to $C(u, w)$.
        ii. Paths branching off *branchPath*(w, v) after node w belong to $C(w, v)$.
        iii. Paths branching off P after node w belong to $C(w, t_P)$.
        iv. Paths branching off both P and *branchPath*(u, v) at node w belong to $C(w)$.

4. For each new or changed equivalence class (at most four), compute the shortest path from s to t that belongs to the class. Insert these paths into the heap.

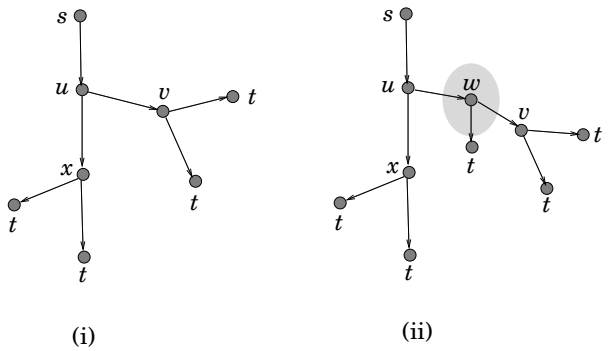

Figure 3: Illustration of how the equivalence class partition and branching structure change with the addition of a new path. The left figure shows the structure before adding the new path. There is an equivalence class for each non-leaf node and branch of the tree. The right figure shows the portion of the structure that is affected if the newly added path came from the class belonging to branch (u, v). The classes corresponding to the node and three branches that are inside the shaded oval are modified.

LEMMA 3.1. *Algorithm k-ShortestPaths correctly computes the ith shortest path, the branching structure $\mathcal{T}_i$, and the equivalence class partition of the candidate paths R_i, for each i from 1 to k.*

The complexity of the algorithm described above is dominated by Step 4. Step 1 takes only $O(\log k)$ time per iteration of the repeat loop, and Steps 2 and 3 take $O(n)$ time for path manipulation. The redistribution of candidate paths among equivalence classes is conceptual—the hard work is computing the *minimum* element in each class in Step 4. In the following section, we discuss how to implement this step efficiently.

Remark: Our algorithm is conceptually similar to those of Yen and Lawler. The main difference is that our algorithm partitions the candidate paths into equivalence classes determined by the path branching structure, and those algorithms do not. This additional structure together with the fast replacement paths subroutine (Section 4) is the key to our algorithm's efficiency.

4 The Replacement Paths Problem

The key step in our k shortest paths algorithm is the computation of the best path in each equivalence class, which can be formulated as a replacement paths problem. Let $H = (V, E)$ be a directed graph with non-negative edge weights, and let x, y be two specified nodes. Let $P = \{v_1, v_2, \ldots, v_m\}$, where $v_1 = x$, and $v_m = y$, be the shortest path from x to y in H. We want to compute the shortest path from x to y that *does not include* the edge (v_i, v_{i+1}), for each $i \in \{1, 2, \ldots, m-1\}$. We call this the *best replacement path for* (v_i, v_{i+1}); the reference to the source x and target y is implicit.

A naïve algorithm would require $m-1$ invocations of the single-source shortest path computation: run the shortest path algorithm in graph H_{-i}, where H_{-i} is the graph H with edge (v_i, v_{i+1}) deleted. The following algorithm does batch computation to determine all the replacement paths in $O(|E| + |V| \log |V|)$ time; as mentioned earlier, it can fail for some directed graphs, but the failure can easily be detected. This algorithm is a slight simplification of the one in [7]. For full details of the algorithm's data structures, refer to that paper.

ALGORITHM REPLACEMENT

1. In the graph H, let X be a shortest path tree from the source x to all the remaining nodes, and let Y be a shortest path tree from all the nodes to the target y. Observe that P, the shortest path from x to y, belongs to both X and Y.
2. For every edge $e_i = (v_i, v_{i+1}) \in P$
   (a) Let $X_i = X \setminus e_i$. Let E_i be the set of all edges $(a, b) \in E \setminus e_i$ such that a and b are in different components of X_i, with a in the same component as x.
   (b) For every edge $(a, b) \in E_i$
   Let $pathWeight(a, b) = d(x, a) + c(a, b) + d(b, y)$. Observe that $d(x, a)$ and $d(b, y)$ can be computed in constant time from X and Y.
   (c) The replacement distance for e_i is the minimum of $pathWeight(a, b)$ over all $(a, b) \in E_i$.

The quantity $pathWeight(a, b)$ is the total weight of the concatenation of $path(x, a)$, (a, b), and $path(b, y)$. By sweeping over the edges of P from one end of P to the other while maintaining a priority queue on the

edges of E_i, with $pathWeight(e)$ as the key of each edge $e \in E_i$, the entire algorithm takes the same asymptotic time as one shortest path computation.

Let us now consider how this algorithm may fail in some directed graphs. It is clear that $pathWeight(e)$ is the length of the shortest path from x to y that uses edge $e = (a, b) \in E_i$, and hence the algorithm finds the shortest path that uses an edge of E_i. However, this path may not be the path we seek, because the suffix $path(b, t)$ *may traverse* e_i. A simple example of this pathological behavior is shown in Figure 4.

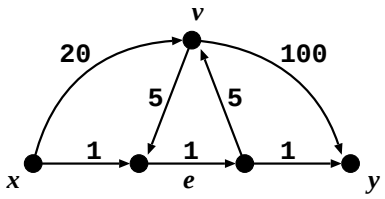

Figure 4: A directed graph for which the replacement paths algorithm fails. The shortest path from v to y goes through the edge e, which causes our algorithm to compute an incorrect path for the replacement edge candidate (x, v). The correct replacement path for e uses the *second* shortest path from v to y, which does not go through e.

Define $low(v)$ to be the smallest i such that $path(v, y)$ contains vertex v_i. (In the original paper [7], $low(v)$ is replaced by an equivalent but more complicated quantity called $minblock(v)$, for reasons specific to that paper.) We say that $(a, b) \in E_i$ is the *min-yielding cut edge for* e_i if (a, b) has the minimum $pathWeight()$ over all cut edges in E_i. We say that (a, b) is *valid* if $low(b) > i$. The following theorem identifies when the replacement paths algorithm may fail.

THEOREM 4.1. *The algorithm* REPLACEMENT *correctly computes the replacement path for* e_i *if the min-yielding cut edge for* e_i *is valid.*

In undirected graphs, all min-yielding cut edges are valid. In directed graphs, exceptions can arise. However, an exception is easily detected—the $low()$ labels for all the vertices can be computed by a preorder traversal of the shortest path tree Y, and so we certify each min-yielding cut edge in constant time. When an exception is detected, our algorithm falls back to the slower method of running separate shortest path algorithms for each failing e_i.

5 The Shortest Path in an Equivalence Class

We describe briefly how the replacement path subroutine is used to compute the shortest path in an equivalence class. Consider the four equivalence classes created in step (3), in which P branches off from $branchPath(u, v)$ at a vertex w.

First consider a branch's equivalence class. Let (a, c) be a branch in $\mathcal{T}$, and choose b such that $lead(a, c) = (a, b)$. The paths in $C(a, c)$ follow $prefixPath(c)$ up through b, then branch off strictly before c. Thus it suffices to find the shortest *suffix* starting at b, ending at t, subject to the constraints that the suffix (1) is vertex-disjoint from $prefixPath(a)$ and (2) branches off $branchPath(a, c)$ before c. We determine this path using the replacement path problem in a subgraph H of G, defined by deleting from G all the vertices on $prefixPath(a)$, including a.

The shortest path in the node's equivalence class $C(w)$ is easier to find: We obtain a graph H by deleting from G all the vertices in $prefixPath(w)$ except w, plus all the lead edges that leave from w. We compute the shortest path from w to t in H, then append it to $prefixPath(w)$.

If the next shortest path P belongs to a node equivalence class $C(u)$ (step (2) of Algorithm k-ShortestPaths), then $C(u)$ is modified and a new equivalence class $C(u, t_P)$ is created. We can find the shortest paths in $C(u, t_P)$ and $C(u)$ as above. (In the latter case, we simply remove one more edge $lead(u, t_P)$ from H and recompute the shortest path from u to t.)

Thus, the overall complexity of the k shortest paths algorithm is dominated by $O(k)$ invocations of the replacement paths subroutine. In the optimistic case, this takes $O(m + n \log n)$ time per invocation; in the pessimistic case, it takes $O(n(m + n \log n))$ time per invocation.

6 Implementation and Empirical Results

6.1 Implementation

We have implemented our algorithm using Microsoft Visual C++ 6.0, running on a 1.5 Ghz Pentium IV machine. The implementation follows the pseudo-code in Section 3 and the more detailed algorithm description of the replacement paths algorithm in [7]. We list a few of the notable features of the implementation here:

1. The Fibonacci heap data structure is used in both Dijkstra's shortest path algorithm and our replace-

ment paths subroutine. Fibonacci heaps therefore contribute to the performance of both our k shortest paths algorithm and Yen's algorithm. We implemented the Fibonacci heap from scratch, based on Fredman and Tarjan's paper [5].

2. Our graph data structure is designed to reduce memory allocation of small structures, since measurements showed it to be a significant cost. The chief components of the graph data structure are an array of nodes, an array of edges, and scratch space for use in creating subgraphs. We store two arrays of pointers to edges, one sorted by source node and one sorted by destination node. Each node gets access to its incident edges by pointing to the appropriate subsequences in these arrays.

   A primary operation for the k shortest paths algorithm is producing subgraphs efficiently. Since memory allocation/deallocation is relatively expensive and most of the information in a subgraph is the same as that in the parent graph, a subgraph borrows structures from the parent graph, uses these structures to compute some path, and then returns them to the parent graph. Because a subgraph generally has nodes or edges eliminated, we maintain a separate array of edges as scratch space in the parent graph for use by the subgraph.

3. Our program sometimes chooses to use a naïve algorithm instead of the replacement paths algorithm of Section 4. The naïve algorithm deletes each shortest path edge in turn, and finds the shortest path from the source to the sink in this new subgraph. Because the replacement paths algorithm calculates two shortest path trees and also performs a priority queue operation for every graph edge, we estimated that each naïve shortest path computation should take about 1/3 of the time of the whole replacement paths algorithm. Therefore, our k shortest paths implementation is a hybrid: it chooses whichever replacement paths subroutine is likely to be faster, using a threshold of 3 for the branch path length.

   Subsequent measurement suggested that a threshold closer to 5 might be more appropriate. See Figure 5—the crossover point between the two algorithms appears to be around five. Future work will explore this more fully.

6.2 Experiments

We compared our new algorithm with an implementation of Yen's k shortest paths algorithm. We implemented Yen's algorithm ourselves, using state of the art data structures (Fibonacci heap) and optimized memory management. Our test suite had three different kinds of experimental data: real GIS data for road networks in the United States, synthetic graphs modeling wireless networks, and random graphs.

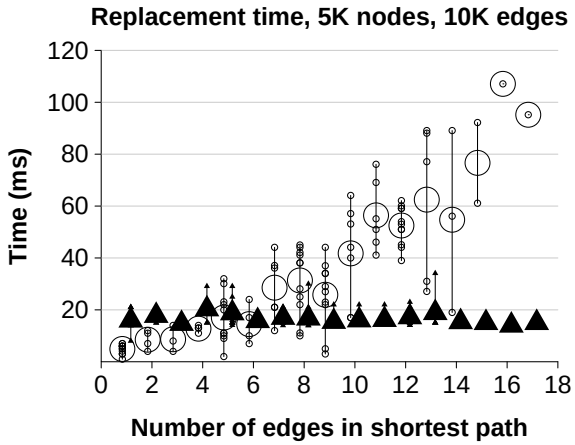

Figure 5: Time to compute replacement paths by the naïve algorithm (circles) and our algorithm (triangles). The small glyphs show the raw data; the large ones show the average time value for each shortest path edge count. Note the large variance in the runtime of the naïve algorithm, and the essentially constant runtime of our algorithm. There are equal numbers of small circles and triangles for each x-value; the low y-variance of the triangles means some small triangles are hidden by the large ones.

GIS Road Networks. We obtained data on major roads from the Defense Mapping Agency. These graphs represent the major roads in a latitude/longitude rectangle. The edge weights in these graphs are road lengths. The first graph contains the road system in the state of California, and the second contains the road system in the northeastern part of the U.S.

The experiments show that in the California graph, for 250 shortest paths from San Diego to Piercy, our algorithm is about 8 times faster than Yen's. For a closer source–destination pair (Los Angeles, San Francisco), the speedup factor is about 4. Finally, when the source and destination are fairly close (Washington, D.C., and New York), the relative speed advantage is about 20%. Figure 6 summarizes the results of these experiments.

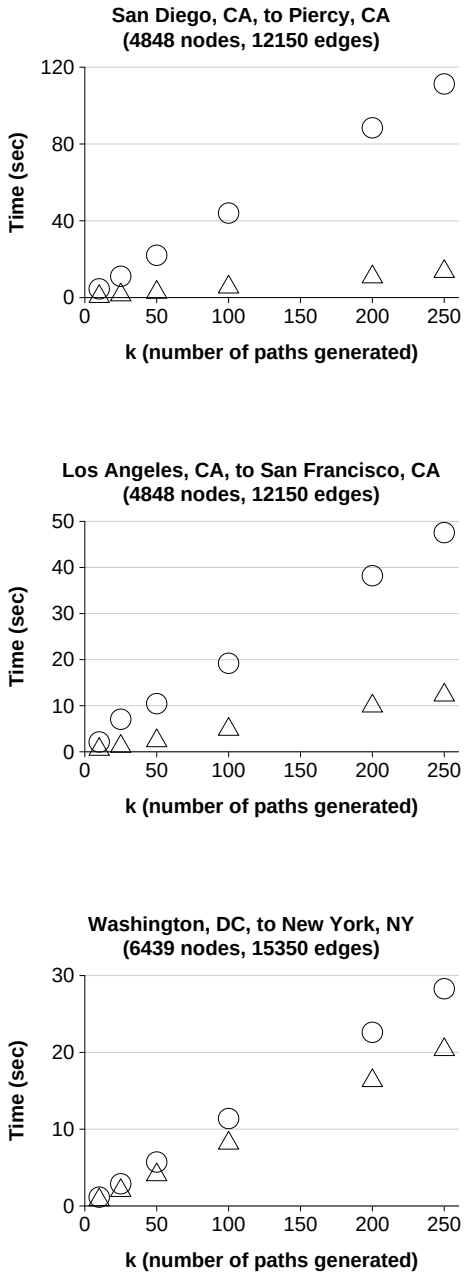

Figure 6: Time to compute k shortest paths on GIS graphs. Circles represent Yen's algorithm; triangles represent our algorithm.

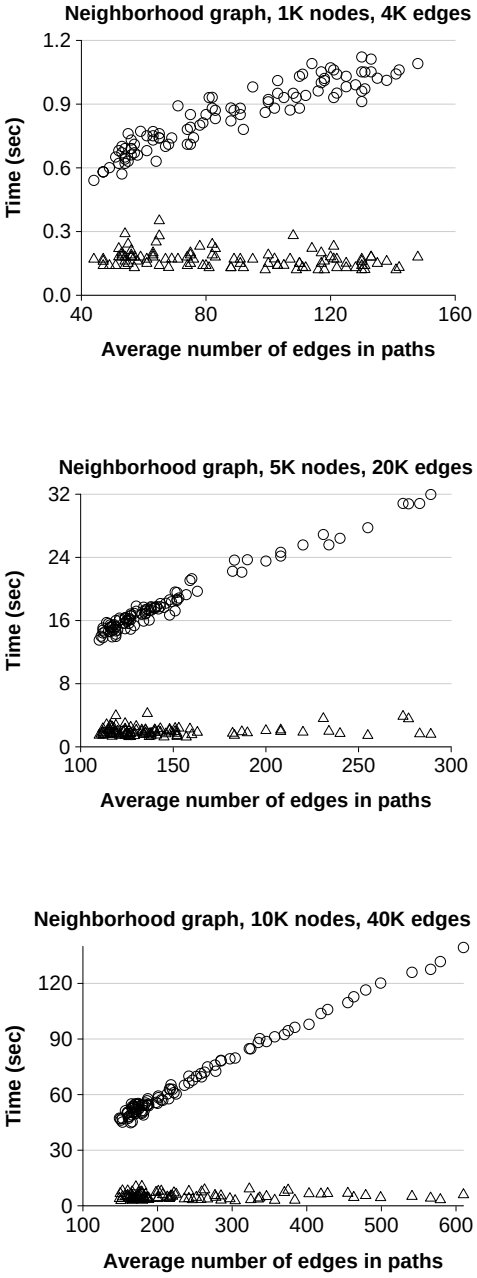

Figure 7: Time to compute 100 shortest paths on neighborhood graphs, plotted versus the average number of edges in all the paths. Each plot summarizes 100 trials on graphs with the same (n, m), but with the grid rectangle's aspect ratio varied to control the average shortest path length. Circles represent Yen's algorithm, triangles our algorithm. The charts for $m = 8n$ are similar to those for $m = 4n$, and are omitted to save space.

Geometric Neighborhood Graphs. We generated synthetic models of ad hoc wireless networks, by considering nodes in a rectangular grid. The aspect ratio of the rectangle was varied to create graphs of varying diameter. The source and target were chosen to be opposite corners of the grid. In each case, we considered two models of edges: in one case, all 8 neighbors were present, and their distances were chosen uniformly at random in $[0, 1000]$; in the second case, 4 of the 8 neighbors were randomly selected.

Our experiments, summarized in Figure 7, show that our new algorithm is faster by a factor of at least 4. In the large graphs (10000 nodes, 40000 edges), the speedup is around *twentyfold.*

Random Graphs. The new algorithm achieves its theoretical potential most fully when the average shortest path has many edges. This is clear from the experiments on the GIS and neighborhood data. Random graphs, on the other hand, tend to have small diameter. (In particular, a random graph on n nodes has expected diameter $O(\log n)$.) These graphs, therefore, are not good models for systems with long average paths. Even in these graphs, our new algorithm does better than Yen's in most cases, although the speed advantage is not substantial, as expected.

Each random graph is generated by selecting edges at random until the desired number of edges is generated. Edge weights are chosen uniformly at random in $[0, 1000]$. We tried three random graph classes: (1K nodes, 10K edges), (5K nodes, 20K edges), and (10K nodes, 25K edges). We plot the time needed to generate 100 shortest paths between a random (s, t) pair, against the *average number of edges* in the 100 paths. See Figure 8.

6.3 Discussion

We can characterize our results according to the following broad generalizations.

Average Number of Edges in Shortest Paths. The efficiency of the new algorithm derives mainly from performing batch computations when finding the best path in an equivalence class. The relative gain is proportional to the number of edges in the branch path where the replacement paths subroutine is applied. Thus, if the replacement paths subroutine works without failover, our algorithm is likely to deliver a speedup that grows *linearly with the average number of edges* in the k

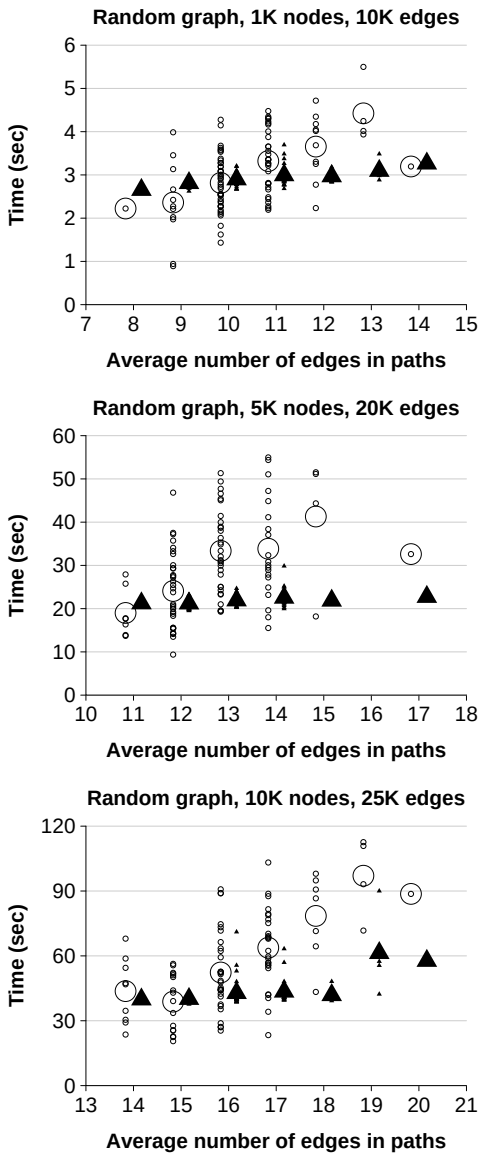

Figure 8: Time to compute 100 shortest paths on random graphs, plotted versus the average number of edges in all the paths. Each plot summarizes 100 random (s, t) trials in random graphs with the same (n, m) parameters. Circles represent Yen's algorithm, triangles our algorithm. The x-coordinate of each glyph is the nearest integer to the average number of edges in all 100 paths. The small glyphs show the raw data; the large ones show the average time value for each shortest path edge count. Note the heavy concentration of the average path edge count around a mean value that increases with the graph size, probably logarithmically, and also note the large variance in the runtime of Yen's algorithm. There are equal numbers of small circles and triangles for each x-value; the low y-variance of the triangles means some small triangles are hidden by the large ones.

shortest paths. This advantage is minimized for random graphs, because the expected diameter of a random graph is very small. This is corroborated by the data in Figure 8. In geometric graphs, such as those obtained from GIS or ad hoc networks, shortest paths are much more likely to have many edges, and our algorithm has a corresponding advantage. This is borne out by Figures 6 and 7.

Replacement Path Failure. Our experiments show that the replacement paths algorithm rarely fails. When averaged over many thousands of runs, the replacement paths subroutine failed 1.2% of the time on random graphs, 0.5% on neighborhood graphs, and never on GIS graphs. Thus, in practice our k shortest paths algorithm shows an asymptotic speedup over Yen's algorithm. It also exhibits far more consistency in its runtime.

Contraindications. Yen's algorithm optimizes over the same set of candidate paths as our algorithm. If the average path length is small, our hybrid algorithm does essentially the same work as Yen's algorithm, running Dijkstra's algorithm repeatedly. In this case our algorithm is slightly less efficient than Yen's because of the extra bookkeeping needed to decide which subroutine to use, but the relative loss is only about 20% in speed.

In other cases, the replacement paths algorithm may be beaten by repeated Dijkstras even when the shortest path length is greater than three. This seems to occur most often in dense random graphs where Dijkstra's algorithm can find one shortest path without building the whole shortest path tree; the replacement paths algorithm, on the other hand, always builds two complete shortest path trees.

7 Concluding Remarks and Future Work

We have presented a new algorithm for enumerating the k shortest simple paths in a directed graph and reported on its empirical performance. The new algorithm is an interesting mix of traditional worst-case analysis and optimistic engineering design. Our theoretical advantage comes from a new subroutine that can perform batch computation in a specialized equivalence class of paths. However, this subroutine is known to fail for some directed graphs. Nevertheless, our experiments show that the strategy of using this fast subroutine optimistically and switching to a slower algorithm when it fails works very well in practice.

We are exploring several additional directions for further improvements in the algorithm's performance.

1. When should we switch to the naïve replacement paths algorithm? Is (path length ≤ 3) the right cutoff, or would a more sophisticated condition give better results?

   To help answer this question, we ran the k shortest paths algorithm on 140 different random and neighborhood graphs, measuring the runtime for each threshold value between 2 and 7. Figure 9 combines the results for all experiments. For each test case, we computed the minimum running time over the six threshold values. We then computed a normalized runtime for each of the threshold values by dividing the actual runtime by the minimum runtime. Figure 9 shows the average normalized runtime over all 140 test cases.

   Averaged normalized runtime emphasizes the importance of test cases for which the threshold matters. A test case for which the threshold choice makes little difference has little effect on the average normalized time, because all its normalized times will be near 1.0. A test case for which one threshold is clearly better will assign high normalized weights to the other thresholds, and hence will select against them strongly.

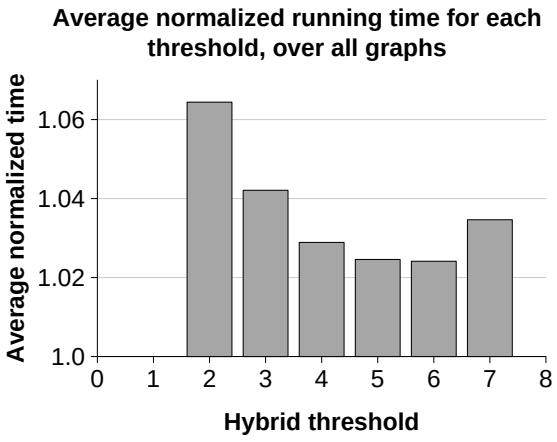

Figure 9: Average normalized runtime for all test cases.

   This chart suggests that 5 and 6 are the best thresholds. They should give about a 2% improvement in runtime over the threshold of 3 that we used in the experiments of Section 6.2. We

plan further measurements to confirm this expectation. Note that the chart also shows that no single threshold is ideal for all the test cases: the best thresholds (5 and 6) give a runtime 2% worse than would be obtained by an optimal threshold choice for each experiment.

2. We have discovered an improved version of the algorithm that makes only two calls to the replacement paths subroutine after each new path is discovered. Currently, our algorithm makes three calls to the subroutine, plus one Dijkstra call. This change should improve the running time of our algorithm by about 40%.

References

[1] A. Brander and M. Sinclair. A comparative study of K-shortest path algorithms. In *Proc. of 11th UK Performance Engineering Workshop*, pages 370–379, 1995.

[2] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[3] B. L. Fox. k-th shortest paths and applications to the probabilistic networks. In *ORSA/TIMS Joint National Mtg.*, volume 23, page B263, 1975.

[4] M. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5:83–89, 1976.

[5] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[6] E. Hadjiconstantinou and N. Christofides. An efficient implementation of an algorithm for finding K shortest simple paths. *Networks*, 34(2):88–101, September 1999.

[7] J. Hershberger and S. Suri. Vickrey prices and shortest paths: What is an edge worth? In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*, pages 252–259, 2001.

[8] J. Hershberger and S. Suri. Erratum to "Vickrey prices and shortest paths: What is an edge worth?". In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.

[9] J. Hershberger, S. Suri, and A. Bhosle. On the difficulty of some shortest path problems. In *Proc. 20th Sympos. Theoret. Aspects Comput. Sci.*, Lecture Notes Comput. Sci. Springer-Verlag, 2003.

[10] W. Hoffman and R. Pavley. A method for the solution of the Nth best path problem. *Journal of the ACM*, 6:506–514, 1959.

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

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

[13] E. Martins and M. Pascoal. A new implementation of Yen's ranking loopless paths algorithm. Submited for publication, Universidade de Coimbra, Portugal, 2000.

[14] E. Martins, M. Pascoal, and J. Santos. A new algorithm for ranking loopless paths. Technical report, Universidade de Coimbra, Portugal, 1997.

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

[16] M. Pollack. The kth best route through a network. *Operations Research*, 9:578, 1961.

[17] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

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

[19] J. Y. Yen. Another algorithm for finding the K shortest loopless network paths. In *Proc. of 41st Mtg. Operations Research Society of America*, volume 20, 1972.

[20] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.