FINDING THE HIDDEN PATH: TIME BOUNDS FOR ALL-PAIRS SHORTEST PATHS

DAVID R. KARGER*, DAPHNE KOLLER†, AND STEVEN J. PHILLIPS‡

Abstract. We investigate the all-pairs shortest paths problem in weighted graphs. We present an algorithm—the Hidden Paths Algorithm—that finds these paths in time $O(m^*n+n^2\log n)$, where m^* is the number of edges participating in shortest paths. Our algorithm is a practical substitute for Dijkstra's algorithm. We argue that m^* is likely to be small in practice, since $m^* = O(n\log n)$ with high probability for many probability distributions on edge weights. We also prove an $\Omega(mn)$ lower bound on the running time of any *path-comparison based* algorithm for the all-pairs shortest paths problem. Path-comparison based algorithms form a natural class containing the Hidden Paths Algorithm, as well as the algorithms of Dijkstra and Floyd. Lastly, we consider generalized forms of the shortest paths problem, and show that many of the standard shortest paths algorithms are effective in this more general setting.

1. Introduction.

Oh what a peaceful life is theirs
Who worldly strife and noise have flown
And follow the hidden path from cares
Which none but the wise of the world have known

Fray Luis de Leon, c. 1527 - 1591

Let G be a weighted directed graph with n vertices and m edges. The *all-pairs shortest paths problem* is to find a shortest path between each pair of vertices in G.

Our contributions to this problem lie in three areas. In Section 2 we present a new algorithm for all-pairs shortest paths, called the *Hidden Paths Algorithm*, that works on graphs with non-negative edge weights. Let an edge be called *optimal* if it is a shortest path, and let m^* be the number of optimal edges in the graph. The Hidden Paths Algorithm runs in time $O(m^*n + n^2\log n)$ if we use a Fibonacci heap [12] to implement a priority queue; the running time increases to $O(m^*n\log n)$ if a standard heap is used instead. The algorithm operates by running Dijkstra's single-source shortest paths algorithm [7] in parallel from all nodes in the graph, using information gained at one node to reduce the work done at other nodes. Our algorithm is practical and simple to implement. Is is also likely to be fast in practice, because it is known [14, 18, 27] that $m^* = O(n\log n)$ with high probability when the input graph is the complete graph with edge weights chosen independently from any of a large class of probability distributions, including the uniform distribution on the real interval $[0,1]$ or the uniform distribution on the range $\{1,\ldots,n^2\}$. The Hidden Paths Algorithm is also useful when the graph has (possibly negative) integer edge weights. Such a graph can be reweighted using an $o(mn)$ runtime scaling algorithm for single-source shortest paths (for example, Gabow and Tarjan [15], Goldberg [16]) before applying the Hidden Paths Algorithm. If the edge weights are polynomial sized integers, the combined running time of Goldberg's scaling algorithm and the Hidden Paths Algorithm is $O(m^*n + n^2\log n + \sqrt{n}m\log n)$.

Our second contribution is a new lower bound, given in Section 3. Many algorithms for the all-pairs shortest paths problem use the edge weight function only in comparing the weights of paths in the graph. We call such algorithms *path-comparison*

* Department of Computer Science, Stanford University, Stanford, California. Supported by a National Science Foundation Graduate Fellowship.

† Department of Computer Science, Stanford University, Stanford, California, and IBM Almaden Research Center, San Jose, California.

‡ Department of Computer Science, Stanford University, Stanford, California. Supported by NSF grants CCR-8858030-03 and CCR-9010517.

based, and prove that on directed graphs any path-comparison based algorithm requires $\Omega(mn)$ time in the worst case (in the worst case $m^* = m$). The idea of the lower bound is to construct a weighted directed graph with $\Theta(mn)$ directed paths. If an algorithm $\mathcal{A}$ fails to examine any path, we show how to modify the weight function so that the unexamined path is optimal, without $\mathcal{A}$ detecting the change. The directed graph we use is acyclic, so the lower bound holds even for this restricted class of directed graphs.

In proving the lower bound, we show that any path-comparison based algorithm requires $\Omega(mn)$ time even to verify the output of an all-pairs shortest paths algorithm. Thus, in the path-comparison model, verification is as hard as finding the paths. We also show a lower bound of $\Omega(n^3)$ for verifying that the edge weights of a graph satisfy the triangle inequality. These lower bounds also hold for randomized path-comparison based algorithms.

Finally, we investigate generalized shortest paths problems where the weight of a path is not necessarily the sum of the weights of its edges. For example, one might define the weight of a path to be the maximum of the weights of its edges. In Section 4 we show that many all-pairs shortest paths algorithms work even for generalized path weight functions, in their normal time bounds. We also extend our lower bound proof to show an $\Omega(mn)$ lower bound on the running time of any algorithm that solves the all-pairs shortest paths problem for certain generalized path weight functions.

Previous Work. The most widely known algorithms for the all-pairs shortest paths problem are those of Floyd [9] and Dijkstra [7]. Floyd's algorithm works by dynamic programming and runs in time $\Theta(n^3)$. On graphs with non-negative edge weights, Dijkstra's algorithm for the single-source shortest path problem can be run from each vertex (as noted by Johnson [21]), resulting in a running time of $\Theta(mn + n^2 \log n)$ if Fibonacci heaps [12] are used to implement priority queues. A variant of Dijkstra's algorithm developed by Spira [31] has an expected running time of $O(n^2 \log^2 n)$ if the edge weights are independently and identically distributed random variables. Bloniarz [3] provided an algorithm with an expected running time of $O(n^2 \log n \log^* n)$. Another algorithm, developed by Frieze and Grimmet [14], achieves an expected running time of $O(n^2 \log n)$, but is suitable only for random graphs. All of these algorithms are path-comparison based, so by the lower bound of Section 3 they have a worst case running time of $\Omega(mn)$. Fast algorithms exist for special cases of the all-pairs shortest paths problem, for instance when the graph is unweighted [8] or planar [10].

Other researchers have looked at shortest paths from the perspective of matrix multiplication. Fredman [11] shows that $O(n^{5/2})$ comparisons between sums of edge weights suffice to solve the all-pairs shortest paths problem. He uses this fact to do preprocessing, producing an algorithm that runs in time $O(n^3(\log\log n / \log n)^{1/3})$. Fredman's algorithm was simplified and the running time was decreased slightly by Takaoka [33]. For the important special cases where the graph is unweighted or the edge-weights are bounded integers, the algorithms of Alon, Galil, and Margalit [1] and Seidel [30] use fast matrix multiplication to find all-pairs shortest distances very quickly. For example, Seidel's algorithm finds all-pairs shortest distances in an unweighted undirected graph in time $n^\omega \log n$, where ω is the exponent of matrix multiplication (the current bound is $\omega < 2.376$, due to Coppersmith and Winograd [5]). Alon, Galil, Margalit and Naor [2] show how to extend these algorithms to find the paths, rather than just the distances, with a polylogarithmic slowdown. These results for matrix-based algorithms should be contrasted with the lower bound in Section 3;

such algorithms are not path-comparison based, since they perform comparisons involving sums of weights of edges that do not form a path. The use of such comparisons distinguishes the algebraic decision tree model from the path-comparison based model. It is surprising that one must allow such comparisons in order to improve on the $\Omega(n^3)$ bound.

An algorithm similar to the Hidden Paths Algorithm, with the same time bound, has been developed independently by McGeoch [28]. A variant of our algorithm has been developed independently by Jakobsson [20] as a transitive closure algorithm. Both these algorithms require more complex data structures than those used by the Hidden Paths Algorithm.

Lower bounds on the computational complexity of the all-pairs shortest paths problem have been proved in some other models. If the permissible operations are addition and minimum in a straight line computation, Kerr [23] shows that any algorithm requires $\Omega(n^3)$ running time. Regarding algebraic decision tree complexity, Spira and Pan [32] show that $\Omega(n^2)$ comparisons between sums of edge weights are necessary to solve the single-source shortest paths problem.

Generalized weight functions have been studied extensively in various contexts, and standard algorithms have been extended to work in generalized settings (see, for example, Frieze [13]). In particular, the all-pairs shortest path problem has been studied in arbitrary semirings, generalizing the semiring of reals with minimum and addition (see Zimmerman [35] for a survey and further references). This generalization provides a common framework for such problems as the construction of regular expressions for the languages accepted by finite automata [19]. Lengauer and Theune [26] extend standard algorithms to a yet more general framework. Knuth [24], generalizes the notion of paths to allow for compound edges, extending Dijkstra's algorithm to apply to derivations in context-free grammars.

2. The Hidden Paths Algorithm. In this section we describe the Hidden Paths Algorithm, which solves the all-pairs shortest paths problem in a directed graph $G = (V, E)$ with nonnegative edge weights. In order to present our algorithm, we need to make the following definitions.

2.1. Preliminary Definitions. We use $(u_1, u_2, \ldots, u_k)$ to denote a path from u_1 to u_k going through the vertices $u_2, \ldots, u_{k-1}$. The symbol $(u \rightsquigarrow v)$ denotes some path from u to v (which may be the edge (u, v) if it exists). The symbol $(u \rightsquigarrow v \rightsquigarrow w)$ denotes the concatenation of the paths represented by $(u \rightsquigarrow v)$ and $(v \rightsquigarrow w)$, and $(u, v \rightsquigarrow w)$ denotes the concatenation of the edge (u, v) to the path $(v \rightsquigarrow w)$. The *length* of a path $(u_1, \ldots, u_k)$ is

$$|(u_1, \ldots, u_k)| = k - 1 \ .$$

Let $\|(u, v)\|$ denote the *weight* of the edge (u, v). We extend the weight function by setting $\|(u, v)\| = \infty$ for any $(u, v) \notin E$, and by setting $\|(u, u)\| = 0$, so that $\|(u, v)\|$ is defined for all pairs u, v. The weight of a path $(u_1, \ldots, u_k)$ is

$$\|(u_1, \ldots, u_k)\| = \sum_{i=1}^{k-1} \|(u_i, u_{i+1})\|$$

DEFINITION 1. *A path* $(u \rightsquigarrow v)$ *is* optimal *if for any other path* $(u \rightsquigarrow' v)$, $\|(u \rightsquigarrow v)\| \leq \|(u \rightsquigarrow' v)\|$.

DEFINITION 2. *An edge is* optimal *if it is an optimal path between its endpoints.*

DEFINITION 3. *The number of optimal edges in G is denoted by $m^*(G)$.*

FACT 2.1. *An edge is optimal iff it participates in some shortest path.*

Note that in an unweighted graph each edge is optimal, so $m^* = m$ and our algorithm provides no improvement.

DEFINITION 4. *The* distance $d(u, v)$ *between two vertices u and v is the weight of an optimal path between them.*

2.2. Description of the Algorithm. The Hidden Paths Algorithm presented in this section finds a shortest path between every pair of vertices in a directed graph. Essentially it runs Dijkstra's single-source shortest paths algorithm in parallel for all points in the graph. The different single-source threads are integrated in a way that allows the use of intermediate results from one thread to reduce the work done by another. There is a similarity here to the all-pairs min-cut algorithm of Gomory and Hu [17], which uses the information gained during one min-cut computation to speed up the other computations. In a sense, the Hidden Paths Algorithm discovers the hidden "shortest path structure" of the graph by pruning away the unnecessary edges. We note that the algorithm actually constructs each path in reverse order, by adding edges to the tail of the path. This facilitates forward traversal on the constructed paths. It is simple to modify the algorithm to construct the paths in the form typically used in Dijkstra's algorithm. Here we give an intuitive description of the Hidden Paths Algorithm; a precise presentation can be found in Figures 1 and 2.

- For each vertex v, a list E_v^* of the optimal edges directed into v that have been found so far.
- A path array P, which for every pair u, v describes the current best path from u to v.
  Each entry $P[u, v]$ consists of
  first The first vertex on this path (other than u itself).
  weight The weight of the path.
  heaptr A pointer to the heap entry for this path, if one exists.
- A heap H of pointers to candidate paths that appear in the path array, ordered by weight.

FIG. 1. *Data structures maintained by the Hidden Paths Algorithm.*

The Hidden Paths Algorithm maintains a heap containing, for every two vertices $u \neq v$, the best path from u to v found so far. The heap is ordered according to path weight. It is initialized to contain for each pair $u \neq v$ a path of weight $\|(u, v)\|$ (recall that if $(u, v) \notin E$ then $\|(u, v)\| = \infty$). The path at the top of the heap is always an optimal path.

At each iteration, the algorithm removes a path, say $(u \rightsquigarrow v)$, from the top of the heap (a delete-min operation). This is an optimal path from u to v. This path is now used to construct a set of new *candidate paths*. If a new candidate path $(x \rightsquigarrow z)$ is shorter, it replaces the current best path from x to z in the heap. This maintains the optimality of the path at the top of the heap.

The complexity of the algorithm is a function of the number of candidate paths created, so we do not want to create too many. If $(u \rightsquigarrow v)$ is an edge, then the candidate paths are all those paths of the form $(u, v \rightsquigarrow w)$.[1] If $(u \rightsquigarrow v)$ is a path that is not an edge, then the candidate paths are all those of the form $(t, u \rightsquigarrow v)$ where

[1] A minor optimization is to consider only vertices w such that $(v \rightsquigarrow w)$ has already been discovered to be optimal. The algorithm remains correct, and fewer arithmetic operations are performed.

Initialize
    Initialize the heap of pairs $u \neq v$ ordered by the weight $\|(u,v)\|$.
    For all u, v
        Set $P[u,v].weight := \|(u,v)\|$.
        Set $P[u,v].heaptr$ to point to appropriate heap entry.
    For all v, set E_v^* to be empty.

While (heap not empty and weight of top item $\neq \infty$) do
    Step 1 Remove the top element $(u \rightsquigarrow v)$ from the heap.
    Step 2 If $(u \rightsquigarrow v)$ is an edge, then
        Add (u,v) to E_v^*.
        For each w, Update(u,v,w).
    Step 3 For all edges $(t,u) \in E_u^*$,
        Update(t,u,v).

Procedure Update(x,y,z):
    If $P[x,y].weight + P[y,z].weight < P[x,z].weight$ then
        Set $P[x,z].weight := P[x,y].weight + P[y,z].weight$.
        Set $P[x,z].first := y$.
        Change the priority of x,z in the heap to $P[x,z].weight$.
        Modify $P[x,z].heaptr$ accordingly.

FIG. 2. *The Hidden Paths Algorithm.*

(t,u) is an optimal edge that has already been found. Note that in the construction of candidate paths, the edge is always concatenated to the tail.

The following theorem shows that constructing this limited set of candidate paths suffices for finding a shortest path between each pair of vertices in the graph. The analysis in Section 2.3 shows that the resulting complexity is $O(m^*n + n^2 \log n)$.

Before stating the theorem, it is convenient to make one more definition.

DEFINITION 5. *For any two paths p and q, $p \prec q$ if $(\|p\|, |p|) < (\|q\|, |q|)$ according to lexicographic order.*

THEOREM 2.2. *The Hidden Paths Algorithm finds an optimal path between every connected pair of vertices in the graph. Furthermore, it discovers them in order of increasing weight.*

Proof. Let *Opt* be the set of optimal paths found so far. We prove the theorem by induction on the size of *Opt*. The inductive hypothesis is that at the beginning of each iteration, when p is the item at the top if the heap

1. p is an optimal path.
2. *Opt* contains an optimal path between each pair of vertices of distance less than $\|p\|$.
3. For each pair of vertices u and v for which an optimal path has not yet been found, the heap contains a path of minimal weight among those of the form (i) the edge (u,v) and (ii) paths having the form $(u, w \rightsquigarrow v)$ for (u,w) and $(w \rightsquigarrow v)$ in *Opt*.

The inductive hypothesis holds trivially at the beginning of the first iteration, since at that time p is the shortest edge in the graph, *Opt* is empty, and the heap

An extra field *opt* in the path array P can be used to mark whether or not a path has already been found to be optimal.

Operation	# of ops	cost for reg. heap	cost for Fib. heap
create	$O(n^2)$	$O(n^2)$	$O(n^2)$
priority change	$O(m^*n)$	$O(m^*n \log n)$	$O(m^*n)$
delete-min	$O(n^2)$	$O(n^2 \log n)$	$O(n^2 \log n)$
Total	—	$O(m^*n \log n)$	$O(m^*n + n^2 \log n)$

TABLE 1
The running time of the Hidden Paths Algorithm

contains all the edges. The construction of candidate paths in Steps 2 and 3 of the algorithm (see Figure 2) ensures that condition 3 always holds at the beginning of an iteration.

It remains to prove that when $p = (u \rightsquigarrow v)$ is the item on top of the heap, conditions 1 and 2 hold for p. This can be false only if there exists an optimal path $r = (w \rightsquigarrow y)$, such that $\|r\| < \|p\|$, and no optimal path from w to y has been placed in the heap. Condition 1 is violated if $w = u$ and $y = v$; condition 2 is violated otherwise. Let r be a smallest such path according to $\prec$. Since all edges were placed in the heap during the initialization step, r must be a path of length at least two. Assume $r = (w, x \rightsquigarrow y)$. It is clear that $(w, x) \prec r$ and $(x \rightsquigarrow y) \prec r$ and that both are necessarily optimal. By our choice of r, some optimal path $(x \rightsquigarrow' y)$ must have been placed in the heap. The edge (w, x) was placed in the heap during initialization, and is never replaced by a shorter path. Since $\|(x \rightsquigarrow' y)\| = \|(x \rightsquigarrow y)\| \leq \|r\| < \|p\|$, the path $(x \rightsquigarrow' y)$ must have already been deleted from the heap, and placed in *Opt*. Similarly, the edge (w, x) must also be in *Opt*. Therefore, by condition 3 of the inductive hypothesis, the path $(w, x \rightsquigarrow' y)$, whose weight is equal to $\|r\|$, must have been constructed as a candidate path, and thus a path from w to y of no greater weight must be in the heap. This path is an optimal path from w to y, contradicting the assumption that no such path was placed in the heap. □

2.3. Running Time Analysis. Let us count the number of operations used by the algorithm. In the initialization step, a heap of size $O(n^2)$ is created; it is clear that the cost of the other operations in this step is subsumed by the cost of the heap operations. The while loop is iterated at most $n(n-1)$ times, since in each iteration an optimal path is found. Therefore, the algorithm executes at most $n(n-1)$ delete-min operations in Step 1.

It remains to count only the time taken by Steps 2 and 3. Note that in each of these steps the calls to Procedure Update subsume all other operations. It therefore suffices to count calls to Procedure Update. We amortize the calls over the edges of the graph. In Step 2, charge the $O(n)$ calls to the edge (u, v). In Step 3, charge a call to Update(t, u, v) to the edge (t, u). Observe that, in either case, it is an optimal edge that is being charged. We claim that in fact only $O(n)$ updates are charged to any edge. This is clear for Step 2, since no edge is removed from the heap more than once. For Step 3, note that at most $n - 1$ optimal paths $(u \rightsquigarrow v)$ are found leaving any particular vertex u, and that Procedure Update is only called when $(u \rightsquigarrow v)$ is such an optimal path. Since only optimal edges are charged, and each is charged $O(n)$ updates, the total number of updates charged is $O(m^*n)$. Procedure Update requires a constant number of primitive operations, plus at most one priority change operation.

The complexity of the algorithm depends on the implementation of the heap (see Table 1). Using a standard heap implementation, the time for a priority change

operation is $O(\log n)$ and we get a total complexity of $\Theta(m^* n \log n)$. Using Fibonacci heaps (described in [12]), priority change operations take constant amortized time, and we therefore get a complexity of $\Theta(n^2 \log n + m^* n)$.

The Hidden Paths Algorithm is also very simple and easy to implement, and thus provides a practical substitute for Dijkstra's algorithm.

2.4. Extensions and Refinements. The Hidden Paths Algorithm can easily be transformed to one that finds the k nearest pairs of vertices in the graph. The revised algorithm will initialize the heap to contain only the actual edges in the graph. Then, when comparing candidate paths to existing paths (in Procedure Update), the algorithm will sometimes have to do an insert rather than a priority change operation. The algorithm terminates when $|Opt|$ reaches k. Because at most k^2 candidate paths are created, the running time of this algorithm (using Fibonacci heaps) is $O(m + k \log n + k^2)$.

The Hidden Paths Algorithm requires that the graph have non-negative edge weights. However, in the case of an integer edge weight function, several scaling algorithms (such as Gabow, Tarjan [15] and Goldberg [16]) transform the weight function into a non-negative weight function, that induces the same shortest paths structure on the graph. We can solve the shortest paths problem on such graphs by first using one of these algorithms to make the edge weights positive, and then applying the Hidden Paths Algorithm. Let $-N$, $N > 0$, be the weight of the smallest (most negative) edge in the graph. The combined running time of Goldberg's scaling algorithm and the Hidden Paths Algorithm is $O(m^* n + n^2 \log n + \sqrt{n} m \log N)$. Thus the Hidden Paths Algorithm may also improve on the $O(mn)$ upper bound in graphs with negative edge weights.

Although Fibonacci heaps have a better asymptotic running time than standard heaps, they may be undesirable in applications due to their increased complexity. It is therefore useful to point out cases in which the Hidden Paths Algorithm can be used with regular heaps without increasing the asymptotic complexity. This can happen if not every candidate path created necessitates a priority change operation. We show below that this situation occurs when the edge weights in the graph are small integers.[2] Such graphs are an interesting special case that can occur frequently in practice.

LEMMA 2.3. *Let $(u \rightsquigarrow v)$ be a non-edge candidate path in the heap. Let a be the weight of the longest optimal edge in the graph. Then*

$$\|(u \rightsquigarrow v)\| \leq d(u,v) + a \ .$$

Proof. This is clear if $(u \rightsquigarrow v)$ is optimal. On the other hand, a non-optimal path $(u \rightsquigarrow v) = (u, w \rightsquigarrow v)$ can become a path in the heap only if at some stage of the algorithm (u,w) and $(w \rightsquigarrow v)$ are in Opt but no optimal u,v path is in Opt. The edge (u,w) is optimal and therefore $\|(u,w)\| \leq a$. Since $(w \rightsquigarrow v)$ is optimal and in Opt, and since optimal paths are discovered in increasing order, we conclude that

$$\|(w \rightsquigarrow v)\| \leq d(u,v) \ .$$

Therefore

$$\|(u \rightsquigarrow v)\| = \|(u,w)\| + \|(w \rightsquigarrow v)\| \leq a + d(u,v) \ . \quad \square$$

[2] It actually suffices that the weights of the optimal edges be small integers.

THEOREM 2.4. *If the edge weights are integers, and a is the largest weight of an optimal edge, then there can be at most $(a+1)n^2$ priority changes, and the Hidden Paths Algorithm has a running time of $O(an^2 \log n + m^*n)$, using an ordinary heap.*

Proof. We will prove that for any pair of vertices u and v, the entry for u, v in the heap can be modified at most $a+1$ times. The original entry for u, v in the heap is the edge (u, v). This entry can only be replaced by a non-edge path $(u \rightsquigarrow v)$. By Lemma 2.4, $\|(u \rightsquigarrow v)\|$ is at most $d(u, v) + a$. By integrality, the priority of (u, v) can then be decreased at most a times. Overall, there can be at most $a+1$ priority change operations for the pair u, v. □

In fact, an even better bound can be achieved if we modify the algorithm slightly. The modified algorithm is similar to Dial's implementation [6] of Dijkstra's algorithm. Note that if the maximum edge weight is a, the weight of any optimal path is at most $a(n-1)$. Rather than using a heap to store candidate paths, we can use an array of size $a(n-1)+1$. Paths of weight w will be kept in a bucket at index w of the array. Initially only edges are inserted. Later, each time a candidate path $(u \rightsquigarrow v)$ improves the path from u to v, we remove the old path from the array and insert the new path at index $\|(u \rightsquigarrow v)\|$. Rather than iteratively deleting entries from the heap, we traverse the array from index 0 to index $a(n-1)$. As before, each path which we encounter as we traverse the array will be an optimal path. Also as before, we construct $O(m^*n)$ candidate paths. Thus the total running time of this modified algorithm is $O(an + m^*n)$.

If space is a concern, it can be conserved in the following fashion. It is easy to prove, using the techniques of lemma 2.3, that if p is the path at the top of the heap, then there is no non-edge path of weight exceeding $\|p\| + a$ in the heap. It follows that at any given time, we need only use $a+1$ buckets of our array to store candidate paths. Therefore our array can be mapped onto a circular array of size $a+1$ such that no collisions of buckets ever take place. It follows that only $a+1$ space is needed to store the buckets. If $a = O(n^2)$, it follows that the space used is no more than that needed in the heap implementation; if in addition the diameter by weight of the graph is $O(m^*n)$ (and certainly if $a = O(m^*)$), then the running time of the algorithm remains $O(m^*n)$.

Another easy consequence is the following:

LEMMA 2.5. *If $(u \rightsquigarrow v)$ is a non-edge non-optimal candidate path in the heap, then*

$$\|(u \rightsquigarrow v)\| \leq 2\, d(u, v) \ .$$

Proof. As in Lemma 2.3, the path $(u \rightsquigarrow v) = (u, w \rightsquigarrow v)$ can be placed in the heap only if (u, w) and $(w \rightsquigarrow v)$ were both found before any optimal u, v path. This implies that

$$\begin{aligned} \|(u, w)\| &\leq d(u, v) \\ \|(w \rightsquigarrow v)\| &\leq d(u, v) \end{aligned}$$

and therefore that

$$\|(u \rightsquigarrow v)\| = \|(u, w)\| + \|(w \rightsquigarrow v)\| \leq 2\, d(u, v) \ . \quad \square$$

This shows that at any stage in the algorithm, any non-edge path in the heap has at most twice the optimal weight. This property is useful for "anytime" applications,

which might require the algorithm to be stopped in the middle of execution with good intermediate results.

Finally, we note that the Hidden Paths Algorithm considers some unnecessary candidate paths. One possible improvement, which was also developed independently by Jakobsson [20], creates only candidate paths of which every subpath is optimal. More specifically, a path $p = (u, v \rightsquigarrow w, t)$ is made a candidate path iff $(u, v \rightsquigarrow w)$ and $(v \rightsquigarrow w, t)$ are already known to be optimal. This will clearly reduce the number of candidate paths formed, but there does not seem to be a simple expression for the reduced running time achieved by this algorithm. More complex data structures are required, but it is still possible to achieve a running time of $\Theta(c + n^2 \log n)$, where c is the number of candidate paths formed.

2.5. m^* in a Random Graph. To predict the behavior of the Hidden Paths Algorithm in practice, we need to study the quantity m^* for "typical" graphs. It is easy to construct graphs for which $m^* = O(n)$, while m is $\Theta(n^2)$. It is also easy to construct graphs for which $m^* = m$. In this section we note that for a large class of probability distributions on random graphs, $m^* = O(n \log n)$ with high probability.

Consider a distribution F on non-negative edge weights, which does not depend on n, such that $F(0) = 0$, and $F'(0)$ exists and is positive. In particular, the uniform distribution on $[0, 1]$ and the exponential distribution with mean λ both satisfy these conditions. The work of Frieze and Grimmet [14] implies that $m^*(G) = O(n \log n)$ with high probability. In particular, they prove the following result:

THEOREM 2.6 (FRIEZE AND GRIMMET). *Let G be a complete directed graph, whose edge weights are chosen independently according to F. Consider the set S of edges defined by placing (v, w) in S iff it is one of the p shortest edges originating at v, where $p = \min\{n - 1, 20 \log_2 n\}$. Then with probability $1 - O(n^{-1})$, S contains every optimal edge in G.*

Hence, under the conditions of the theorem, $m^*(G) = O(n \log n)$ with probability $1 - O(n^{-1})$ (and therefore $E[m^*(G)] = O(n \log n)$). Similar results are derived in a different context by Luby and Ragde [27]. Hassin and Zemel [18] prove a similar theorem (with a different constant) for both directed and undirected graphs, when the edge weights are uniformly distributed. The constant factors given by these analyses are small. In fact, empirical studies by McGeoch [28] indicate that when the edge weights are uniformly distributed, $m^*(G)$ grows approximately as $0.5n \ln n + 0.3n$. Furthermore, Theorem 2.6 holds for some discrete edge weight distributions, for instance if the edge weights are integers chosen uniformly and independently from the range $1, \ldots, n^2$.

COROLLARY 2.7. *If the edge weights of G are chosen independently according to F, then with high probability the running time of the Hidden Paths Algorithm is $O(n^2 \log n)$.*

This time bound is an improvement over earlier algorithms by Spira [31] and Bloniarz [3], and matches the performance of the algorithm of Frieze and Grimmet. However, the Hidden Paths Algorithm can be effectively used in any situation where m^* is significantly less than m, whereas the algorithm of Frieze and Grimmet is designed specifically for random graphs.

3. A Lower Bound. Many algorithms for the shortest paths problem use edge weights only to compute and compare the weights of paths. We therefore define a version of the decision tree model that captures this behavior.

DEFINITION 6. *A* path-comparison based *all-pairs shortest paths algorithm $\mathcal{A}$ accepts as input a graph G and a weight function. The algorithm $\mathcal{A}$ can perform*

all standard operations. However, the only way it can access the edge weights is to compare the weights of two different paths.

We can think of a path-comparison based algorithm as being given only the graph and a black-box path weight comparator. The path weights can be accessed only through the black box. The algorithm must output a reasonable encoding of the shortest paths in G.[3]

It should be noted that the algorithms of Floyd, Dijkstra, Spira, Bloniarz, and Frieze & Grimmet, as well as the Hidden Paths Algorithm, all fit into this path-comparison based model. On the other hand, Fredman's $o(n^3)$ algorithm [11] is not path-comparison based because it adds weights of edges that do not form a single path. This algorithm conform to the more general algebraic decision tree model.

We show a lower bound of $\Omega(mn)$ on the running time of any path-comparison based algorithm running on a graph with n vertices and m edges. For simplicity, we first show a lower bound of $\Omega(n^3)$ on the running time of any path-comparison based shortest paths algorithm running on a certain graph of $\Theta(n^2)$ edges. We then show how the construction can be modified to yield a lower bound of $\Omega(mn)$ for a graph with m edges, where on these graphs $m = m^*$. An obvious modification allows us to construct graphs with arbitrary values of m^* and m ($m^* \leq m$), for which $\Omega(m^*n)$ time would be required.

To show the $\Omega(n^3)$ lower bound, we construct a directed graph of $3n$ vertices on which any path-comparison based shortest paths algorithm must perform $\Omega(n^3)$ comparisons. The directed graph G has $\Omega(n^3)$ paths. We show that if $\mathcal{A}$ fails to examine one of these paths, then we can modify the weight function to make that path optimal without $\mathcal{A}$ being able to detect the change.

The graph G is a directed tripartite graph on vertices u_i, v_j, and w_k where i, j, and k range from 0 to $n-1$. The edge set for G is $\{(u_i, v_j)\} \cup \{(v_j, w_k)\}$ (see Figure 3). Therefore, the only paths are individual edges and paths (u_i, v_j, w_k) of length two.

To define the weight function, we work in base $n+1$ notation, generalized to allow negative digits. Define

$$[a_r, \ldots, a_0]_b = \sum_{i=0}^{r} a_i b^i$$

The edge weights are

$$\begin{aligned} \|(u_i, v_j)\| &= [1, \ 0, \ \ i, \ 0, \ \ j, \ \ \ 0, \ 0]_{n+1} \\ \|(v_j, w_k)\| &= [0, \ 1, \ 0, \ \ k, \ 0, \ -j, \ 0]_{n+1} \end{aligned}$$

and thus

$$\|(u_i, v_j, w_k)\| = [1, \ 1, \ \ i, \ \ k, \ \ j, \ -j, \ 0]_{n+1}.$$

Note that we allow negative digits to appear in the numbers. The standard positive digit representation of these numbers would require that a carry be taken from the next number to the left. This does not affect the correctness of the upcoming proofs. The following lemma is an immediate consequence of the definitions:

LEMMA 3.1. *Let $<$ denote the lexicographic ordering on tuples of integers, with the leftmost integer being the most significant. For all i, j, j', k, k' :*

[3] We require that the output have no information about path weights. For example, the weighted graph itself is not a reasonable encoding of the solution.

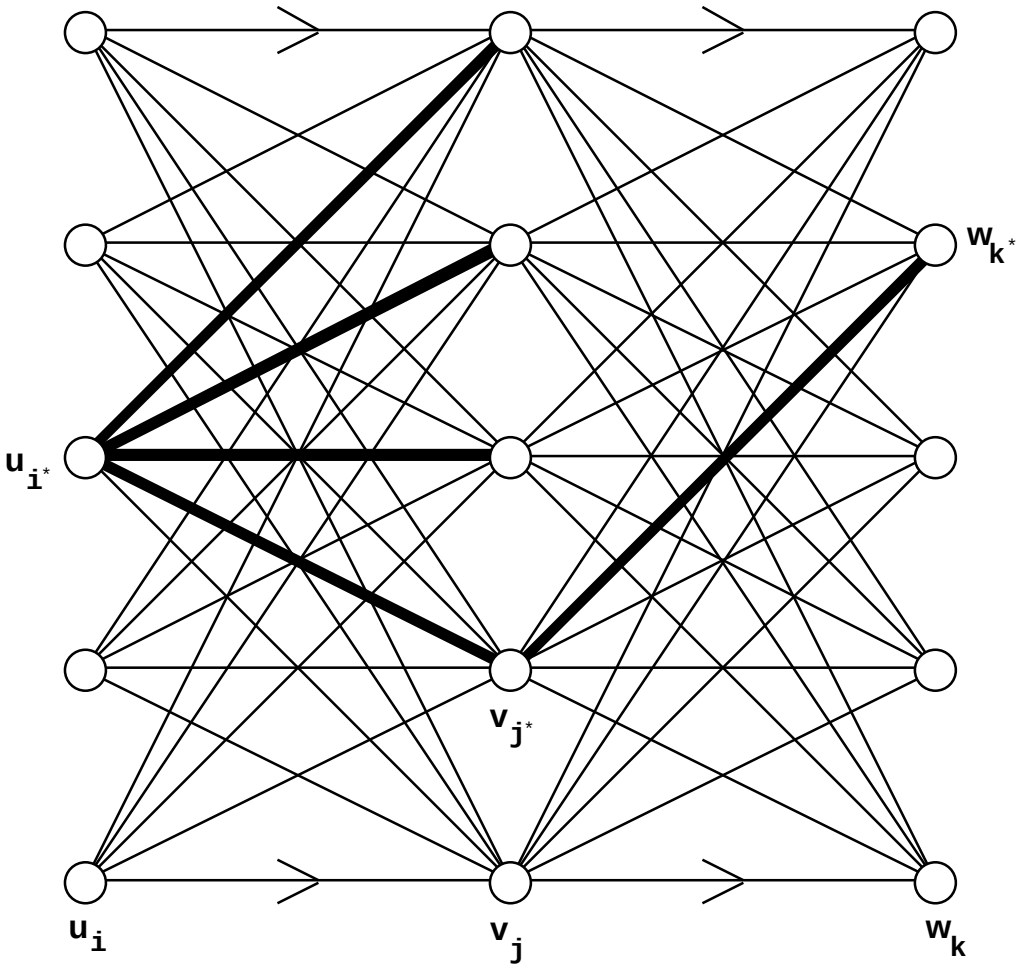

FIG. 3. *The graph used in the lower bound*

1. $\|(u_i, v_j)\| < \|(u_{i'}, v_{j'})\|$ *iff* $(i, j) < (i', j')$
2. $\|(v_j, w_k)\| < \|(v_{j'}, w_{k'})\|$ *iff* $(k, -j) < (k', -j')$
3. $\|(v_j, w_k)\| < \|(u_i, v_{j'})\|$
4. $\|(u_{i'}, v_{j'})\| < \|(u_i, v_j, w_k)\|$
5. $\|(u_i, v_j, w_k)\| < \|(u_{i'}, v_{j'}, w_{k'})\|$ *iff* $(i, k, j) < (i', k', j')$.

Proof. Immediate from the base $n+1$ notation. For example, item 3 follows from the fact that $\|(v_j, w_k)\| < [1, 0, 0, 0, 0, 0, 0]_{n+1} \leq \|(u_i, v_j)\|$. □

It follows that the unique optimal path from u_i to w_k goes through v_0, and has weight $[1, 1, i, k, 0, 0, 0]_{n+1}$. Define L to be the set of optimal paths.

Consider providing $(G, \|\cdot\|)$ as input to $\mathcal{A}$, and suppose that $\mathcal{A}$ runs correctly. It must therefore output the set of optimal paths L. Suppose further that a non-optimal path $p^* = (u_{i^*}, v_{j^*}, w_{k^*})$ with $j^* > 0$ was never one of the operands in any comparison operation performed by $\mathcal{A}$. We define a weight function $\|\cdot\|'$ in which p^* is the unique shortest path from u_{i^*} to w_{k^*}, but the ordering by weight of all the other paths remains the same. If we run $\mathcal{A}$ on $(G, \|\cdot\|')$, all path comparisons not involving

p^* give the same result as they did using $\|\cdot\|$. Therefore, since $\mathcal{A}$ never performed a comparison involving p^* while running on $\|\cdot\|$, we deduce that $\mathcal{A}$ still outputs L, which is now incorrect. If $\mathcal{A}$ never examined an optimal path (u_{i^*}, v_0, w_{k^*}), we can apply the above construction with $j^* = 1$. The algorithm will then fail because the only comparison that has a different result is between (u_{i^*}, v_0, w_{k^*}) and (u_{i^*}, v_1, w_{k^*}), which by hypothesis was not performed.

The weight function $\|\cdot\|'$ is $\|\cdot\|$ with the following modifications (in Figure 3, the edges with modified weights are marked by thicker lines). For all $j \le j^*$, we decrease the weight of the edge (u_{i^*}, v_j):

$$\|(u_{i^*}, v_j)\|' = [1, 0, i^*, 0, 0, j, j]_{n+1} \ .$$

We also decrease the weight of the edge (v_{j^*}, w_{k^*}):

$$\|(v_{j^*}, w_{k^*})\|' = [0, 1, 0, k^*, 0, -j^*, -n]_{n+1} \ .$$

Thus

$$\|(u_{i^*}, v_{j^*}, w_{k^*})\|' = [1, 1, i^*, k^*, 0, 0, j^* - n]_{n+1} < \|(u_{i^*}, v_0, w_{k^*})\|' \ .$$

LEMMA 3.2. *In G, the conditions of Lemma 3.1 continue to hold for $\|\cdot\|'$, except that the single path $p^* = (u_{i^*}, v_{j^*}, w_{k^*})$ directly precedes (u_{i^*}, v_0, w_{k^*}) in the ordering. Thus, under $\|\cdot\|'$ the path p^* is optimal.*

Proof. We show the conditions of Lemma 3.1 one at a time. Clearly we need only consider comparisons for which one or both operands have changed; *e.g.* only comparisons between operands involving i^*, j^*, or k^*.

1. The four most significant digits of the base $n+1$ representation of the weights remain unchanged and the three least significant digits still increase with j.
2. Only $\|(v_{j^*}, w_{k^*})\|$ has changed, and only by n; but the edge whose weight was closest differed by $n+1$.
3. The two most significant digits are unchanged.
4. This also is enforced by the two most significant digits.
5. If $i \neq i'$ or $k \neq k'$, the inequality is enforced by the four most significant digits. It is also simple to verify that for each i and k, $\|(u_i, v_j, w_k)\|$ increases with j (with the exception of $(u_{i^*}, v_{j^*}, w_{k^*})$). □

We have therefore proved the following:

THEOREM 3.3. *There exists a directed graph of $3n$ vertices on which any path-comparison based shortest paths algorithm must perform at least $n^3/2$ path weight comparisons.*

Note that since all edge weights are polynomial in n, the input graph G is not hard merely because unusually large edge weights increase the input size. As the graph constructed is a directed acyclic graph, the lower bound holds even for this restricted class of graphs.

We now adapt the above proof to show an $\Omega(mn)$ lower bound for graphs of m edges. Let $m \ge 4n$, and assume without loss of generality that $2n$ divides m. We perform the same construction as before, but of the middle vertices we use only $v_0, \ldots, v_{m/2n-1}$, connecting each of them to all the vertices u_i and w_k. This requires m edges and creates $mn/2$ two-edge paths. We also use the same weight function as before, restricted to the edges we include.

THEOREM 3.4. *There exists a directed graph with $2n + m/2n$ vertices and m edges, on which any path-comparison based shortest paths algorithm must perform at least $mn/2$ path weight comparisons.*

COROLLARY 3.5. *If $m = \Omega(n)$ then there exists a directed graph G with n vertices and m edges on which any path-comparison based shortest paths algorithm must perform $\Omega(mn)$ path weight comparisons.*

For $m = \Omega(n \log n)$ this lower bound is tight, since it matches the upper bound achieved by Dijkstra's algorithm.

We can in fact show that even the shortest paths *verification* problem requires $\Omega(mn)$ time for path-comparison based algorithms. A verification algorithm $\mathcal{A}$ accepts as input a graph, a weight function (which we again think of as a black-box comparator), and an encoding L that describes, for each pair of vertices, a path between them. Note that the standard description of shortest paths can be encoded in $O(n^2)$ space, so the input size imposes no non-trivial lower bound. The algorithm $\mathcal{A}$ accepts its input if and only if each path in L is a shortest path.

To show the lower bound, we use the same construction as before. We let L be the shortest paths under $\|\cdot\|$, *i.e.* $L = \{(u_i, v_0, w_k) \mid i, k = 0 \ldots n-1\}$, and provide $(G, \|\cdot\|, L)$ as input to $\mathcal{A}$. If $\mathcal{A}$ accepts using fewer than mn comparisons, we use the same modification as before and pass $(G, \|\cdot\|', L)$ to $\mathcal{A}$, which will incorrectly accept this modified input.

COROLLARY 3.6. *Any path-comparison based algorithm for verification of shortest paths requires time $\Omega(mn)$ on G.*

If we add edges from each u_i to each w_k (thus producing $\Omega(n^2)$ edges), and set $\|(u_i, w_k)\| = \|(u_i, v_0, w_k)\|$, we can similarly deduce that

COROLLARY 3.7. *Any path-comparison based algorithm for verifying that the edge weights satisfy the triangle inequality requires time $\Omega(n^3)$ on graphs of n vertices.*

The construction of Theorem 3.4 can be applied to randomized algorithms for the shortest paths problem. For suppose that the expected number of comparisons performed by such an algorithm is $o(mn)$. Then the probability that a randomly selected path is checked by the algorithm approaches 0 as n goes to ∞. Thus if we take the graph G and select a single path $(u_{i^*}, v_{j^*}, w_{k^*})$ uniformly at random and apply the $\|\cdot\|'$ construction, the algorithm detects our modification with probability approaching 0. We thus have

THEOREM 3.8. *If a randomized path-comparison based shortest paths algorithm performs $o(mn)$ expected comparisons on graphs with m edges and n vertices then there is a weighted graph on which the algorithm will almost surely fail to be correct.*

The following corollaries are randomized counterparts of Corollaries 3.6 and 3.7:

COROLLARY 3.9. *Any randomized path-comparison based shortest paths verification algorithm must perform $\Omega(mn)$ expected comparisons.*

COROLLARY 3.10. *Any randomized path-comparison based algorithm for verifying that all edge weights satisfy the triangle inequality must perform $\Omega(n^3)$ expected comparisons.*

We conjecture that the lower bounds in this section also hold for path-comparison based algorithms running on undirected graphs, but this remains to be proved.

4. Generalized Weight Functions.

4.1. Definitions and Basic Properties. Many shortest paths algorithms in fact solve a much more general problem. In particular, we consider the following generalized shortest paths problem: Given a graph G, and a *generalized weight function* $\|\cdot\|$ that maps every path p to a weight $\|p\|$ in some totally ordered set (with the ordering denoted by $\leq$), find for each pair of vertices a path between them of minimum weight. To make this problem tractable, we impose restrictions on the weight function.

DEFINITION 7. *Consider a weight function* $\|\cdot\|$*:*

- *it is* monotonic *if for all* u, v, w*,*

$$\|(v \rightsquigarrow w)\| \leq \|(v \rightsquigarrow' w)\| \implies \|(u \rightsquigarrow v \rightsquigarrow w)\| \leq \|(u \rightsquigarrow v \rightsquigarrow' w)\|,$$

  and similarly for $\|(u \rightsquigarrow v)\| \leq \|(u \rightsquigarrow' v)\|$*.*
- *it is* non-negative *if for all* u, v, w

$$\|(u \rightsquigarrow v \rightsquigarrow w)\| \geq \max(\|(u \rightsquigarrow v)\|, \|(v \rightsquigarrow w)\|) \ .$$

- *it is* acyclic *if for all* v *the empty path from* v *to* v*, denoted* $v \overset{\emptyset}{\rightsquigarrow} v$*, is optimal.*
- *it is* inductive *if there exists a* concatenation function f *such that for all* u, v, w*,*

$$\|(u \rightsquigarrow v \rightsquigarrow w)\| = f(\|u \rightsquigarrow v\|, \|v \rightsquigarrow w\|).$$

The standard path weight function is monotonic and inductive. It is acyclic if there are no negative weight cycles. It is non-negative if all path weights are non-negative. An example of a monotonic, non-negative, inductive, nonstandard weight function is one that assigns to every path a weight equal to the weight of the maximal edge on the path. Solving single-source shortest paths under this weight function is referred to as the *bottleneck path problem* in [34].

In the literature on generalizing shortest paths to semirings (see [35]), the semiring axioms imply both inductiveness and monotonicity of the weight function. Frieze [13] restricts to a narrower class, which essentially consists of monotonic, acyclic, inductive weight functions over the reals. Lengauer and Theune [26] study extensions of the shortest paths problem to situations where the path weights are only partially ordered; this allows them to deal with certain non-monotonic weight functions.

FACT 4.1. *Any non-negative weight function is also acyclic.*

Proof. Let v be any vertex, and $(v \rightsquigarrow v)$ any cycle. Then

$$\begin{aligned} \|(v \rightsquigarrow v)\| &= \|(v \rightsquigarrow v \overset{\emptyset}{\rightsquigarrow} v)\| \\ &\geq \max(\|v \rightsquigarrow v\|, \|v \overset{\emptyset}{\rightsquigarrow} v\|) \quad \text{by non-negativity} \\ &\geq \|(v \overset{\emptyset}{\rightsquigarrow} v)\| \ . \end{aligned}$$

Therefore, $(v \overset{\emptyset}{\rightsquigarrow} v)$ is optimal. □

FACT 4.2. *If* $\|\cdot\|$ *is monotonic and acyclic, then any path can be trimmed, by removing cycles, to obtain a simple path of smaller or equal weight.*

Proof. If a path p contains a cycle, replace the cycle with the appropriate empty path, which by acyclicity has no greater weight than the cycle; by monotonicity this replacement does not increase the weight of the path. Continue this procedure until no cycles remain. □

FACT 4.3. *Under a monotonic and acyclic weight function, any connected pair of vertices is connected by a simple optimal path.*

Proof. If two vertices are connected, the set of simple paths between them is nonempty and finite, and thus must contain some path of minimal weight. By Fact 4.2, this path is also shorter than all non-simple paths between these vertices, and is thus optimal. □

FACT 4.4. *The shortest paths under any monotonic acyclic weight function can be encoded in the standard manner in* $O(n^2)$ *space.*

Proof. In the (u, v) entry, store the first edge on any minimal length shortest path from u to v. □

We argue that monotonicity is the major defining characteristic of the shortest paths problem, for it ensures that the shortest path between two vertices can be constructed from other shortest paths. The property of acyclicity is also important, since it ensures that a simple optimal path exists between every pair of connected vertices. We shall therefore restrict our attention to monotonic acyclic weight functions.

The following theorem will be used in our extention of the Hidden Paths Algorithm to generalized weight functions.

DEFINITION 8. *A path is* taut *if every edge in it is optimal. A weight function is taut if every connected pair of vertices is connected by a taut optimal path.*

LEMMA 4.5. *Any monotonic and acyclic weight function is taut.*

Proof. Define a non-optimal edge (u, v) to be *strongly non-optimal* if there is no taut optimal path from u to v, *i.e.* if every optimal path from u to v contains a non-optimal edge. For the lemma to be false, there must be some pair of vertices all of whose optimal paths contain a strongly non-optimal edge. Otherwise, using the monotonicity condition, we could take a path containing no strongly non-optimal edge, and replace each non-optimal edge with a taut optimal subpath, thus producing a taut optimal path. It therefore suffices to prove that no strongly non-optimal edges exist.

Assume that there exists some strongly non-optimal edge (u, v). By the same argument as above, an edge is strongly non-optimal if and only if every optimal path between its endpoints contains a strongly non-optimal edge. Since, by Fact 4.3, there exists an optimal path between every pair of connected vertices, we can construct an infinite sequence of edges $\{(u, v) = (u_0, v_0), (u_1, v_1), (u_2, v_2), \ldots\}$ such that (u_{i+1}, v_{i+1}) is a strongly non-optimal edge contained in some optimal path $(u_i \overset{i}{\rightsquigarrow} u_{i+1}, v_{i+1} \overset{i}{\rightsquigarrow} v_i)$ from u_i to v_i. Since the set of edges is finite, we have that $(u_i, v_i) = (u_{i+k}, v_{i+k})$ for some i, k. Now observe that by induction on j, using the monotonicity condition, $(u_i \overset{i}{\rightsquigarrow} u_{i+1} \overset{i+1}{\rightsquigarrow} \ldots \overset{i+j-1}{\rightsquigarrow} u_{i+j}, v_{i+j} \overset{i+j-1}{\rightsquigarrow} \ldots \overset{i+1}{\rightsquigarrow} v_{i+1} \overset{i}{\rightsquigarrow} v_i)$ is an optimal path. In particular, this is true for $j = k$. Since $u_{i+k} = u_i$, this means we have an optimal path of the form $(u_i \rightsquigarrow u_i, v_i \rightsquigarrow v_i)$. By the conditions of monotonicity and acyclicity, this implies that $(u_i \overset{\emptyset}{\rightsquigarrow} u_i, v_i \overset{\emptyset}{\rightsquigarrow} v_i) = (u_i, v_i)$ must be optimal, contradicting the fact that (u_i, v_i) is strongly non-optimal. □

4.2. Algorithms. An algorithm for a generalized shortest paths problem receives as input the graph and a black box for the weight function. We assume that the black box takes constant time to compute the weight of any path. Many path-comparison based shortest paths algorithms also work for generalized weight functions. We consider here Floyd's algorithm, Dijkstra's algorithm, and the Hidden Paths Algorithm.

THEOREM 4.6. *Floyd's algorithm works on any monotonic and acyclic weight function.*

Proof. Recall that Floyd's algorithm iteratively finds for each i the best path between every pair of vertices u, v that uses (except for the endpoints) only the first i vertices in the graph. It does this by comparing, at stage i, the best path that uses only the first $i-1$ vertices with all paths of the form $(u \rightsquigarrow w \rightsquigarrow v)$ where w is the i'th vertex, and both $(u \rightsquigarrow w)$ and $(w \rightsquigarrow v)$ use only the first $i-1$ vertices.

The proof is by induction on i. The inductive hypothesis is that after stage i, the algorithm has found for every pair of vertices u, v a best path among those using only

the first i vertices. Note that once an optimal path is found, it is never replaced.

The base case, $i = 0$, is obvious. Assume that the inductive hypothesis holds for stage $i-1$, and let w be the i'th vertex. Let u, v be a pair of vertices for which any best path that uses only the first i vertices uses the vertex w. Let this path be $(u \rightsquigarrow v) = (u \rightsquigarrow w \rightsquigarrow v)$. By Fact 4.2, any cycle in a path can be eliminated without increasing the weight of the path, so we may assume that neither $(u \rightsquigarrow w)$ nor $(w \rightsquigarrow v)$ contains w as an interior point. In other words, the paths $(u \rightsquigarrow w)$ and $(w \rightsquigarrow v)$ use only the first $i-1$ vertices. By the inductive hypothesis, by the end of stage $i-1$ Floyd's algorithm found a best path $(u \rightsquigarrow' w)$ among all the paths from u to w using only the first $i-1$ vertices. Therefore,

$$\|(u \rightsquigarrow' w)\| \leq \|(u \rightsquigarrow w)\| .$$

Similarly, the algorithm found a path $(w \rightsquigarrow' v)$ such that

$$\|(w \rightsquigarrow' v)\| \leq \|(w \rightsquigarrow v)\| .$$

Using monotonicity, we deduce that

$$\|(u \rightsquigarrow' w \rightsquigarrow' v)\| \leq \|(u \rightsquigarrow w \rightsquigarrow' v)\| \leq \|(u \rightsquigarrow w \rightsquigarrow v)\| .$$

Therefore, in phase i Floyd's algorithm finds a path from u to v which is optimal among paths using only the first i vertices.

At the conclusion of stage n, the algorithm has found for each pair u, v a path which is best among those using all vertices, *i.e.* an optimal path. □

Note as well that Floyd's algorithm can be used to verify the acyclicity of a path weight function, since it will find a cycle which is better than an empty path if such a cycle exists.

THEOREM 4.7. *Dijkstra's algorithm works on any monotonic non-negative weight function.*

Proof. Dijkstra's algorithm is run from a single *source* vertex s, and can be thought of as maintaining a set *Opt* of optimal paths $(s \rightsquigarrow v)$ from s to some of the other vertices in the graph. Let $V(Opt)$ denote the set of endpoints of paths in *Opt*. At each iteration the algorithm adds to *Opt* a path $(s \rightsquigarrow u, v)$ that minimizes $\{\|(s \rightsquigarrow' u, v)\| \mid (s \rightsquigarrow' u) \in Opt, (u, v) \in E, v \notin V(Opt)\}$. It suffices to show by induction that the path $(s \rightsquigarrow u, v)$ is in fact an optimal path. To show this, consider some other path from s to v, $(s \rightsquigarrow x, y \rightsquigarrow v)$, where (x, y) is the first edge on the path such that $x \in V(Opt)$ and $y \notin V(Opt)$. Such an edge must exist since $s \in V(Opt)$ and $v \notin V(Opt)$. As $x \in V(Opt)$, there exists an optimal path $(s \rightsquigarrow' x) \in Opt$. Then

$$\begin{aligned} \|(s \rightsquigarrow x, y \rightsquigarrow v)\| &\geq \|(s \rightsquigarrow x, y)\| && \text{by non-negativity} \\ &\geq \|(s \rightsquigarrow' x, y)\| && \text{by monotonicity} \\ &\geq \|(s \rightsquigarrow' u, v)\| && \text{by choice of } u, v. \end{aligned}$$

Thus $\|(s \rightsquigarrow' u, v)\|$ is in fact optimal, as desired. □

THEOREM 4.8. *The Hidden Paths Algorithm works on any monotonic non-negative weight function.*

Proof. We modify the inductive hypothesis in the proof of Theorem 2.2 as follows:

1. If p is the item at the top of the heap, then p is an optimal path.
2. *Opt* contains a taut optimal path between each pair of vertices of distance less than $\|p\|$.

3. For each pair of vertices u and v for which an optimal path has not yet been found, the heap contains a path of minimal weight among those of the form (i) the edge (u, v) and (ii) taut paths having the form $(u, w \rightsquigarrow v)$ for (u, w) and $(w \rightsquigarrow v)$ in *Opt*.

An examination of the proof shows that the nature of the weight function is used only in the proof of Conditions 1 and 2. Assume one of them to be false and let $r = (x \rightsquigarrow w)$ be a minimal (under $\prec$) taut path such that no optimal path from x to w has yet been placed in the heap. As before, r cannot consist of a single edge, so assume that $r = (x, y \rightsquigarrow w)$. Due to non-negativity, $(x, y) \prec r$ and $(y \rightsquigarrow w) \prec r$. Therefore, $d(y, w) \leq \|r\|$. If $d(y, w) < \|r\|$, then by the minimality of r and Lemma 4.5, there exists an optimal path $(y \rightsquigarrow' w)$ in *Opt*. If $d(y, w) = \|r\|$, then $(y \rightsquigarrow w)$ is taut and optimal, and we can again deduce by the choice of r and the definition of $\prec$ that an optimal path $(y \rightsquigarrow' w)$ must already be in *Opt*. The edge $(x, y) \prec r$ and is optimal by the tautness of r, and must therefore also be in *Opt*. But then $(x, y \rightsquigarrow' w)$, which due to monotonicity is also an optimal path from x to w, must have been placed in the heap. This contradicts the assumption that no optimal path from x to w has been found. □

4.3. Lower Bounds. The lower bound in Section 3 can be adapted to the situation of generalized weight functions. We show a lower bound for the class of monotonic non-negative weight functions, even on undirected graphs.

THEOREM 4.9. *Any algorithm to solve the generalized shortest paths problem for arbitrary monotonic non-negative weight functions requires* $\Omega(mn)$ *path weight queries.*

Proof. We consider a modified version of the construction from Section 3. Use the same graph G, with middle vertices $v_0, \ldots, v_{m/2n-1}$ but with undirected edges, and let $\|\cdot\|$ be defined as follows:

$$\begin{aligned} \|(v, v)\| &= 0 \text{ for all vertices } v \\ \|(u_i, v_j)\| &= 2 \\ \|(v_j, w_k)\| &= 2 \\ \|(u_i, v_j, w_k)\| &= 4 \ . \end{aligned}$$

All other paths have length 5. Suppose as before that some path $(u_{i^*}, v_{j^*}, w_{k^*})$ does not have its weight queried. Change the weight of this path to be 3. It is simple to verify that the modified weight function remains monotonic and non-negative. □

This lower bound can also be extended to the case of inductive weight functions studied in [13]. To do this, assign to each edge e in the graph a unique weight $\|e\|$. We are then free to assign arbitrary weights to paths of length 2, because each such path contains a different pair of subpath weights. Our concatenation function is defined by the weights we want to assign to these length 2 paths. To assign weight ω to the path $(u \rightsquigarrow v \rightsquigarrow w)$, set $f(\|u \rightsquigarrow v\|, \|v \rightsquigarrow w\|) = \omega$. We can now proceed almost exactly in the previous case. Assign weights $1, \ldots, n^2$ to the individual edges. Assign weight $4n^2$ to all paths of length 2, and $5n^2$ to all other paths (so that $f(4n^2, y) = f(x, 4n^2) = 5n^2$). If any path of length 2 is not examined, change the weight of this path to be $3n^2$. It is trivial to verify that these weight functions are inductive (as well as non-negative and monotonic). We have proved the following theorem:

THEOREM 4.10. *Any algorithm to solve the generalized shortest paths problem for arbitrary monotonic non-negative inductive weight functions requires* $\Omega(mn)$ *path weight queries.* Note that Theorem 4.9 holds true even when the weight function is

restricted to take integer values in a bounded range. It is easy to see that Theorem 4.10 requires unbounded edge weights.

Corollaries parallel to Corollaries 3.6 and 3.7 also hold. Thus any subcubic solution to the standard shortest paths problem must take advantage of more specific properties of path weights than monotonicity, acyclicity, non-negativity, and induction.

5. Conclusion. We have produced a new algorithm, the Hidden Paths Algorithm, and identified a new measure m^*—the number of edges that participate in shortest paths. The Hidden Paths Algorithm runs in time $O(m^*n + n^2 \log n)$. The question arises: are there finer measures of the shortest-paths difficulty of a graph? In particular, the Hidden Paths Algorithm essentially runs in time proportional to the number of candidate paths formed. The improved algorithm mentioned in Section 2.4 forms fewer such paths than the Hidden Paths Algorithm. Is there a simple measure for this quantity? We conjecture that no path-comparison based algorithm for all-pairs shortest paths can perform fewer path weight comparisons than the improved algorithm.

The expected value of m^* has been shown to be significantly less than m in the case of independent uniformly distributed edge weights. This suggests that there are many situations in which the Hidden Paths Algorithm will be significantly faster than Dijkstra's algorithm. One can think of the optimal edges as forming a *certificate* of the shortest path structure of the graph, that must be revealed. The philosophy of the Hidden Paths Algorithm is thus similar to recent algorithms for connectivity [4, 29], that work by first finding a sparse subgraph (or certificate) with the same connectivity.

We have shown a lower bound of $\Omega(mn)$ on the running time of path-comparison based algorithms for all-pairs shortest paths. It is of particular interest that the construction and verification algorithms have the same worst case complexity. Compare this to the situation for the minimum spanning tree problem, where there is a linear-time algorithm to verify a minimum spanning tree [25], although no algorithm is known that finds one in linear-time. The comparison based lower bound shows that any improvement in the worst case complexity of shortest paths algorithms, such as Fredman's $o(n^3)$ algorithm, must take advantage of the numerical aspects of the problem, in addition to the ordering of path weights.

The obvious open problem arising from the lower bound is to extend the construction to the case of undirected graphs. Another goal would be to decrease the gap in the algebraic decision tree complexity, between Spira and Pan's $\Omega(n^2)$ lower bound, and Fredman's $O(n^{5/2})$ upper bound. Also, our lower bound would be strengthened if we could show that it held for all graphs of a certain structure and varying weights rather than for a single graph.

Finally, in Section 4 we introduced the notion of a generalized weight function, and defined some natural properties of such functions. We showed that the $\Omega(mn)$ lower bound also holds for a certain class of generalized weight functions, even for undirected graphs. It would be interesting to find tighter classes of weight functions for which the lower bound still holds. We have shown that many existing all-pairs shortest paths algorithms make little use of numerical properties of path weights, and hence work even for generalized weight functions. These algorithms are all path-comparison based. We feel that there is a strong connection between the algorithmic property of being path-comparison based, and the ability of an algorithm to work on generalized weight functions. Further work on this topic could lead to a better understanding of how properties of a path weight function affect the complexity of graph algorithms,

and what properties of the standard weight function allow the path-comparison based lower bound to be circumvented.

Acknowledgements. We would like to thank M. Luby and C. McGeoch for pointing out references to work on expected shortest path lengths in graphs with random edge weights.

REFERENCES

[1] N. Alon, Z. Galil, and O. Margalit, *"On the exponent of the all pairs shortest path problem"*, in Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, 1991, pp. 569–575.

[2] N. Alon, Z. Galil, O. Margalit, and M. Naor, *"Witnesses for boolean matrix multiplication and for shortest paths"*, Tech. Report RJ 8744, IBM, 1992.

[3] P. A. Bloniarz, *"A shortest-path algorithm with expected time $O(n^2 \log n \log^* n)$"*, Tech. Report 80-3, Department of Computer Science, State University of New York at Albany, Aug. 1980.

[4] J. Cheriyan and R. Thurimella, *"Algorithms for parallel k-vertex connectivity and sparse certificates"*, in Proceedings of the 23rd ACM Symposium on Theory of Computing, 1991, pp. 391–401.

[5] D. Coppersmith and S. Winograd, *"Matrix multiplication via arithmetic progressions"*, Journal of Symbolic Computation, 9 (1990), pp. 251–280.

[6] R. B. Dial, *"Algorithm 360: Shortest path forest with topological ordering"*, Communications of the ACM, 12 (1969), pp. 632–633.

[7] E. W. Dijkstra, *"A note on two problems in connection with graphs"*, Numerische Mathematik, 1 (1959), pp. 260–271.

[8] T. Feder and R. Motwani, *"Clique partitions, graph compression and speeding-up algorithms"*, in Proceedings of the 23rd ACM Symposium on Theory of Computing, 1991, pp. 123–133.

[9] R. W. Floyd, *"Algorithm 97: Shortest path"*, Communications of the ACM, 5 (1962), p. 345.

[10] G. N. Frederickson, *"Planar graph decomposition and all pairs shortest paths"*, Journal of the ACM, 38 (1991), pp. 162–204.

[11] M. L. Fredman, *"New bounds on the complexity of the shortest path problem"*, SIAM Journal on Computing, 5 (1976), pp. 83–89.

[12] M. L. Fredman and R. E. Tarjan, *"Fibonacci heaps and their uses in improved network optimization algorithms"*, Journal of the ACM, 36 (1986), pp. 596–615.

[13] A. Frieze, *"Minimum paths in directed graphs"*, Operations Research Quarterly, (1977).

[14] A. M. Frieze and G. R. Grimmet, *"The shortest-path problem for graphs with random arc-lengths"*, Discrete Applied Mathematics, 10 (1985), pp. 57–77.

[15] H. N. Gabow and R. E. Tarjan, *"Faster scaling algorithms for network problems"*, SIAM Journal on Computing, (1989), pp. 1013–1036.

[16] A. V. Goldberg, *"Scaling algorithms for the shortest paths problem"*, Tech. Report STAN-CS-92-1429, Stanford University, 1992.

[17] R. E. Gomory and T. C. Hu, *"Multi-terminal network flows"*, SIAM Journal on Applied Mathematics, 9 (1961), pp. 551–570.

[18] R. Hassin and E. Zemel, *"On shortest paths in graphs with random weights"*, Mathematics of Operations Research, 10 (1985), pp. 557–564.

[19] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Series in Computer Science, Addison-Wesley, 1979.

[20] H. Jakobsson, *"Mixed-approach algorithms for transitive closure"*, in Proceedings of the 10th ACM Symposium on Principles of Database Systems, 1991, pp. 199–205.

[21] D. B. Johnson, *"Efficient algorithms for shortest paths in sparse networks"*, Journal of the ACM, 24 (1977), pp. 1–13.

[22] D. R. Karger, D. Koller, and S. J. Phillips, *"Finding the hidden path: Time bounds for all-pairs shortest paths*, in Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, 1991, pp. 560–568.

[23] L. R. Kerr, *The Effect of Algebraic Structure on the Computational Complexity of Matrix Multiplications*, PhD thesis, Cornell University, 1970.

[24] D. E. Knuth, *"A generalization of Dijkstra's algorithm"*, Information Processing Letters, 6 (1977), pp. 1–5.

[25] J. Komlos, *"Linear verification for spanning trees"*, Combinatorica, 5 (1985), pp. 57–65.

[26] T. Lengauer and D. Theune, *"Efficient algorithms for path problems with general cost criteria"*, in Proceedings of the 18th International Colloquium on Automata, Languages and Programming, 1991, pp. 314–326.

[27] M. Luby and P. Ragde, *"A bidirectional shortest-path algorithm with good average case behavior"*, Algorithmica, 4 (1989), pp. 551–567.

[28] C. C. McGeoch, *"A new all-pairs shortest-path algorithm"*, Tech. Report 91-30, DIMACS, 1991. to appear in Algorithmica.

[29] H. Nagamochi and T. Ibaraki, *"Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph"*, Algorithmica, to appear, (1991).

[30] R. Seidel, *"On the all-pairs-shortest-path problem"*, in Proceedings of the 24th ACM Symposium on Theory of Computing, 1992, pp. 745–749.

[31] P. M. Spira, *"A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$"*, SIAM Journal on Computing, 2 (1973), pp. 28–32.

[32] P. M. Spira and A. Pan, *"On finding and updating shortest paths and spanning trees"*, in Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory, 1973.

[33] T. Takaoka, *"A new upper bound on the complexity of the all pairs shortest path problem"*, in Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science, 1991, pp. 209–213.

[34] R. E. Tarjan, *Data Structures and Network Algorithms*, vol. 44 of CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, 1983.

[35] U. Zimmerman, *Linear and Combinatorial Optimization in Ordered Algebraic Structures*, vol. 10 of Annals of Discrete Mathematics, North-Holland Publishing Company, 1981.