SIAM J. COMPUT.
Vol. 37, No. 5, pp. 1637–1655

A PRACTICAL SHORTEST PATH ALGORITHM WITH LINEAR EXPECTED TIME*

ANDREW V. GOLDBERG†

Abstract. We present an improvement of the multilevel bucket shortest path algorithm of Denardo and Fox [*Oper. Res.*, 27 (1979), pp. 161–186] and justify this improvement both theoretically and experimentally. We prove that if the input arc lengths come from a natural probability distribution, the new algorithm runs in linear average time while the original algorithm does not. We also describe an implementation of the new algorithm. Our experimental data suggests that the new algorithm is preferable to the original one in practice. Furthermore, for integral arc lengths that fit into a word of today's computers, the performance is close to that of breadth-first search, suggesting limitations on further practical improvements.

Key words. algorithms, data structures, shortest paths, experimental evaluation

AMS subject classifications. 68W40, 68Q25

DOI. 10.1137/070698774

1. Introduction. The shortest path problem with nonnegative arc lengths (*the NSP problem*) is very common in practice, and algorithms for this problem have been extensively studied both from theoretical, e.g., [2, 6, 9, 10, 12, 13, 17, 25, 29, 30, 34, 37, 38, 39, 40, 42, 43, 44, 45], and computational, e.g., [4, 5, 11, 19, 20, 24, 27, 28, 31, 35, 36, 46], viewpoints. Efficient implementations of Dijkstra's algorithm [12], in particular, have received a lot of attention.

Suppose that the input graph has n vertices and m arcs. To state some of the previous results, we assume that the input arc lengths are integral. Let U denote the biggest arc length. We define C to be the ratio between U and the smallest nonzero arc length, δ. Note that if the lengths are integral, then $C \leq U$. Modulo precision problems and arithmetic operation complexity, our results apply to real-valued arc lengths as well. To simplify comparing time bounds with and without U (or C), one can make the following similarity assumption [18]: $\log U = O(\log n)$.

Several algorithms for the problem have near-linear worst-case running times, although no algorithm has a linear running time if the graph is directed and the computational model is well established. In the pointer model of computation, the Fibonacci heap data structure of Fredman and Tarjan [16] leads to an $O(m + n \log n)$ implementation of Dijkstra's algorithm. In a RAM model with unit-time word operations, the fastest currently known algorithms achieve the following bounds: $O(m + n(\log U \log\log U)^{1/3})$ [39], $O(m + n(\sqrt{\log n}))$ [38], $O(m \log\log U)$ [25], and $O(m \log \log n)$ [43].

For undirected graphs, Thorup's algorithm [42] has a linear running time in a word RAM model. A constant-time priority queue of [3] yields a linear-time algorithm for directed graphs, but only in a nonstandard computation model that is not supported by any existing computer.

Our work has been motivated by a recent paper of Meyer [30]. The paper gives an NSP algorithm with a linear average time for input arc lengths drawn independently

*Received by the editors November 21, 2001; accepted for publication (in revised form) July 29, 2007; published electronically February 14, 2008. Preliminary versions of theoretical and experimental results presented in this paper appeared in [21, 22] and [23], respectively.
http://www.siam.org/journals/sicomp/37-5/69877.html
†Microsoft Research, 1075 la Avenida, Mountain View, CA 94043 (goldberg@microsoft.com).

from a uniform distribution on $[1, \ldots, M]$. It also proves that, under the same assumptions, the running time is linear with high probability (w.h.p.). Meyer's algorithm may scan some vertices more than once, and its worst-case time bound, $O(nm \log n)$, is far from linear. Both the algorithm and its analysis are complicated.

In this paper we show that a natural improvement of the *multilevel bucket (MLB)* shortest path algorithm of [9] has an average running time that is linear and a worst-case time of $O(m+n \log C)$. Our average-time bound holds for arc lengths distributed uniformly on $[1, \ldots, M]$; the lengths do not need to be independent. We also show that if the lengths are independent, the algorithm running time is linear w.h.p. We refer to the new algorithm as the *smart queue algorithm*. Unlike the MLB algorithm, our algorithm is not an implementation of Dijkstra's algorithm: a vertex selected for scanning is not necessarily a minimum labeled vertex. However, the selected vertex distance label is equal to the correct distance, and each vertex is scanned at most once. A similar relaxation of Dijkstra's algorithm was originally introduced by Dinic [13] and used in its full strength by Thorup [42].

A recent paper of Hagerup [26] gives a different NSP algorithm with linear expected time. This algorithm uses the power of word operations to a greater extent than our algorithm, which allows the use of a simpler bucket structure.

The practical importance of the NSP problem motivated extensive computational work. Implementations of Dijkstra's algorithm based on the classical binary heap data structure [45] were often used as a benchmark to compare other work against. This algorithm is easy to code and has reasonably constant factors.[1] In practice binary heaps usually outperform Fibonacci heaps [16] and often outperform pairing heaps [40]. However, the running time of the algorithm based on binary heaps grows superlinearly with the number of elements on the heap, and the algorithm is not hard to beat on bigger problems.

In many practical problems, label-correcting algorithms of Pape [36], Pallottino [35], and Glover and Klingman [20] perform extremely well. However, these algorithms have bad worst-case time bounds and sometimes perform poorly. Furthermore, unlike implementations of Dijkstra's algorithm and the smart queue algorithm, these algorithms cannot be terminated early if one desires a shortest path between a pair of vertices.

Dail's bucket-based algorithm [10] works well if U is small. Cherkassky, Goldberg, and Radzik [5] show that the two-level bucket algorithm has a reasonable performance for moderately large U. Further theoretical and experimental work [6, 24] produced more robust two- and three-level implementations. Although they have much better worst-case performance than the label-correcting algorithms mentioned above, these MLB implementations are somewhat slower on many practical problems. For example, based on real-life road networks, Zhan and Noon [46] recommend Pallottino's algorithm for the NSP problem and bucket-based algorithms for the single pair problem.

We describe a practical implementation of the smart queue algorithm. As a step toward this implementation, we develop a new implementation of the MLB algorithm that is more efficient than the previous implementations if the number of bucket levels is large. A simple modification of this implementation yields an efficient implementation of the smart queue algorithm. Our experimental results show that the smart queue algorithm with a large number of levels is practical. In particular, an implementation with the number of levels optimized for the worst-case theoretical performance

[1] Implementations based on 4-heaps have better constants.

works well on both typical and bad-case inputs. For 32-bit arc lengths, the code runs in time less than 2.5 times that of breadth-first search for all inputs we tried. These inputs included ones designed to be hard for our implementation. Our results lead to a better understanding of NSP algorithm implementations and show how close their performance is to the lower bound provided by breadth-first search.

Our algorithm has been motivated by a theoretical average-case analysis. The fact that the algorithm is also practical is an interesting example of a situation where probabilistic analysis leads to improved practical performance.

2. Preliminaries. The input to the NSP problem is a directed graph $G = (V, A)$ with n vertices, m arcs, a source vertex s, and nonnegative arc lengths $\ell(a)$. The goal is to find shortest paths from the source to all vertices of the graph. Unless mentioned otherwise, we assume that arc lengths are integers in the interval $[1, \ldots, U]$, where U denotes the biggest arc length. Let δ be the smallest nonzero arc length, and let C be the ratio of the biggest arc length to δ. If all arc lengths are zero or if $C < 2$, then the problem can be solved in linear time [13] (for arc lengths in the range $[L, U]$ with $U < 2L$, the linear-time algorithm uses width L single-level buckets). Without loss of generality, we assume that $C \geq 2$ and $\log C \geq 1$, which is the only technical reason for making the assumption, as it simplifies the bounds. This implies that $\log U \geq 1$. We say that a statement holds *with high probability (w.h.p.)* if the probability that the statement is true approaches one as $m \to \infty$.

We assume the *word RAM* model of computation (see, e.g., [1]). To efficiently implement the MLB data structure [9], we need array addressing and the following unit-time word operations: addition, subtraction, comparison, and arbitrary shifts. To allow a higher-level description of our algorithm, we use a *strong RAM* computation model that also allows word operations including bitwise logical operations and the operation of finding the index of the most significant bit in which two words differ. The latter operation is in AC0; see [8] for a discussion of a closely related operation. The use of this more powerful model does not improve the amortized operation bounds but simplifies the description.

3. Labeling method and related results. The labeling method for the shortest path problem [14, 15] works as follows (see, e.g., [41]). The method maintains for every vertex v its distance label $d(v)$, parent $p(v)$, and status $S(v) \in \{$`unreached`, `labeled`, `scanned`$\}$. Initially $d(v) = \infty$, $p(v) = nil$, and $S(v) =$ `unreached`. The method starts by setting $d(s) = 0$ and $S(s) =$ `labeled`. While there are labeled vertices, the method picks such a vertex v, scans all arcs out of v, and sets $S(v) =$ `scanned`. To scan an arc (v, w), one checks if $d(w) > d(v) + \ell(v, w)$ and, if true, sets $d(w) = d(v) + \ell(v, w)$, $p(w) = v$, and $S(w) =$ `labeled`.

If the length function is nonnegative, the labeling method always terminates with correct shortest path distances and a shortest path tree. The efficiency of the method depends on the rule to choose a vertex to scan next. We say that $d(v)$ is *exact* if the distance from s to v is equal to $d(v)$. It is easy to see that if the method always selects a vertex v such that, at the selection time, $d(v)$ is exact, then each vertex is scanned at most once.

Dijkstra [12] observed that if ℓ is nonnegative and v is a labeled vertex with the smallest distance label, than $d(v)$ is exact. However, a linear-time implementation of Dijkstra's algorithm in the strong RAM model is at least as hard as linear-time sorting. Dinic [13] and Thorup [42] use a relaxation of Dijkstra's selection rule to get linear-time algorithms for special cases of the NSP problem. To describe a related

relaxation that we use, define the *caliber* of a vertex v, $c(v)$, to be the minimum length of an arc entering v, or infinity if no arc enters v.

LEMMA 3.1 (caliber lemma). *Suppose ℓ is nonnegative, and let μ be a lower bound on distance labels of labeled vertices. Let v be a vertex such that $\mu + c(v) \geq d(v)$. Then $d(v)$ is exact.*

The lemma follows from the observation that for any labeled vertex u, such that $(u, v) \in A$, $d(u) + \ell(u, v) \geq \mu + c(v) \geq d(v)$.

4. Algorithm description and correctness. Our algorithm is based on the MLB implementation of Dijkstra's algorithm modified to use Lemma 3.1 to detect and scan vertices with exact (but not necessarily minimum) distance labels. Our algorithm is a labeling algorithm. During the initialization, the algorithm also computes $c(v)$ for every vertex v. The algorithm keeps labeled vertices in one of two places: a set F and a priority queue B. The former is implemented to allow constant time additions and deletions, for example, as a doubly linked list. The latter is implemented using multi-level buckets as described below. The priority queue supports operations `insert`, `delete`, `decrease-key`, and `extract-min`. However, the `insert` operation inserts vertices into either B or F, and the `decrease-key` operation may move vertices from B to F.

At a high level, the algorithm works as follows. Vertices in F have exact distance labels and if F is nonempty, we remove and scan a vertex from F. If F is empty, we remove and scan a vertex from B with the minimum distance label. Suppose a distance label of a vertex u decreases. Note that u cannot belong to F. If u belongs to B, then we apply the `decrease-key` operation to u. This operation either relocates u within B or discovers that u's distance label is exact and moves u to F. If u is in neither B nor F, we apply the `insert` operation to u, and u is inserted either into B or, if $d(u)$ is determined to be exact, into F.

Next we describe the bucket structure B. For a given integer parameter $\Delta \geq 2$, B contains $k + 1$ levels of buckets, where $k = \lceil \log_\Delta U \rceil$. Except for the top level, a level contains Δ buckets. Conceptually, the top level contains infinitely many buckets. However, at any given time all buckets on that level are empty except at most three consecutive ones (see Lemma 4.1 below), and one can maintain only these buckets by wrapping around modulo three at the top level.[2] We denote bucket j at level i by $B(i, j)$; i ranges from 0 (bottom level) to k (top), and j ranges from 0 to $\Delta - 1$, except at the top level discussed above. A bucket contains a set of vertices maintained in a way that allows constant-time insertion and deletion, e.g., in a doubly linked list. At each level i, we maintain the number of vertices at this level.

We maintain μ such that μ is a lower bound on the distance labels of labeled vertices. Initially $\mu = 0$. Every time an `extract-min` operation removes a vertex v from B, we set $\mu = d(v)$. Consider the base Δ representation of the distance labels and number digit positions starting from 0 for the least significant digit. Let $\mu_{i,j}$ denote the ith through jth least significant digit of μ, and let μ_i denote the ith least significant digit. Similarly, $d_i(u)$ denotes the ith least significant digit of $d(u)$, and likewise for the other definitions. Note that μ and the $k + 1$ least significant digits of the base Δ representation of $d(u)$ uniquely determine $d(u)$: $d(u) = \mu + (d_{0,k} - \mu_{0,k})$ if $d_{0,k} > \mu_{0,k}$ and $d(u) = \mu + \Delta^k + (d_{0,k} - \mu_{0,k})$ otherwise.

For a given μ, let $\underline{\mu_i}$ and $\overline{\mu_i}$ be μ with the i least significant digits replaced by 0 or $\Delta - 1$, respectively. Each level $i < k$ corresponds to the range of values $[\underline{\mu_{i+1}}, \overline{\mu_{i+1}}]$.

[2] For low-level efficiency, one may want Δ to be a power of two and wrap around modulo four.

Each bucket $B(i,j)$ corresponds to the subrange containing all integers in the range with the ith digit equal to j. At the top level, a bucket $B(k,j)$ corresponds to the range $[j \cdot \Delta^k, (j+1) \cdot \Delta^k)$. The *width* of a bucket at level i is equal to Δ^i: the bucket contains Δ^i distinct values. We say that a vertex u *is in the range of* $B(i,j)$ if $d(u)$ belongs to the range corresponding to the bucket.

The position of a vertex u in B depends on μ: u belongs to the lowest-level bucket containing $d(u)$. More formally, let i be the index of the most significant digit in which $d(u)$ and $\mu_{0,k}$ differ, or 0 if they match. Note that $\underline{\mu_i} \le d(u) \le \overline{\mu_i}$. Given μ and u with $d(u) \ge \mu$, we define the *position of* u by $(i, d_i(u))$ if $i < k$ and $B(k, \lfloor (d(u) - \mu)/\Delta^k \rfloor)$ otherwise. If u is inserted into B, it is inserted into $B(i,j)$, where (i,j) is the position of u. For each vertex in B, we store its position.

Figure 1 gives an example of the bucket structure. In this example, $\Delta = 2$, $k = 3$, and $\mu = 10$. For instance, to find the position of a vertex v with $d(v) = 14$, we note that the binary representations of 10 and 14 differ in bit 2 (remember that we start counting from 0) and the bit value is 1. Thus v belongs to bucket 1 at level 2.

Our modification of the MLB algorithm uses Lemma 3.1 during the `insert` operation to put vertices into F whenever the lemma allows it. The details are as follows.

`insert`. Insert a vertex u into $B \cup F$ as follows. If $\mu + c(u) \ge d(u)$, put u into F. Otherwise compute u's position (i,j) in B and add u to $B(i,j)$.

`decrease-key`. Decrease the key of an element u in position (i,j) as follows. Remove u from $B(i,j)$. Set $d(u)$ to the new value and insert u as described above.

`extract-min`. Find the lowest nonempty level i. Find j, the first nonempty bucket at level i, by setting j to the index of the bucket at level i that contains μ, and increment j until the bucket $B(i,j)$ is nonempty. If $i = 0$, delete a vertex u from $B(i,j)$. (In this case $\mu = d(u)$.) Return u. If $i > 0$, examine all elements of $B(i,j)$ and delete a minimum element u from $B(i,j)$. Note that in this case $\mu < d(u)$; set $\mu = d(u)$. Since μ increased, some vertex positions in B may have changed. We do *bucket expansion* of $B(i,j)$ and return u.

To understand bucket expansion, note that the vertices with changed positions are exactly those in $B(i,j)$. To see this, let μ' be the old value of μ and consider a vertex v in B. Let (i',j') be v's position with respect to μ'. By the choice of $B(i,j)$, if $(i,j) \ne (i',j')$, then either $i < i'$ or $i = i'$ and $j < j'$. In both cases, the common prefix of μ' and $d(v)$ is the same as the common prefix of $d(u)$ and $d(v)$, and the position of v does not change.

On the other hand, vertices in $B(i,j)$ have a longer common prefix with $d(u)$ than they have with μ' and these vertices need to move to a *lower* level. Bucket expansion

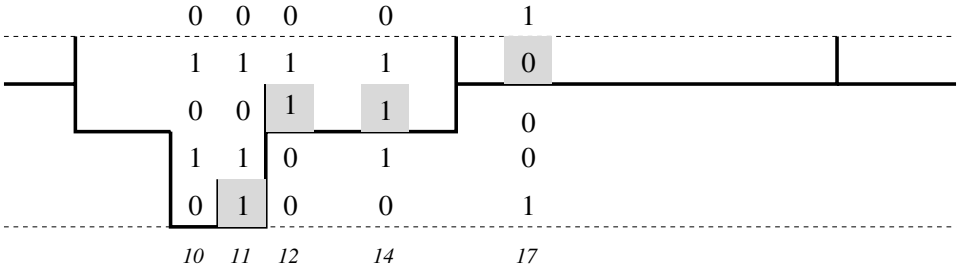

FIG. 1. *MLB example.* $\Delta = 2$, $k = 3$, $\mu = 10$. *Values on the bottom are in decimal. Values on top are in binary, with the least significant bit on the bottom. Shaded bits determine positions of the corresponding elements.*

deletes these vertices from $B(i)$ and uses the `insert` operation to add the vertices back into B or into F, as appropriate. The vertex u in $B(i,j)$ with the minimum distance label is not inserted into B but is returned and scanned. Note that we do bucket expansions only when F is empty and the expanded bucket contains a labeled vertex with the minimum distance. Thus μ is updated correctly.

Although the formal description of the algorithm is nontrivial, the algorithm itself is simple: At each step, remove a vertex from F or, if F is empty, then remove the minimum-labeled vertex from B. In the latter case, expand the bucket from which the vertex has been removed, if necessary. Scan the vertex and update its neighbors if necessary. Terminate when both F and B are empty.

The width of a top-level bucket is at least U. In the original MLB algorithm, at any point of the execution all labeled vertices are contained in at most two consecutive top-level buckets. A slightly weaker result holds for our algorithm.

LEMMA 4.1. *At any point of the execution, all labeled vertices are in the range of at most three consecutive top-level buckets.*

Proof. Let μ' be the current value of μ and let $B(k,j)$ be the top-level bucket containing μ'. Except for s (for which the result holds trivially), a vertex v becomes labeled during a scan of another vertex u removed from either B or F. In the former case, at the time of the scan $d(u) = \mu \leq \mu'$, $d(v) = \mu + \ell(u,v) \leq \mu' + U$, and therefore v is contained in either $B(k,j)$ or $B(k,j+1)$. In the latter case, when u has been added to F, the difference between $d(u)$ and μ was at most $c(u) \leq U$, and thus $d(u) \leq \mu' + U$, $d(v) \leq d(u) + U \leq \mu' + 2 \cdot U$, and thus v belongs to $B(k,j)$, $B(k,j+1)$, or $B(k,j+2)$. □

Algorithm correctness follows from Lemmas 3.1 and 4.1 and the observations that μ is always set to the minimum distance label of a labeled vertex, μ remains a lower bound on the labeled vertex labels (and therefore is monotonically nondecreasing), and F always contains vertices with exact distance labels.

Remark. An alternative interpretation of the algorithm is as follows. For each vertex v added to F, we apply a potential transformation as follows. Let $\delta = d(v) - \mu$, where the value of μ is taken at the time v is added to F. We reduce the length of all arcs into v by δ and increase the length of all arcs out of v by δ. Note that the transformed lengths are nonnegative as $d(v) - \mu \leq c(v)$. With respect to the transformed lengths, our algorithm follows Dijkstra's rule of always scanning a vertex with the minimum distance label. This interpretation gives an alternative proof of correctness of the algorithm.

5. Worst-case analysis. In this section we prove a worst-case bound on the running time of the algorithm. Our analysis is similar to that for the MLB algorithm given in [6], except the resulting bound has a C instead of a U. Some definitions and lemmas introduced in this section will also be used in the next section.

We start the analysis with the following lemmas.

LEMMA 5.1 (see [6]).

- *Given μ and u, we can compute the position of u with respect to μ in constant time.*
- *We can find the lowest nonempty level of B in constant time.*

LEMMA 5.2. *The algorithm runs in $O(m + n + \Phi_1 + \Phi_2)$ time, where Φ_1 is the total number of times a vertex moves from a bucket of B to a lower-level bucket and Φ_2 is the number of empty buckets examined by the algorithm.*

Proof. Since each vertex is scanned at most once, the total scan time is $O(m+n)$. A vertex is added to and deleted from F at most once, so the total time devoted

to maintaining F is $O(n)$. An `insert` operation takes constant time, and these operations are caused by inserting vertices into B for the first time by `decrease-key` operations and by `extract-min` operations. The former take $O(n)$ time; we account for the remaining ones jointly with the other operations. A `decrease-key` operation takes constant time and is caused by a decrease of $d(v)$ due to a scan of an arc (u, v). Since an arc is scanned at most once, these operations take $O(m)$ total time. The work we accounted for so far is linear.

Next we consider the `extract-min` operations. Consider an `extract-min` operation that returns u. The operation takes $O(1)$ time, plus the time proportional to the number of empty buckets examined, plus the time proportional to the number of vertices in the expanded bucket, excluding u. Each of these vertices moves to a lower level in B. Thus we get the desired time bound. ∎

Note that $\Phi_1 = O(nk)$ and $\Phi_2 = O(n\Delta)$ since after examining less than Δ empty buckets we discover a nonempty one and scan a vertex from it. Setting Δ to $\lceil \frac{\log U}{\log\log U} \rceil$ balances Φ_1 and Φ_2 and yields the $O(m + n\frac{\log U}{\log\log U})$ worst-case time bound. To get a better bound, we define $k' = \lfloor \log_\Delta \delta \rfloor$.

LEMMA 5.3. *Buckets at level k' and below are never used.*

Proof. Let (i, j) be the position of a vertex v of caliber $c(v) \geq \delta$. If $i \leq k'$, then $d(v) - \mu < \Delta^i \leq \Delta^{k'} \leq \delta \leq c(v)$ and the algorithm adds v to F, not B. ∎

The lemma implies that the algorithm uses $O(\log_\Delta U - \log_\Delta \delta) = O(\log_\Delta C)$ bucket levels.

Setting Δ to $\lceil \frac{\log C}{\log\log C} \rceil$ and applying Lemma 5.3, we get the following result.

THEOREM 5.4. *The worst-case running time of the algorithm is $O(m+n\frac{\log C}{\log\log C})$.*

Note that setting $\Delta = 2$ yields an $O(m + n \log C)$ bound.

Our optimization can also be used to improve other data structures based on multilevel buckets, such as radix heaps [2] and hot queues [6]. For these data structures, the equivalent of Lemma 5.3 allows one to replace time bound parameter U by C. In particular, the bound of the hot queue implementation of Raman [39] improves to $O(m + n(\log C \log\log C)^{1/3})$. The modification of Raman's algorithm to obtain this bound is straightforward given the results of this section.

6. Average-case analysis. In this section we prove that for $\Delta = 2$ (or any other constant), the smart queue algorithm runs in linear average time under the assumption that the input arc lengths are uniformly distributed on $[1, \ldots, M]$.[3] We also show that the running time is linear w.h.p. with the additional assumption that the lengths are independent.

Since $\Delta = 2$, $\Phi_2 = O(n)$; it remains to bound Φ_1.

A key lemma for our analysis is as follows.

LEMMA 6.1. *The algorithm never inserts a vertex v into a bucket at a level less than or equal to* $\log c(v) - 1$.

Proof. Suppose during an `insert` operation, v's position in B is (i, j) with $i \leq \log c(v) - 1$. Then the most significant digit in which $d(v)$ and μ differ is digit i and $d(v) - \mu < \Delta^{i+1} \leq c(v)$. Therefore `insert` puts v into F, not B. ∎

The above lemma motivates the following definitions. The *weight of an arc* a, $w(a)$, is defined by $w(a) = k - \lfloor \log \ell(a) \rfloor$. The *weight of a vertex* v, $w(v)$, is defined to be the maximum weight of an incoming arc or zero if v has no incoming arcs.

[3] As we shall see, if M is large enough, then the result also applies to the range $[0, \ldots, M]$.

Lemma 6.1 implies that the number of times v can move to a lower level of B is at most $w(v)+1$, and therefore $\Phi_1 \leq m + \sum_V w(v)$. Note that k depends on the input, and thus the weights are defined with respect to a given input.

For the probability distribution of arc weights defined above, we have $\mathbf{Pr}[\lfloor \log \ell(a) \rfloor = i] = 2^i/M$ for $i = 0, \ldots, k-1$. The definition of w yields

$$\mathbf{Pr}[w(a) = t] = 2^{k-t}/M \quad \text{for } t = 1, \ldots, k. \tag{1}$$

Since $M \geq U$, we have $M \geq 2^{k-1}$, and therefore

$$\mathbf{Pr}[w(a) = t] \leq 2^{-t+1} \quad \text{for } t = 1, \ldots, k. \tag{2}$$

THEOREM 6.2. *If arc lengths are uniformly distributed on* $[1, \ldots, M]$*, then the average running time of the algorithm is linear.*

Proof. Since $\Phi \leq m + \sum_V w(v)$, it is enough to show that $\mathbf{E}[\sum_V w(v)] = O(m)$. By the linearity of expectation and the definition of $w(v)$, we have $\mathbf{E}[\sum_V w(v)] \leq \sum_A \mathbf{E}[w(a)]$. The expected value of $w(a)$ is

$$\mathbf{E}[w(a)] = \sum_{i=1}^{k} i\mathbf{Pr}[w(a) = i] \leq \sum_{i=1}^{\infty} i2^{-i+1} = 2\sum_{i=1}^{\infty} i2^{-i} = O(1).$$

Note that this bound holds for any k. Thus $\sum_A \mathbf{E}[w(a)] = O(m)$. ∎

Remark. Note that Theorem 6.2 does not require arc lengths to be independent. Our proof of its high-probability variant, Theorem 6.7, requires this independence.

Remark. The proof of the theorem works for any arc length distribution such that $\mathbf{E}[w(a)] = O(1)$. In particular, the theorem holds for real-valued arc lengths selected uniformly from $(0, 1]$. (We exclude zero so that w is well defined.) In fact, for this distribution the high-probability analysis below is simpler (given the independence assumption). However, the integer distribution is somewhat more interesting, because some test problem generators use this distribution and most practical problems have integral lengths.

Next we show that the algorithm running time is linear w.h.p. by showing that $\sum_A w(a) = O(m)$ w.h.p. First, we show that w.h.p. U is not much smaller than M and δ is close to Mm^{-1} (Lemmas 6.3 and 6.4). Let S_t be the set of all arcs of weight t and note that $\sum_A w(a) = \sum_t t|S_t|$. We show that as t increases, the expected value of $|S_t|$ goes down exponentially. For small values of t, the value of $|S_t|$ is also bounded by an exponentially decreasing function w.h.p. To deal with large values of t, we show that the total number of arcs with large weights is small, and so is the contribution of these arcs to the sum of arc weights.

Proofs of the following two lemmas are fairly standard; we include them to make the paper self-contained.

LEMMA 6.3. *W.h.p.,* $U \geq M/2$.

Proof. For an arc a, $\mathbf{Pr}[\ell(a) < M/2] < 1/2$, and by the independence of arc lengths,

$$\mathbf{Pr}[U < M/2] \leq 2^{-m} \to 0 \quad \text{as } m \to \infty.$$

Thus $\mathbf{Pr}[U \geq M/2] \to 1$ as $m \to \infty$. ∎

LEMMA 6.4. *W.h.p.,* $\delta \geq Mm^{-4/3}$. *If* $M \geq m^{2/3}$, *then w.h.p.* $\delta \leq Mm^{-2/3}$.

Proof. For an arc a, we have

$$\mathbf{Pr}\Big[\ell(a) \geq Mm^{-4/3}\Big] = (M - Mm^{-4/3})/M \geq 1 - m^{-4/3}.$$

Since the arc lengths are independent, we have

$$\mathbf{Pr}\Big[\delta \geq Mm^{-4/3}\Big] \geq (1 - m^{-4/3})^m \to 1 \quad \text{as } m \to \infty.$$

Similarly,

$$\mathbf{Pr}\Big[\ell(a) > Mm^{-2/3}\Big] = (M - \lfloor Mm^{-2/3} \rfloor)/M \leq 1 - \frac{m^{-2/3}}{2},$$

and by the independence

$$\mathbf{Pr}\Big[\delta > Mm^{-2/3}\Big] \geq \left(1 - \frac{1}{2m^{2/3}}\right)^m \to 0 \quad \text{as } m \to \infty. \qquad \square$$

From (1), we have $\mathbf{E}[|S_t|] = m2^{k-t}/M$. Since the weights are independent, we can apply the Chernoff bound (see, e.g., [7, 33]) to conclude that $\mathbf{Pr}\big[|S_t| \geq 2m2^{k-t}/M\big] < (\frac{e}{4})^{m2^{k-t}/M}$. Since $M \geq 2^{k-1}$, we have

$$\mathbf{Pr}\big[|S_t| \geq 4m2^{-t}\big] < \left(\frac{e}{4}\right)^{2m2^{-t}}.$$

As mentioned above, we bound the contributions of arcs with large and small weights to $\sum_A w(a)$ differently. We define $\beta = \log(m^{2/3})$ and partition A into two sets—A_1, containing the arcs with $w(a) \leq \beta$, and A_2, containing the arcs with $w(a) > \beta$.

LEMMA 6.5. $\sum_{A_1} w(a) = O(m)$ *w.h.p.*

Proof. Assume that $\delta \geq Mm^{-4/3}$ and $U \geq M/2$; by Lemmas 6.3 and 6.4 this happens w.h.p. This assumption implies $C \leq m^{4/3}$. The probability that for some t, $1 \leq t \leq \beta$, $|S_t| \geq 4m2^{-t}$ is, by the union bound and the fact that the probability is maximized for $t = \beta$, less than

$$\beta \left(\frac{e}{4}\right)^{m2^{-\beta}} \leq \log(m^{2/3}) \left(\frac{e}{4}\right)^{mm^{-2/3}} \leq \log m \left(\frac{e}{4}\right)^{m^{1/3}} \to 0 \quad \text{as } m \to \infty.$$

Thus w.h.p., for all t, $1 \leq t \leq \beta$, we have $|S_t| < 4m2^{-t}$ and

$$\sum_{A_1} w(a) = \sum_{t=1}^{t \leq \beta} t|S_t| \leq 4m \sum_{t=1}^{\infty} t2^{-t} = O(m). \qquad \square$$

LEMMA 6.6. $\sum_{A_2} w(a) = O(m)$ *w.h.p.*

Proof. If $M < m^{2/3}$, then $k \leq \beta$ and A_2 is empty, so the lemma holds trivially.

Now consider the case $M \geq m^{2/3}$. By Lemmas 6.3 and 6.4, w.h.p. $Mm^{-4/3} \leq \delta \leq Mm^{-2/3}$ and $U \geq M/2$; assume that this is the case. The assumption implies $m^{2/3}/2 \leq C \leq m^{4/3}$. Under this assumption, we also have $2^{k-1} \leq M \leq 2^{k+1}$. Combining this with (1) we get $2^{-2-t} \leq \mathbf{Pr}[w(a) = t] \leq 2^{1-t}$. This implies that

$$2^{-2-\beta} \leq \mathbf{Pr}[w(a) > \beta] \leq 2^{2-\beta};$$

therefore,

$$\frac{m^{-2/3}}{8} \leq \mathbf{Pr}[w(a) > \beta] \leq 4m^{-1/3}$$

and

$$\frac{m^{1/3}}{8} \leq \mathbf{E}[|A_2|] \leq 4m^{2/3}.$$

Using the independence and applying the Chernoff bound, we get

$$\mathbf{Pr}[|A_2| > 2\mathbf{E}[|A_2|]] < \left(\frac{e}{4}\right)^{\mathbf{E}[|A_2|]}.$$

Replacing the first occurrence of $\mathbf{E}[|A_2|]$ by the upper bound on its value and the second occurrence by the lower bound (since $e/4 < 1$), we get

$$\mathbf{Pr}\left[|A_2| > 8m^{2/3}\right] < \left(\frac{e}{4}\right)^{m^{1/3}/8} \to 0 \quad \text{as } m \to \infty.$$

For all arcs a, $\ell(a) \geq \delta$, and thus

$$w(a) = k - \lfloor \ell(a) \rfloor \leq 1 + \log U + 1 - \log \delta = 2 + \log C \leq 2 + (4/3)\log m.$$

Therefore, w.h.p.,

$$\sum_{A_2} w(a) \leq 8m^{2/3}(2 + (4/3)\log m) = o(m). \qquad \square$$

Thus we have the following theorem.

THEOREM 6.7. *If arc lengths are independent and uniformly distributed on* $[1, \ldots, M]$*, then w.h.p., the algorithm runs in linear time.*

Remark. The expected and high-probability bounds also apply if the arc lengths come from $[0, \ldots, U]$ and $U = \omega(m)$, as in this case w.h.p. no arc has zero length.

7. Algorithm implementation. In this section we describe efficient implementations of the MLB algorithm and smart queue algorithms, including algorithm engineering considerations involved in their development. We also discuss how the MLB implementation differs from the previous implementations [6, 24]. The MB code implements the algorithm MLB algorithm, and the SQ code implements the smart queue algorithm.

Previous implementations of the MLB algorithm, as well as those described below, use the following *wide bucket* heuristic. Pick w such that $0 < w \leq \delta$. Then the MLB algorithm remains correct if one multiplies the bucket width on every level by w. Both MB and SQ codes use the wide bucket heuristic. We refer to the modifications needed to convert the MLB algorithm to the smart queue algorithm as the *caliber heuristic*. This heuristic is the only difference between MB and SQ.

Our implementation of MB is very similar to that of [24], except in the details of the `insert` operation. The previous implementation maintained a range of distance values for each level, updating the ranges when the value of μ changed. To insert a vertex, one looks for the lowest level to which the vertex belongs and then computes the offset of the bucket to which the vertex belongs. In contrast, MB and SQ compute the vertex position with respect to μ as described in section 4. This is slightly more

efficient when the number of levels is large. The efficiency gain is bigger for SQ because it does not necessarily examine all levels for a given value of μ. The new implementation is also simpler than the old one.

We always set Δ to a power of two. This allows us to use bit shifts instead of divisions. Our codes set w to the biggest power of two not exceeding δ, or to one if $\delta \leq 1$. We use an array to represent each level of buckets.

One can give MB either k or Δ as a parameter. Then MB sets the other parameter based on the input arc lengths. We refer to the code with the number of levels k set to two by MB2L, and to the code with Δ set to two by MB2D. These are the two extreme cases that we study. (We do not study the single-level case because it often would have needed too much memory and time.) Alternatively, one can let MB choose the values of both k and Δ based on the input. We refer to this adaptive variant as MB-A. The adaptive variant of the algorithm uses the relationship $\Delta = \Theta(k)$ suggested by the worst-case analysis. To choose the constant hidden by the Θ location, we observe the following. Examining empty buckets involves looking at a single pointer and has good locality properties as we access the buckets sequentially. Moving vertices to lower levels, on the other hand, requires changing several pointers, and has poor locality. This suggests that Δ should be substantially greater than k, and experiments confirm this.

In more detail, MB-A sets k and Δ as follows. First we find the smallest value of k such that k is a power of two and $(16k)^k \geq U/w$. Then we set Δ to $16k$. At this point, however, both Δ and k may be larger than they need to be. While $(\Delta/2)^k \geq U/w$ we reduce Δ. Finally, while $(\Delta)^{k-1} \geq U/w$ we reduce k. This typically leads to $16k \leq \Delta \leq 128k$ and works well in our tests.

We obtain our SQ code by adding the caliber heuristic to MB. The modification of MB is relatively straightforward. We use a stack to implement the set F needed by the caliber heuristic. The adaptive variant of the code, SQ-A, uses the same procedure to set k and Δ as MB-A does.

8. Experimental methodology and setup. Following Moret and Shapiro [32], we use a baseline code—breadth-first search (BFS) in our case—and measure running times of our shortest path codes on an input relative to the BFS running time on this input. Our BFS code computes distances and a shortest path tree for the unit length function. The breadth-first search problem is a special case of NSP and, modulo low-level details, the BFS running time is a lower bound on the NSP codes. Baseline running times give a good indication of how close to optimal the running times are and reduces dependency on low-level implementation and architecture details.

However, some of the dependencies, in particular, cache dependencies, remain. Our codes put arc and vertex records in consecutive locations. Input IDs of the vertices determine their ordering in memory. In general, breadth-first search examines vertices in a different order than an NSP algorithm. This may—and in some cases does—lead to very different caching behavior of the two codes for certain vertex orderings. To deal with this dependency on input IDs, our generators permute the IDs at random. Thus all our problem generators are randomized.

For every input problem type and any set of parameter values, we run the corresponding generator five times and report the averages. We report the baseline BFS time in seconds and all other times in units of the BFS time. In addition, we count operations that determine Φ_1 and Φ_2 in Lemma 5.2. For each of these operations, we give the number of the operations divided by the number of vertices so that the

amortized operation cost is immediate. The two kinds of operations we count are examinations of empty buckets and the number of vertices processed during bucket expansion operations.

We use 64-bit integers for internal representation of arc lengths and distances. If the graph contains simple paths longer than 2^{64}, our codes may get overflows. Note that for 32-bit input arc lengths, no overflow can happen unless the number of vertices exceeds 2^{32}, which is too many to fit into the memory of a modern computer.

Our experiments have been conducted on a 933 MHz Pentium III machine with 512M of memory, 256K cache, and running RedHat Linux 7.1. All our shortest path codes and the baseline code are written in C++, in the same style, and compiled with the `gcc` compiler using the `-O6` optimization option. Our BFS code uses the same data structures as the MB code.

9. Problem families. We report data on seven problem families produced by three problem generators. These problem families have been selected as the most interesting from many more families we experimented with. Since we are interested in the efficiency of the data structures, we restrict our study to sparse graphs, for which the data structure manipulation time is most apparent.

Our first generator, SPRAND, builds a Hamiltonian cycle and then adds arcs at random. The generator may produce parallel arcs but not self-loops. Arc lengths are chosen independently and uniformly from $[\ell, u]$. Vertex 1 is the source. If the number of arcs is large enough, SPRAND graphs are expanders and the average number of vertices in the priority queue during a shortest path computation is large.

We use SPRAND to generate two problem families, RAND-I and RAND-C. For both families, $\ell = 1$ and $m = 4n$. For RAND-I, $u = n$, and n increases by a factor of two from one set of parameter values to the next one. We chose the initial value of n large enough so that the running time is nonnegligable and the final value is as large as possible subject to the constraint that all our codes run without paging. For RAND-C, $n = 2^{20}$ and $u = 2^i$; i starts at 1 and then takes on integer multiples of four from 4 to 32. Up to $i = 20$, the minimal arc length δ in all test inputs is one. For $i = 24$, δ is greater than one for some inputs. For $i = 28$ and 32, δ is always greater than one. Note that the expected value of C does not change for $i \geq 28$, and therefore the results for $i > 32$ would have been very similar to those for $i = 28$ and 32.

Our second generator, SPGRID, produces grid-like graphs. An x, y grid graph contains $x \cdot y$ vertices, $[i, j]$, for $0 \leq i < x$ and $0 \leq j < y$. A vertex $[i, j]$ is connected to the adjacent vertices in the same layer, $[i, j + 1 \bmod y]$ and $[i, j - 1 \bmod y]$. In addition, for $i < x-1$, each vertex $[i, j]$ is connected to the vertex $[i+1, j]$. Arc lengths are chosen independently and uniformly from $[\ell, u]$. Vertex $[0, 0]$ is the source. We use SPGRID to generate two problem families, LONG-I and LONG-C. Both families contain *long* grid graphs with $y = 8$ and x a parameter. For these graphs, the average number of vertices in the priority queue is small.

The LONG-I and LONG-C problem families are similar to the RAND-I and RAND-C families. For LONG-I, $u = n$, and n increases by a factor of two from the value that yields a reasonable running time to the maximum value that does not cause paging. The LONG-C problem family uses the same values of u as the RAND-C problem family.

Our last problem generator is SPHARD. This generator produces problems aimed to be hard for MLB algorithms for certain values of k and Δ. Graphs produced by this generator consist of $2k + 1$ vertex-disjoint paths, with the source connecting to the beginning of each path. (See Figure 2 for an example.) These paths have the same

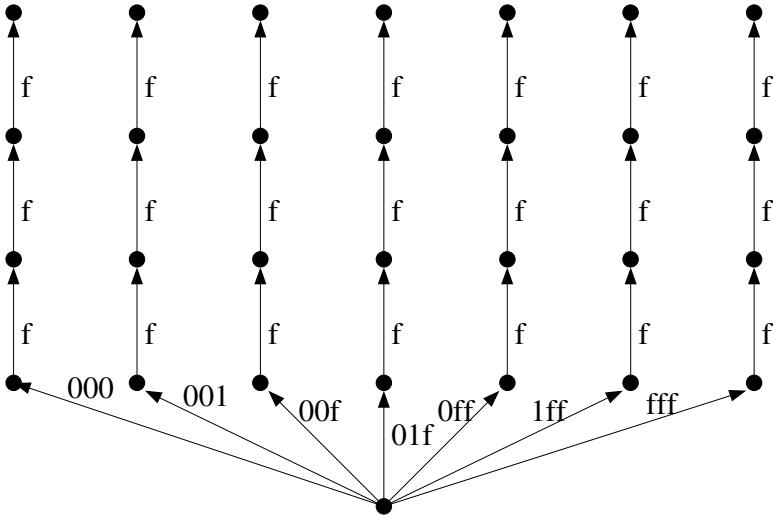

FIG. 2. *An example of a hard problem instance;* $k = 3$ *and* $\Delta = 16$. *Arc lengths are given in hexadecimal. We omit the extra vertex with arcs designed to manipulate vertex calibers.*

number of arcs, which can be adjusted to get a graph of the desired size. Path arcs have a length of Δ. The lengths of the source arcs are as follows. One arc has zero length. Out of the remaining arcs, k arcs have the following base-Δ representation. For $1 \le i \le k$, the first i digits are $\Delta - 1$ and the remaining digits are 0. The last k arcs, for $1 \le j \le k$, have the first $j-1$ digits $\Delta-1$, the jth digit 1, and the remaining digits 0. The graph also contains an extra vertex with no incoming arcs connected to every other vertex of the graph. The length of the arc connecting the vertex to the source is zero to make sure that the minimum arc length is zero. The lengths of the other arcs are all the same. These lengths can be zero (to force every vertex caliber to zero) or large (so that the calibers are determined by the other arcs).

Note that if the SPHARD generator with parameters k and Δ produces an input, our adaptive codes may select different parameter values. For $D = \log \Delta$, a problem produced by SPRAND has $(k \cdot D)$-bit lengths. These lengths determine parameter values selected by the adaptive codes.

The three SPHARD problem families we study are HARD1, HARD0, and HARDEST-SQ. The first two problem families differ only in the length of the arcs which determine vertex calibers: the length is large for the first family and zero for the second. All problems in this family have approximately 2^{20} vertices, and the number of arcs is approximately the same in all problems. To create a problem in this family, we choose k and D such that $k \cdot D = 36$ and generate a problem which is hard for MB with k levels and $\Delta = 2^D$. Each HARDEST-SQ problem also has approximately 2^{20} vertices. Problems in this family differ by the k and Δ values. These values are selected so that both the generator and the adaptive codes use the same k and Δ parameters.

10. Experimental results. This section discusses our experimental results.

Caliber heuristic effectiveness. Our analysis shows that work of the caliber heuristic is amortized over other work performed by the algorithm and therefore the heuristic cannot hurt the performance by much. The heuristic can, however, significantly

improve performance. Experimental data confirms this fact. In particular, data for the HARD1 family in Table 1 shows how drastic performance improvement can be. The HARD0 family data in Table 1 shows that if all vertex calibers are forced to zero and the caliber heuristic never helps, its cost is just a few percent of the running time.

Making more levels practical. Our previous work [5, 6] showed that 2- and 3-level MLB implementations and their variants perform well except on certain types of graphs with very large lengths. Increasing the number of levels improved performance on bad examples but hurt performance on "typical" problems somewhat. Comparing MB and SQ code performance on graphs with random arc weights (Tables 2 and 3), we observe that, as the theory would suggest, the caliber heuristic helps more if the number of bucket levels is higher. This is especially apparent if one compares data for MB2D and SQ2D on RAND-C problems (Table 2). This makes the bucket structures with the higher number of levels, in particular, the adaptively selected number of levels, practical. While the random arc length data illustrates how the caliber heuristic helps in "typical" cases, Table 1 shows the effectiveness of adaptive parameter selection on hard problems: Note that for problems with 36-bit lengths, our adaptive codes set $k = 6$.

Operation counts and code tuning. As the analysis suggests, poor performance of the MLB codes is caused by either a large number of the empty bucket examinations or a high cost of bucket expansion operations. See, for example, Tables 3 and 1. The data shows that if the number of empty bucket examinations per vertex is moderate (e.g., ten), they are well amortized by other operations on vertices and do not have a noticeable effect on the running time. When the number of these operations reaches one hundred per vertex, they do have an effect. See, e.g., Table 2. Table 1 shows that processing vertices during bucket expansion is more expensive. Processing one vertex

TABLE 1
HARD1 (left) and HARD0 (right) family data.

k		BFS	MB	SQ
2	time	0.63	1189.20	1.38
	emp.	sec.	52428.94	0.40
	exp.		0.60	0.30
3	time	0.63	10.66	1.39
	emp.	sec.	1170.14	0.15
	exp.		1.14	0.57
4	time	0.62	2.68	1.44
	emp.	sec.	170.44	0.11
	exp.		1.56	0.67
6	time	0.63	2.25	1.53
	emp.	sec.	24.31	0.08
	exp.		2.46	0.77
9	time	0.62	2.87	1.60
	emp.	sec.	6.37	0.05
	exp.		3.89	0.85
12	time	0.63	3.70	1.66
	emp.	sec.	3.12	0.04
	exp.		5.36	0.88
18	time	0.63	5.86	1.82
	emp.	sec.	1.41	0.03
	exp.		8.32	0.93
36	time	0.62	16.32	2.32
	emp.	sec.	0.49	0.01
	exp.		17.75	0.97

k		BFS	MB	SQ
2	time	0.62	1199.67	1201.13
	emp.	sec.	52428.94	52428.94
	exp.		0.60	0.60
3	time	0.62	9.48	9.39
	emp.	sec.	1170.14	1170.14
	exp.		1.14	1.14
4	time	0.63	2.67	2.82
	emp.	sec.	170.44	170.44
	exp.		1.56	1.56
6	time	0.62	2.25	2.45
	emp.	sec.	24.31	24.31
	exp.		2.46	2.46
9	time	0.62	2.87	3.08
	emp.	sec.	6.37	6.37
	exp.		3.89	3.89
12	time	0.62	3.72	3.93
	emp.	sec.	3.12	3.12
	exp.		5.36	5.36
18	time	0.63	5.86	6.05
	emp.	sec.	1.41	1.41
	exp.		8.32	8.32
36	time	0.63	16.24	16.43
	emp.	sec.	0.49	0.49
	exp.		17.75	17.75

TABLE 2
RAND-C (top) and RAND-I (bottom) family data.

u		BFS	MB2L	SQ2L	MB2D	SQ2D	MB-A	SQ-A
1	time	2.97	1.39	1.35	1.39	1.36	1.38	1.35
	emp.	sec.	0.00	0.00	0.00	0.00	0.00	0.00
	exp.		0.48	0.48	0.48	0.48	0.48	0.48
4	time	2.99	1.74	1.59	1.99	1.73	1.47	1.42
	emp.	sec.	0.00	0.00	0.00	0.00	0.00	0.00
	exp.		1.04	0.88	1.64	1.15	0.42	0.39
8	time	3.03	1.81	1.69	2.80	1.96	1.79	1.70
	emp.	sec.	0.00	0.00	0.00	0.00	0.00	0.00
	exp.		1.26	1.05	3.51	1.49	1.26	1.05
12	time	3.00	1.86	1.73	3.41	2.07	1.85	1.74
	emp.	sec.	0.01	0.01	0.01	0.00	0.01	0.01
	exp.		1.35	1.13	5.49	1.56	1.35	1.13
16	time	3.00	1.84	1.74	3.91	2.16	1.95	1.71
	emp.	sec.	0.14	0.07	0.08	0.00	0.12	0.04
	exp.		1.37	1.16	7.44	1.56	2.04	1.08
20	time	2.99	1.82	1.75	4.52	2.27	1.89	1.71
	emp.	sec.	1.84	0.56	0.57	0.00	1.35	0.02
	exp.		1.37	1.16	8.97	1.56	2.11	1.04
24	time	2.99	2.08	1.80	4.95	2.30	2.11	1.76
	emp.	sec.	18.43	1.32	0.93	0.00	6.12	0.03
	exp.		1.33	1.13	9.23	1.56	2.99	1.12
28	time	2.92	2.59	1.85	5.22	2.37	2.13	1.75
	emp.	sec.	55.33	1.35	0.95	0.00	11.72	0.04
	exp.		1.33	1.13	9.23	1.56	2.79	1.06
32	time	2.98	2.49	1.85	5.11	2.36	2.13	1.74
	emp.	sec.	55.33	1.36	0.95	0.00	11.72	0.04
	exp.		1.33	1.13	9.23	1.56	2.79	1.06

n		BFS	MB2L	SQ2L	MB2D	SQ2D	MB-A	SQ-A
2^{17}	time	0.15	1.55	1.56	3.86	2.26	1.88	1.63
	emp./n	sec.	3.01	1.09	0.74	0.01	2.05	0.06
	exp./n		1.09	1.04	7.14	1.56	2.00	1.05
2^{18}	time	0.30	1.73	1.64	4.15	2.26	1.93	1.73
	emp./n	sec.	3.03	0.79	0.74	0.01	2.04	0.04
	exp./n		1.45	1.19	7.65	1.56	2.36	1.19
2^{19}	time	0.62	1.68	1.63	4.41	2.31	1.89	1.71
	emp./n	sec.	3.34	1.12	0.74	0.00	2.36	0.05
	exp./n		1.06	1.01	8.15	1.58	1.96	1.01
2^{20}	time	1.30	1.83	1.79	4.64	2.35	1.94	1.79
	emp./n	sec.	3.34	0.79	0.74	0.00	2.37	0.03
	exp./n		1.41	1.20	8.65	1.58	2.09	1.06
2^{21}	time	2.90	1.83	1.77	4.73	2.29	2.00	1.78
	emp./n	sec.	3.67	1.13	0.74	0.00	2.36	0.03
	exp./n		1.16	1.03	9.13	1.56	2.33	1.16

influences the running time roughly as much as scanning a hundred empty buckets. These observations justify the choice of k and Δ in our adaptive algorithms.

Most robust code. Our data also suggests that SQ-A is a very robust code. Often it is the fastest code, and its running time is always within 10% the fastest code. When designing the HARDEST-SQ problem family, our goal was to produce problems which are hard for the SQ-A code. If one believes that these problems are close to the worst case, then Table 4 shows that even for large lengths, SQ-A performs very well. For example, for 49-bit lengths, its running time exceeds that of BFS by less than a factor

TABLE 3
LONG-C (top) and LONG-I (bottom) family data.

u		BFS	MB2L	SQ2L	MB2D	SQ2D	MB-A	SQ-A
1	time	1.62	1.35	1.35	1.34	1.34	1.34	1.34
	emp.	sec.	0.13	0.06	0.13	0.06	0.13	0.06
	exp.		0.44	0.44	0.44	0.44	0.44	0.44
4	time	1.63	1.54	1.54	1.72	1.60	1.42	1.49
	emp.	sec.	0.61	0.34	0.55	0.23	0.69	0.66
	exp.		0.88	0.71	1.29	0.80	0.39	0.38
8	time	1.63	1.60	1.56	2.06	1.76	1.60	1.56
	emp.	sec.	2.64	0.77	0.96	0.29	2.64	0.77
	exp.		0.77	0.54	1.51	0.83	0.77	0.54
12	time	1.63	1.56	1.55	2.28	1.87	1.56	1.55
	emp.	sec.	5.82	2.44	1.00	0.29	5.82	2.44
	exp.		0.52	0.42	1.51	0.83	0.52	0.42
16	time	1.63	1.56	1.57	2.52	1.99	1.61	1.54
	emp.	sec.	13.68	9.45	1.00	0.29	8.87	1.40
	exp.		0.42	0.38	1.51	0.83	0.52	0.34
20	time	1.63	1.67	1.69	2.75	2.10	1.57	1.54
	emp.	sec.	43.27	37.60	1.00	0.29	9.30	2.64
	exp.		0.39	0.37	1.51	0.83	0.34	0.25
24	time	1.62	2.22	2.15	2.99	2.23	1.64	1.61
	emp.	sec.	146.96	136.51	1.00	0.30	6.10	2.30
	exp.		0.35	0.33	1.51	0.84	0.51	0.40
28	time	1.62	3.08	3.02	3.05	2.33	1.65	1.64
	emp.	sec.	212.05	205.08	1.00	0.35	10.96	2.72
	exp.		0.31	0.30	1.51	0.91	0.43	0.38
32	time	1.63	3.18	3.08	3.06	2.38	1.66	1.67
	emp.	sec.	212.00	206.77	1.00	0.36	10.96	2.91
	exp.		0.31	0.31	1.51	0.92	0.43	0.40

n		BFS	MB2L	SQ2L	MB2D	SQ2D	MB-A	SQ-A
2^{17}	time	0.08	1.71	1.71	2.71	2.14	1.86	1.71
	emp./n	sec.	8.59	4.77	1.00	0.29	4.88	1.00
	exp./n		0.46	0.39	1.51	0.83	0.61	0.41
2^{18}	time	0.17	1.66	1.61	2.80	2.13	1.81	1.68
	emp./n	sec.	12.45	5.10	1.00	0.29	4.17	1.31
	exp./n		0.27	0.22	1.51	0.83	0.63	0.46
2^{19}	time	0.35	1.65	1.65	2.74	2.17	1.73	1.61
	emp./n	sec.	13.65	9.43	1.00	0.29	8.89	1.40
	exp./n		0.42	0.38	1.51	0.83	0.52	0.34
2^{20}	time	0.75	1.59	1.60	2.72	2.10	1.64	1.60
	emp./n	sec.	17.89	10.09	1.00	0.29	6.98	1.47
	exp./n		0.23	0.21	1.51	0.83	0.45	0.31
2^{21}	time	1.61	1.60	1.63	2.65	2.06	1.62	1.59
	emp./n	sec.	23.59	18.84	1.00	0.29	5.88	2.44
	exp./n		0.40	0.37	1.51	0.83	0.52	0.42

of three. We estimate that for 32-bit lengths, SQ-A running time is always within a factor of 2.5 of the BFS time.

11. Concluding remarks. The worst-case bound for the smart queue algorithm is achieved for $\Delta = \theta(\frac{\log C}{\log \log C})$, when the work of moving vertices to lower levels balances the work of scanning empty buckets during bucket expansion. Our average-case analysis reduces the former but not the latter. We get a linear running time when Δ is constant and the empty bucket scans can be charged to vertices in nonempty

TABLE 4
HARDEST-SQ family data.

bits	$\log \Delta$	k		BFS	SQ-A
4	4	1	time	0.62	1.37
			emp.	sec.	0.33
			exp.		0.04
6	3	2	time	0.63	1.50
			emp.	sec.	1.60
			exp.		0.80
8	4	2	time	0.62	1.51
			emp.	sec.	3.20
			exp.		0.80
15	5	3	time	0.62	1.73
			emp.	sec.	9.00
			exp.		1.14
18	6	3	time	0.62	1.78
			emp.	sec.	18.14
			exp.		1.14
24	6	4	time	0.62	1.98
			emp.	sec.	21.11
			exp.		1.56
30	6	5	time	0.62	2.18
			emp.	sec.	23.00
			exp.		2.00
35	7	5	time	0.62	2.32
			emp.	sec.	46.27
			exp.		2.00
42	7	6	time	0.62	2.55
			emp.	sec.	48.92
			exp.		2.46
49	7	7	time	0.62	2.80
			emp.	sec.	50.87
			exp.		2.93

buckets. An interesting open question is if one can get a linear average running time and a better worst-case running time, for example, using techniques from [2, 6, 9], without running several algorithms "in parallel."

Our optimization is to detect vertices with exact distance labels before these vertices reach the bottom level of buckets and place them into F. This technique can be used not only in the context of multilevel buckets but also in the context of radix heaps [2] and hot queues [6].

The fact that SQ-A performance is close to that of BFS limits potential improvements one would consider. For example, a search of a graph to determine better parameter values would not pay for itself, unless it can be amortized over many shortest path computations, e.g., in the context of the all-pairs shortest path problem.

We would like to note that on the machine used in the experiments, fetching a value from the main memory takes on the order of a hundred processor clock cycles. Our experimental results imply that data structure manipulation times are comparable to the data fetch times. As the gap between the processor and memory speeds widens every year, the relative time spent on data structure operation will decrease, and our shortest path algorithm performance will be getting even closer to that of BFS.

An alternative to comparing a shortest path algorithm to BFS is as follows. Run the algorithm and save the sequence of vertices it scanned. Then rerun the algorithm using the saved sequence instead of the data structures to select the next vertex to

scan. The difference between running times of the latter and the former algorithms is a measure of the data structure overhead. It can be compared to the running time of the original algorithm.

Informal experiments show that for problems that are easy for the label-correcting algorithms, the smart queue algorithm works almost as well. It would be interesting to have a more formal comparison of these implementations on real-life problems, such as those in [46]. We also implemented an algorithm that combines the ideas behind smart queues and hot queues [6]. Informal experiments show that the resulting code performs a little better on the hard instances but slightly worse on "typical" instances.

Our results suggest that the smart queue algorithm should be considered in practice when arc lengths are nonnegative integers. The shortest path codes and generators used in this study are available via http://www.avglab.com/andrew/soft.html.

Acknowledgments. Part of this work was done at STAR Lab., InterTrust Technologies Corp., 4750 Patrick Henry Dr., Santa Clara, CA 95054. The author would like to thank Jim Horning, Rajeev Raman, Bob Tarjan, and Eva Tardos for useful discussion and comments on a draft of this paper. We are also grateful to an anonymous referee of a conference version of the paper [22] for pointing out that Theorem 6.2 does not need arc lengths to be independent.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, MA, 1974.

[2] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan, *Faster algorithms for the shortest path problem*, J. ACM, 37 (1990), pp. 213–223.

[3] A. Brodnik, S. Carlsson, J. Karlsson, and J. I. Munro, *Worst case constant time priority queues*, in Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 2001, pp. 523–528.

[4] B. V. Cherkassky and A. V. Goldberg, *Negative-cycle detection algorithms*, Math. Program., 85 (1999), pp. 277–311.

[5] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, *Shortest paths algorithms: Theory and experimental evaluation*, Math. Programming, 73 (1996), pp. 129–174.

[6] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein, *Buckets, heaps, lists, and monotone priority queues*, SIAM J. Comput., 28 (1999), pp. 1326–1346.

[7] H. Chernoff, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statistics, 23 (1952), pp. 493–507.

[8] R. Cole and U. Vishkin, *Deterministic coin tossing with applications to optimal parallel list ranking*, Inform. and Control, 70 (1986), pp. 32–53.

[9] E. V. Denardo and B. L. Fox, *Shortest–route methods:* 1. *Reaching, pruning, and buckets*, Oper. Res., 27 (1979), pp. 161–186.

[10] R. B. Dial, *Algorithm* 360*: Shortest path forest with topological ordering*, Comm. ACM, 12 (1969), pp. 632–633.

[11] R. B. Dial, F. Glover, D. Karney, and D. Klingman, *A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees*, Networks, 9 (1979), pp. 215–248.

[12] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[13] E. A. Dinic, *Economical algorithms for finding shortest paths in a network*, in Transportation Modeling Systems, Yu. S. Popkov and B. L. Shmulyian, eds., Institute for System Studies, Moscow, 1978, pp. 36–44 (in Russian).

[14] L. R. Ford, Jr., *Network Flow Theory*, Technical report P-932, Rand Corporation, Santa Monica, CA, 1956.

[15] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[16] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. ACM, 34 (1987), pp. 596–615.

[17] M. L. Fredman and D. E. Willard, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. Comput. System Sci., 48 (1994), pp. 533–551.
[18] H. N. Gabow, *Scaling algorithms for network problems*, J. Comput. System Sci., 31 (1985), pp. 148–168.
[19] G. Gallo and S. Pallottino, *Shortest paths algorithms*, Ann. Oper. Res., 13 (1988), pp. 3–79.
[20] F. Glover, R. Glover, and D. Klingman, *Computational study of an improved shortest path algorithm*, Networks, 14 (1984), pp. 25–37.
[21] A. V. Goldberg, *A Simple Shortest Path Algorithm with Linear Average Time*, Technical report STAR-TR-01-03, STAR Lab., InterTrust Tech., Inc., Santa Clara, CA, 2001.
[22] A. V. Goldberg, *A simple shortest path algorithm with linear average time*, in Proceedings of the 9th ESA, Lecture Notes in Comput. Sci. 2161, Springer-Verlag, Berlin, 2001, pp. 230–241.
[23] A. V. Goldberg, *Shortest path algorithms: Engineering aspects*, in Proceedings of the ISAAC '01, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 2001, pp. 502–513.
[24] A. V. Goldberg and C. Silverstein, *Implementations of Dijkstra's algorithm based on multi-level buckets*, in Lecture Notes in Econom. and Math. Systems 450 (Refereed Proceedings), P. M. Pardalos, D. W. Hearn, and W. W. Hages, eds., Springer-Verlag, Berlin, 1997, pp. 292–327.
[25] T. Hagerup, *Improved shortest paths in the word RAM*, in 27th International Colloq. on Automata, Languages and Programming, Geneva, Switzerland, 2000, pp. 61–72.
[26] T. Hagerup, *Simpler computation of single-source shortest paths in linear average time*, in Proceedings of STACS 2004, ACM, New York, 2004, pp. 362–369.
[27] M. S. Hung and J. J. Divoky, *A computational study of efficient shortest path algorithms*, Comput. Oper. Res., 15 (1988), pp. 567–576.
[28] H. Imai and M. Iri, *Practical efficiencies of existing shortest-path algorithms and a new bucket algorithm*, J. Oper. Res. Soc. Japan, 27 (1984), pp. 43–58.
[29] D. B. Johnson, *Efficient algorithms for shortest paths in sparse networks*, J. ACM, 24 (1977), pp. 1–13.
[30] U. Meyer, *Single-source shortest paths on arbitrary directed graphs in linear average time*, in Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 2001, pp. 797–806.
[31] J.-F. Mondou, T. G. Crainic, and S. Nguyen, *Shortest path algorithms: A computational study with the C progremming language*, Comput. Oper. Res., 18 (1991), pp. 767–786.
[32] B. M. E. Moret and H. D. Shapiro, *An empirical analysis of algorithms for constructing a minimum spanning tree*, in Proceedings of the 2nd Workshop on Algorithms and Data Structures, Springer-Verlag, Berlin, 1991, pp. 99–117.
[33] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, UK, 1995.
[34] K. Noshita, *A theorem on the expected complexity of Dijkstra's shortest path algorithm*, J. Algorithms, 6 (1985), pp. 400–408.
[35] S. Pallottino, *Shortest-path methods: Complexity, interrelations and new propositions*, Networks, 14 (1984), pp. 257–267.
[36] U. Pape, *Implementation and efficiency of Moore-algorithms for the shortest route problem*, Math. Programming, 7 (1974), pp. 212–222.
[37] R. Raman, *Fast Algorithms for Shortest Paths and Sorting*, Technical report TR 96-06, King's Colledge, London, 1996.
[38] R. Raman, *Priority queues: Small, monotone and trans-dichotomous*, in Proceedings of the 4th Annual European Symposium on Algorithms, Lecture Notes in Comput. Sci. 1136, Springer-Verlag, Berlin, 1996, pp. 121–137.
[39] R. Raman, *Recent results on single-source shortest paths problem*, SIGACT News, 28 (1997), pp. 81–87.
[40] J. T. Stasko and J. S. Vitter, *Pairing heaps: Experiments and analysis*, Comm. ACM, 30 (1987), pp. 234–249.
[41] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conf. Ser. in Appl. Math. 44, SIAM, Philadelphia, 1983.
[42] M. Thorup, *Undirected single-source shortest paths with positive integer weights in linear time*, J. ACM, 46 (1999), pp. 362–394.
[43] M. Thorup, *On RAM priority queues*, SIAM J. Comput., 30 (2000), pp. 86–109.
[44] R. A. Wagner, *A shortest path algorithm for edge-sparse graphs*, J. ACM, 23 (1976), pp. 50–57.
[45] J. W. J. Williams, *Algorithm 232 (Heapsort)*, Comm. ACM, 7 (1964), pp. 347–348.
[46] F. B. Zhan and C. E. Noon, *Shortest path algorithms: An evaluation using real road networks*, Transp. Sci., 32 (1998), pp. 65–73.