A Simple Shortest Path Algorithm with Linear Average Time

Andrew V. Goldberg[1]

STAR Laboratory, InterTrust Technologies Corp.
4750 Patrick Henry Dr., Santa Clara, CA 95054, USA
`goldberg@intertrust.com`

Abstract. We present a simple shortest path algorithm. If the input lengths are positive and uniformly distributed, the algorithm runs in linear time. The worst-case running time of the algorithm is $O(m + n \log C)$, where n and m are the number of vertices and arcs of the input graph, respectively, and C is the ratio of the largest and the smallest nonzero arc length.

1 Introduction

The shortest path problem with nonnegative arc lengths 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, 22, 28, 26, 27, 29, 31, 32], and computational, e.g., [4, 5, 11, 19, 21, 20, 33], viewpoints. Efficient implementations of Dijkstra's algorithm [12], in particular, received a lot of attention. At each step, Dijkstra's algorithm selects for processing a labeled vertex with the smallest distance label. For a nonnegative length function, this selection rule guarantees that each vertex is scanned at most once.

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. 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), we make the 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 word operations, the fastest currently known algorithms achieve the following bounds: $O(m+n(\log U \log\log U)^{1/3})$ [29], $O(m+n(\sqrt{\log n}))$ [28], $O(m \log\log U)$ [22], and $O(m \log\log n)$ [32].

For undirected graphs, Thorup's algorithm [31] 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 non-standard computation model that is not supported by any currently existing computers.

In a recent paper [23, 24], Meyer gives a shortest path algorithm with a linear average time for input arc lengths drawn independently from a uniform distribution on $[1, \dots, M]$. He also proves that, under the same conditions, the running time is linear with high probability. 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 an improvement of the multi-level bucket 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 $[1, \dots, M]$. We also show that if the arc lengths are independent, the algorithm running time is linear with high probability. Both the algorithm and its analysis are natural and simple. 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. This relaxation of Dijkstra's algorithm was originally introduced by Dinitz [13], used in its full strength by Thorup [31], and also used in [23]. Our technique can also be used to improve the worst-case running time of the above-mentioned $O(m+n(\log U \log\log U)^{1/3})$ algorithm of Raman [29] to $O(m+n(\log C \log\log C)^{1/3})$. (This new bound also applies if the input arc lengths are real-valued.)

Our results advance understanding of near-linear shortest path algorithms. Since many computational studies use graphs with uniform arc lengths, these results show that such problems are easy in a certain sense. Although we prove our results for the uniform arc length distribution, our algorithm achieves improved bounds on some other distributions as well. Our results may have practical implications in addition to theoretical ones. The multi-level bucket algorithm works well in practice [5, 21] and our improvement of this algorithm is natural and easy to implement. It is possible that a variant of our algorithm is competitive with the current state-of-the-art implementations on all inputs while outperforming these implementations on some inputs. Although our algorithm looks attractive, the competing implementations are highly optimized and practicality of our results cannot be claimed without a careful implementation and experimentation.

2 Preliminaries

The input to the shortest path problem we consider 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. We assume that arc lengths are integers in the interval $[1, \dots, U]$, 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]; without loss of generality, we assume that $C \geq 2$ (and $\log C \geq 1$). This implies $\log U \geq 1$. 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]). Our data structures 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.

Our shortest path algorithm uses a variant of the multi-level bucket data structure of Denardo and Fox [9]. Although we describe our result in the strong RAM model, following [6], one can also follow [9, 21] and obtain an implementation of the algorithm in the weaker word RAM model. Although somewhat more complicated to describe formally, the latter implementation appears more practical.

3 Labeling Method and Related Results

The labeling method for the shortest path problem [14, 15] works as follows (see e.g., [30]). The method maintains for every vertex v its distance label $d(v)$, parent $p(v)$, and status $S(v) \in \{\text{unreached}, \text{labeled}, \text{scanned}\}$. Initially $d(v) = \infty$, $p(v) = nil$, and $S(v) = \text{unreached}$. The method starts by setting $d(s) = 0$ and $S(s) = \text{labeled}$. While there are labeled vertices, the method picks such a vertex v, scans all arcs out of v, and sets $S(v) = \text{scanned}$. 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 $S(w) = \text{labeled}$.

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 chose 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 appears to be hard. Dinitz [13] and Thorup [31] use a relaxation of Dijkstra's selection rule to get linear-time algorithms for special cases of the shortest path problem. To describe this relaxation, we 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. The following *caliber lemma* is implicit in [13, 29].

Lemma 1. *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.*

4 Algorithm Description and Correctness

Our algorithm is based on the multi-level bucket implementation [9] of Dijkstra's algorithm, but we use Lemma 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. Our 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 may insert the vertex into B or F, and the `decrease-key` operation may move the vertex 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 was neither in 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.

The bucket structure B contains $k+1$ levels of buckets, where $k = \lceil \log U \rceil$. Except for the top level, a level contains two buckets. Conceptually, the top level contains infinitely many buckets. However, at most three consecutive top-level buckets can be nonempty at any given time (Lemma 2 below), and one can maintain only these buckets by wrapping around modulo three at the top level. (For low-level efficiency, one may want to have wrap around modulo four, which is a power of two.)

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 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 binary representation of the distance labels and number bit positions starting from 0 for the least significant bit. Let $\mu_{i,j}$ denote the i-th through j-th least significant bit of μ and let μ_i denote the i-th least significant bit. Similarly, $d_i(u)$ denotes the i-th least significant bit of $d(u)$, and likewise for the other definitions. Note that μ and the $k+1$ least significant bits of the binary representation of $d(u)$ uniquely determine $d(u)$: $d(u) = \mu + (u_{0,k} - \mu_{0,k})$ if $u_{0,k} > \mu_{0,k}$ and $d(u) = \mu + 2^k + (u_{0,k} - \mu_{0,k})$ otherwise.

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

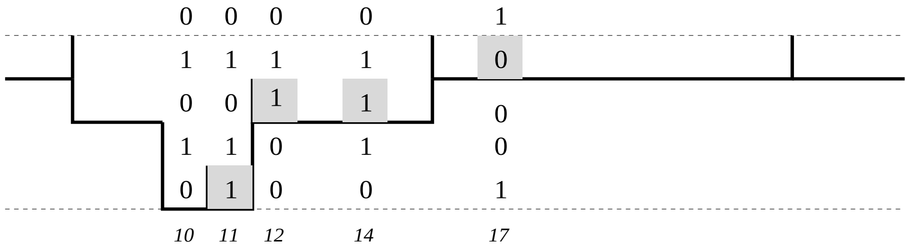

Fig. 1. Multi-level bucket example. $k = 3$, $\mu = 10$. Values on the bottom are in decimal. Values on top are in binary, with the least significant bit on the bottom. Shaded bits determine positions of the corresponding elements.

Each bucket $B(i, j)$ corresponds to the subrange containing all integers in the range with the i-th bit equal to j. At the top level, a bucket $B(k, j)$ corresponds to the range $[j \cdot 2^k, (j + 1) \cdot 2^k)$. The *width* of a bucket at level i is equal to 2^i: the bucket contains 2^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 bit 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 $B(k, \lfloor d(u) - \mu/2^k \rfloor)$ 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, $k = 3$ 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 multi-level bucket algorithm uses Lemma 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. 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 $(i,j) \neq (i',j')$, 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 deletes these vertices from $B(i)$ and uses the `insert` operation to add the vertices back into B or into F, as appropriate.

Although the formal description of the algorithm is a little complicated, the algorithm itself is relatively 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.

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.

In the original multi-level bucket 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 2. *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 $d(u) = \mu \leq \mu'$, $d(v) = \mu + \ell(u,v) \leq \mu' + 2^k$, and therefore v is contained either in $B(k,j)$ 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 $c(u) \leq 2^k$, thus $d(u) \leq \mu' + 2^k$, $d(v) \leq d(u) + 2^k \leq \mu' + 2 \cdot 2^k$, and thus v belongs to $B(k,j)$, $B(k,j+1)$, or $B(k,j+2)$.

Algorithm correctness follows from Lemmas 1 and 2, and the observations that μ is always set to a 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.

5 Worst-Case Analysis

In this section we prove a worst-case bound on the running time of the algorithm. Some definitions and lemmas introduced in this section will be also used in the next section.

We start the analysis with the following lemmas.

Lemma 3. [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 $O(1)$ time.*

Lemma 4. *The algorithm runs in $O(m + n + \Phi)$ time, where Φ is the total number of times a vertex moves from a bucket of B to a lower level bucket.*

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 an amount of 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.

The $O(m + n \log U)$ worst-case time bound is easy to see. To show a better bound, we define $k' = \lfloor \log \delta \rfloor$.

Lemma 5. *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 $d(v) - \mu < 2^i \leq 2^{k'} \leq \delta \leq c(v)$ and the algorithm adds v to F, not B.

The above lemma implies the following bound.

Theorem 1. *The worst-case running time of the algorithm is $O(m + n \log C)$.*

Note that the lemma also implies that the algorithm needs only $O(\log C)$ words of memory to implement the bucket data structure.

Our optimization can also be used to improve other data structures based on multi-level buckets, such as radix heaps [2] and hot queues [6]. For these data structures, the equivalent of Lemma 5 allows one to replace time bound parameter U by C. In particular, the bound of the hot queue implementation of Raman [29] 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 the current section.

6 Average-Case Analysis

In this section we prove linear-time average and high-probability time bounds for our algorithm under the assumption that the input arc lengths are uniformly distributed on $[1, \ldots, M]$.[1]

A key lemma for our analysis is as follows.

Lemma 6. *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 bit in which $d(v)$ and μ differ is bit i and $d(v) - \mu < 2^{i+1} \leq c(v)$. 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. Lemma 6 implies that the number of times v can move to a lower level of B is at most $w(v) + 1$ and therefore $\Phi \leq m + \sum_V w(v)$. Note that k depends on the input, 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 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

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

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

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 independently and uniformly from $[0, 1]$. In fact, for this distribution the high-probability analysis below is simpler. However, the integer distribution is

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

somewhat more interesting, for example because some test problem generators use this distribution.

Remark Note that Theorem 2 does not require arc lengths to be independent. Our proof of its high-probability variant, Theorem 3, requires the independence.

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 7 and 8). 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, this is also true 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.

Because of the space limit, we omit proofs of the following two lemmas.

Lemma 7. *W.h.p.,* $U \geq M/2$.

Lemma 8. *W.h.p.,* $\delta \geq Mm^{-4/3}$*. If* $M \geq m^{2/3}$*, then w.h.p.* $\delta \leq Mm^{-2/3}$.

From (1) and the independence of arc weights, we have $\mathbf{E}[|S_t|] = m2^{k-t}/M$. By the Chernoff bound (see e.g. [7, 25]), $\mathbf{Pr}\left[|S_t| \geq 2m2^{k-t}/M\right] < \left(\frac{e}{4}\right)^{m2^{k-t}/M}$. Since $M \geq 2^{k-1}$, we have

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

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

Proof. Assume that $\delta \geq Mm^{-4/3}$ and $U \geq M/2$; by Lemmas 7 and 8 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

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

Lemma 10. $\sum_{A_2} w(a) = O(m)$ *w.h.p.*

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

Now consider the case $M \ge m^{2/3}$. By Lemmas 7 and 8, w.h.p. $Mm^{-4/3} \le \delta \le Mm^{-2/3}$ and $U \ge M/2$; assume that this is the case. The assumption implies $m^{2/3}/2 \le C \le m^{4/3}$. Under this assumption, we also have $2^{k-1} \le M \le 2^{k+1}$. Combining this with (1) we get $2^{-2-t} \le \mathbf{Pr}[w(a) = t] \le 2^{1-t}$. This implies that

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

therefore

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

and by the independence of arc weights,

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

By the Chernoff bound,

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

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, $\ell(a) \ge \delta$, and thus

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

Therefore w.h.p.,

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

Thus we have the following theorem.

Theorem 3. *If arc lengths are independent and uniformly distributed on* $[1, \dots, M]$*, then with high probability, the algorithm runs in linear time.*

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

7 Concluding Remarks

We described our algorithm for the binary multi-level buckets, with two buckets at each level except for the top level. One can easily extend the algorithm for base-$\varDelta$ buckets, for any integer $\varDelta \geq 2$. One gets a worst-case bound for $\varDelta = \theta(\frac{\log C}{\log \log C})$ [9] when the work of moving vertices to lower levels balances with 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 $\varDelta$ is constant and the empty bucket scans can be charged to vertices in nonempty 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 multi-level buckets, but in the context of radix heaps [2] and hot queues [6].

Acknowledgments

The author would like to thank Jim Horning, Anna Karlin, Rajeev Raman, Bob Tarjan, and Eva Tardos for useful discussion and comments on a draft of this paper. We are also greatful to an anonymous referee for pointing out that Theorem 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, 1974.
2. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster Algorithms for the Shortest Path Problem. *J. Assoc. Comput. Mach.*, 37(2):213–223, April 1990.
3. A. Brodnik, S. Carlsson, J. Karlsson, and J. I. Munro. Worst case constant time priority queues. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 523–528, 2001.
4. B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. *Math. Prog.*, 85:277–311, 1999.
5. B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996.
6. B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. *SIAM J. Comput.*, 28:1326–1346, 1999.
7. H. Chernoff. A Measure of Asymptotic Efficiency for Test of a Hypothesis Based on the Sum of Observations. *Anals of Math. Stat.*, 23:493–509, 1952.
8. R. Cole and U. Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 70:32–53, 1986.
9. E. V. Denardo and B. L. Fox. Shortest–Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.
10. R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.
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*, 9:215–248, 1979.

12. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.
13. E. A. Dinic. Economical algorithms for finding shortest paths in a network. In Yu.S. Popkov and B.L. Shmulyian, editors, *Transportation Modeling Systems*, pages 36–44. Institute for System Studies, Moscow, 1978. In Russian.
14. L. Ford. Network Flow Theory. Technical Report P-932, The Rand Corporation, 1956.
15. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.
16. M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.
17. M. L. Fredman and D. E. Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comp. and Syst. Sci.*, 48:533–551, 1994.
18. H. N. Gabow. Scaling Algorithms for Network Problems. *J. of Comp. and Sys. Sci.*, 31:148–168, 1985.
19. G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.
20. F. Glover, R. Glover, and D. Klingman. Computational Study of an Improved Shortest Path Algorithm. *Networks*, 14:25–37, 1984.
21. A. V. Goldberg and C. Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. In P. M. Pardalos, D. W. Hearn, and W. W. Hages, editors, *Lecture Notes in Economics and Mathematical System 450 (Refereed Proceedings)*, pages 292–327. Springer Verlag, 1997.
22. T. Hagerup. Improved Shortest Paths in the Word RAM. In *27th Int. Colloq. on Automata, Languages and Programming, Geneva, Switzerland*, pages 61–72, 2000.
23. U. Meyer. Single-Source Shortest Paths on Arbitrary Directed Graphs in Linear Average Time. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 797–806, 2001.
24. U. Meyer. Single-Source Shortest Paths on Arbitrary Directed Graphs in Linear Average Time. Technical Report MPI-I-2001-1-002, Max-Planck-Institut für Informatik, Saarbrüken, Germany, 2001.
25. R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.
26. K. Noshita. A Theorem on the Expected Complexity of Dijkstra's Shortest Path Algorithm. *J. Algorithms*, 6:400–408, 1985.
27. R. Raman. Fast Algorithms for Shortest Paths and Sorting. Technical Report TR 96-06, King's Colledge, London, 1996.
28. R. Raman. Priority Queues: Small, Monotone and Trans-Dichotomous. In *Proc. 4th Annual European Symposium Algorithms*, pages 121–137. Springer-Verlag, Lect. Notes in CS 1136, 1996.
29. R. Raman. Recent Results on Single-Source Shortest Paths Problem. *SIGACT News*, 28:81–87, 1997.
30. R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
31. M. Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. Assoc. Comput. Mach.*, 46:362–394, 1999.
32. M. Thorup. On RAM Priority Queues. *SIAM Journal on Computing*, 30:86–109, 2000.
33. F. B. Zhan and C. E. Noon. Shortest Path Algorithms: An Evaluation using Real Road Networks. *Transp. Sci.*, 32:65–73, 1998.