Shortest Path Algorithms: Engineering Aspects

Andrew V. Goldberg

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

Abstract. We review shortest path algorithms based on the multi-level bucket data structure [6] and discuss the interplay between theory and engineering choices that leads to efficient implementations. Our experimental results suggest that the caliber heuristic [17] and adaptive parameter selection give an efficient algorithm, both on typical and on hard inputs, for a wide range of arc lengths.

1 Introduction

The three most common ways to evaluate algorithm performance are worst-case analysis, average-case analysis, and experimental evaluation. The most effective way to evaluate an algorithm is to use all three approaches. Good algorithm engineering combines theoretically justified ideas, common-sense heuristics, and experimental feedback to develop an efficient and robust code.

In this paper we study the important problem of finding shortest paths from a source to all other vertices in a directed graph with nonnegative arc lengths (the *NSP problem*). This problem has been well-studied. The worst-case bounds for the problem have a long history; see e.g., [1, 2, 5–7, 9, 10, 14, 19, 20, 24–26, 28–30]. The currently best bounds are near-linear. Let n and m denote the number of vertices and arcs in the input graph, respectively. If the input arc lengths are integral, let U denote the maximum arc length. Let C denote the ratio between the biggest and the smallest nonzero arc lengths. In the pointer model of computation, one can get an $O(m + n \log n)$ time bound [13]. In a RAM model with word operations, the fastest currently known algorithms achieve the following bounds: $O(m + n(\log U \log\log U)^{1/3})$ [26], $O(m + n(\sqrt{\log n}))$ [25], $O(m \log\log U)$ [19], and $O(m \log\log n)$ [29]. In the special case when the graph is undirected, Thorup's algorithm [28] runs in linear time.

The average-case results for the NSP problem are interesting because they apply to natural input distributions or to potentially practical algorithms. In particular, Noshita [23] shows that under relatively weak

assumptions on the input distribution, the average-case bound on the binary heap implementation of Dijkstra's algorithm is better than the worst-case bound. Mayer [21] shows that the problem can be solved in linear average time if input arc lengths are independent and uniformly distributed. Goldberg [17] shows that a simple modification of the algorithm of [6] yields an algorithm with average running time on the uniform arc length distribution (without the independence assumption). Below we refer to the algorithm of [17] as the *smart queue* algorithm.

Computational work on the NSP problem has received a lot of attention as well; see, e.g., [3, 4, 8, 15, 18, 16, 31]. This work leads to the general agreement that the problem can be solved well in practice. However, this work does not show how close existing NSP algorithm implementations are to a practical lower bound.

We focus our attention on NSP algorithms based on the *multi-level bucket (MLB)* data structure and its variants. This data structure was originally proposed by Denardo and Fox [6]. Enhancements to it have been proposed in [1, 5]. For a long time, the folklore, originating from the original paper [6] and the fact that computer memories were small, was that the MLB data structure was not competitive in practice. The work of Cherkassky et al. [4] and the followup work of [5, 18] suggest that two- or three-level MLB variants are competitive on many problems.

In this paper we review the MLB data structure and discuss an implementation of the MLB data structure that takes advantage of the newer theoretical results and improves upon previous implementations. We also discuss an implementation of the smart queue version of the algorithm. Our experimental results show that the use of smart queues makes the MLB algorithm with a large number of levels practical. In particular, an implementation with the number of levels optimized for the worst-case theoretical performance works well on both typical and bad-case inputs. For example, for 32-bit arc lengths, the code runs in time less than 2.5 times that of breadth-first search, even on the hardest problems we were able to construct. Our results lead to better understanding of NSP algorithm implementations and show how close their performance is to the lower bound provided by breadth-first search.

2 The Multi-Level Bucket Data Structure

Assuming that the reader is familiar with the labeling method [11, 12] (see also [27]) and Dijkstra's algorithm [9], we review the MLB algorithm and related results. For more details, see [5, 6, 17].

The MLB structure B implements the priority queue operations `insert`, `decrease-key`, and `extract-min`. The bucket structure has two parameters, the number of levels k and the base Δ. A k-level structure contains k regular levels and a special (top) level. Given an input graph with maximum arc length U, the two parameters are related by $k = \lceil \log_\Delta U \rceil$. Except for the top level, a level contains Δ buckets. Conceptually, the top level contains infinitely many buckets. However, one can show that at most two consecutive top-level buckets can be nonempty at any given time, and that one can maintain only these buckets. 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 $\Delta - 1$. A bucket contains a set of vertices maintained as a doubly linked list.

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 $\mu = d(v)$, the distance label of v. Consider the base-Δ representation of the distance labels and number digit positions starting from 0 for the least significant digit. One can show that μ and the $k + 1$ least significant digits of the base-Δ representation of $d(u)$ uniquely determine $d(u)$.

Next we define the position of a vertex u in B with respect to μ. Let μ' and $d'(u)$ be μ and $d(u)$, respectively, truncated to the $k + 1$ least significant digits. Let i be the index of the most significant digit in which μ' and $d'(u)$ differ, or 0 if they match. Let j be the digit of $d(u)$ in position i. Then we say that the *position of* u *in* B is (i, j). When u is inserted into B, it is inserted into $B(i, j)$. For each vertex in B, we store its position.

Each bucket $B(i, j)$ corresponds to a range of values that depends on μ. Suppose the position of u in B is (i, j). We say that u belongs to the range of $B(i', j')$ if $(i', j') = (i, j)$ or if $i' > i$. The *width* of a bucket at level i is equal to Δ^i: the bucket contains Δ^i distinct values.

Given μ, one can compute the position of a vertex u from $d(u)$ in constant time. This gives a simple constant-time implementation of `insert` and `decrease-key` operations. To implement the `extract-min` operation, we find the lowest nonempty level i and the first nonempty bucket $B(i, j)$ on this level. We do the latter by starting from a bucket whose range contains μ and scanning consecutive buckets until a nonempty one is found. If the level is the bottom level, we delete and return a vertex from $B(i, j)$. Otherwise we *expand* the bucket by finding and deleting a vertex with the smallest label in the bucket, setting μ to the distance label of this vertex, and moving remaining vertices from $B(i, j)$ to their new locations. An important fact is that the vertices always more to *lower* levels.

The analysis of the algorithm amortizes all work over vertex scans except for the work of scanning empty buckets and the work involved in moving vertices during the bucket expansion operation.

Theorem 1. *The algorithm runs in $O(m + n + \rho + \phi)$ time, where ρ is the number of empty bucket scans and ϕ is the number of times vertices move from a bucket to a lower level bucket.*

If we charge scanning empty buckets to the vertex the algorithm finds and scans immediately afterwords, we get an $O(n\Delta)$ bound on ρ. The fact that vertices move to lower levels of B implies an $O(kn)$ bound on ϕ. To improve efficiency, one has to balance ρ and ϕ. From the worst-case analysis point of view, setting $\Delta = \Theta(k)$ minimizes the running time. If $\Delta = k = \Theta\left(\frac{\log U}{\log\log U}\right)$, then the algorithm runs in $O(m + n\frac{\log U}{\log\log U})$ time.

Next we discuss two heuristics that improve the MLB algorithm performance. 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. We say that a distance label of a labeled vertex is *exact* if the label is equal to the distance from the source to the vertex. The following *caliber lemma* is implicit in [10, 26] and explicitly stated in [17]. The lemma allows us to relax Dijkstra's minimum distance label selection rule while maintaining the invariant that each vertex is scanned at most once.

Lemma 1. *Suppose ℓ is nonnegative and let μ be a lower bound on the distance labels of labeled vertices. Let v be a vertex such that $\mu + c(v) \geq d(v)$. Then $d(v)$ is exact.*

Previous implementations of the MLB algorithm, as well as those described below, use the following *wide bucket* heuristic. Let L be the smallest nonzero arc length and pick w such that $0 < w \leq L$. Then the MLB algorithm remains correct if one multiplies the bucket width on every level by w. With this heuristic and $w = \Theta(L)$, the algorithm needs only $\lceil \log_\Delta(U/L) \rceil = O(\log_\Delta C)$ levels, and one can replace "U" by "C" in the time bounds.

The *caliber heuristic* [17] uses Lemma 1 to detect and scan vertices with exact distance labels. To use the heuristic, we modify the MLB algorithm to keep labeled vertices in one of two places: a set F and a priority queue B implemented using the MLB structure. We refer to the modified algorithm as the *smart queue* algorithm.

At a high level, the algorithm works as follows. Vertices in F have exact distance labels. 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 into F, depending on whether Lemma 1 applies or not.

The caliber heuristic provably improves algorithm performance for certain input distributions. For example if Δ is a constant and arc lengths are distributed uniformly over $[1, M]$ for some M that is the same for all arcs, then the expected running time of the MLB algorithm with the heuristic is $O(n + m)$.

3 Algorithm Implementations

We implemented several variants of the MLB algorithm. The MB code implements the algorithm with the wide bucket heuristic, and the SQ code adds the caliber heuristic. Next we discuss these codes and engineering considerations involved in their development.

Our implementation of MB is very similar to that of [18], 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. The MB implementation computes the vertex position with respect to μ as described above. 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 L. 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 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 chose 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 chose 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.

4 Experimental Methodology and Setup

Following Moret and Shapiro [22], 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 BSF code computes distances and a shortest path tree for the unit length function. The breadth-first search problem is a simple special case of NSP and, modulo BFS implementation efficiency, 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 removes dependencies on some 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 ρ and ϕ in Theo-

rem 1. 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 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 modern computers.

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 MLB code.

5 Problem Families

We report data on seven problem families produced by three problem generators. Since we are interested in the efficiency of shortest path 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 0 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 as large as possible subject to the constraint that all our codes run without paging. For RAND-C, $n = 2^{20}$, u starts at 1 and then takes on integer multiples of four from 4 to 32. Up to $u = 20$, the minimal arc length L in all test inputs is one. For $u = 24$, L is greater than one for some inputs. For $u = 28$ and 32, L is always greater than one. Note that the expected value of C does not

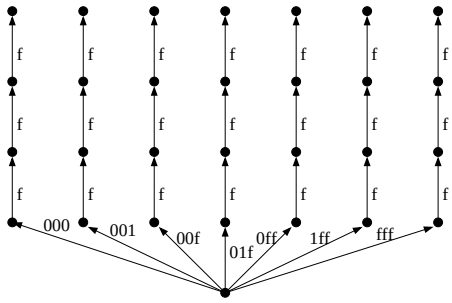

Fig. 1. An example of a hard problem instance; $k = 3$ and $\Delta = 16$. Arc lengths are given in hexadecimal. We omit the extra vertex with arcs designed to manipulate vertex calibers.

change for $u \geq 28$, and therefore the results for $u > 32$ would have been very similar to those for $u = 28$ 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 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 large. 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, and for the same reasons.

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 1 for an example.) These paths have the same 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 $1 \leq i \leq k$, the first i digits are $\Delta - 1$ and the remaining digits are 0. The last k arcs, for $1 \leq j \leq k$, have the first $j - 1$ digits $\Delta - 1$, the j-th 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

n (= C)		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

n (= C)		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

Table 1. RAND-I (left) and LONG-I (right) family data

source is zero to make sure that the minimum arc length is zero. 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 parameters. For $D = \log \Delta$, a problem produced by SPRAND has $(k \cdot D)$-bit lengths. These lengths determine parameters 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 chose k and D such that $k \cdot D = 36$ and generate a problem which is hard for MB with k levels and $\Delta = 2^D$. (We exclude $k = 1$ as always.) 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.

6 Experimental Results

Previous work [4, 18] suggests that MB2L performs well except on some problems with large arc lengths. Our experimental results confirm this,

bits		BFS	MB2L	SQ2L	MB2D	SQ2D	MLB-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
4 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

bits		BFS	MLB2L	SQ2L	MLB2D	SQ2D	MLB-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

Table 2. RAND-C (left) and LONG-C (right) family data

and suggest that the caliber heuristic, combined with adaptive selection of parameters, leads to a more robust code. Empty bucket scans cause bad performance of MB2L. For example, the data in Table 2 shows that MB2L performs similarly to SQ-A when C is small, but the latter code is faster when C is large. HARD1 and HARD0 problems (Table 3) show that in the worst case the difference is huge.

Next we compare performance of our MB and SQ codes. Data for HARD1 and HARD0 problems, given in Table 3, shows that the caliber heuristic can give significant savings in the number of operations or no savings at all. In the former case, SQ is much faster than MB. In the latter case, SQ is a little slower. Looking at the uniform arc length data, Tables 1 – 2, we see that the caliber heuristic provides significant improvement when the number of levels is large, as theoretical analysis

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

bits	log Δ	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

Table 3. hard1 (left) and hard0 (center) and hard-sq (right) data.

predicts. This heuristic makes bucket structures with many levels more practical. In particular, it makes SQ practical.

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 noticable effect on the running time. When the number of these operations reaches a hundred per vertex, they do have an effect. See e.g., Table 2. The same table shows that processing vertices during bucket expansion is more expensive. Processing one vertex influences the running time more than scanning a hundred empty buckets. These observations justify our choice of k and Δ in our adaptive algorithms.

HARD0 problems (Table 3) illustrate that adaptive selection of k and Δ is important from the worst-case point of view. (For 36-bit lengths used in HARD1 and HARD0 problems, our adaptive codes set $k = 6$.) As far as typical performance goes, our adaptive codes are never significantly slower, and sometimes significantly faster, than the corresponding non-adaptive codes. See, e.g., Table 2.

7 Concluding Remarks

Our data suggests that the caliber heuristic makes the MLB algorithm with adaptive parameter selection practical. Even on the hardest problem we used, when input arc lengths fit into 32-bit words, the running time of SQ-A is always within a factor of 2.5 of the running time of breadth-first search. We conjecture that for no input will this code run longer than the baseline code by more than a factor of three, given that the problem's size is much bigger than the cache size but smaller than the memory size. On many real-life problems, this code will run within a factor of two of the baseline code.

We also experimented with a hot queue [5] version of our code. Although we do not give details due to the lack of space, this code performs better than SQ-A on bad-case problems and almost as well on typical problems.

References

1. 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.
2. P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Math. Systems Theory*, 10:99–127, 1977.
3. B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. *Math. Prog.*, 85:277–311, 1999.
4. B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996.
5. B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. *SIAM J. Comput.*, 28:1326–1346, 1999.
6. E. V. Denardo and B. L. Fox. Shortest–Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.
7. R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.
8. 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.
9. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.

10. 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.
11. L. Ford. Network Flow Theory. Technical Report P-932, The Rand Corporation, 1956.
12. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.
13. 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.
14. 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.
15. G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.
16. F. Glover, R. Glover, and D. Klingman. Computational Study of an Improved Shortest Path Algorithm. *Networks*, 14:25–37, 1984.
17. 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, USA, 2001.
18. 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.
19. T. Hagerup. Improved Shortest Paths in the Word RAM. In *27th Int. Colloq. on Automata, Languages and Programming, Geneva, Switzerland*, pages 61–72, 2000.
20. D.B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networs. *J. Assoc. Comput. Mach.*, 24:1–13, 1977.
21. 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. A full version available as TR MPI-I-2001-1-002, Max-Planck-Institut für Informatik, Saarbrüken, Germany.
22. B. M.E. Moret and H. D. Shapiro. An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree. In *Proc. 2nd Workshop on Algorithms and Data Structures*, 1991.
23. K. Noshita. A Theorem on the Expected Complexity of Dijkstra's Shortest Path Algorithm. *J. Algorithms*, 6:400–408, 1985.
24. R. Raman. Fast Algorithms for Shortest Paths and Sorting. Technical Report TR 96-06, King's Colledge, London, 1996.
25. 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.
26. R. Raman. Recent Results on Single-Source Shortest Paths Problem. *SIGACT News*, 28:81–87, 1997.
27. R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
28. M. Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. Assoc. Comput. Mach.*, 46:362–394, 1999.
29. M. Thorup. On RAM Priority Queues. *SIAM Journal on Computing*, 30:86–109, 2000.
30. J. W. J. Williams. Algorithm 232 (Heapsort). *Comm. ACM*, 7:347–348, 1964.
31. F. B. Zhan and C. E. Noon. Shortest Path Algorithms: An Evaluation using Real Road Networks. *Transp. Sci.*, 32:65–73, 1998.