Buckets, Heaps, Lists, and Monotone Priority Queues

Boris V. Cherkassky *
Central Econ. and Math. Inst.
Krasikova St. 32
117418, Moscow, Russia
cher@cemi.msk.su

Andrew V. Goldberg
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
avg@research.nj.nec.com

Craig Silverstein †
Computer Science Department
Stanford University
Stanford, CA 94305
csilvers@theory.stanford.edu

Abstract

We introduce the heap-on-top (hot) priority queue data structure that combines the multi-level bucket data structure of Denardo and Fox and a heap. We use the new data structure to obtain an $O(m + n(\log C)^{\frac{1}{3}+\epsilon})$ expected time implementation of Dijkstra's shortest path algorithm, improving the previous bounds. We can implement hot queues even more efficiently in practice by using sorted lists to represent small priority queues. Our experimental results in the context of Dijkstra's algorithm show that this implementation of hot queues performs very well and is more robust than implementations based only on heap or multi-level bucket data structures.

1 Introduction

A priority queue is a data structure that maintains a set of elements and supports operations `insert`, `decrease-key`, and `extract-min`. Priority queues are fundamental data structures with many applications. Typical applications include graph algorithms (*e.g.* [14]) and event simulation (*e.g.* [5]).

An important subclass of priority queues used in applications such as event simulation and in Dijkstra's shortest path algorithm [13] is the class of monotone priority queues. Intuitively, a priority queue is *monotone* if at any time keys of elements on the queue are at least as big as the key of the most recent element extracted from the queue. In this paper we deal with monotone priority queues.

Unless mentioned otherwise, we refer to priority queues whose operation time bounds depend only on the number of elements on the queue as *heaps*. The fastest implementations of heaps are described in [4, 14, 19]. Alternative implementations of priority queues use buckets (*e.g.* [2, 7, 11, 12]). Operation times for bucket-based implementations depend on the maximum event duration C, defined in Section 2. See [3] for a related data structure.

Heaps are particularly efficient when the number of elements on the heap is small. Bucket-based priority queues are particularly efficient when the maximum event duration C is small. Furthermore, some of the work done in bucket-based implementations can be amortized over elements in the buckets, yielding better bounds if the number of elements is large. In this sense, heaps and buckets complement each other.

We introduce *heap-on-top priority queues (hot queues)*, which combine the multi-level bucket data structure of Denardo and Fox [11] and a heap. These queues use the heap instead of buckets when buckets would be sparsely occupied. The resulting implementation takes advantage of the best performance features of both data structures. We also give an alternative and more insightful description of the multi-level bucket data structure. (Concurrently and independently, a similar description has been given by Raman [17].)

*This work was done while the author was visiting NEC Research Institute.

†Supported by the Department of Defense, with partial support from NSF Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

Hot queues are related to radix heaps *(RH)*[1] of Ahuja *et al.* [2]. An RH is similar to the multi-level buckets, but uses a heap to find nonempty buckets. To get the best bounds, the heap operation time in an RH should depend on the number of *distinct* keys on the heap. The most complicated part of [2] is modifying Fibonacci heaps [14] to meet this requirement. In contrast, the hot queue bounds do not require anything special from the heap. We can use Fibonacci heaps with no modifications and achieve the same bounds as RH.

Using the heap of Thorup [19], we obtain even better bounds. As a side-effect, we obtain an $O(m + n(\log C)^{\frac{1}{3}+\epsilon})$ expected time implementation of Dijkstra's shortest path algorithm, improving the previous bounds. Since Thorup's bounds depend on the total number of elements on the heap, RH cannot take immediate advantage of this data structure.

We believe that data structures are especially interesting if they work well both in theory and in practice. A preliminary version of the hot queue data structure [6] did not perform well in practice. Based on experimental feedback, we modified the data structure to be more practical. We also developed techniques that make hot queues more efficient in practice.

We compare the implementation of hot queues to implementations of multi-level buckets and k-ary heaps in the context of Dijkstra's shortest paths algorithm. Our experimental results show that hot queues perform best overall and are more robust than either of the other two data structures. This is especially significant because a multi-level bucket implementation of Dijkstra's algorithm compared favorably with other implementations of the algorithm in a previous study [7] and was shown to be very robust. For many problem classes, the hot queue implementation of Dijkstra's algorithm is the best both in theory and in practice.

Due to the page limit, we omit some proofs, details, and experimental data. A full version of the paper appears in [8].

2 Preliminaries

A *priority queue* is a data structure that maintains a set of elements and supports operations `insert`, `decrease-key`, and `extract-min`. We assume that elements have *keys* used to compare the elements and denote the key of an element u by $\rho(u)$. Unless mentioned otherwise, we assume that the keys are integral. By the value of an element we mean the key of the element. The `insert` operation adds a new element to the queue. The `decrease-key` operation assigns a smaller value to the key of an element already on the queue. The `extract-min` operation removes a minimum element from the queue and returns the element. We denote the number of `insert` operations in a sequence of priority queue operations by N.

To gain intuition about the following definition, think of event simulation applications where keys correspond to processing times. Let u be the latest element extracted from the queue. An *event* is an `insert` or a `decrease-key` operation on the queue. Given an event, let v be the element inserted into the queue or the element whose key was decreased. The *event duration* is $\rho(v) - \rho(u)$. We denote the maximum event duration by C. An application is *monotone* if all event durations are nonnegative. A *monotone* priority queue is a priority queue for monotone applications. To make these definitions valid for the first insertion, we assume that during initialization, a special element is inserted into the queue and deleted immediately afterwards. Without loss of generality, we assume that the value of this element is zero. (If it is not, we can subtract this value from all element values.)

In this paper, by *heap* we mean a priority queue whose operation time bounds are functions of the number of elements on the queue. We assume that heaps also support the `find-min` operation, which returns the minimum element on the heap.

We call a sequence of operations on a priority queue *balanced* if the sequence starts and ends with an empty queue. In particular, implementations of Dijkstra's shortest path algorithm produce balanced operation sequences.

In this paper we use the RAM model of computation [1]. The only nonobvious result about the model we use appears in [9], where it is attributed to B. Schieber. The result is that given two machine words, we can find, in constant time, the index of the most significant bit in which the two words differ.

3 Multi-Level Buckets

In this section we describe the k-level bucket data structure of Denardo and Fox [11]. We give a simpler description of this data structure by treating the element keys as base-Δ numbers for a certain parameter Δ. Consider a bucket structure B that contains k levels of buckets, where k is a positive integer. Except for the top level, a level contains an array of Δ buckets. The top level contains infinitely many buckets. Each top level bucket corresponds to an interval $[i\Delta^k, (i+1)\Delta^k - 1]$. We choose Δ so that at most Δ consecutive buckets at the top level can be nonempty; we need to maintain only

[1] Radix heap operation time bounds depend on C.

these buckets.[2]

We denote bucket j at level i by $B(i,j)$. A bucket contains a set of elements in a way that allows constant-time additions and deletions, *e.g.* in a doubly linked list.

Given k, we choose Δ as small as possible subject to two constraints. First, each top level bucket must contain at least $(C+1)/\Delta$ integers. Then, by the definition of C, keys of elements in B belong to at most Δ top level buckets. Second, Δ must be a power of two so that we can manipulate base-Δ numbers efficiently using RAM operations on words of bits. With these constraints in mind, we set Δ to the smallest power of two greater or equal to $(C+1)^{1/k}$.

We maintain μ, the key of the latest element extracted from the queue. Consider the base-Δ representation of the keys and an element u in B. By definitions of C and Δ, μ and the k least significant digits of the base-Δ representation of $\rho(u)$ uniquely determine $\rho(u)$. If $\overline{\mu}$ and $\overline{\rho}$ are the numbers represented by the k least significant digits of μ and $\rho(u)$, respectively, then $\rho(u) = \mu + (\overline{\rho} - (\overline{\mu}))$ if $\overline{\rho} \geq \overline{\mu}$ and $\rho(u) = \mu + \Delta^k + (\overline{\rho} - \overline{\mu})$ otherwise. For $1 \leq i < k$, we denote by μ_i the i-th least significant digit of the base-Δ representation of μ. We denote the number obtained by deleting the $k-1$ least significant digits of μ by μ_k. Similarly, for $1 \leq i < k$, we denote the i-th least significant digits of $\rho(u)$ by u_i and we denote the number obtained by deleting $k-1$ least significant digits of $\rho(u)$ by u_k.

The levels of B are numbered from k (top) to 1 (bottom) and the buckets at each level are numbered from 0 to $\Delta - 1$. Let i be the index of the most significant digit in which $\rho(u)$ and μ differ or 1 if $\rho(u) = \mu$. Given μ and u with $\rho(u) \geq \mu$, we say that the *position of u with respect to μ* is (i, u_i). If u is inserted into B, it is inserted into $B(i, u_i)$. For each element in B, we store its position. If an element u is in $B(i,j)$, then all except for $i+1$ most significant digits of $\rho(u)$ are equal to the corresponding digits of μ and $u_i = j$.

The following lemma follows from the fact that keys of all elements on the queue are at least μ.

LEMMA 3.1. *For every level i, buckets $B(i,j)$ for $0 \leq j < \mu_i$ are empty.*

At each level i, we maintain the number of elements at this level. We also maintain the total number of elements in B.

The `extract-min` operation can change the value of μ. As a side-effect, positions of some elements in B may change. Suppose that a minimum element is deleted and the value of μ changes. Let μ' be the value of μ before the deletion and let μ'' be the value of μ after the deletion. By definition, keys of the elements on the queue after the deletion are at least μ''. Let i be the position of the least significant digit in which μ' and μ'' differ. If $i = 1$ (μ' and μ'' differ only in the last digit), then for any element in B after the deletion its position is the same as before the deletion. If $i > 1$, than the elements in bucket $B(i, \mu''_i)$ with respect to μ' are exactly those whose position is different with respect to μ''. These elements have a longer prefix in common with μ'' than with μ' and therefore they belong to a lower level with respect to μ''.

The *bucket expansion* procedure moves these elements to their new positions. The procedure removes the elements from $B(i, \mu''_i)$ and puts them into their positions with respect to μ''. The two key properties of bucket expansions are as follows:

- After the expansion of $B(i, \mu''_i)$, all elements of B are in correct positions with respect to μ''.
- Every element of B moved by the expansion is moved to a lower level.

Now we are ready to describe the multi-level bucket implementation of the priority queue operations.

- `insert`

  To insert an element u, compute its position (i,j) and insert u into $B(i,j)$.

- `decrease-key`

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

- `extract-min`

  (We need to find and delete the minimum element, update μ, and move elements affected by the change of μ.)

  Find the lowest nonempty level i.

  Find the first nonempty bucket at level i.

  If $i = 1$, delete an element from $B(i,j)$, set $\mu = \rho(u)$, and return u. (In this case old and new values of μ differ in at most the last digit and all element positions remain the same.)

  If $i > 1$, examine all elements of $B(i,j)$ and delete a minimum element u from $B(i,j)$. Set $\mu = \rho(u)$ and expand $B(i,j)$. Return u.

Next we deal with efficiency issues.

LEMMA 3.2. *Given μ and u, we can compute the position of u with respect to μ in constant time.*

[2] The simplest way to implement the top level is to "wrap around" modulo Δ.

Iterating through the levels, we can find the lowest nonempty level in $O(k)$ time. Using binary search, we can find the level in $O(\log k)$ time. We can do even better using the power of the RAM model:

LEMMA 3.3. *If $k \leq \log C$, then the lowest nonempty level of B can be found in $O(1)$ time.*

As we will see, the best bounds are achieved for $k \leq \log C$.

A simple way of finding the first nonempty bucket at level i is to go through the buckets. This takes $O(\Delta)$ time.

LEMMA 3.4. *We can find the first nonempty bucket at a level in $O(\Delta)$ time.*

Remark. One can do better [11]. Divide buckets at every level into groups of size $\lceil \log C \rceil$, each group containing consecutive buckets. For each group, maintain a $\lceil \log C \rceil$-bit number with bit j equal to 1 if and only if the j-th bucket in the group is not empty. We can find the first nonempty group in $O\left(\frac{\Delta}{\log C}\right)$ time and the first nonempty bucket in the group in $O(1)$ time. This construction gives a $\log C$ factor improvement for the bound of Lemma 3.4. By iterating this construction p times, we get an $O\left(p + \frac{\Delta}{\log^p C}\right)$ bound.

Although the above observation improves the multi-level bucket operation time bounds for small values of k, the bounds for the optimal value of k do not improve. To simplify the presentation, we use Lemma 3.4, rather than its improved version, in the rest of the paper.

THEOREM 3.1. *Amortized bounds for the multi-level bucket implementation of priority queue operations are as follows: $O(k)$ for* `insert`*, $O(1)$ for* `decrease-key`*, $O(C^{1/k})$ for* `extract-min`*.*

Proof. The `insert` operation takes $O(1)$ worst-case time. We assign it an amortized cost of k because we charge moves of elements to a lower level to the insertions of the elements.

The `decrease-key` operation takes $O(1)$ worst case time and we assign it an amortized cost of $O(1)$.

For the `extract-min` operation, we show that its worst-case cost is $O(k + C^{1/k})$ plus the cost of bucket expansions. The cost of a bucket expansion is proportional to the number of elements in the bucket. This cost can be amortized over the `insert` operations, because, except for the minimum element, each element examined during a bucket expansion is moved to a lower level.

Excluding bucket expansions, the time of the operation is $O(1)$ plus the $O(\Delta)$ for finding the first nonempty bucket. This completes the proof since $\Delta = O(C^{1/k})$. ■

Note that in any sequence of operations the number of `insert` operations is at least the number of `extract-min` operations. In a balanced sequence, the two numbers are equal, and we can modify the above proof to obtain the following result.

THEOREM 3.2. *For a balanced sequence, amortized bounds for the multi-level bucket implementation of priority queue operations are as follows: $O(1)$ for* `insert`*, $O(1)$ for* `decrease-key`*, $O(k + C^{1/k})$ for* `extract-min`*.*

For $k = 1$, the `extract-min` bound is $O(C)$. For $k = 2$, the bound is $O(\sqrt{C})$. The best bound of $O\left(\frac{\log C}{\log \log C}\right)$ is obtained for $k = \lceil \frac{\log C}{\log \log C} \rceil$.

Remark. The k-level bucket data structure uses $\Theta(kC^{1/k})$ space.

4 Hot Queues

A *hot queue* uses a heap H and a multi-level bucket structure B. Intuitively, the hot queue data structure works like the multi-level bucket data structure, except we do not expand a bucket containing less than t elements, where t is a parameter set to optimize performance. Elements of the bucket are copied into H and processed using the heap operations. If the number of elements in the bucket exceeds t, the bucket is expanded. In the analysis, we charge scans of buckets at the lower levels to the elements in the bucket during the expansion into these levels and obtain an improved bound.

A k-level hot queue uses the k-level bucket structure with an additional *special* level $k+1$, which is needed to account for scanning of buckets at level k. Only two buckets at the top level can be nonempty at any time, μ_{k+1} and $\mu_{k+1} + 1$. Note that if the queue is nonempty, then at least one of the two buckets is nonempty. Thus bucket scans at the special level add a constant amount to the work of processing an element found. We use wrap-around at level $k+1$ instead of k.

An *active* bucket is the bucket whose elements are in H. At most one bucket is active at any time, and H is empty if and only if there is no active bucket. We denote the active bucket by $B(a, b)$. We make a bucket active by making H into a heap containing the bucket elements, and inactive by reseting the heap to an empty heap. (Elements of the active bucket are both in the bucket and in H.)

To describe the details of hot queues, we need the following definitions. We denote the number of elements in $B(i, j)$ by $c(i, j)$. Given μ, $i : 1 \leq i \leq k+1$, and $j : 0 \leq j < \Delta$, we say that an element u is *in the range of $B(i, j)$* if $\underline{\mu}_i \leq \rho(u) \leq \overline{\mu}_i$, where $\underline{\mu}_i$ ($\overline{\mu}_i$) is obtained by replacing each of the $i - 1$ least significant digits of μ by 0 ($\Delta - 1$). Using RAM operations, we can check if an element is in the range of a bucket in constant time.

We maintain the invariant that μ is in the range of $B(a,b)$ if there is an active bucket.

The detailed description of the queue operations is as follows.

- `insert`

  If H is empty or if the element u being inserted is not in the range of the active bucket, we insert u into B as in the multi-level case.

  Otherwise u belongs to the active bucket $B(a,b)$. If $c(a,b) < t$, we insert u into H and $B(a,b)$. If $c(a,b) \geq t$, we make $B(a,b)$ inactive, add u to $B(a,b)$, and expand the bucket.

- `decrease-key`

  Decrease the key of an element u as follows. If u is in H, decrease the key of u in H. Otherwise, let (i,j) be the position of u in B. Remove u from $B(i,j)$. Set $\rho(u)$ to the new value and insert u as described above.

- `extract-min`

  If H is not empty, extract and return the minimum element of H. Otherwise, proceed as follows.

  Find the lowest nonempty level i. Find the first nonempty bucket at level i by examining buckets starting from $B(i,\mu_i)$.

  If $i = 1$, delete an element from $B(i,j)$, set $\mu = \rho(u)$, and return u.

  If $i > 1$, examine all elements of $B(i,j)$ and delete a minimum element u from $B(i,j)$. Set $\mu = \rho(u)$. If $c(i,j) > t$, expand $B(i,j)$. Otherwise, make $B(i,j)$ active. Return u.

Correctness of the hot queue operations follows from the correctness of the multi-level bucket operations, Lemma 3.1, and the observation that if u is in H and v is in B but not in H, then $\rho(u) < \rho(v)$.

LEMMA 4.1. *The cost of finding the first nonempty bucket at a level, amortized over the* `insert` *operations, is $O(k\Delta/t)$.*

Proof. We scan at most one nonempty bucket during a search for the first nonempty bucket. We scan an empty bucket at level i at most once during the period of time while the prefix of μ including all except the last $i-1$ digits remains the same. Furthermore, we scan the buckets only when level i is nonempty. This can happen only if a higher-level bucket has been expanded during the period the prefix of μ does not change. We charge bucket scans to insertions of these elements into the queue. Over t elements that have been expanded are charged at most k times each, giving the desired bound. ∎

THEOREM 4.1. *Let $I(N)$, $D(N)$, $F(N)$, and $X(N)$ be the time bounds for heap* `insert`, `decrease-key`, `find-min`, *and* `extract-min` *operations. Then amortized times for the hot queue operations are as follows: $O(k + I(t))$ for* `insert`, *$O(D(t) + I(t))$ for* `decrease-key`, *and $O(F(t)+X(t)+\frac{kC^{1/k}}{t})$ for* `extract-min`.

Proof. Two key facts are crucial for the analysis. The first fact is that the number of elements on H never exceeds t since each level accounts for at most t elements. The second fact is Lemma 4.1. Given the first fact and Theorem 3.1, the bounds are straightforward. ∎

For Fibonacci heaps [14], the amortized time bounds are $I(N) = D(N) = F(N) = O(1)$, and $X(N) = O(\log N)$. This gives $O(k)$, $O(1)$, and $O(\log t + \frac{kC^{1/k}}{t})$ amortized bounds for the queue operations `insert`, `decrease-key`, and `extract-min`, respectively. Setting $k = \sqrt{\log C}$ and $t = 2^k$, we get $O(\sqrt{\log C})$, $O(1)$, and $O(\sqrt{\log C})$ amortized bounds. Radix heaps achieve the same bounds but are more complicated.

For Thorup's heaps [19], the expected amortized time bounds are $I(N) = D(N) = F(N) = O(1)$, and $X(N) = O(\log^{\frac{1}{2}+\epsilon} N)$. This gives $O(k)$, $O(1)$, and $O(\log^{\frac{1}{2}+\epsilon} t + \frac{kC^{1/k}}{t})$ expected amortized time bounds for the queue operations `insert`, `decrease-key`, and `extract-min`, respectively. Here ϵ is any positive constant. Setting $k = \log^{\frac{1}{3}} C$ and $t = 2^{(k^2)}$, we get $O(\log^{\frac{1}{3}} C)$, $O(1)$, and $O(\log^{\frac{1}{3}+\epsilon} C)$ expected amortized time.

Similarly to Theorem 3.2, we can get bounds for a balanced sequence of operations.

THEOREM 4.2. *Let $I(N)$, $D(N)$, $F(N)$, and $X(N)$ be the time bounds for heap* `insert`, `decrease-key`, `find-min`, *and* `extract-min` *operations, and consider a balanced sequence of the hot queue operations. The amortized bounds for the operations are as follows: $O(I(t))$ for* `insert`, *$O(D(t) + I(t))$ for* `decrease-key`, *and $O(k + F(t) + X(t) + \frac{kC^{1/k}}{t})$ for* `extract-min`.

Using Fibonacci heaps, we get $O(1)$, $O(1)$, and $O(k + \log t + \frac{kC^{1/k}}{t})$ amortized bounds for the queue operations. Consider `extract-min`, the only operation with nonconstant bound. Setting $k = 1$ and $t = \frac{C}{\log C}$, we get an $O(\log C)$ bound. Setting $k = 2$ and $t = \frac{\sqrt{C}}{\log C}$, we get an $O(\log C)$ bound. Setting $k = \sqrt{\log C}$ and $t = 2^k$, we get an $O(\sqrt{\log C})$ bound.

Remark. All bounds are valid only when $t \leq n$. For $t > n$, one should use a heap.

Remark. Consider the 1- and 2-level implementations. Although the time bounds are the same, the two-level implementation has two advantages: It uses less space and its time bounds remain valid for a wider range of values of C.

Using Thorup's heaps and setting $k = \log^{\frac{1}{3}} C$ and $t = 2^{(k^2)}$, we get $O(1)$, $O(1)$, and $O(\log^{\frac{1}{3}+\epsilon} C)$ expected amortized time bounds.

The above time bounds allow us to get an improved bound on Dijkstra's shortest path algorithm. Suppose we are given a graph with n vertices, m arcs, and integral arc lengths in the range $[0, C]$. The running time of Dijkstra's algorithm is dominated by a balanced sequence of priority queue operations that includes $O(n)$ `insert` and `extract-min` operations and $O(m)$ `decrease-key` operations (see *e.g.* [18]). The maximum event duration for this sequence of operations is C. The bounds for the queue operations immediately imply the following result.

THEOREM 4.3. *On a network with N vertices, m arcs, and integral lengths in the range $[0, C]$, the shortest path problem can be solved in $O(m + n(\log C)^{\frac{1}{3}+\epsilon})$ expected time.*

This improves the deterministic bound of $O(m + n\sqrt{\log C})$ of [2]. (The hot queue implementation based on Fibonacci heaps matches this deterministic bound.)

5 Implementation Details

Our previous papers [7, 15] describe implementations of multi-level buckets. Our implementation of hot queues augments the multi-level bucket implementation of [15]. See [15] for details of the multi-level bucket implementation.

Consider a k-level hot queue. As in the multi-level bucket implementation, we set Δ to the smallest power of two greater or equal to $C^{1/k}$. Based on the analysis of Section 4 and experimental results, we set t, the maximum size of an active bucket, to $\lceil \frac{C^{1/k}}{4 \log C} \rceil$.

The number of elements in an active bucket is often small. We take advantage of this fact by maintaining elements of an active bucket in a sorted list instead of a heap until operations on the list become expensive. At this point we switch to a heap. We use a k-heap with $k = 4$, which worked best in our tests. (See *e.g.* [10].)

To implement priority queue operations using a sorted list, we use doubly linked list sorted in non-decreasing order. Our implementation is designed for the shortest path application. In this application, the number of `decrease-key` operations on the elements of the active bucket tends to be very small (in [16], this fact is proven for random graphs). Because of this, elements inserted into the list or moved by the `decrease-key` operation tend to be close to the beginning of the list. A different implementation may be better for a different application.

The `insert` operation searches for the element's position in the list and puts the element at that position. One can start the search in different places. Our implementation starts the search at the beginning of the list. Starting at the end of the list or at the point of the last insertion may work better in some applications.

The `extract-min` operation removes the first element of the list.

The `decrease-key` operation removes the element from the list, finds its new position, and puts the element in that position. Our implementation starts the search from the beginning of the list. Starting at the previous position of the element, at the end of the list, or at the place of the last insertion may work better in some applications.

When a bucket becomes active, we put its elements in a list if the number of elements in the bucket is below T_1 and in a heap otherwise. (Our code uses $T_1 = 2,000$.) We switch from the list to the heap using the following rule, suggested by Satish Rao (personal communication): Switch if an `insert` or a `decrease-key` operation examines more than $T_2 = 5,000$ elements. An alternative, which may work better in some applications but performed worse in ours, is to switch when the number of elements in the list exceeds T_1.

6 Experimental Setup

Our experiments were conducted on a Pentium Pro with a 166 MHz processor running Linux 1.3.68. The machine has 64 Meg. of memory and all problem instances fit into main memory. Our code was written in C++ and compiled with the Linux `gcc` compiler version 2.7.0 using the `-O6` optimization option.

We made an effort to make our code efficient. In particular, we set the bucket array sizes to be powers of two. This allows us to use word shift operations when computing bucket array indices.

The full paper reports on experimental results for five types of graphs. Two of the graph types were chosen to exhibit the properties of the algorithm at two extremes: one where the paths from the start vertex to other vertices tend to be order $\Theta(n)$, and one in which the path lengths are order $\Theta(1)$. The third graph type was random sparse graphs. The fourth type was constructed to have a lot of `decrease-key` operations in the active bucket. This is meant to test the robustness of our implementations when we violate the assumption (made in Section 5) that there are few `decrease-key` operations. The fifth type of graphs is meant to be easy or hard for a specific implementation with a specific number of bucket levels.

We tested each type of graph on seven implementations: k-ary heaps, with k=4; k-level buckets, with k ranging from 1 to 3, and k-level hot queues, with k ranging from 1 to 3. Each of these has parameters to tune, and the results we show are for the best parameter values we tested.

Most of the problem families we use are the same as in our previous paper [15]. The next two sections describe the problem families.

6.1 The Graph Types Two types of graphs we explored were grids produced using the GRIDGEN generator [7]. These graphs can be characterized by a length x and width y. The graph is formed by constructing x layers, each of which is a path of length y. We order the layers, as well as the vertices within each layer, and we connect each vertex to its corresponding vertex on adjacent layers. All the vertices on the first layer are connected to the source.

The first type of graph we used, the LONG GRID, has a constant width — 16 vertices in our tests. We used graphs of different lengths, ranging from 512 to $32,768$ vertices. The arcs had lengths chosen independently and uniformly at random in the range from 1 to C. C varied from 1 to $100,000,000$.

The second type of graph we used was the WIDE GRID type. These graphs have length limited to 16 layers, while the width can vary from 512 to $32,768$ vertices. C was the same as for LONG GRIDS.

The third type of graphs includes random graphs with uniform arc length distribution. A random graph with n vertices has $4n$ arcs.

The fourth type of graphs is the only type that is new compared to [15]. These are based on a cycle of n vertices, numbered 1 to n. In addition, each vertex is connected to $d-1$ distinct vertices. The length of an arc (i, j) is equal to $2k^{1.5}$, where k is the number of arcs on the cycle path from i to j.

The fifth type of graphs includes HARD graphs. These are parameterized by the number of vertices, the desired number of levels k, and a maximum arc length C. From C we compute p, the number of buckets in each level assuming the implementation has k levels. The graphs consist of two paths connected to the source. The vertices in each path are at distance p from each other. The distance from the source to path 1 is 0; vertices in this path will occupy the first bucket of bottom level bins. The distance from the source to path 2 is $p-1$, making these vertices occupy the last bucket in each bottom-level bin. In addition, the source is connected to the last vertex on the first path by an arc of length 1, and to the last vertex of the second path by an arc of length C.

A summary of our graph types appears in Table 1.

6.2 Problem Families For each graph type we examined how the relative performance of the implementations changed as we increased various parameters. Each type of modification constitutes a *problem family*. The families are summarized in Table 2. In general, each family is constructed by varying one parameter while holding the others constant. Different families can vary the same parameter, using different constant values.

7 Experimental Results

The 2- and 3-level bucket structures are very robust [7, 15]. In most cases, 2- and 3-level hot queues perform similarly to, although usually slightly better than, the corresponding multi-level bucket structures. One level hot queues are significantly more robust than one level buckets, but not as robust as 2- and 3-level hot queues.

Due to the shortage of space, we present experimental results for the hard problems only. These problems separate hot queues from multi-level buckets.

In the tables, k denotes the implementation: "h" for heap, "bi" for buckets with i levels, and "hi" for hot queue with i levels. We report running times and counts for operations that give insight into algorithm performance. For the heap implementation, we count the total number of `insert` and `decrease-key` operations. For the bucket implementations, we count the number of empty buckets examined (*empty operations*). For the hot queue implementations, we count the number of empty operations and the number of `insert` and `decrease-key` operations on the active bucket. We plot the data in addition to tabulating it.

We were unable to run 1-level bucket and hot queue implementations on some problems because of memory limitations. We leave the corresponding table entries blank.

Tables 3 and 4 give data for the HARD-2 and HARD-3 families, designed to be hard for 2- and 3-level bucket implementations, respectively. With at most two elements on the heap at any time, the heap implementation is the most efficient on the hard problems.

For the hot queue implementations, no bucket is expanded and the action is confined to the two special top level buckets. Thus hot queues perform almost as well as heaps. The only exception is `h1` for the largest value of C it could handle, where its running time is about 1.5 times greater than for other value of C. We have no explanation for this discrepancy.

The HARD-2 problems are hard for `b1` and `b2`, and, as expected, these implementations do poorly on this family. Similarly, all bucket implementation do worse than the other implementations on the HARD-3 family.

8 Concluding Remarks

In theory, the hot queue data structure is better than both the heap and the multi-level bucket data structures. Our experiments show that the resulting implementation is more robust than the heap or the multi-level bucket data structures.

The new heap of Raman [17] instead of Thorup's heap improves our time bound: a factor of $\log^{\epsilon} C$ is replaced by $\sqrt{\log \log C}$.

Hot queues seem more practical than radix heaps. The latter data structure requires more bookkeeping. In addition, the hot queue heap usually contains much fewer elements, and our implementation takes advantage of this fact.

The 2-level hot queue data structure seems as robust as the 3-level hot queue and is usually somewhat faster. This data structure should be best in most applications. The 3-level structure may be more robust for large values of C because the value of t is much smaller, reducing the sensitivity to the parameters for active buckets. The 1-level hot queue may be useful in event-simulation applications because it can be viewed as a robust version of the calendar queue data structure.

Acknowledgments

We would like to thank Bob Tarjan for stimulating discussions and insightful comments, Satish Rao for suggesting an adaptive strategy for switching from lists to heaps, and Harold Stone for useful comments on a draft of this paper.

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] P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Math. Systems Theory*, 10:99–127, 1977.

[4] G. S. Brodal. Worst-Case Efficient Priority Queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58, 1996.

[5] R. Brown. Calandar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem. *Comm. ACM*, 31:1220–1227, 1988.

[6] B. V. Cherkassky and A. V. Goldberg. Heap-on-Top Priority Queues. Technical Report 96-042, NEC Research Institute, Princeton, NJ, 1996. Available via URL http://www.neci.nj.nec.com/homepages/avg/pub/neci-tr-96-042.ps.

[7] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996.

[8] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. Technical Report 96-070, NEC Research Institute, Princeton, NJ, 1996. Available via URL http://www.neci.nj.nec.com/homepages/avg/pub/neci-tr-96-070.ps.

[9] R. Cole and U. Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 70:32–53, 1986.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[11] E. V. Denardo and B. L. Fox. Shortest–Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[12] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.

[13] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.

[14] 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.

[15] A. V. Goldberg and C. Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. Technical Report 95–187, NEC Research Institute, Princeton, NJ, 1995.

[16] A. V. Goldberg and R. E. Tarjan. Expected Performance of Dijkstra's Shortest Path Algorithm. Technical Report 96–062, NEC Research Institute, Princeton, NJ, 1996.

[17] R. Raman. Fast Algorithms for Shortest Paths and Sorting. Technical Report TR 96-06, King's Colledge, London, 1996.

[18] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

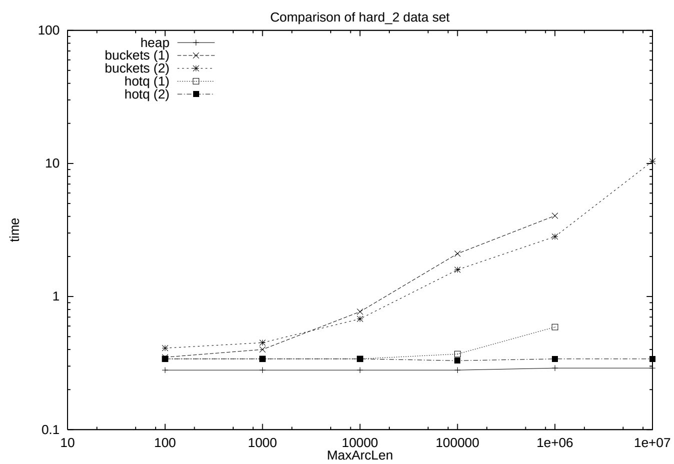
[19] M. Thorup. On RAM Priority Queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.

Name	type	description	salient feature
long grid	grid	16 vertices high $n/16$ vertices long	path lengths are $\Theta(n)$
wide grid	grid	$n/16$ vertices high 16 vertices long	path lengths are $\Theta(1)$
random	random	degree 4	path lengths are $\Theta(\log n)$
cycle	cycle	$d(i,j) = (i-j)^2$	results in many `decrease-key` operations
hard	two paths	$d(S, \text{path } 1) = 0$ $d(S, \text{path } 2) = p - 1$	vertices occupy first and last buckets in bottom level bins

Table 1: The graph types used in our experiments. p is the number of buckets at each level.

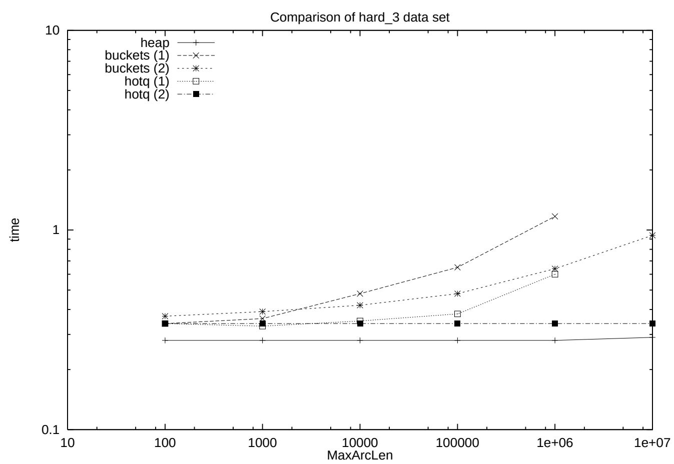
Graph type	Graph family	Range of values	Other values
long grid	Modifying C	$C = 1$ to $1,000,000$	$x = 8192$
	Modifying x	$x = 512$ to 32768	$C = 16$
			$C = 10,000$
			$C = 100,000,000$
	Modifying C and x	$x = 512$ to 32768	$C = x$
wide grid	Modifying C	$C = 1$ to $1,000,000$	$y = 8192$
	Modifying y	$y = 512$ to 32768	$C = 16$
			$C = 10,000$
			$C = 100,000,000$
	Modifying C and y	$y = 512$ to 32768	$C = y$
random graph	Modifying C	$C = 1$ to $1,000,000$	$n = 131072$
	Modifying n	$n = 8,192$ to $524,288$	$C = 16$
			$C = 10,000$
			$C = 100,000,000$
	Modifying C and n	$n = 8,192$ to $524,288$	$C = n$
cycle	Modifying n	$n = 300$ to 1200	$d = 20$
			$d = 200$
hard	Modifying C	$C = 100$ to $10,000,000$	$n = 131072, p = 2$
			$n = 131072, p = 3$

Table 2: The problem families used in our experiments. C is the maximum arc length; x and y the length and width, respectively, of grid graphs; d is the degree of vertices in the cycle graph; and p is the number of buckets at each level.



k	MaxArcLen	100	1000	10000	100000	1000000	10000000
h	**time**	**0.28 s**	**0.28 s**	**0.28 s**	**0.28 s**	**0.29 s**	**0.29 s**
	heap ops.	262143	262143	262143	262143	262143	262143
b1	**time**	**0.35 s**	**0.40 s**	**0.77 s**	**2.10 s**	**4.05 s**	
	empty	917488	1966048	8257408	33422848	66976768	
b2	**time**	**0.41 s**	**0.45 s**	**0.68 s**	**1.59 s**	**2.82 s**	**10.38 s**
	empty	917488	1966048	8257408	33422848	66976768	268300288
b3	**time**	**0.41 s**	**0.39 s**	**0.40 s**	**0.42 s**	**0.41 s**	**0.44 s**
	empty	3	7	131085	393209	393209	917489
h1	**time**	**0.34 s**	**0.34 s**	**0.34 s**	**0.37 s**	**0.59 s**	
	empty	0	0	0	0	0	
	act. bucket	122880	129024	130560	130816	131008	
h2	**time**	**0.34 s**	**0.34 s**	**0.34 s**	**0.33 s**	**0.34 s**	**0.34 s**
	empty	0	0	0	0	0	0
	act. bucket	126976	129024	130560	130944	131008	131056
h3	**time**	**0.34 s**	**0.34 s**	**0.33 s**	**0.34 s**	**0.34 s**	**0.33 s**
	empty	0	0	0	0	0	0
	act. bucket	129024	130560	130816	130944	131040	131056

Table 3: Hard problems for the 2-level implementation. $n = 131072$.



k	MaxArcLen	100	1000	10000	100000	1000000	10000000
h	**time**	**0.28 s**	**0.28 s**	**0.28 s**	**0.28 s**	**0.28 s**	**0.29 s**
	heap ops.	262143	262143	262143	262143	262143	262143
b1	**time**	**0.34 s**	**0.36 s**	**0.48 s**	**0.65 s**	**1.17 s**	
	empty	393208	917488	1966048	4063168	8257408	
b2	**time**	**0.37 s**	**0.39 s**	**0.42 s**	**0.48 s**	**0.64 s**	**0.94 s**
	empty	393208	917488	1966048	4063168	8257408	16645888
b3	**time**	**0.40 s**	**0.42 s**	**0.46 s**	**0.54 s**	**0.69 s**	**0.99 s**
	empty	393208	917488	1966048	4063168	8257408	16645888
h1	**time**	**0.34 s**	**0.33 s**	**0.35 s**	**0.38 s**	**0.60 s**	
	empty	0	0	0	0	0	
	act. bucket	126976	130048	130944	131040	131064	
h2	**time**	**0.34 s**	**0.34 s**	**0.34 s**	**0.34 s**	**0.34 s**	**0.34 s**
	empty	0	0	0	0	0	0
	act. bucket	129024	130048	130944	131056	131064	131071
h3	**time**	**0.33 s**	**0.33 s**	**0.34 s**	**0.34 s**	**0.33 s**	**0.34 s**
	empty	0	0	0	0	0	0
	act. bucket	130048	130816	131008	131056	131068	131071

Table 4: Hard problems for the 3-level implementation. $n = 131072$.