Performance of Priority Queues Under a Variant of the Hold Model

Mauricio Marín†
Escuela de Computación
Universidad de Magallanes
Angamos esq. Zenteno
Punta Arenas, Chile

Ricardo Baeza-Yates‡
Depto. de Cs. de la Computación
Universidad de Chile
Blanco Encalada 2120
Santiago, Chile

Abstract

In the last decades a number of near optimal priority queues have been developed for simulations based on discrete events. Many of these priority queues have been compared and analyzed using a model for a list of events in an empirical manner. This paper studies the performance of priority queues under a variant of this model which mimics the requirements of a new class of applications: multiple-object simulation. Our results show that an unusual adaptation of a strictly balanced binary tree gives the best performance in all the cases considered.

1 Introduction

Efficient solutions for the implementation of priority queues (PQs) have been proposed by several authors [1]-[7]. Most of these PQs have been empirically compared using different approaches and applications [8]-[12]. From this it has been concluded that no single implementation is the best for all cases and applications. For example, the tests reported in [12] show that the pairing heap [5] is the best structure for the minimum spanning tree problem. But the tests presented in [10] show that the splay tree [3] is the best structure under the empirical hold model [13] with several probability distributions for the priorities. The hold model is a simple model for simulations based on discrete events. Moreover, our results in hard-particle simulations — a particular case of multiple-object simulation — show that the complete binary tree (CBT) [6] is by far the best structure in this class of applications [14, 15].

It is interesting to note that hard-particle simulations impose requirements over the PQ that are quite similar to the empirical hold model used to compare PQs. In fact, a hard-particle simulation may be seen as a sort of hold model with exponential distribution for the priorities.

This paper is intended to provide additional insight into the performance of some of the fastest PQs reported in the literature by using a particular variant of the empirical hold model and ranging through several probability distributions for the priorities. The aim was to mimic the PQ work-load in the general arena of multiple-object simulation. We conclude that the CBT still maintains its optimal performance being clearly more efficient than all of the other PQs tested in this work. So the results here presented extend the ones in [14, 15] drawing

---

*Partially funded by Chilean CONICYT project 1950622.
†E-mail: mmarin@cs.f.umag.cl
‡E-mail: rbaeza@dcc.uchile.cl
conclusions from a more general viewpoint and complement the empirical work presented in [8]:[12] (note that we also include results for some PQs with no empirical tests reported in the literature).

The next sections are organized as follows: section 2 explains the hold model and the new variant. Section 3 presents the details about the experiments performed and section 4 presents the conclusions obtained of the experimental results.

2 A Model for Discrete Event Simulations

Any discrete event simulation needs to manage a list of events. The event with minimal time in the list is the next event to happen. To handle efficiently the extraction of the minimal value, a priority queue must be used (in this case, priority is given by the event time). To compare different priority queues is then necessary to model the list of events. In other words, it is necessary to have a generator of events that allows direct measurement of the efficiency attained by the PQ while performing event management. One simple model to simulate an event list is the hold model [13].

The hold model has became a standard in the analysis of priority queues and pending event set implementations. It has been used, for example, in [8, 9, 10, 13] and its origin is related with the use of PQs for the efficient management of events generated during discrete-event simulations. In this case, the priority of each item in the PQ represents the time at which some event is scheduled to occur. Most typical PQ operations used for event-management has been the INSERT operation which schedules an event in the PQ and the EXTRACTMIN operation which retrieves the next chronological event (i.e. the item with highest priority) from the PQ or pending event set.

The hold model is based on an idealized simulation strategy where each event that takes place causes the scheduling of exactly one new future event. So after inserting $N$ items in the PQ this model consists on performing a big loop where in each cycle it is performed an EXTRACTMIN followed by an INSERT. The priority value (event-time) of the new item inserted is calculated as $key + X$ where $key$ is the priority of the item most recently dequeued by EXTRACTMIN, and $X$ is a random priority increment taken from a probability distribution. This has the advantage of allowing direct measurement of the PQ performance as a function of the number of items enqueued in the data structure (the size of the PQ remains constant among hold-cycles). Such a model has been accepted as a valid and useful formalism to analyze and compare PQs by many authors interested in the efficient implementation of general-purpose pending event sets. It is important to note that even under this simple model, analytical analysis of priority queues has proven very hard [13].

Discrete-event simulation of multiple-object systems can be considered as a good candidate to be described by the standard hold-model. However, this kind of simulation has a well defined behavior which can be described using a better model. The model that we use in this work views the system as composed by $N$ objects where each object has only one event scheduled in the PQ. Some of these events marks the instants at which two objects have an interaction, and other events are only associated with the object owner so that its occurrence has no effects in other objects of the system. We model this by incorporating a probability $\rho$ to the standard hold model. After the next event is extracted and a new event is inserted, with probability $\rho$ a random event is taken out of the event list and a new one inserted. In both cases, the time of the new event is computed as in the standard model (always using the time of the current
event as base time). When only the next event is deleted, we have the standard model. That is, the event does not affect the rest of the system. If a second event is deleted, means that another object was affected by the current event (for example, a collision between two objects). This probability $\rho$ can be seen as modeling the density of the objects in the system. When $\rho$ is 0, we have the standard model (that is, all events are independent of other objects). On the other hand, when $\rho$ is one, all events affect another object. So, these two types of events are alternated in a random manner as shown in the pseudo-code given in figure 1.

```
InitQueue( N_nodes );
// Start measuring running time
Loop( number_of_trials )
  (Object,Time) = ExtractMin();
  Insert( Object, Time+Random() );
  If ( Uniform() < Rho ) // Probability parameter
    Object = Random_Object();
    Delete( Object );
    Insert( Object, Time+Random() );
  ENDIf
ENDLoop
// Stop measuring running time
```

Figure 1: Pseudo-code for the variant of the hold model.

3 Experimental Design

The PQs tested together the CBT were the *implicit heap* [6, 7], the *skew heap* with top-down variant [4], the *binary priority queue* [7], the leftist tree [6, 7], the binomial queue [2], the priority tree [1, 7], the pairing heap [5], the splay tree with bottom-up splaying [3] and the binary search tree [7] used as a priority queue.

Note that the CBT has not been considered in the literature as a useful data structure in practice until [16]. However, this structure seems to be very efficient in the class of problems which we are interested, namely, situations in multiple-object simulation where the EXTRACT-MIN operation may be followed by the possible elimination of an arbitrary node, and then the insertion of one or two new nodes (details can be found in [14, 15]). Also, each object maintains an one-to-one correspondence with the nodes in the PQ and the data structure must support the bi-directional mapping $Object \leftrightarrow Priority$.

Notice that if the number of objects is constant it is not necessary to perform node deletions since we can make node displacements as the object priorities change. This was exploited in the implementation of the *complete binary tree* and *implicit heap*, although we also obtained results making node deletions in these PQs. We have tested other heap ordered structures for which deletions can be avoided, but we have not considered such an alternative because the update algorithms (up/down shifting) become identical to the *implicit heap*.

---

1In [6] the *complete binary tree* is described only as a first approximation to the *implicit heap*. 
<table>
<thead>
<tr>
<th>Distribution</th>
<th>Exprasion used to compute the random priority increments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>$-\ln \mu$</td>
</tr>
<tr>
<td>Uniform 0.0-2.0</td>
<td>$2\mu$</td>
</tr>
<tr>
<td>Biased 0.9-1.1</td>
<td>$0.9 + 0.2\mu$</td>
</tr>
<tr>
<td>Bimodal</td>
<td>$0.095238\mu + \text{if } \mu &lt; 0.1 \text{ then } 9.5238 \text{ else } 0.0$</td>
</tr>
<tr>
<td>Triangular</td>
<td>$1.5\mu^{0.5}$</td>
</tr>
</tbody>
</table>

Table 1: Probability distributions where $\mu$ is a random value uniformly distributed between 0.0 and 1.0.

The function Random() used in Figure 1 is built with different probability distributions. We have used the similar distributions of [10] for the priority increments, namely the biased, bimodal, exponential, triangular and uniform distributions all with expected value 1.0. Table 1 shows the details of the distributions used. For each distribution we obtained results for $\rho = 0.0, 0.3, 0.6$ and 0.9. The number of trials was $10^6$ in all the experiments.

To obtain the execution times of each priority queue the standard clock() function was used. Each experiment was repeated 10 times observing an error rate below 0.1%. This error is defined as the ratio between the standard deviation and the average.

In each experiment and before performing any time measuring the PQ is initialized with $N$ nodes taking random priorities, and then a loop of $10^6$ hold operations is performed in order to reach the steady state in the PQ (shape and priorities). After this, a new loop of $10^6$ hold operations is performed with void PQ operations (functions without code) to measure the overhead involved in the code no related with the PQ (main loop, random number generation, etc.). Finally, a loop of $10^6$ hold operations is performed to measure the average cost of each hold cycle. For the two loop above the same sequence of random numbers is used and before each loop all the priorities are decreased in $x$ to avoid overflow in floating point operations, where $x$ is the smallest priority in the PQ.

The experiments were carried out in a workstation DG Avion with UNIX at minimal load (no other users and no background processing), and programs written in C language compiled with the gcc compiler at the maximal speed option O2. To avoid the effects of paging activity in the total running time a relatively small number of nodes ($10 \to 10^4$) was used. To avoid calls to the operating system due to priority queues that use dynamic memory allocation, all the memory required by these priority queues was pre-allocated before the start of each experiment. The programming style was the traditional one with minimal calls to subroutines and no recursive algorithms. All code used in this work is available from the authors by e-mail. The C-Language implementation for the complete binary tree that we tested is given in the Appendix at the end of this paper.
4 Conclusions

In Figures 2-6 we show the empirical results obtained. Note that all our results are presented as the ratio, $T_{PQ}/T_{CBT(a)}$, of the running time $T_{PQ}$ associated with one of the priority queues over the total running time obtained with the complete binary tree implementation which does not perform node deletions ($T_{CBT(a)}$).

It is shown in Figures 2-6 that the CBT without node deletions is the best structure in all the cases. This includes the standard hold model ($\rho = 0$). An important fact is that this structure seems to be more efficient than other PQs as the number of nodes increase, namely the ratio $T_{PQ}/T_{CBT(a)}$ increases with the number of nodes. There is, however, an inflection in $N = 5000$ for all the curves showed in the graphs. It is because 5000 is a more distant of a power of two than 10000 or 100000, which makes the CBT a bit more slower, reducing a little the ratio $T_{PQ}/T_{CBT(a)}$ (this does not imply that the other PQs are more efficient for this particular $N$). Note that only the ratio $T_{CBT(b)}/T_{CBT(a)}$ decreases all the time with the number of nodes. All these properties make the CBT very convenient for the simulation of large systems.

For the node deletion case, however, several other structures were more efficient than CBT(b) for systems with $N < 5000$. Node deletion is useful in systems where the objects are created and eliminated dynamically during simulation. But it is by far less probable to have a system where the objects are created and eliminated in a manner as dramatic as in the hold model used in our experiments. So in systems with variable number of objects it is better to implement an hybrid structure between the cases with node deletions and without node deletions. That is, the Delete() operation is applied only when an object is really removed from the system and the no-node-deletion case (CBT(a)) is used during all the life-time of the object.

Without considering the CBT, our results for other PQs are not similar to the ones presented by Jones [10] in many aspects (the results of [10] must be compared with our results for $\rho = 0.0$). For example, our implementation of the implicit heap seems to be more efficient than the one used in [10]. But perhaps the most interesting difference is related with the performance of the splay tree (ST). In our results the ST was not the most efficient structure for all distributions as in [10]. In our case, the ST was one of the less efficient PQs tested (note that for the ST we have used a C-code written by Jones, available through Internet). Also, conversely to Jones, we have included results for the binary search tree (BST) where it is shown that in the case of exponential distribution the BST is more efficient than the ST (the ST may be considered as a BST with a balance heuristic). This confirms the analytical results of [13] since the BST remains balanced for exponential distribution doing evident the overhead involved in the code oriented to balance the tree. We attribute our differences with [10] to the different programming languages and architectures used; Pascal over a CISC architecture for Jones versus C-Language over a RISC architecture in our experiments.

Also it is interesting to observe in Figures 2-6 how sensible are some PQs to the probability distributions and how this sensibility decreases as $\rho$ increases. Our results confirm that for the standard hold model ($\rho = 0.0$) there are PQs with poor performance under the biased, bimodal, triangular and uniform distributions. Only under the exponential distribution all the PQs have relatively good performance.

Finally, another important fact to be observed from Figures 2-6 is how efficient are the different PQs in performing the DELETE operation. This can be observed seeing how the relative position of each PQ changes as the value of $\rho$ increases (larger $\rho$ implies larger arbitrary-node deletions).
Figure 2: Total running time for the Biased Distribution. Each curve shows the ratio $(T_{PQ}/T_{CBT(a)})$ between the running time with a priority queue PQ and the running time with the complete binary tree that does not perform node deletions. In the graphics each priority queue PQ is identified by a letter: (a) CBT without node deletions, (b) CBT with node deletions, (c) Heap without node deletions, (d) Heap with node deletions, (e) Binomial Queue, (f) Binary Priority Queue, (g) Binary Search Tree, (h) Leftist Tree, (i) Pairing Heap, (j) Priority Tree, (k) Skew Heap and (l) Splay Tree.
Figure 3: Total running time for the Bimodal Distribution. Each curve shows the ratio $(T_{PQ}/T_{CBT}(a))$ between the running time with a priority queue PQ and the running time with the complete binary tree that does not perform node deletions. In the graphics each priority queue PQ is identified by a letter: (a) CBT without node deletions, (b) CBT with node deletions, (c) Heap without node deletions, (d) Heap with node deletions, (e) Binomial Queue, (f) Binary Priority Queue, (g) Binary Search Tree, (h) Leftist Tree, (i) Pairing Heap, (j) Priority Tree, (k) Skew Heap and (l) Splay Tree.
Figure 4: Total running time for the Exponential Distribution. Each curve shows the ratio \( T_{PQ}/T_{CBT(a)} \) between the running time with a priority queue PQ and the running time with the complete binary tree that does not perform node deletions. In the graphics each priority queue PQ is identified by a letter: (a) CBT without node deletions, (b) CBT with node deletions, (c) Heap without node deletions, (d) Heap with node deletions, (e) Binomial Queue, (f) Binary Priority Queue, (g) Binary Search Tree, (h) Leftist Tree, (i) Pairing Heap, (j) Priority Tree, (k) Skew Heap and (l) Splay Tree.
Figure 5: Total running time for the Triangular Distribution. Each curve shows the ratio $(T_{PQ}/T_{CBT(a)})$ between the running time with a priority queue PQ and the running time with the complete binary tree that does not perform node deletions. In the graphics each priority queue PQ is identified by a letter: (a) CBT without node deletions, (b) CBT with node deletions, (c) Heap without node deletions, (d) Heap with node deletions, (e) Binomial Queue, (f) Binary Priority Queue, (g) Binary Search Tree, (h) Leftist Tree, (i) Pairing Heap, (j) Priority Tree, (k) Skew Heap and (l) Splay Tree.
Figure 6: Total running time for the Uniform Distribution. Each curve shows the ratio $(T_{PQ}/T_{CBT(o)})$ between the running time with a priority queue PQ and the running time with the complete binary tree that does not perform node deletions. In the graphics each priority queue PQ is identified by a letter: (a) CBT without node deletions, (b) CBT with node deletions, (c) Heap without node deletions, (d) Heap with node deletions, (e) Binomial Queue, (f) Binary Priority Queue, (g) Binary Search Tree, (h) Leftist Tree, (i) Pairing Heap, (j) Priority Tree, (k) Skew Heap and (l) Splay Tree.
Appendix: Complete Binary Tree as a Priority Queue

To obtain the node with highest priority (event with lesser time in multiple-object simulation) a complete binary tree (CBT) which performs a binary tournament between all the priorities is used. Each leaf has an object number (i.e. object identifier) and each internal node (recursively up to the root) has the object number with highest priority of its two children. Therefore the root of the tree has the object number with highest priority (see Figure 7.a). Every time a new priority is computed for an object \( i \), the tournament is updated for all nodes in the path from the leaf labeled with \( i \) to the root (see Figure 7.b).

![Diagram of a CBT for ten objects (priority values in parentheses).](image)

![Diagram of a CBT updated after changing the priority for object 4 to 0.18.](image)

Figure 7: a) CBT for ten objects (priority values in parentheses).

b) CBT updated after changing the priority for object 4 to 0.18.

The CBT is implemented using an array of \( 2N - 1 \) integers. A node at position \( n \) has its children in positions at \( 2n \) and \( 2n + 1 \). The parent of node \( n \) is in the position \( \lfloor \frac{n}{2} \rfloor \) of the array. All internal nodes are stored between positions 1 and \( N - 1 \).

Deletions from the CBT are performed by removing the rightmost leaf and exchanging it with the target leaf to be deleted. Then we have to update the CBT considering the above changes. Insertions are performed by appending a new rightmost leaf and updating the CBT. In this case it is necessary to have an additional array to map from objects to leaves.
The CBT is updated with the following subroutine,

```c
void UpdateCBT(i)
    int i; /* object number */
{
    int f, /* father */
        l, /* left child */
        r, /* right child */
        w; /* old winner */

    /* At least it is necessary to cover the old path of object i */
    for( f=CBT[i+N-1]/2; f>0; f=f/2 ) {
        if ( CBT[f]!=i ) break; /* jumps to the next "for" */
        l = CBT[f*2];
        r = CBT[f*2+1];
        if ( Priority[l] < Priority[r] )
            CBT[f] = l; /* if true then l has greater priority than r */
        else
            CBT[f] = r;
    }

    /* Now the event time comparisons are stopped as soon as possible */
    for( ; f>0; f=f/2 ) {
        w = CBT[f]; /* old winner */
        l = CBT[f*2];
        r = CBT[f*2+1];
        if ( Priority[l] < Priority[r] )
            CBT[f] = l;
        else
            CBT[f] = r;
        if ( CBT[f] == w ) return; /* end of priority comparisons */
    }
} /* End of UpdateCBT */
```
References


