A NEW FORM OF EFFICIENT TREE-BASED PRIORITY QUEUES FOR DISCRETE EVENT SIMULATION

Rick Siow Mong Goh[*] • Ian Li- Jin Thng[+] • Wai Teng Tang • Marie Therese Quieta
Department of Electrical and Computer Engineering, National University of Singapore
3 Engineering Drive 3, CCN Laboratory, Singapore 117576
Email: <[*]engp1815@nus.edu.sg,[+]eletlj@nus.edu.sg>

KEYWORDS

Priority queue, splay tree, skew heap, calendar queue.

ABSTRACT

A priority queue plays an important role in stochastic discrete event simulations for as much as 40% of a simulation execution time is consumed by the *pending event set* management. This article describes a new form of tree-based priority queues which employs the *demarcation* procedure to systematically split a single tree-based priority queue into many smaller trees, each divided by a logical time boundary. These new *Demarcate Construction* priority queues offer an average speedup of more than twice over the single tree-based counterparts and outperform the current expected $O(1)$ Calendar Queue in many scenarios. Their superior performance renders them suitable for many applications such as discrete event simulators.

INTRODUCTION

In stochastic discrete event simulation (DES), we often observe that the known kinds of efficient tree-based priority queues such as the Splay Tree (Sleator and Tarjan 1985) and Skew Heap (Sleator and Tarjan 1986) only have at best an *amortized* time bound of $O(\log(n))$ per operation, where by amortized time is meant the time per operation averaged over a worst-case sequence of operations (Tarjan 1985). Comparatively, multilist-based priority queues such as the Calendar Queue (CQ) (Brown 1988) and its variant Dynamic CQ (DCQ) (Oh and Ahn 1998) offer an "expected" $O(1)$ average time bound per operation, where by "expected" is meant that the CQs are not theoretically proven to be $O(1)$ but rather displays an $O(1)$ performance in numerous scenarios. However, the drawback of employing the CQs is that the worst-case time bound per operation can be as poor as $O(n)$ (Rönngren and Ayani 1997). That said, the CQ has been chosen as the *pending event set* (PES) structure in various simulators such as the popular Network Simulator v2 (Fall and Varadhan 2002).

In DES, the PES is defined as the set of all events generated during a DES and of which the events have not been simulated yet. The basic operations of the enqueue and dequeue of events define the PES as a priority queue of events with the minimum time-stamp having the highest priority and maximum time-stamp having the least priority. Comfort (Comfort 1984) has revealed that up to 40% of the computational effort in a simulation may be devoted on the management of the PES alone, where the *enqueue* and *dequeue* operations account for as much as 98% of all operations on the PES. A DES frequently operates in a three-step cycle: *dequeue* – removal of an event with the highest priority from the PES; *execute* – processing this dequeued event; *enqueue* – insertion of new event/s resulting from the execution into the PES. The two basic operations, enqueue and dequeue, have run-time complexity closely dependent on the total number of events in the PES. Therefore, a PES structure should be efficient especially for large-scale simulations that involve large number of events during simulation jobs.

In most applications the metric of interest for a priority queue is often the time required to perform the most common operations. This metric is referred to as *access* time. In DES, the total run-time of the simulation job is by far more important than the individual times of the operations, except for real-time applications. Therefore, the amortized (or average) access time per operation is by far more important than the worst-case access time for each individual operation. Fine-grain simulations, such as but not limited to ATM network simulations, are time-consuming due to the huge number of events to process (Oh and Ahn 1998). The faster and the larger the networks, the higher the number of events would be in the PES and the longer run-times these network simulations would require, which may take days or weeks to yield results with an acceptable level of statistical error. For example, experiments conducted in Tcpsim (Dupuy et al. 1990) for a three-minute simulated time over Sun Ultra 1 took more than one day execution time on average (Oh and Ahn 1998). Therefore, to speed up simulation jobs, one approach is to develop high-performance priority queue structures for the PES.

In this article we develop the *Demarcate Construction* (*Demarco*) priority queue, a multilist-based structure which is made up of two building blocks. The name *Demarco* arises from the word "demarcate" which means to divide and separate clearly as if by boundaries. The primary structure is an array of buckets, where each bucket may contain a tree holding *near-future* events. The secondary structure is made up of a simple unsorted linked list to hold *far-future* events. *Demarcation* refers to the process of constructing the primary structure and transferring events from the secondary structure to the primary. In an amortized sense, this demarcation process ensures that a tree-based priority queue has

Proceedings 18th European Simulation Multiconference
Graham Horton (c) SCS Europe, 2004
ISBN 3-936150-35-4 (book) / ISBN 3-936150-36-2 (CD)

comparable performance or better, than one which does not undergo demarcation.

DEMARCATE CONSTRUCTION

The *Demarcate Construction* (*Demarco*) has four essential principles. First and foremost, the concept of demarcation is to have many trees each containing a small number of events. In contrast, a tree-based priority queue manages only a single tree containing all the nodes or events. Upon applying demarcation, an array of logical buckets is constructed. Each bucket spans equal time-interval and these buckets systematically enable the events to be demarcated and distributed in the buckets. Thus on the average, the tree in each bucket will have a smaller number of events leading to a much reduced height as compared to a single tree priority queue.

Secondly, *Demarco* defers the sorting of events until necessary. At the onset, all enqueued events are appended in the secondary tier (*SecT*) of *Demarco*. These events are not sorted according to their timestamps. During the first dequeue operation, the primary tier (*PriT*) is constructed and the events are inserted into the corresponding buckets in *PriT* where they are sorted according to the tree-based priority queue's native enqueue algorithm.

Thirdly, unlike other multilist-based priority queues (e.g. the CQs), *Demarco* does not rely on sampling heuristics to obtain structure parameters. The parameters used when constructing *PriT* are obtained from the events distribution in the *SecT*.

Lastly, the algorithm of *Demarco* proceeds in demarcation *cycles* where by a *cycle* is defined as the duration when: the events in *SecT* are transferred to the *PriT*, more events are enqueued in *PriT* and *SecT*, and all the events in the *PriT* are dequeued.

Basic Structure of Demarco

The main building blocks of the *Demarcate Construction* (*Demarco*) consist of:

1. Primary Tier (*PriT*) – an array of buckets where each bucket may contain a tree. Each tree-node contains an event holding a near-future (i.e. soon to be dequeued) timestamp. Within each bucket, the events are sorted according to the algorithm of the tree-based priority queue. The parameters used in creating the *PriT* are obtained from the events distribution in *SecT*.
2. Secondary Tier (*SecT*) – an unsorted singly linked list. Acting as an overflow list to contain far-future events, *SecT* buffers events that do not affect the *PriT*. This reduces the number of events in the *PriT* and thus, on the average, the number of events in each bucket decreases as simulation time progresses. Since the performance of tree-based priority queues depends on the height (or number of levels), reducing the number of events in *PriT* will eventually lead to a reduction in the height of the tree in the buckets in *PriT*. This leads to a superior overall performance.

The Demarco Algorithm

Though the *Demarco* is a multilist-based structure alike the CQs, *Demarco* marks the first departure from the CQs' resize triggers and sampling heuristics to obtain structure parameters such as the number of buckets and the bucketwidth. Instead of the static methodologies used in the CQs, *Demarco* employs a dynamic approach of updating its structure parameters by making *PriT* structure parameters (i.e. bucketwidth and number of buckets) to be dependent on the events distribution in *SecT*. Since the *Demarco* proceeds in cycles, the structure parameters of *PriT* gets renewed according to the most current events and are not affected by the past events. This process removes the need to have costly resize operations found in the CQs. This becomes more vivid when the enqueue and dequeue operations are described.

The *Demarco* structure keeps a set of variables to function and they are defined as follow:

PriT_Start – Used for calculating the bucket-index of the event which is to be enqueued in *PriT*. It is set to *SecT_Min* during each Demarcation process, where by events are transferred from *SecT* to *PriT*.
PriT_Num – Number of events in *PriT*.
PriT_Bw – Bucketwidth of *PriT*.
PriT_Index – Bucket-index of the first non-empty bucket bucket in *PriT*.
SecT_Cur – Minimum timestamp of an event that can be enqueued in *SecT*. This value will be set equal to *SecT_Max* at each transfer of events from *SecT* to *PriT*.
SecT _Min – Minimum timestamp in *SecT*.
SecT _Max – Maximum timestamp in *SecT*.
SecT _Num – Number of events in *SecT*.

The Demarco Algorithm – Dequeue Operation

At the onset, all enqueued events are placed in *SecT* in a FIFO manner without time-order thus leaving *PriT* being empty. On the first dequeue operation, *PriT* is constructed and thereafter, all the events are transferred from *SecT* to *PriT*. The bucketwidth of *PriT*, an important structure parameter, is dynamically assigned using equation (1).

$$PriT\_Bw = Bucketwidth = \frac{SecT\_Max - SecT\_Min}{SecT\_Num} \quad (1)$$

The number of buckets to be created in *PriT* is set to be *SecT_Num*, giving an average of one event per bucket on the assumption that the event distribution is a uniform distribution. Though in practical scenarios this may not be true, the *Demarco* will still perform well because the enqueue of events into *PriT* is $O(\log(n_B))$ per event whereby n_B is the number of events in a bucket. For most scenarios, $n_B << N$, where N is the total number of events in the *Demarco* structure.

After the construction of *PriT*, the events in *SecT* are transferred to *PriT*. Transferring of an event into *PriT* is alike enqueuing an event into *PriT* which utilizes the tree-based priority queue's native enqueue algorithm. Thereafter, the highest priority event would be in the first bucket in *PriT* (where *PriT_Index* = 0 and that *PriT*_Start = *SecT_Min* have been initialized). On each dequeue, the highest priority event would be removed from the first bucket in *PriT* by employing the tree-based priority queue's native dequeue algorithm. Subsequently, when the first bucket is empty, it is *invalidated* and the second bucket is then considered, where at the same time, parameter *PriT*_Index is incremented by one. If the second bucket is empty, *PriT_Index* is incremented again until a non-empty bucket is found and the current highest priority event is dequeued. After all the events in *PriT* are dequeued, i.e. all the buckets are empty, the demarcation cycle repeats itself with *SecT* treating the next dequeue to be alike the first dequeue as mentioned.

The Demarco Algorithm – Enqueue Operation

For each enqueue operation, *Demarco* checks if that event timestamp is greater than *SecT_Cur*. If so, the event is simply placed at the end of the linked list in *SecT*. If the event is not inserted in *SecT*, then the event is enqueued in *PriT*. On enqueuing in *PriT*, the bucket-index of the bucket where this event is to be inserted in *PriT* is:

$$Bucket\_index = \left\lfloor \frac{timestamp - PriT\_Start}{PriT\_Bw} \right\rfloor \qquad (2)$$

and the event is enqueued according to the tree's native enqueue algorithm.

PERFORMANCE MEASUREMENT TECHNIQUES

The performance of priority queues are often measured by the average access time of the enqueue and dequeue operations under different load conditions. The parameters to be varied for each priority queue performance benchmark are: the access pattern, the priority increment distribution and the queue size. The access pattern models used are the Classic Hold and Up/Down. They emulate the steady-state and the transient phase of a typical simulation respectively. The various priority increment distributions tested are as shown in Figure 1 and the queue size ranges from 100 to 1 million. These benchmark scenarios had also been commonly applied in (Jones 1986; Rönngren et al. 1993; Rönngren and Ayani 1997; Oh and Ahn 1998). The experiments were carried out on an AMD Athlon MP 1.2GHz dual-processor server running the priority queues sequentially. Required memory was pre-allocated. All code was written in the C programming language. 10 runs of each experiment were done.

The Camel(x,y) distribution is used to model bursty traffic in computer and communication networks which represents a highly-skewed distribution. The parameters used for Camel(x,y) result in two humps with x probability mass being concentrated in the two humps. The duration of the humps makes up y of an interval, where x and y are (0,1). Change(A,B,x) is a compound distribution that combines priority distribution A and B, with x priority increments being alternately drawn by A and B. In our experiments, Camel(0.999,0.001) and Change(Exp(1),Triangle,2000) are used.

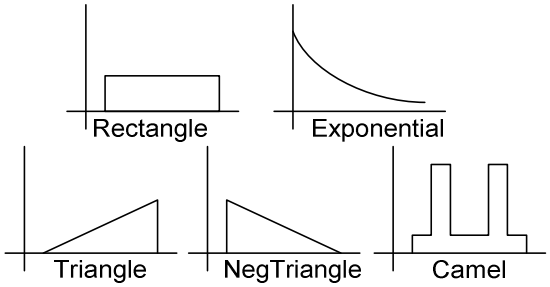


Figure 1: Priority Increment Distributions

EXPERIMENTAL RESULTS

The objectives of this section are firstly to present the performance of tree-based priority queues with and without *Demarco*. Secondly, we compare *Demarco* priority queues with the current fastest multilist-based queues – CQ and DCQ. Lastly, we would like to determine *Demarco* priority queues' generality and sensitivity in the six priority increment distributions using the Classic Hold and Up/Down models, as well as when the queue size increases from 100 to 1 million. Note that a logarithmic scale has been used for the queue-size axis which leads to logarithmic complexity for linear plots.

Steady-State Phase Experiments

Figures 2(a) to 2(f) show the results obtained under the Classic Hold experiment which is commonly employed to test the steady-state performance of the priority queues. Note that the obvious knee seen in the graphs is due to the declining cache performance and occurs when the queue size is about 10,000. This phenomenon is also observed in the graphs in (Rönngren and Ayani 1997) where the experiments were done on SUN and Intel architectures.

Figures 2 show vividly that the performance of *Demarco* structures, i.e. *Demarco*-Skews and *Demarco*-Splays, outperform the tree-based priority queues; Skew Heap and Splay Tree, where by *Demarco*-Skews/-Splays is made up of a *Demarco* structure where each bucket in *PriT* of *Demarco* may contain a Skew Heap/Splay Tree. At larger queue sizes, the performance speedup that *Demarco* offers is more than three times. Figures 2(a) to 2(d) show that the performance of the *Demarco* structures are comparable to the expected $O(1)$ complexity multilist-based priority queues, i.e. CQ and DCQ. Furthermore, Figures 2(e) and 2(f) demonstrate clearly that the *Demarco* structures outperform the CQs which have erratic performance for skewed distributions such as the Camel and Change.

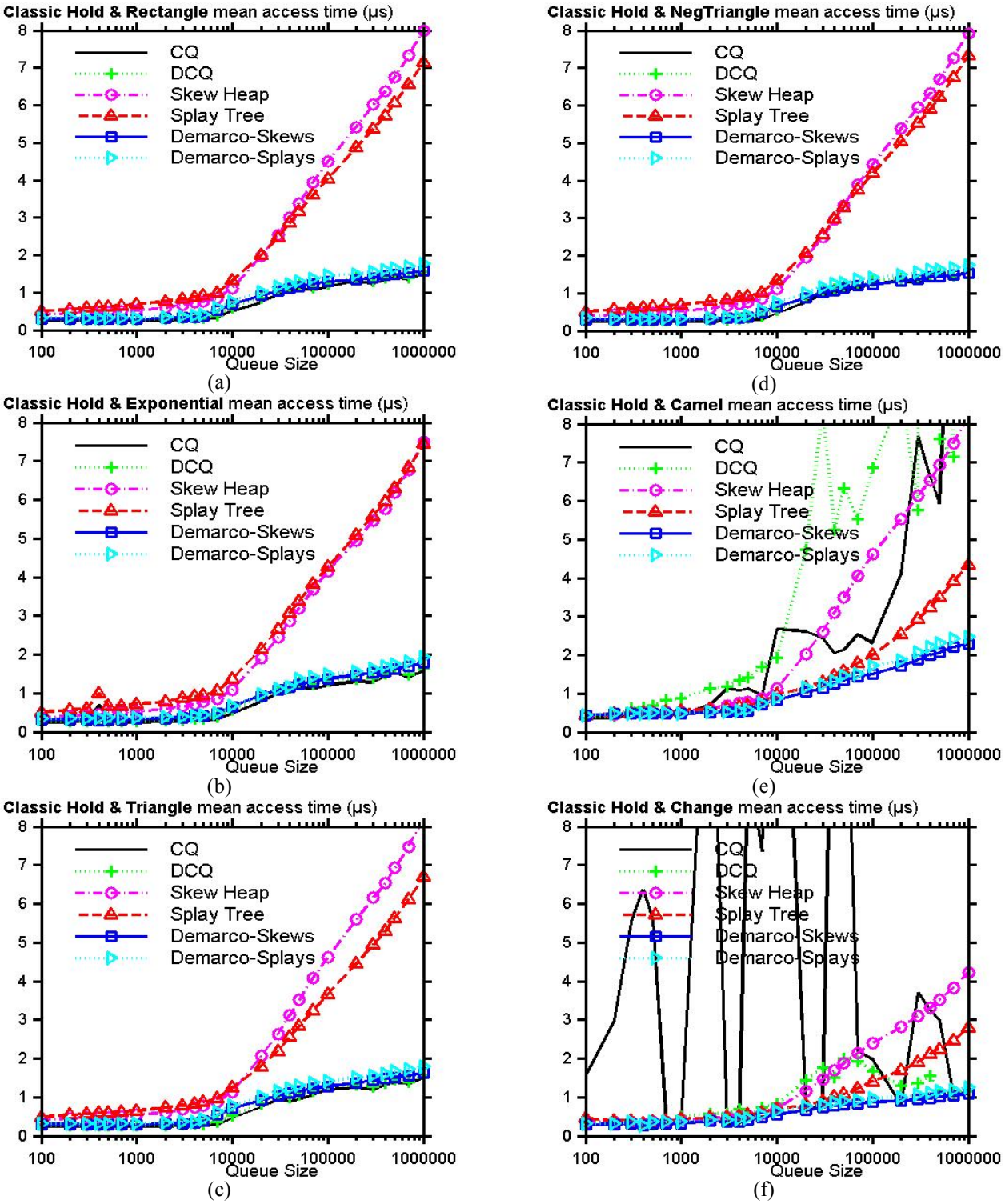


Figure 2: Performance Graphs for Classic Hold Model Experiments.

Transient Phase Experiments

The Up/Down model which tests the performance of priority queue structures during transient periods when the queue size fluctuates frequently, reveals the weaknesses of the CQs. Figures 3(a) to 3(d) show that the CQs experience several peaks and these suggest strongly that the resize operations found in the CQs can be costly since the CQs resize whenever the queue size fluctuates by factors of two. The form of triggers found in the CQs are clearly inflexible because even though the CQs can be performing well with its existing operating parameters, but because of their static triggers, they still have to resize whenever the queue size fluctuates by factors of two. Figures 3(e) and 3(f) again demonstrate that the CQs are sensitive to skewed distributions. The *Demarco* structures outperform all the priority queues in all these experiments.

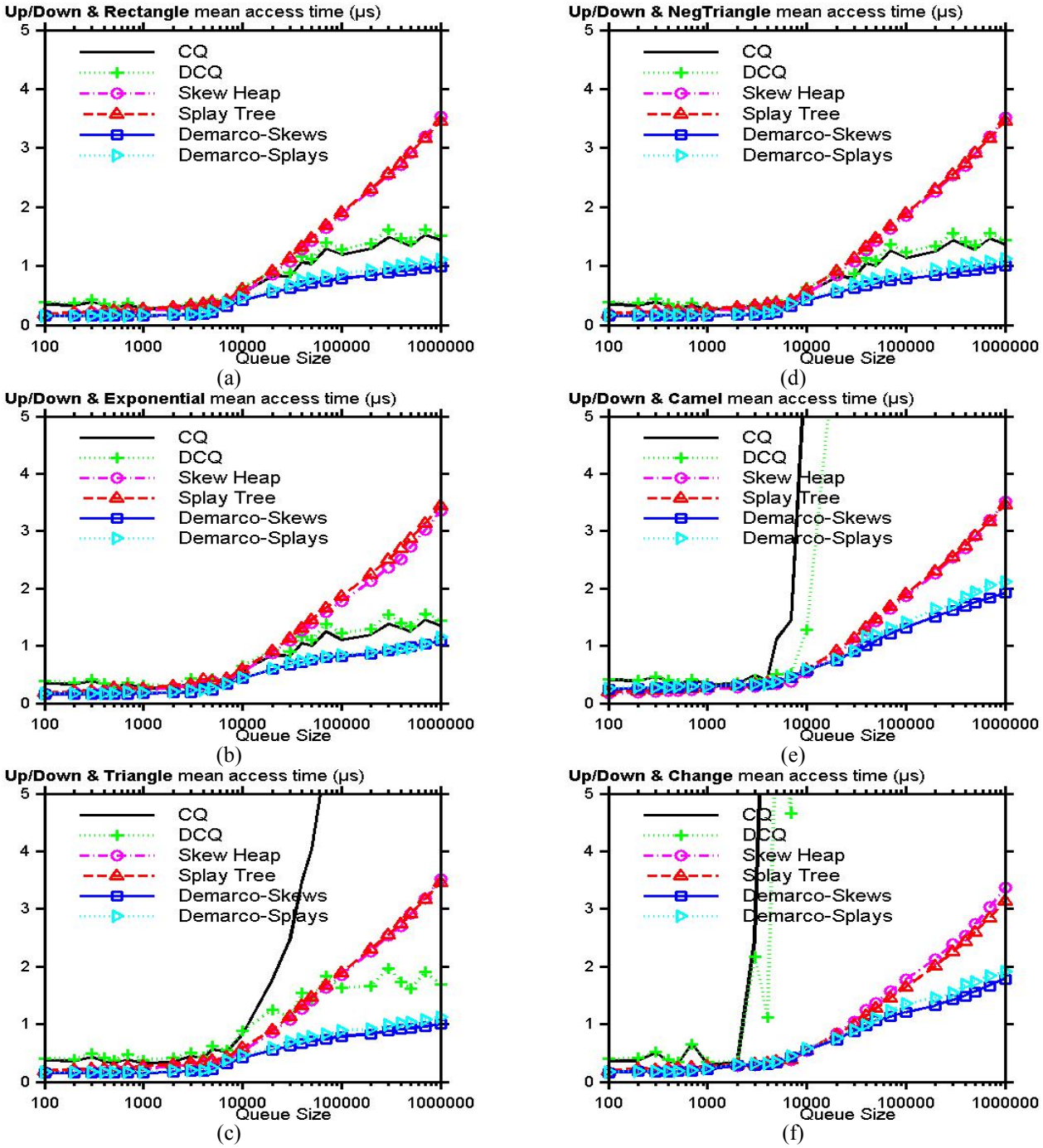


Figure 3: Performance Graphs for Up/Down Model Experiments.

Overall Performance Comparison

This section illustrates numerically the performance speedup of the *Demarco* structures over the normal single tree-based priority queues. In addition, we compare the relative performance of the *Demarco* structures and tree-based priority queues versus the multilist-based CQ and DCQ.

Table 1 shows that the speedup offered by the *Demarco* structure is more than two times, average over all queue sizes and distributions. Table 2 illustrates the relative performance of all the priority queues considered. The *Demarco*-Skews and *Demarco*-Splays outperform their tree-based counterparts and are generally more stable than the CQs.

Table 1: Speedup Offered by the *Demarco* Structure Normalized Over a Single Tree-Based Priority Queue – Comparison by Priority Increment Distribution

Distribution	*Demarco*-Skews	*Demarco*-Splays
Rectangle	2.99	2.61
Exponential	2.62	2.66
Triangle	3.01	2.46
NegTriangle	2.99	2.66
Camel	2.05	1.31
Change	1.91	1.39
Average	**2.60**	**2.18**

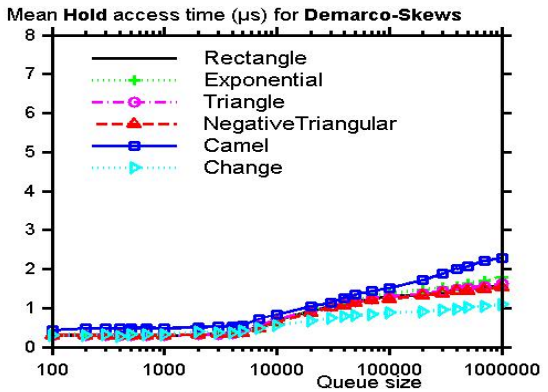
Table 2: Relative Average Performance for All Distributions
(Normalized Respect to Fastest Access Time where 1.00 is the Fastest)

Model	Queue Size	*Demarco*-Skews	*Demarco*-Splays	Skew Heap	Splay Tree	CQ	DCQ
Classic Hold	100	1.07	1.14	1.31	1.61	1.59	1.00
	1000	1.15	1.26	1.76	2.15	1.00	1.41
	10000	1.00	1.10	1.55	1.70	5.32	1.23
	100000	1.00	1.12	3.27	2.59	1.20	1.81
	1000000	1.00	1.11	4.44	3.61	1.90	NA*
	Average	1.04	1.15	2.47	2.33	2.20	NA*
Up/Down	100	1.00	1.09	1.01	1.12	2.17	2.38
	1000	1.00	1.06	1.25	1.42	1.56	1.75
	10000	1.00	1.08	1.16	1.27	19.55	15.96
	100000	1.00	1.10	1.93	1.95	NA*	NA*
	1000000	1.00	1.10	2.67	2.63	NA*	NA*
	Average	1.00	1.09	1.60	1.68	NA*	NA*
Total Average		**1.02**	**1.12**	**2.04**	**2.01**	NA*	NA*

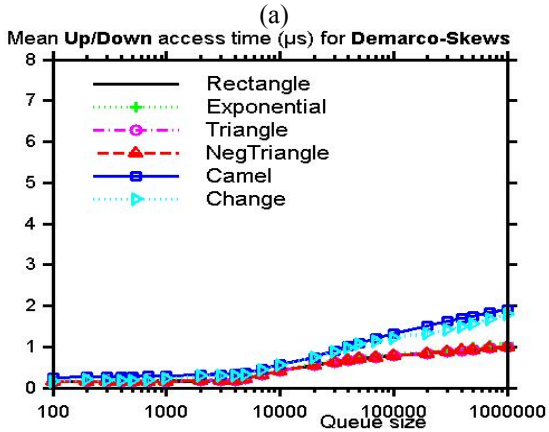
* NA is meant that some of the access times are too high in at least one or more distributions. Thus the results are not considered in this comparison.

Generality and Sensitivity of Demarco Structures

Figures 4(a) and 4(b) show the generality and insensitivity of *Demarco*-Skews under the various distributions and queue sizes (*Demarco*-Splays has similar graphs and is thus not included). Though the performance of *Demarco*-Skews may differ by as much as twice for different distributions, the complexity is still considered near $O(1)$. Furthermore, the graphs show that it is stable for all the distributions unlike the CQs which is near $O(n)$ for skewed distributions. This superior performance is made possible because of the four essential principles mentioned.



(a)



(b)

Figure 4: Performance Graphs for *Demarco*-Skews

CONCLUSION

Demarcate Construction is a new form of tree-based priority queues which employs the *demarcation* process. These new priority queues offer an average speedup of more than twice over the single tree-based counterparts and outperform the current expected $O(1)$ Calendar Queues in many scenarios. Its generality in small to large queue sizes (100 to 1 million events), insensitivity to priority increment distributions and low overhead costs, make it a superior priority queue for many applications such as the pending event set structure in discrete event simulators.

REFERENCES

Brown, R. 1988. "Calendar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem." *Commun. ACM 24*, 12 (Dec.), 825-829.

Comfort, J. C. 1984. "The Simulation of a Master-Slave Event Set Processor." *Simulation 42*, 3 (March), 117-124.

Dupuy, A., Schwartz, J., Yemini, Y. and Bacon, D. 1990. "NEST: A Network Simulation and Prototyping Testbed." *Commun. ACM 33*, 10 (Oct.), 63-74.

Fall, K. and Varadhan, K. 2002. The ns Manual. UCB/LBNL/VINT Network simulator v2. http://www.isi.edu/nsnam/ns/.

Jones, D. W. 1986. "An Empirical Comparison of Priority-Queue and Event-Set Implementations." *Commun. ACM 29*, 4 (April), 300-311.

Oh, S., and Ahn, J. 1998. "Dynamic Calendar Queue." In *Proceedings of the 32nd Annual Simulation Symposium*, 20-25.

Rönngren, R. and Ayani, R. 1997. "A Comparative Study of Parallel and Sequential Priority Queue Algorithms." *ACM Trans. Model. Comput. Simul. 7*, 2 (April), 157-209.

Schwetman, H. 1996. CSIM18 User's Guide. Austin, TX: Mesquite Software, Inc.

Rönngren, R., Riboe, J., and Ayani, R. 1993. "Lazy Queue: New Approach to Implementing the Pending Event Set." *Int. J. Computer Simulation 3*, 303-332.

Sleator, D. D. and Tarjan, R. E. 1985. "Self-Adjusting Binary Search Trees." *Journal of the ACM 32*, 3 (July), 652-686.

Sleator, D. D. and Tarjan, R. E. 1986. "Self-Adjusting Heaps." *SIAM Journal of Computing 15*, 1 (Feb.), 52-69.

Tarjan, R.E. 1985. "Amortized Computational Complexity." *SIAM Journal on Algebraic and Discrete Meth. 6*, 2 (April), 306-318.