

Revisiting Priority Queues for Image Analysis

Cris L. Luengo Hendriks

*Centre for Image Analysis, Swedish University of Agricultural Sciences,
Box 337, SE-75105 Uppsala, Sweden.
phone: +46 18 471 3471
fax: +46 18 55 34 47*

Abstract

Many algorithms in image analysis require a priority queue, a data structure that holds pointers to pixels in the image, and which allows efficiently finding the pixel in the queue with the highest priority. However, very few articles describing such image analysis algorithms specify which implementation of the priority queue was used. Many assessments of priority queues can be found in the literature, but mostly in the context of numerical simulation rather than image analysis. Furthermore, due to the ever-changing characteristics of computing hardware, performance evaluated empirically 10 years ago is no longer relevant. In this paper I revisit priority queues as used in image analysis routines, evaluate their performance in a very general setting, and come to a very different conclusion than other authors: implicit heaps are the most efficient priority queues. At the same time, I propose a simple modification of the hierarchical queue (or bucket queue) that is more efficient than the implicit heap for extremely large queues.

Key words: Image Analysis, Priority Queue, Heap, Binary Search Tree, AVL Tree, Splay Tree, Red-Black Tree, Ladder Queue, Hierarchical Heap, Grey-Weighted Distance Transform, Watershed

1. Introduction

A priority queue is a data container that supports insertion of a new data element and extraction of the element with the highest priority. These operations are often called enqueue and dequeue, or `insert` and `delete-min` (since lower values are usually associated to higher priority). Various implementations complement these with other operations, such as `find-min`, `merge`, `delete` and `decrease-key`. Each of these operations can be performed in a fraction of a microsecond on modern hardware. However, a typical program might perform millions of these operations, and spend an important fraction of total execution time accessing the queue. It is therefore important to select an efficient queue implementation.

Many algorithms for image analysis, especially those that can be described with the “recursive propagation” paradigm, can be implemented efficiently using priority queues. Examples are the distance transform in non-convex domains [1], the grey-weighted distance transform [2, 3], fast marching level sets [4], morphological reconstruction [5], area and attribute openings, closings and thinnings [6, 7], the watershed transform [8], region growing [9] and skeletonisation [10]. All these algorithms are implemented using only `insert` and `delete-min`. Therefore, this paper considers only these two operations.

For all of these algorithms, the priority given to the elements in the queue are either pixel values in the input image or propagation distances. Whether this priority value is given in integer or floating-point representation depends on the application.

The advantages of using integer values are reduced memory usage and faster arithmetic operations. Both these advantages are becoming less significant: modern computers can perform floating-point calculations just as fast as integer calculations, and memory prices no longer are a limiting factor. The advantages of using a floating-point representation are the increased accuracy of the result and the near elimination of problems related to overflow, underflow and rounding. For these reasons, this paper explicitly examines only priority queues that use a floating-point priority value. This excludes the very efficient hierarchical queue (also known as bucket queue), in which there is a separate list for each possible priority. In this paper I present a slight modification of the hierarchical queue that makes it applicable when the priority value has a very large range or is not integer. The computational overhead makes it more efficient than the heap only when the queue size is very large. Note that, for some applications, slightly altering the processing order of pixels introduces an error smaller than the accuracy of the algorithm. For these cases it is possible to quantise priority values, enabling the use of the hierarchical queue [11].

Some priority queue algorithms are stable, meaning that elements with identical priority are dequeued in the same order that they were enqueued (in FIFO order, “first in, first out”). Non-stable queues can be made stable with the addition of one integer to each enqueued element [12]. Stability hasn’t been considered in this paper, though it is important for several algorithms such as the watershed and skeletonisation algorithms.

Algorithm performance can be evaluated theoretically or empirically. Theoretical performance, derived from e.g. the number of comparisons, is used extensively in the literature [for

Email address: cris@cb.uu.se (Cris L. Luengo Hendriks)

example, see [13, 14, 15, 16] but is not representative of performance in practice, due to the complex multilevel caching [17] and pipelined computation cores of modern computing architectures. On the other hand, empirically evaluated performance depends on the design of the input data set, and is relevant only for the class of computer used in the evaluation. The empirical evaluation presented here uses a very simple random model for the input data, and shows how the performance is affected by the choice of input. Because of this input data model and because only the two operations needed for image analysis algorithms are considered in the evaluation, the results presented in this paper apply specifically to image analysis applications. For an application that stores larger blocks of data with each key, and for applications that require other than `insert` and `delete-min` operations, the best option for priority queue will be different than that recommended here.

Good priority queue comparison papers appear occasionally [18, 19, 20, 21], though I have been able to find only one dedicated explicitly to image analysis applications [12]. In that paper, Breen and Monro conclude that the splay tree and the AVL tree, two types of self-adjusting binary search trees, perform better than the heap data structure. These three priority queue algorithms are included in the present comparison. They also evaluated the hierarchical heap and propose the SplayQ. Both were found to outperform other data structures, but are by definition limited to a relatively small set of possible priorities, and are therefore not considered here.

Jones [18] concluded from his empirical comparison of priority queues that “implicit heaps are [...] consistently worse than many other priority queue implementations.” In contrast, LaMarca and Ladner [17] concluded the opposite: “the low memory overhead of implicit heaps makes them an excellent choice as a priority queue.” The difference between these two studies is 10 years of computer hardware development. LaMarca and Ladner used a computer with 3 levels of cache, and for which main memory access is expensive compared to comparisons. Their theoretical algorithm evaluation is based on analysing cache misses rather than number of comparisons. This paper will discover what changes are introduced by another 12 years of hardware development.

2. Priority Queue Algorithms

2.1. Implicit Heaps

The implicit heap is widely used as a priority queue because it does not have any memory overhead. It is also the oldest priority queue implementation with an $O(\log N)$ performance (with N the number of items in the queue) [18]. The heap is not stable, but can be made stable with the addition of an integer to each element that stores the enqueue order. In short, the N elements in the queue are stored in an array, which implicitly represents a complete binary tree of height $\lceil \log_2 N \rceil$. Each array element i has its two children, at array locations $2i + 1$ and $2i + 2$. The heap property, that each element has higher priority (lower value) than its two children, is always maintained. Thus, the highest priority element is always at the root of the

tree (array element 0). When dequeuing, the last element in the array is placed at the root and iteratively swapped with its highest-priority child until the heap property is restored. To enqueue an element, it is appended to the array, then percolated up the tree until the heap property is restored. Both operations are $O(\log N)$. The dequeue operation typically needs to move an element from the top all the way down, performing roughly $2 \log_2 N$ comparisons, since that element was originally at the bottom of the tree. An enqueue operation will move the new item up 1.6 levels on average, independent of N , if all the priority values are from the same uniform distribution [22]. However, in the image analysis algorithms mentioned in the introduction, new priorities tend to be lower than old priorities, further reducing the cost of the enqueue operation.

The heap can be generalised to a k -way heap. The heap described above is a 2-way heap, since each element has two children. It is trivial to adapt this such that each node i has k children at $ki + 1$, $ki + 2$, ... $ki + k$. Increasing k reduces the height of the tree, but increases the number of comparisons that need to be done at each node. The advantage to using a larger k is increased memory locality of the elements compared, which can increase speed when reading data from cached memory [17] (see Subsection 3.3).

2.2. Binary Search Trees

When dequeuing, the implicit heap requires taking the last element in the array, an element that is at the bottom of the tree, moving it to the root, then repeatedly exchanging it with one of its children until the heap property is restored, most probably when the element is back at the bottom of the tree. The reason this is necessary is because the heap requires the binary tree to be complete, otherwise the array it is stored in would have gaps. That is, each level, except for the last one, is full, and the last level has all elements as far to the left as possible. When representing a binary tree using explicit pointers to child nodes, there is more flexibility in the placing of the elements in the tree. Furthermore, moving nodes around only requires swapping child pointers, not copying the data stored in the nodes. Of course, with small data items as used for image analysis algorithms, swapping the data is not much more expensive than swapping the pointers.

A binary search tree uses such explicit child pointers to create a tree where a node’s left child is smaller than the node itself, and its right child is larger. To find an element, one compares the value searched for with the root node’s value, and moves to its left or right child depending on the result. This comparison is repeated until the required value is found. For a priority queue, where the element with the highest priority typically is the one with the lowest value, no comparisons are necessary. One simply descends iteratively to the left child until a node is found without left child. This is the lowest-valued item in the tree. Deleting this node is accomplished simply by replacing it with its right child, if it has one. To enqueue an element, a search is performed, descending all the way to the bottom of the tree, where the new element can be appended. Binary search trees are stable priority queues if an element to be inserted is

considered larger than any equal valued element already in the tree.

These operations are all $O(\log N)$ as long as the tree is balanced (i.e. its leaves are all at the same depth). When the tree becomes severely unbalanced, performance will degrade significantly. In the extreme, the tree can become a linear linked list.

Many techniques have been described to maintain the tree balanced, most notably the AVL tree [23], the red-black tree [23] and the splay tree [24]. They all use node rotations to rebalance the tree, but use different techniques to determine which rotations are needed. Self-balancing trees are also stable.

The AVL tree adds a balance value to each node, that keeps track of the difference in height of its left and right branches. When this difference is larger than 1, one or two node rotations bring the balance at that node back to 0. After inserting or deleting a node, the algorithm walks back to the root, updating, checking and rebalancing nodes.

The red-black tree adds a colour to each node, and thereby the binary tree can simulate a 2-3-4 tree (i.e. a tree where each internal node has either two, three or four children). A 2-3-4 tree has all leafs at the same level. In the red-black tree, each red node is part of the same 2-3-4 node as its parent. Therefore, each red node must have only black children. To keep this invariant, rotations of nodes need to be performed when inserting or deleting a node, and thereby the tree is kept in balance. The code for a red-black tree is very complex when compared to either the AVL tree or the splay tree. Sedgwick [25] recently presented a small modification to the red-black tree, in which, if a node has only one red child, it is required that the red child is to the left. This greatly reduces the number of possible configurations to examine when keeping the tree invariance, thereby simplifying the code. He calls this structure the left-leaning red-black tree. This is the version of the red-black tree used in the present comparison.

Finally, the splay tree, introduced by Sleator and Tarjan [24], has the advantage of not requiring an extra integer to be stored with each node. Instead, both insertions and deletions are performed through an operation called splaying, in which the accessed node is moved to the root of the tree. For insertion, the node closest in value to the new element is splayed to the root, then the new item is inserted as the new root. For dequeuing, the lowest-valued node is splayed to the root and removed. The splay tree is not designed as a priority queue, but rather as a search tree in which elements are accessed multiple times. The splay operation keeps frequently accessed elements near the root of the tree, thereby optimising the search tree. However, Breen and Monro [12] found the splay tree to work better as a priority queue than other self-balancing binary trees.

There exist many more variations on the tree theme than the three described above. They are not included in the present study because their main strength is not relevant to the algorithms used for image analysis. The Fibonacci heap [26] is expected to be efficient when `merge` or `decrease-key` are needed. The relaxed heap [27] and the 2-3 heap [28] are improvements on the Fibonacci heap. All three are based on the premise of

delaying the sorting of enqueued elements. This idea is carried further by several much more complex data structures, such as the ladder queue.

2.3. Ladder Queue

The two-list structure [29] simply divides the set of queued elements into near future (NF) and far future (FF). Elements in the NF are sorted, those in the FF are not. When enqueueing, only elements that fall in the NF section need to be sorted, other elements can simply be appended to the FF list. When dequeuing, the first element for the sorted list can directly be extracted. Only when the NF becomes empty is it necessary to do any work. In this case, a new threshold is determined and a subset of the FF is sorted into the NF list.

This structure has been refined as the lazy queue [30] and later the ladder queue [31]. These two queues divide the data set into three groups. The NF is again a sorted list. The FF is some data structure with many unsorted bins, such that elements in the first bin all have higher priority than the elements in the second bin, and so forth. Finally, the very far future (VFF) is an unsorted list, which is completely emptied into the FF when needed. Adding the middle layer improves performance, and the structure of this middle layer is the difference between the lazy queue and the ladder queue. The ladder queue is a multi-scale structure, where, if a bin is too large to sort into the NF list, a new rung in the ladder can be added. The bin can now be sorted over many smaller bins, one of which can be sorted into the NF. The ladder queue is the only queue in this comparison that is expected to have an $O(1)$ amortized cost for enqueue and dequeue operations [31].

As opposed to the heap and the binary search trees, the ladder queue expects newly enqueued elements to be larger than the last dequeued element, and becomes highly inefficient when this requirement is not met. Similarly, the queue is most efficient when all newly enqueued elements can be inserted into the VFF rather than the FF or NF. Thus, this data structure is potentially very efficient for algorithms such as the grey-weighted distance transform, which tends to produce new priority values that are larger than most of the values currently in the queue, but is not at all applicable to an algorithm such as the seeded watershed, which can enqueue pixels with a lower priority value than any seen previously.

2.4. Hierarchical Queue

When the priority values are limited to small integers (e.g. digital images often have 8-bit or 12-bit integers as pixel values when they come off the imaging sensor), it is possible to allocate a FIFO queue (bucket) for each possible priority value. An array contains a pointer to each of these buckets, and when enqueueing an element, the correct bucket can be directly indexed using the priority value. Both enqueue and dequeue are $O(1)$ operations. This is called a hierarchical queue in the literature [e.g. 12]. As mentioned in the introduction, the limitation of priority values to small integers makes this data structure inapplicable in a general algorithm, though it is the best choice under certain circumstances. Breen and Monro [12] suggest the

SplayQ as an alternative to the hierarchical queue. The SplayQ is a splay tree with a FIFO queue at each node, and its advantage is that it is not limited to small integers. However, enqueue and dequeue still require traversing a binary search tree with k nodes, k being the number of different priority values on the queue. This means that, if k is small compared to the total number of elements N in the queue, the SplayQ is an efficient data structure. However, when priority values are taken from a very large set, k will be close to N , and the algorithmic complexity reverts to that of the splay tree.

2.5. Hierarchical Heap

To make the hierarchical queue applicable when priority values are not limited to small integers, I propose using an implicit heap instead of a FIFO queue for each bucket, and a simple linear mapping of priority values to bucket indices (as in the computation of the histogram of an image). Computing the bucket index i from the priority value p is thus accomplished with

$$i = \lfloor \frac{p - p_{min}}{p_{max} - p_{min}} k \rfloor, \quad (1)$$

where k is the number of buckets, p_{min} is the lowest and p_{max} is the highest expected priority value. This still limits priority values to a predefined range, but it is no longer necessary to have as many buckets as different values exist in that range. For added flexibility, when i exceeds $k - 1$, the element can simply be added to the last bucket. In this case, when most elements exceed the expected priority range, the algorithmic complexity of the hierarchical heap reverts to that of the implicit heap.

By dividing the N elements in the queue over k buckets, the average size of each heap is reduced by a factor k , and the enqueueing and dequeuing operations thus are of order $O(\log N/k)$ if the buckets are chosen correctly. Of course, the actual algorithmic complexity depends strongly on the distribution of priority values. Calculating the bucket index adds a constant time to the enqueue operation that needs to be amortised by the $O(\log k)$ reduction in time to enqueue the element in the implicit heap. It is therefore necessary to choose k large enough.

3. Implementation Aspects

When implementing the algorithms described in Section 2, it is possible to include many optimisations. This section details some of the optimisations mentioned in the literature.

3.1. Dynamically Increasing Array Size

The implicit heap is implemented in an array, a contiguous block of memory of a fixed size. Usually, at the start of the image analysis algorithm it is not possible to know how large the queue will become. Therefore it is important to be able to enlarge the array as necessary. The most trivial implementation [32] is to allocate a new, larger array, and copy the data over. The C function `realloc` can do this efficiently when there is a block of free memory available directly after the array to be

enlarged; it does not need to copy any data in this case. If data needs to be copied, this is an $O(N)$ operation. To amortise this cost and keep the average enqueue operation at $O(\log N)$, the array needs to be doubled in size every time a new element does not fit [33]. That is, the memory block is always between N and $2N$ in size.

An alternative implementation of resizable arrays divides the array into equal-sized blocks. The array index must then be translated to a block number and an index into the block. Some additional overhead is required to manage these blocks: either keep a list of pointers to the blocks, or put the blocks in a linked list or binary tree. For more elegant approaches see e.g. Brodnik et al. [34] or Demaine [33]. A block-based approach to the array might be efficient in environments that do not natively support resizing memory blocks.

3.2. Dedicated Memory Management

The binary search tree and its derivatives are composed of nodes, small blocks of data containing one enqueued element (a pointer) and its priority value (a floating-point number), and two pointers to the node's children. On a 64-bit computer this is a 32 byte block. When allocating such a node through the C library's `malloc` function on my Linux computer, this block actually requires 40 bytes of storage, the overhead is for the system to know how large the block is. This means that 20% of the memory used by the queue is wasted. If instead the nodes are allocated in larger groups of, for example, 1000 nodes, the overhead is reduced to 0.025%.

To manage these allocated but unused nodes, my implementation links them in a list. When a new node is needed, the first one in the linked list is used. When a node is deleted, it is inserted at the beginning of this linked list. Both operations are constant-time operations that do not depend on the number of nodes allocated in each block.

3.3. Memory Cache

Processor speeds have increased far beyond the speed at which data can be moved from the main memory to the processor. To alleviate this bottleneck, processors include a small amount of memory that works at the same clock speed as the processor itself. Often there will be a second or third level cache that is larger but works at a lower speed. When a memory address is accessed, a block of memory around it is moved to the cache, under the assumption that the program will access nearby addresses. Cache management logic has become very complex, to better predict what pieces of memory will be needed and what pieces of memory can be moved out of the cache. Consequently, programs perform differently today than they did when Jones [18] or LaMarca and Ladner [17] did their priority queue comparisons. Currently, caches are better adapted to real-world algorithms, meaning it is less important to tailor the algorithms to match the system's cache. One thing still holds true: if all the data that a program uses fit in the cache, the program will run faster than if the data do not fit. Furthermore, if the data do not fit in the cache, the program will run faster if the subset of the data it needs for one task is grouped together in memory, rather than scattered over the address space.

LaMarca and Ladner [17] and Naor et al. [35] describe methods to cluster the nodes in implicit heaps and binary trees to improve their performance on cached memory.

3.4. Recursion

Heaps and binary search trees, like many other algorithms, can be described very elegantly using recursive functions. Computer science classes typically teach to remove recursion for better performance. This is not necessary if the function is tail recursive, that is, it calls itself just before exiting. In this case, any good optimising compiler will create object code that is just as efficient as the non-recursive version of the code [36] (or even more efficient, as can be seen in the experimental results in Section 4). If the function is not tail-recursive, removing recursion is more complex. The performance hit of function calls is platform-dependent.

4. Comparison

All experiments, except where indicated, were performed on a Hewlett Packard workstation with an Intel Xeon E5430 CPU (a quad-core, 64-bit processor clocked at 2.66 GHz) and 16 GiB¹ of memory, running Red Hat Linux 5. This computer is representative for the systems typically used today for image analysis applications, though measurements vary from system to system. Code was compiled using the GNU C compiler (GCC) version 4.1.2 with the `-O3` option, which turns on all optimisations.

4.1. The Algorithms

I implemented and tested the algorithms listed in Table 1. The code used for this empirical comparison is available online at <http://www.cb.uu.se/~cris/priorityqueues.html>, and includes everything required to reproduce the graphs in this paper.

4.2. Memory Requirements

I used the Linux command `ps` to measure the amount of memory used by a process. Two of the values it can return, resident set size (RSS) and virtual memory size (VSZ), are useful for this task. RSS indicates the amount of memory that a process has in use and is not swapped out to the page file. As long as the process does not exceed the available main memory, this number will indicate the amount of physical memory used by that process. This is not the same as the amount of memory allocated by that process: `malloc` reserves a contiguous block in the address space, but only the portion of this block that has been accessed is mapped to the physical memory. VSZ is the size of the address space reserved for the process. When a program starts, the system reserves a fixed amount of address space for it. A call to `malloc` will increase the virtual memory size when the requested block size exceeds the space available.

Table 1: Algorithms implemented for this comparison, and their names as referenced in the figures.

Name	Description
Heap (PA)	The implicit heap with the array split over blocks, and an array with pointers to these blocks.
Heap	The implicit heap with the array dynamically grown, using <code>realloc</code> .
4-heap	The implicit 4-way heap, using <code>realloc</code> .
H-heap	The hierarchical heap (Subsection 2.5), using 1024 buckets in the range [0,100]. Each bucket is implemented as ‘Heap’.
BST (SM)	The binary search tree, with each node allocated independently using <code>malloc</code> .
BST	The binary search tree, with nodes allocated in groups of 1024.
BST (NR)	The binary search tree, with nodes allocated in groups of 1024, using non-recursive code.
AVL tree	The AVL tree, using code from Pfaff [37], and nodes allocated in groups of 1024.
LLRB tree	Left-leaning red-black tree, using code from Sedgewick [25], and nodes allocated in groups of 1024.
Splay tree	The splay tree, using code from Sleator [38] (top-down splaying, non-recursive code), and nodes allocated in groups of 1024.
Ladder Q	The ladder queue, according to Tang et al. [31]; additional details in Appendix.

Figure 1(a) shows the RSS for the test program when using the various priority queues, for a wide range of queue sizes. For simplicity, only data for five of the algorithms are plotted, each of the other priority queues produced (nearly) identical results to one of the five plotted algorithms.

The various versions of the heap all occupy roughly the same amount of memory, close to the theoretical minimum. The splay tree and the binary search trees need about twice as much memory as the heap needs (the two child pointers take up as much space as the data stored in each node). The left-leaning red-black tree and the the AVL tree occupy 25% more than the binary search tree. This is because these two data structures use one extra bookkeeping value in each node. Both the AVL and the red-black trees can be implemented by writing the bookkeeping values to the lower bits of the child pointers, which are not used because of the alignment to 8-byte boundaries. I have not implemented this, as it complicates the code and increases computational cost. The binary search tree using the system `malloc` requires almost 50% more space than the same tree using a dedicated memory management. This does not match with the overhead estimated in Subsection 3.2. The hierarchical heap has some memory overhead when compared to the implicit heap, because it starts off with 1024 small heaps. As the amount of enqueued elements increases, the relative overhead becomes smaller. Finally, the ladder queue has a huge overhead because the VFF and the FF are both large enough to hold the full initial data set. The imposed limit of 900000 elements in the VFF (see Appendix) can clearly be seen in the horizontal

¹In this paper I use 1 GiB = 2¹⁰ MiB = 2²⁰ KiB = 2³⁰ bytes.

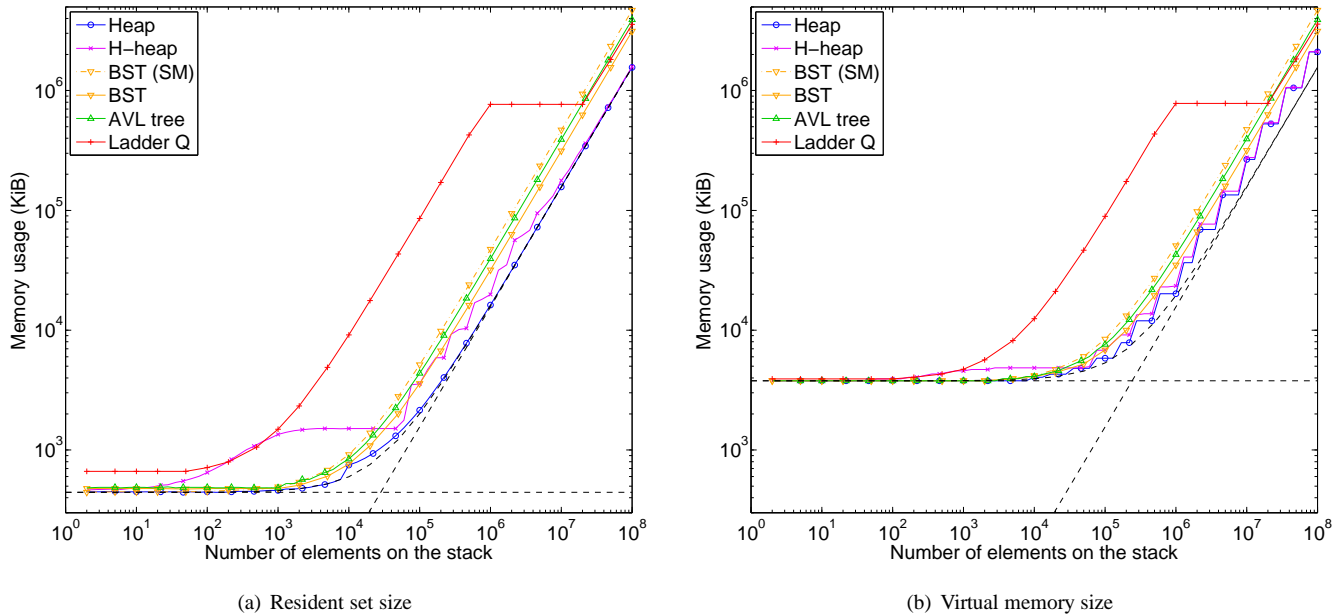


Figure 1: Memory usage of the queue. (a) The resident set size is the non-swapped physical memory used by the program. This does not reflect memory that was allocated but not addressed. (b) The virtual memory size is the total size of the program, and is given by the number of pages assigned to the process by the operating system. In both graphs, the horizontal black dashed line is the size reported for a program that uses the ‘null’ queue (that doesn’t store any information); the diagonal black dashed line is the amount of memory necessary to only store the data enqueued. The black dashed curve is the sum of these two quantities, and represents the smallest possible size of the program. Not shown: ‘4-heap’ is identical to ‘Heap,’ ‘LLRB tree’ is identical to ‘AVL tree,’ and ‘Splay tree’ is identical to ‘BST.’

portion of the graph starting at 10^6 items.

Figure 1(b) shows the VSZ of the test program using the various priority queue algorithms. Note that the initial virtual memory size is very large compared to the amount of memory used by the program. Also note that the heap, which doubles in size when it needs to be expanded, now shows the expected staircase-like behaviour, which it didn’t show in the RSS graph. The amount of memory allocated by the heap is at most twice as much as needed, which makes it at worst equal in size to the binary search tree. The hierarchical heap is composed of heaps, and therefore also shows the same behaviour.

4.3. Time Requirements

In many applications it is acceptable to increase the memory usage if it will make the program faster. Both Jones [18] and Breen and Monro [12] found certain self-balancing trees to outperform the more memory-efficient heap. I performed similar experiments to see if the increased memory usage of these data structures indeed is justifiable.

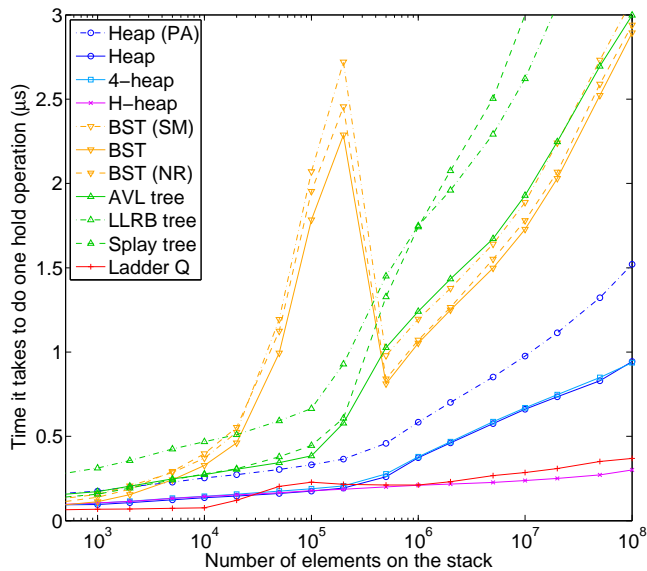
The classic performance experiment uses the hold model [12, 18, 17, 20]. A queue is set up with N elements, then the average time it takes for one hold cycle (one dequeue operation followed by one enqueue operation) is measured. This model is very simple and does not recognise the dynamic nature of queue sizes [21]. I will, non-the-less, use this model as its simplicity avoids making many assumptions on the input data set. The priority queue is filled with random priority values from a uniform distribution in the the interval $[0,50]$. The hold operation dequeues an element, adds a random value to its priority (using a uniform distribution in the interval $[0,2]$) and enqueues

an element with this new priority. As discussed later, the increment used to generate new priorities influences the measured times. Most papers using the hold model apply various probability distributions. The reported differences in performance caused by changing the distribution are relatively small, and therefore only the uniform distribution is considered here.

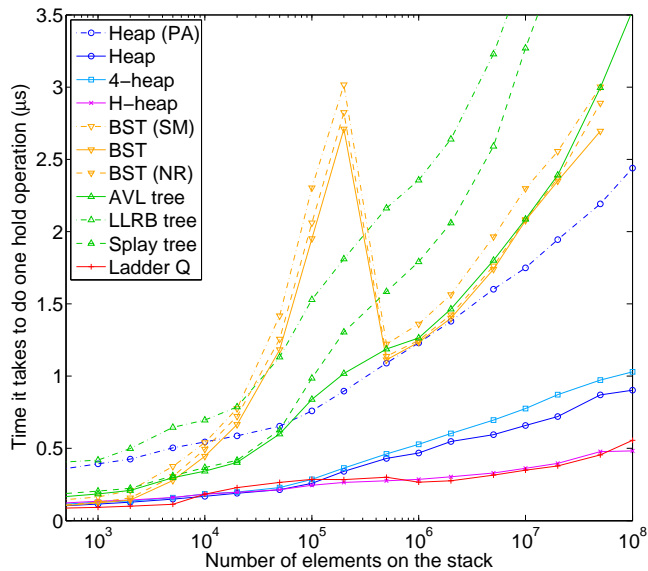
The hold operation was repeated 10^7 times on each queue, so that the C function `clock` can report accurate timing information even though it only has a 10 ms resolution. The timing of a ‘null’ queue, one that doesn’t actually store any data, was measured in the same way to determine the overhead of generating the random numbers. This value was subtracted from all the measured times.

Figure 2(a) shows the measured times for all the priority queue algorithms listed in Subsection 4.1. N , the number of elements in the queue, was varied exponentially from 2 to 10^8 (equivalent of enqueueing all pixels in a large 3D image). Figure 2(b) is the same experiment run on a similar system, a FineTec server with two AMD Opteron 248 processors (a single-core, 64-bit processor clocked at 2.2 GHz), and 4 GiB of memory, running the Fedora Core 6 distribution of Linux. Some minor differences aside, these two graphs are very similar, supporting the claim made earlier that the computer used in these experiments is representative of the systems used today in practice.

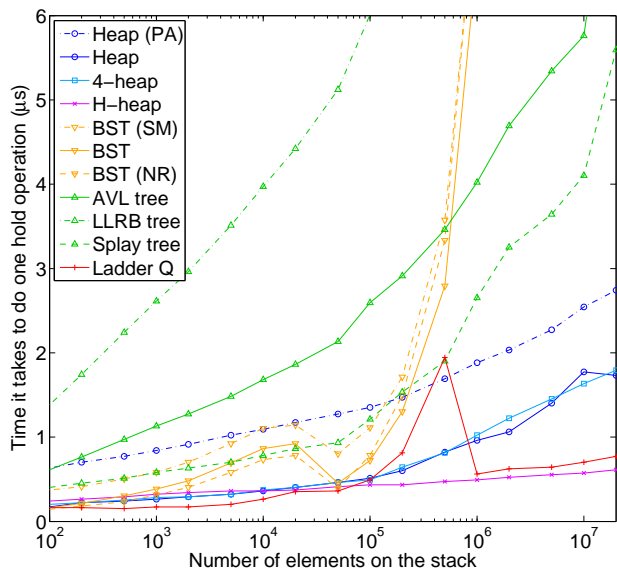
Figure 2(c) and (d) show the results on yet another platform, a Sun Blade 2500 workstation with two UltraSPARC IIIi processors (a single-core, 64-bit processor clocked at 1.6 GHz), and 4 GiB of memory, running SunOS 5.8. Figure 2(c) shows



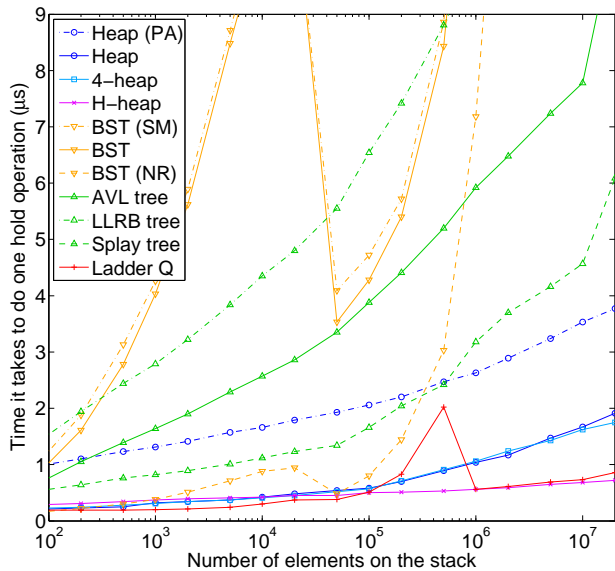
(a) Intel Xeon E5430 / GCC 4.1



(b) AMD Opteron 248 / GCC 4.1



(c) Sun UltraSPARC IIIi / Sun C 5.3



(d) Sun UltraSPARC IIIi / GCC 2.95

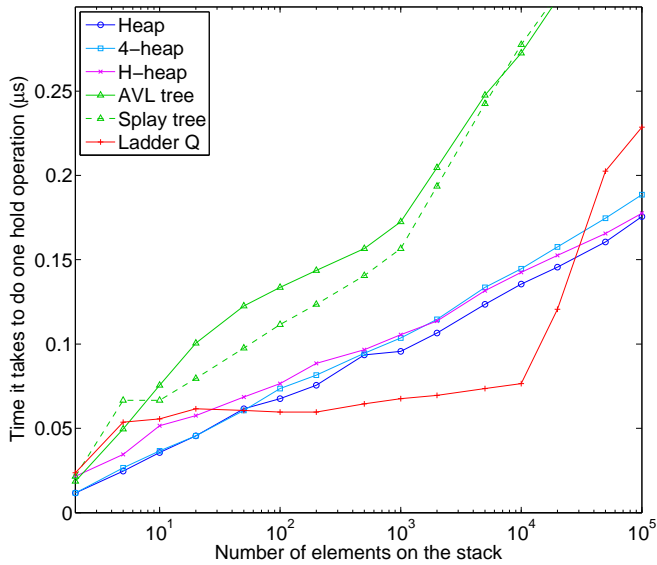
Figure 2: Speed of a hold operation. (a)-(d) Time for a single hold (dequeue one element, enqueue a new element) versus the queue size, for various architectures. All except the binary search tree (yellow) are $O(\log N)$ operations, meaning we expect straight lines in this graphs.

the results on code compiled with Sun WorkShop C version 5.3 and the `-O4` optimising option, Figure 2(d) shows the results on code compiled with GCC version 2.95.2 and the `-O3` optimising option, generating a 32-bit executable. The biggest difference between these two graphs is the vertical axis: most algorithms took 50% longer to run when compiled as 32-bit executables. Many other differences are caused by the compiler quality. Most notably, GCC 2.95 does not do tail recursion optimisation, severely degrading the performance of the binary search tree with recursive code.

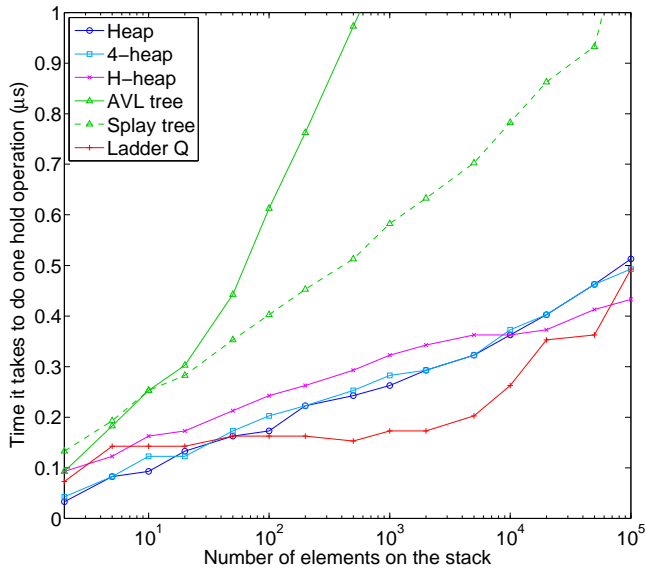
There are several striking things in these graphs, that will be analysed in the remainder of this subsection. Conclusions are

drawn in Section 5.

Except for the binary search tree and the ladder queue, all algorithms are supposed to be $O(\log N)$. This means that, in these semi-logarithmic plots, the time should be represented by a straight line. This is (approximately) only the case for the hierarchical heap. All other algorithms are affected by the cache size. Figure 3 shows the portion of the graphs in Figure 2(a) and (c) for smaller N . Note that these algorithms do behave as expected in this domain. For very large N the plots also form straight lines, but with a higher slope. This reflects the higher cost of memory access when the amount of data exceeds the cache size. Figure 4 plots time against memory usage, with the



(a) Intel Xeon E5430 / GCC 4.1



(b) Sun UltraSPARC IIIi / Sun C 5.3

Figure 3: Speed of a hold operation. (a),(b) Detail of the graphs in Figure 2(a),(c), respectively.

$O(\log N)$ model fitted separately to the data to the left and to the right of the “border” at 6 MiB, the cache size on the test system. The hierarchical heap has the advantage that enqueueing an element does not require moving data across the full length of an array of length N , but only across an array of approximate length N/k .

The execution time of the binary search tree peaks around $N = 2 \cdot 10^5$. This is an artifact of the testing method. After creating the tree, it is fairly well balanced because the data is entered unsorted. During the timing of the hold operation, however, the tree becomes increasingly unbalanced because newly enqueued elements are larger than the dequeued elements. The

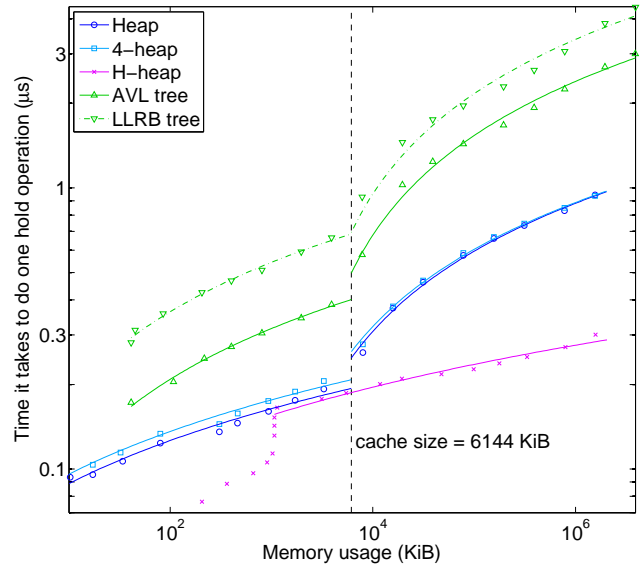


Figure 4: Speed of hold operation versus memory usage. The speed of the algorithms changes dramatically when the memory occupied by the queue becomes larger than the cache size. The data for each algorithm is fitted with two $\log(n)$ lines: one to the points where the queue is smaller than the cache size, and one where it is larger than the cache size. For the hierarchical queue a single line was fitted. The data for the splay tree is not shown for clarity of the graph, the fits were similar to the other balanced binary search trees.

more unbalanced the tree is, the longer the hold operation takes. For a tree that initially is larger than $N = 2 \cdot 10^5$, the chosen priority increment distribution does not debalance the tree significantly with 10^7 hold operations. In these cases, its performance is better than any of the self-balancing trees, which perform node rotations that are unnecessary. When the probability distributions are changed so that the tree is debalanced much quicker (Figure 5, using an initial uniform distribution in the range $[0,1]$, and a uniform increment distribution in the range $[1,2]$), the binary search tree’s performance is degraded enormously, whereas the self-balancing trees perform comparably. On the Sun workstation only 10^6 hold operations were done in each test, and hence the BST graphs peak one decade earlier, at $N = 2 \cdot 10^4$.

Whereas the various heap implementations behave similarly on the various architectures, the binary search trees and self-balancing trees do not. In particular, the splay tree has a slightly steeper slope than the AVL tree on all architectures when the queue size exceeds the cache size, but for small queues the AVL tree performs relatively much worse on the Sun workstation than on the two Linux computers. The splay tree seems to access the memory more frequently in one hold operation, but the AVL tree does many more function calls (it contains recursive code that is not tail-recursive, and hence the compiler cannot optimise away). On the Sun machine the relative costs of these operations favour the splay tree, whereas on the Intel and AMD machines they balance out almost exactly.

The ladder queue produces results that are more difficult to interpret, due to the complexity of the data structure and the delay in sorting. It is possible that the knee at 10^4 elements

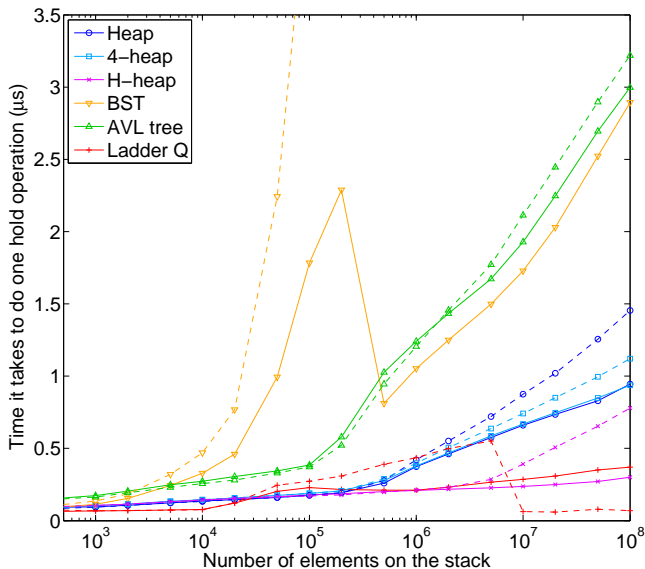


Figure 5: Influence of input on performance. The continuous lines are the corresponding lines from Figure 2(a), the dashed lines are hold operation times for those same algorithms, but with input that quickly surpasses the range of data initially in the queue.

is caused by the need of a second rung in the FF. On the Sun workstation there is a peak presumably caused by the 900000 element limit imposed on the VFF. When this limit was not imposed, the performance on this platform degraded exponentially for larger queues, though it did not affect performance on the Linux machines. I guess that this phenomenon is related to the less efficient `realloc` on the Sun machine, but was unable to test this hypothesis. In the debalancing graph (Figure 5) it can be seen that for 10^7 or more elements the hold operation can be even faster than for very small queue sizes. This is again an artifact of the testing method. Only 10^7 hold operations are performed on the queue, and all of these 10^7 new elements have a priority larger than any on the initial queue. With such a large queue, none of the elements enqueued during the test need to be sorted, since the queue already contains enough elements to satisfy all the dequeue operations. When the time needed to dequeue all elements in the queue is taken into account, this advantage disappears.

4.4. Enqueueing and Dequeueing Times

The hold operation does not represent all of the work done by a priority queue when used in an image analysis routine. Typically, the queue is first initialised by enqueueing some pixels, for example all the pixels on the border of the objects. The pixels in the queue are then processed one by one, each potentially resulting in the enqueueing of some of its neighbours. But at some point all pixels have been processed or are in the queue to be processed, and no further enqueueing happens. Pixels are dequeued until the queue is empty. To measure the performance in this initial enqueueing-only and final dequeueing-only stages, I plotted in Figure 6 the times for filling and emptying the queues used to measure hold times. These values were di-

Table 2: Execution time (in seconds) for the grey-weighted distance transform using various priority queues. The algorithm ran out of memory when using the ladder queue with the largest image size.

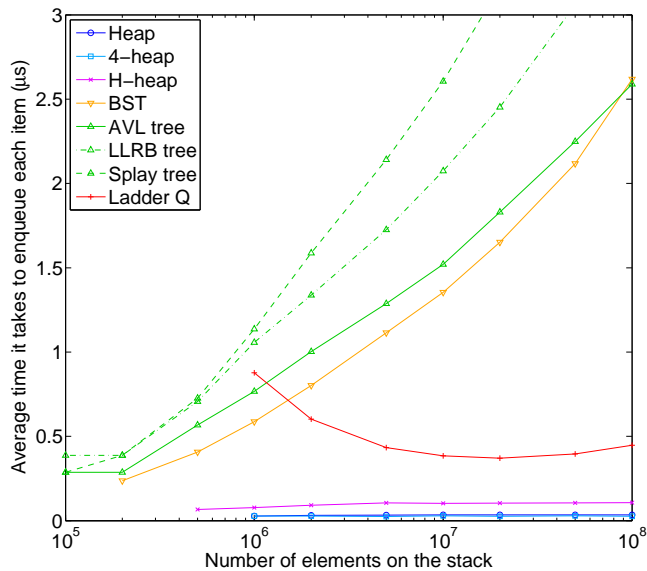
Image size (pixels)	128 ³	256 ³	512 ³
number of seeds	27	216	1728
H-heap	9.25	126.4	1228
Ladder Q	10.77	138.3	-
Heap	11.72	153.4	1517
AVL tree	17.11	217.0	2401
Splay tree	19.65	261.8	2875

vided by N to get an average time per enqueue or dequeue. Note that this is not the time it takes to perform one operation on a queue of size N , but rather the average time to perform the operation on a queue as it grows from 0 to N or shrinks from N to 0. Only the times for large N are plotted, because the `clock` function used for measuring times is rather coarse. The enqueue operation on the heap is, on average, independent of N , as explained by Doberkat [22], whereas the dequeue operation is rather expensive when compared to binary trees. In contrast, the enqueue operation on any of the binary trees takes up much more time than the dequeue operation if the queue is very large (please note the difference in scaling of the vertical axis).

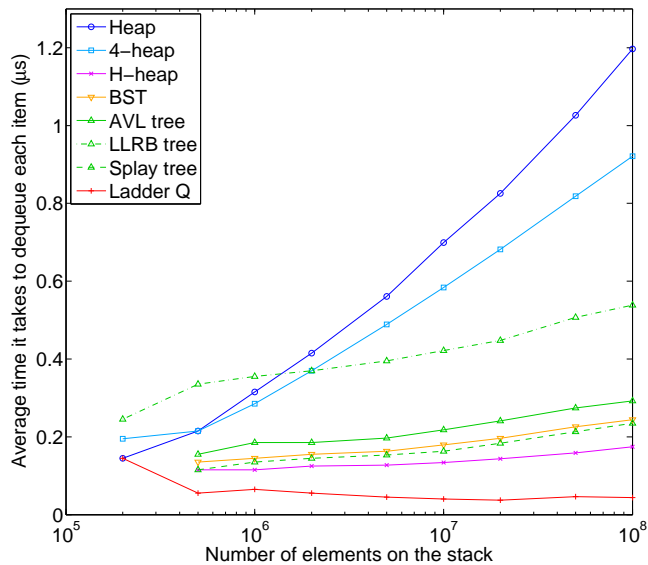
4.5. Priority Queues in Practice

To demonstrate the importance of selecting a good priority queue when implementing an algorithm, I took a grey-weighted distance transform algorithm from the DIPlib library² (Delft University of Technology, The Netherlands), and replaced the calls to the priority queue with calls to the algorithms implemented for this paper. I then generated a small 3D image of size 128³ pixels filled with Normally distributed noise ($\mu = 0$, $\sigma = 10$), smoothed this image using a Gaussian filter ($\sigma = 10$), and added Normally distributed noise again ($\mu = 0$, $\sigma = 1$). To insure a strictly positive image, I subtracted the minimum value and added 10^{-6} . I then generated a seed image by setting 27 randomly selected pixels. The grey-weighted distance transform computes the distance to the nearest seed, where distance is defined as the integral of grey values along the path. When using the splay tree or the AVL tree, as recommended by Breen and Monro [12], this algorithm took, respectively, 19.65 s and 17.11 s to complete. In contrast, using a very simple implicit heap the time is significantly reduced, to 11.72 s. With the hierarchical heap the time is further decreased to 9.25 s. This shows that, in this type of algorithm, the priority queue uses a very significant portion of the computation time. Increasing the image size makes these times more important, see Table 2. In this table one can see that for a large 3D image, computation time can be reduced from 40 minutes to 20 minutes by using a hierarchical heap rather than an AVL tree.

²Obtainable from <http://www.diplib.org/>.



(a) Enqueue speed



(b) Dequeue speed

Figure 6: Average enqueue and dequeue speed per element. (a) Enqueue speed shows measured time to enqueue N elements on an initially empty queue, divided by N . (b) Dequeue speed shows measured time to dequeue all elements from a queue with N elements, divided by N . Note the different scaling of the y axis on the two graphs. Due to the coarseness of the time measurements it was only possible to plot timings for very large N . All the data points are in the right-side domain of Figure 4, that is, the queue is larger than the cache size.

5. Conclusions and Discussion

As can be seen in Figure 2, the implicit heap easily outperforms any of the tested self-balancing trees at all queue sizes, and its performance is very consistent across platforms. Among the self-balancing trees, the AVL tree, the oldest of all trees tested, is also the best in performance. On Solaris the splay tree outperforms the AVL tree probably because the version of the AVL tested uses recursive code that is not tail-recursive. Only for very small queues, up to a few hundred elements, a small time gain might be obtained by using plain binary search trees. If the queue size exceeds the cache size, it is worth while to use the more complex hierarchical heap. However, the added complexity does not pay off for smaller queues. The ladder queue is very efficient at all sizes, though only beats the hierarchical heap in the range of 100 - 10^4 element queue size. Considering the complexity of implementation and its erratic behaviour, it is questionable whether this is a good method for general use.

The improvements to the implicit heap suggested by LaMarca and Ladner [17] to improve data locality and reduce cache misses were also tested. We implemented a four-way heap, that indeed shows a slightly better consistency than the two-way heap for very skewed input data, but only for very large queue sizes. Very large queue sizes are better handled by the hierarchical heap. The other improvement suggested by LaMarca and Ladner is to add an offset to the array that stores the heap data so that the children of one node do not fall in separate cache blocks. In this test, this modification only led to a slightly worse performance (data not shown). This is one of the clear cases where programming tweaks for a specific architecture have adverse effects on other architectures.

A clear lesson here is that simpler algorithms are better suited for simple tasks. Unless the problem is very complex (such as storing huge amounts of data in a priority queue), it is not worth while to implement a complex solution. Advances in hardware and compiler optimisation techniques oftentimes make hand-tweaking of code unnecessary. Hand tweaking not only makes code more difficult to maintain, but indeed can make it more difficult for the compiler to optimise the object code.

It is clearly necessary to occasionally re-evaluate the performance of basic algorithms and techniques, since computer architectures change so rapidly. In this paper I have mainly followed Jones [18], Breen and Monro [12], and LaMarca and Ladner [17]. Of these papers, only the latter suggests the implicit heap as a good (but not optimal) solution. Breen and Monro's paper is the only one that considers the constraints of image analysis problems, though in my view these constraints have changed over the years. For current image analysis programs, the best implementation of the priority queue is the implicit heap. It has the smallest possible memory usage and is faster than all other implementations tested, with the exception of the hierarchical heap and the ladder queue for very large queue sizes. The one disadvantages of the heap that needs to be considered is that it is not inherently stable, requiring the addition of an integer to each enqueued element to obtain a stable priority queue.

Acknowledgements

I would like to thank Dr. David Knowles at Lawrence Berkeley National Laboratory, Berkeley, California, for letting me use

some of his computers to generate the Figures 2(b), (c) and (d), and Figure 3(b).

Appendix: Ladder Queue Implementation

I implemented the ladder queue for this comparison following the details given by Tang et al. [31]. Such a data structure is much more complex than the other structures in the comparisons, and requires much more memory. The test program loads the queue with many data elements before extracting a single one. The ladder queue accumulates all these elements into the VFF list, which makes enqueueing very efficient, but also makes this list grow unreasonably large. To avoid extreme memory usage, the VFF was emptied into the FF the first time it contained 900000 elements. Subsequent enqueueing will happen mostly in the FF, which affects the speed of enqueueing, but does not affect the measured speed of the hold operation. The value of 900000 was chosen ad hoc, and is clearly reflected in the experiment that compares memory usage.

There are two further deviations from the paper. All unsorted lists (that is, VFF and the bins in FF) are simple arrays rather than linked lists. These are initialised to 50 elements and double in size when required. To avoid infinite recursion when a block of elements with identical priority does not fit into the NF, this implementation copies as many elements as can fit in the NF, rather than enlarging the NF as suggested in the paper. This was easier to implement, and since it is not a common occurrence, should not affect the measured performance.

References

- [1] J. Piper, E. Granum, Computing distance transformations in convex and non-convex domains, *Pattern Recognition* 20 (6) (1987) 599–615.
- [2] P. W. Verbeek, B. J. H. Verwer, Shading from shape, the eikonal equation solved by grey-weighted distance transform, *Pattern Recognition Letters* 11 (1990) 681–690.
- [3] L. Ikonen, Priority pixel queue algorithm for geodesic distance transforms, *Image and Vision Computing* 25 (10) (2007) 1520–1529.
- [4] J. A. Sethian, A fast marching level set method for monotonically advancing fronts, *Proceedings of the National Academy of Sciences* 93 (1996) 1591–1595.
- [5] K. Robinson, P. F. Whelan, Efficient morphological reconstruction: a downhill filter, *Pattern Recognition Letters* 25 (15) (2004) 1759–1767.
- [6] E. J. Breen, R. Jones, Attribute Openings, Thinnings, and Granulometries, *Computer Vision and Image Understanding* 64 (3) (1996) 377–389.
- [7] L. Vincent, Grayscale area openings and closings, their efficient implementation and applications, in: J. Serra, P. Salembier (Eds.), *Mathematical Morphology and Its Applications to Signal Processing*, EURASIP, UPC Publications, Barcelona, Spain, 22–27, 1993.
- [8] F. Meyer, Topographic distance and watershed lines, *Signal Processing* 38 (1) (1994) 113–125.
- [9] R. Adams, L. Bischof, Seeded region growing, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16 (6) (1994) 641–647.
- [10] M. Couprie, D. Coeurjolly, R. Zrou, Discrete bisector function and Euclidean skeleton in 2D and 3D, *Image and Vision Computing* 25 (1) (2007) 1543–1556.
- [11] L. Yatziv, A. Bartesaghi, G. Sapiro, O(N) implementation of the fast marching algorithm, *Journal of Computational Physics* 212 (2006) 393–399.
- [12] E. J. Breen, D. Monro, An evaluation of priority queues for mathematical morphology, in: J. Serra, P. Soille (Eds.), *Mathematical Morphology and Its Applications to Image Processing*, Kluwer, Dordrecht, 249–256, 1994.
- [13] G. S. Brodal, R. Fagerberg, Funnel Heap – A Cache Oblivious Priority Queue, in: *Algorithms and Computation*, vol. 2518 of *Lecture Notes in Computer Science*, Springer, 101–131, 2002.
- [14] S. Carlsson, An optimal algorithm for deleting the root of a heap, *Information Processing Letters* 37 (2) (1991) 117–120.
- [15] B. V. Cherkassky, A. V. Goldberg, C. Silverstein, Buckets, heaps, lists, and monotone priority queues, in: *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 83–92, 1997.
- [16] M. Thorup, On RAM priority queues, in: *Proceedings 7th ACM-SIAM Symposium on Discrete Algorithms*, 59–67, 1996.
- [17] A. LaMarca, R. E. Ladner, The influence of caches on the performance of heaps, *ACM Journal of Experimental Algorithms* 1 (1996) 4.
- [18] D. W. Jones, An empirical comparison of priority-queue and event-set implementations, *Communications of the ACM* 29 (4) (1986) 300–311.
- [19] M. Marín, P. Cordero, An empirical assessment of priority queues in event-driven molecular dynamics simulation, *Computer Physics Communications* 92 (1995) 214–224.
- [20] M. Marín, *An Empirical Comparison of Priority Queue Algorithms*, Tech. Rep., Oxford University, 1997.
- [21] R. Rönngren, R. Ayani, A Comparative Study of Parallel and Sequential Priority Queue Algorithms, *ACM Transactions on Modeling and Computer Simulation* 7 (2) (1997) 157–209.
- [22] E.-E. Doberkat, Inserting a new element into a heap, *BIT Numerical Mathematics* 21 (1981) 255–269.
- [23] D. Knuth, Sorting and Searching, vol. 3 of *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, second edn., 1998.
- [24] D. D. Sleator, R. E. Tarjan, Self-Adjusting Binary Search Trees, *Journal of the ACM* 32 (3) (1985) 652–686.
- [25] R. Sedgewick, Left-Leaning Red-Black Trees, presented at Workshop on Analysis of Algorithms, Maresias, Brazil, April, 2008, 2008.
- [26] M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM* 34 (3) (1987) 596–615.
- [27] J. R. Driscoll, H. N. Gabow, R. Shrairman, R. E. Tarjan, Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation, *Communications of the ACM* 31 (11) (1988) 1343–1354.
- [28] T. Takaoka, Theory of 2-3 Heaps, *Discrete Applied Mathematics* 126 (1) (2003) 115–128.
- [29] J. H. Blackstone, G. L. Hogg, D. T. Phillips, A two-list synchronization procedure for discrete event simulation, *Communications of the ACM* 24 (12) (1981) 825–829.
- [30] R. Rönngren, J. Riboe, R. Ayani, Lazy queue: an efficient implementation of the pending-event set, in: *Proceedings of the 24th annual symposium on Simulation*, IEEE Press, 194–204, 1991.
- [31] W. T. Tang, R. S. M. Goh, I. L.-J. Thng, Ladder queue: An O(1) priority queue structure for large-scale discrete event simulation, *ACM Transactions on Modeling and Computer Simulation* 15 (3) (2005) 175–204.
- [32] E. J. Breen, G. H. Joss, K. L. Williams, Dynamic arrays for fast, efficient, data manipulation during image analysis: a new software tool for exploratory data analysis, *Computer Methods and Programs in Biomedicine* 37 (2) (1992) 85–92.
- [33] E. Demaine, Fast and Small Resizable Arrays, *Dr. Dobbs’s Journal* 26 (3) (2001) 132–134.
- [34] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, R. Sedgewick, Resizable Arrays in Optimal Time and Space, in: F. Dehne, A. Gupta, J.-R. Sack, R. Tamassia (Eds.), *Algorithms and Data Structures*, vol. 1663 of *Lecture Notes in Computer Science*, Springer, Berlin, 37–48, 1999.
- [35] D. Naor, C. U. Martel, N. S. Matloff, Performance of Priority Queue Structures in a Virtual Memory Environment, *The Computer Journal* 34 (5) (1991) 428–437.
- [36] A. Bauer, Compilation of Functional Programming Languages using GCC – Tail Calls, Master’s thesis, Institut für Informatik, Technische Universität München, 2003.
- [37] B. Pfaff, GNU libavl 2.0.2a, Computer Program, URL <http://www.stanford.edu/~b1p/avl/>, 2004.
- [38] D. Sleator, An implementation of top-down splaying, Computer Program, URL <http://www.link.cs.cmu.edu/splay/>, retrieved September 2008., 1992.