Advanced Data Structures: Priority Queues

Anders Ingemann (20052979)
Peter E. Hvidgaard (20062546)

September 28, 2011

Project

This project is about priority queues based on the Binary Heap and Fibonacci Heap respectively. The main difference is that the Fibonacci Heap have a amortized $O(1)$ *decrease key* operation, where the Binary Heap have $O(\log n)$.

We will implement both heaps, and use them to implement Dijkstras Algorithm for Single Source Shortest Path. Then we will describe a class of graphs with many *decrease key* operations to investigate whether Fibonacci Heap is faster, despite the much larger constants hidden in the big-O notation.

Binary Heap

We use an array of pointers for our implementation, similar to maximum priority queue in Cormen [1]. Specifically family relations are easily expressed in the following way (assuming the array index start at 1): Given a node of index i: parent index $\lfloor i/2 \rfloor$, left child index $i * 2$ and right child $i * 2 + 1$. The important property of the heap is that given any node, i, we know that all nodes in the entire tree, with i as root, have key at most that of i. Our binary heap implementation support the following operations:

- *initialize* is a simple matter of allocating memory and setting the current size to 0. This is $O(1)$ if we assume that allocating the memory is a constant time operation.
- *insert* the new element into the first available posistion, and then bubbled up by exchanging the element and its parent as long as the heap property is violated. Potentially the entire way upto the root, if it's the new minimum. This takes $O(\log n)$ if we need to move it the all way to the root of the tree.
- *decrease key* takes a pointer to an element, and decrease the key. Then, as in insert, it will bubble up the element until the heap property is restored. This also takes $O(\log n)$ for the same reasons.
- *find min* is a simple operation due to the heap property, it will always be the root element of the tree, i.e. the first element in our array. This takes $O(1)$.
- *delete element* takes a pointer to an element to delete. It will move the last item to the index of the deleted element, and thus the heap property might be violoted. Then a call to the helper function *min heapify* is made to restore the heap property. Therefore the time complexity is determined by *min heapify* - and as we will see, this is $O(\log n)$.
- *min heapify* takes a pointer to the element that might violate the heap property. It works by comparing the the element to its parent first and

    if the parent is the same size we're done.

    if the parent is larger we need to bubble the element up.

    if the parent is smaller we may need to move the element down.

In the cases where we have to move the item, we resursively call the procedure again on the new posistion until we terminate. The work that has to be done, is at most moving the element all the way from the root to a leaf, or from a leaf to the root. This gives a time complexity of $O(\log n)$.

Fibonacci Heap

The Fibonacci heap entirely consists of pointers and only utilizes an array when the scanning of ranks is to be executed. Each node contains four pointers. Two pointing to it's left and right siblings, one pointing at its parent and one to a child. Having a pointer to the left and the right results in the nodes forming a doubly linked list at each level. The heap does not order the lists in any way, since a particular order depending on the rank or the key of the nodes does not have any impact on the correctness or efficiency of heap operations. This doubly linked list allows us to extract and insert nodes in very little (constant) time. Some invariants however need to be maintained:

- Extracting/deleting child nodes result in the parent being marked. If it is already marked, the parent itself has to be extracted and placed at in the root list. This cascades to the parent of that, marking it or recursing further if it is marked as well.
- When deleting the minimum, the heap has to clean up the root list. After this operation there exist no two root nodes with the same rank.

Some of the more curious parts of the implementation also involve the delete_min operation. Since delete_min is iterating a doubly linked list, which is modified while looping, the pointer to the next node is saved before any other operations begin. The loop could otherwise continue in the child list of a node already registered in the scanning array, after the current node has been made a child of it, because it had the same rank.

A custom invariant in the fibonacci heap, set and maintained by us, is the fact that there not ever exists an invalid linked list. Once a node is extracted from a list, it is immedeatly linked to itself on the left and the right. This way succeeding union operations will always be able to link the node into a new list without any dangling pointers.

The amount of memory allocated for the scanning array in the delete_min operation is directly proportional to the number of nodes in the heap. Our intuition however is, that this amount can be drastically reduced. The amount of memory needed is directly proportional to the maximum rank any root node can assume. This rank must be lower than the total number of nodes in the heap, because that node, by definition, contains sub trees of nodes.

We have not investigated this further and do not have examples proving or disproving this intuition. Minimizing the size of the scanning array would decrease the running time of the delete_min operation. This holds true, because inserting a node into the array requires us to check whether the position in the array is free. We do this by checking for the NULL pointer. The only way to ensure that a given (free) address in the array is in fact free, is to clear the allocated memory with zeroes. This is the reason we use calloc() and not malloc(), when allocating the array. calloc() however still needs to run over the entire allocated space, ostensibly taking O(n) time.

Primitive Priority Queue

To test for correctness and to have some “stupid” reference, we implemented a very primitive queue. This queue is a linked list. *Initialize,* insert and *decrease_key are all constant operations. The* find_min and *delete_min operations scan the entire list and, in case of* find_min, returns the minimum element, a $O(n)$ operation.

Testing of implementations

Test Setup

We test by generating the graph in memory, and then run Dijkstas algorithm as described in Fredman and Tarjan [2]. We measure the time it take for Dijkstras Algorithm to finish with each of the three different priority queues. We test a suitable number ot times compared to the input (the smaller input the more trials), and take the average. The test machine was a 2x quad-core Intel Xeon 2.8 GHz with 8 GiB 800MHz DDR2 ram. The program was build using gcc version 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00).

The program can be build by entering the folder **p1_heap** and run the following command: *make -f p1_heap.mk all*. This requires MAKE and GCC to be installed. Then the compiled binary will be in the subfolder **Debug**, and can be run with *./p1_heap*.

Test Data

Graphs with many *decrease key* operations

Constructing a graph with many *decrease key* operations is fairly easy. The problem is making sure the *decrease key* operation force the Binary Heap to move the element. If the element after the *decrease key* doesn't violate the Binary Heap property, it does not need to be moved and will be a constant operation, and thus it will match the Fibonacci Heap.

The graph is generated with the following code:

This graph will call *decrease key* on all edges that is outgoing from the node return by *delete min*, and the node will have an edge to all remaining nodes in the graph. Further more, all *decrease key* calls will force the Binary Heap to bubble the element from a leaf to the root for every call, and thus maximizing the work done.

Algorithm 1 `Generate graph` max decrease calls

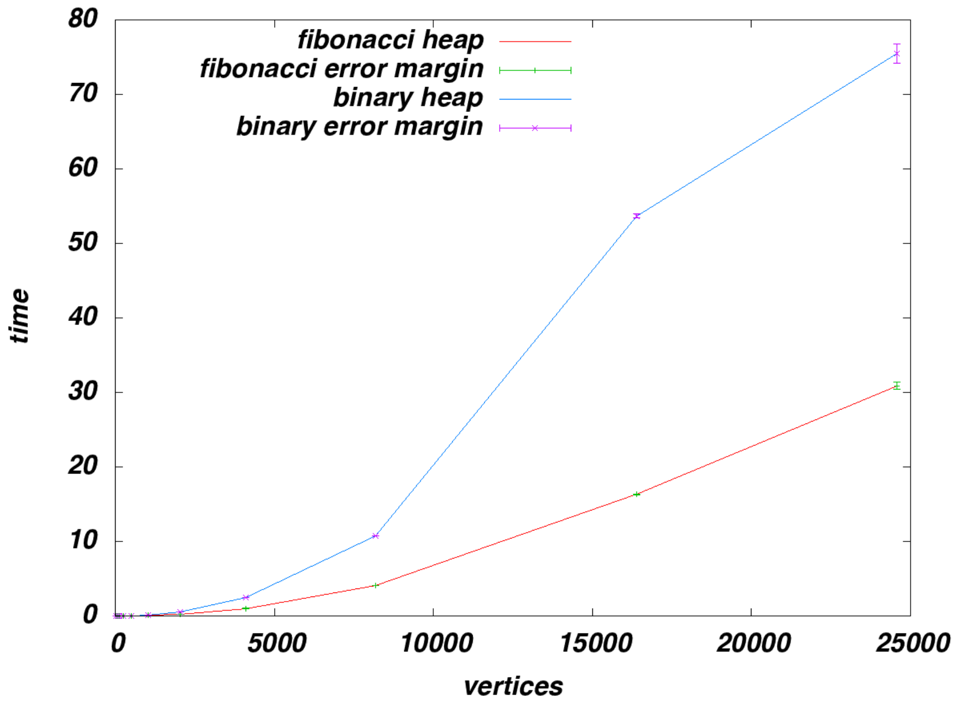
```
Require: number of vertices $n$
1:  for $j := 1; j < n; j := j + 1$ do
2:      $weights[0][j] := n * n - j$
3:  end for
4:  $weights[0][n] := 1$
5:  for $i := n - 1; i > 1; i := i - 1$ do
6:      for $j := 1; j < i; j := j + 1$ do
7:          $weights[i][j] := n^2 - (n - i)n - j + 1$
8:      end for
9:      $weights[i][i - 1] := 1;$
10: end for
11: return $weights$
```

Random Graphs

The random graphs are created by random chance. For every node it has a chance of getting an edge to another node, and the weight is random as well. Futhermore it's possible to set the seed, such that you can always generate the same random graph.

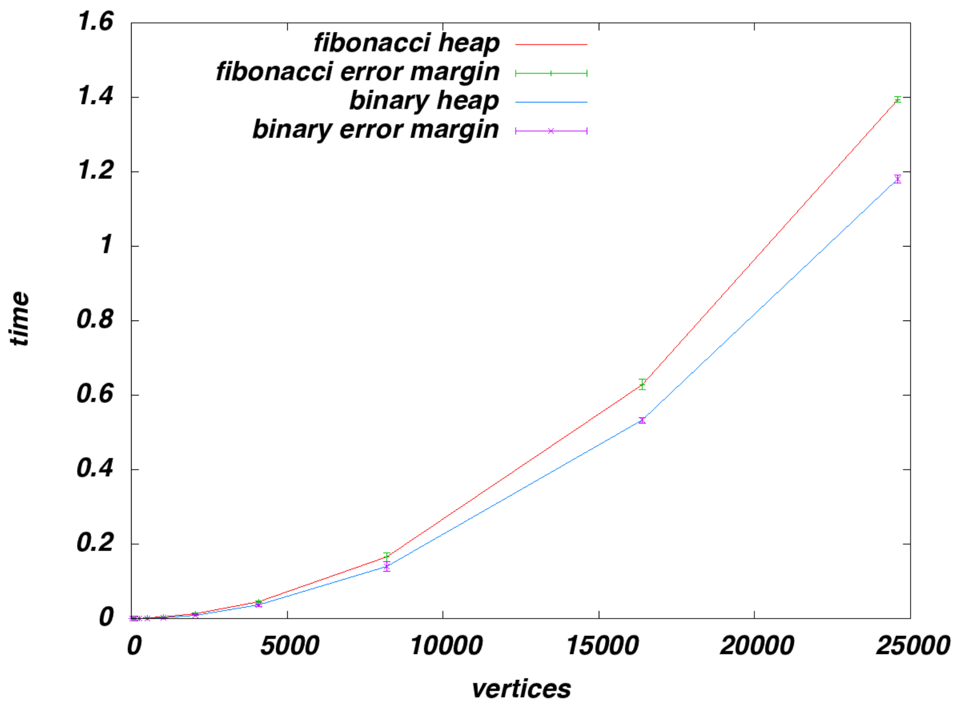
Results

Maximizing decrease key calls



It's rather obvious that binary heap performs significantly worse than Fibonacci Heap, just as expected. The binary heap curve has an odd shape, but we triple checked the results, so our guess is that some underlying VM or hardware played a role, most likely page faults.

Random Graphs



Here we see that the difference between Binary and Fibonacci heaps are almost insignificant, but with binary heap leading. The graphs were created with chance of edge 15% and max weight 20.

Conclusion

It's fairly obvious that unless given a graph, that is constructed in such a way that maximizes the amount of *decrease key* calls and thereyby the work the binary heap does, then the binary heap is to be prefered due to the much smaller constant.

Bibliography

[1] Thomas H. Cormen et. al., *Introduction to algorithms*, 2nd Edition.

[2] M. L. Fredman and R. E. Tarjan *Fibonacci Heaps and Their Use in Improved Network Optimization Algorithms*