Practical complexity of the Fibonacci heap in a simulation and modelling framework

Elwira Johansson
Abstract

Practical complexity of the Fibonacci heap in simulation and modelling framework

Elwira Johansson

Improving a data structure and optimizing memory access can reduce run times in a program, especially in software that access data frequently. This project attempts to speed up the run time of the simulation and modeling framework URDME, by replacing the currently implemented binary heap with the Fibonacci heap and comparing run times. The Fibonacci heap has some interesting features and promising time complexity, but turned out to be efficient only in specific programs that utilize the constant functions of the data structure the majority of the time. The Fibonacci heap presented a major slowdown since the solver utilizes the slower functions frequently. Even with implementation optimizations, the Fibonacci heap will not outperform the binary heap when used in URDME.
1 Introduction

Modeling and simulations play an important role in natural science, allowing users to predict or explain a real-world problem without the need for performing the experiments in real life.

Most simulation and modelling software process large amounts of data, and depending on how the software handles memory access, programs usually spend a noticeable amount of time accessing data. Optimizing data structures have the potential to speed up a program, especially when memory is accessed frequently.

The aim of this thesis is to answer if it is practically efficient to use the Fibonacci heap in the simulation and modelling framework URDME[1], compared to the currently used binary heap. This will be done by implementing the Fibonacci heap in C, measuring run-times by timing example programs provided by URDME software in MATLAB and lastly comparing the run-times and memory usage by the two heaps.

2 Fibonacci heap theory

In order to speed up Dijkstra’s algorithm for the single-source shortest path problem with nonnegative length edges [2], Fredman and Tarjan developed the Fibonacci heap [3].

The Fibonacci heap has similar properties to the binomial heap, such that every binomial heap is a Fibonacci heap, but not the other way around. The Fibonacci heap can contain multiple trees consisting of single nodes, while the binomial heap always consists of trees with $2^n$ nodes, where each tree has a different $n$ value. This is a result of the Fibonacci heap not merging singular nodes into trees when inserting elements, in order to keep the insertion time constant, unlike the binomial heap which eagerly merges every insertion into the tree. The subsections below explain the three most common heap operations in the Fibonacci heap.

2.1 Insertion

When inserting elements into an empty Fibonacci heap, it starts out as a doubly linked list of root-nodes with a pointer saved to the minimum element. The heap will remain a doubly linked list of unordered nodes until the minimal node is extracted from the heap.

2.2 Extract-min

The extraction of the minimum node is a simple procedure, where the children of the minimum node is added to the root-list of the heap and their parent
pointer is removed. The minimum node is linked out from the double linked root list, and the pointer to said node is returned.

2.2.1 Consolidation

Although consolidation is not a common heap operation, it is a crucial help function for the Fibonacci heap. Extracting the minimum element results in a consolidation of the root-nodes in the heap into trees. Each node in the tree can have a maximum of $\log N$ children, where $N$ is the total amount of nodes in the tree [4]. Each parent will have a single pointer to one of its children, while all children will point to its parent. The child nodes will also keep pointers to two other children, the left and right one, resulting in a doubly linked list between the children.

Since the nodes are not stored in any order in the root list, the time to find the new minimum scales with the number of trees in the root list. In order to reduce this time in future operations, the consolidation will merge trees with the same degree, where the degree represents the number of children of the root-node in each tree. The tree with the smallest root-node will adopt the root-node of the other tree, merging the two trees into a tree with previous degree plus one, as seen in Figure 1. From this follows that it is possible for the root-list to contain multiple trees even after consolidation, if no trees in the root-list are of the same degree.

2.3 Delete

Deleting/extracting a node that is not the minimum node is done by decreasing the key of the node to negative infinity, thus making it the new minimum node whereby the function to extract minimum will be called. Decreasing a key is done by setting the key to the decreased value, then moving said node to the root-list as the new minimum node. If the node has children, then the child nodes will also be moved into the root list. When the node is the new minimum node, it is possible to remove the node by extracting the minimum, as previously explained.

3 Implementation

The heap is used in the default solver in URDME, which uses the algorithm next subvolume method (NSM) [5]. The heap is used to store the times until the next event in each subvolume. Two functions are called from the solver related to the heap data structure, initialization and update. The binary heap is implemented as an array object, where the minimum node has index zero, and children of node with index $n$ have index $2n + 1$ and $2n + 2$. The implementation also makes use of two index arrays, that allow the heap to find an element by the original index in the data array where it is possible that elements have switched place multiple times in memory, and the other way around.
3.1 Fibonacci heap

The Fibonacci heap was implemented to match the syntax of the binary heaps to avoid any re-structuring of the solver code. As the solver updates the data in an external array not related to the heap, each node in the Fibonacci heap contains a pointer to the data correlated to the specific node, instead of a copy of the value.

The solver makes use of the binary heap’s property of having the minimum node/data in index zero, which is also a property of the Fibonacci heap in this implementation. The solver will still use the two functions initialize and update, to create and update the heap. It allocates memory for the heap before initialization, meaning that no allocation of nodes is performed in heap operations.

The solver makes use of the heap by calling the two functions fib_initialize_heap() and fib_update().
3.1.1 Initialize heap

`fib_initialize_heap()` inserts all nodes in the doubly linked root list and checks for a new minimum node in each insertion. If there is a new minimum, the heap minimum pointer and the two nodes indices in the index arrays will update accordingly.

3.1.2 Update heap

`fib_update()` function is called after the solver updates element $i$ in the data array. There are a few possible cases when the update function is called: the updated node is the minimum, has parents or/and has children. If the node has a parent, then it is moved from the parents child list to the root list of the heap. If the node has children, then the child nodes are all moved to the root list and the parent pointer is removed. If the minimum node has been updated, then the heap must be consolidated in order to search for a new minimum node. If the updated node is not the minimum, then the minimum key and the key of the updated node will be compared, and the smallest of the two becomes, or remains, the minimum node.

3.1.3 Consolidation

A node pointer array $A$ is allocated, and all pointers are initialized to NULL. The function goes through the root list, storing a pointer to the current node with degree $d$ in $A[d]$ if $A[d]$ is NULL. If $A[d]$ is not NULL, then the node pointed to by $A[d]$ will merge with the current node and remove the pointer from $A[d]$. The node with the smallest key will become the parent of the other node. If $A[d+1]$ is not NULL, the parent node will merge with $A[d+1]$, otherwise a pointer to the parent will be stored in $A[d+1]$ and the next node in the root list is processed. When the nodes in the root list have been traversed, a new root list consisting of the nodes pointed to by all non-NULL pointers in $A$ will be created. For every inserted node in the new root list, the minimum node is updated accordingly, as well as the two index arrays.

3.2 Modified Fibonacci heap

A second version of the Fibonacci heap was implemented where each node contains an extra double to store a local copy of the key value, as well as the pointer to the key value. By doing this, the value at the key pointer and the local key value will differ when an update is made. If the updated node is the minimum node, and the local key value is greater than the value at the key pointer, then the heap requires no updates in structure and no consolidation is needed.
4 Experiments and results

To test the Fibonacci heap implementations, three different simulations with different heap sizes, provided by URDME, were run on a duo-core CPU, Intel i5-4210U, with a clock speed of 1.7 GHz and a L1 cache size of 32KB.

The examples were run in MATLAB and timed with the inbuilt function \texttt{tic}. The times in the tables below are the average running time over ten runs, except for \textit{mincde} that was only measured three times due to the size of the program. The table also provides the size of the heap and the speedup when using the Fibonacci heaps in each program.

4.1 Standard Fibonacci heap

<table>
<thead>
<tr>
<th>Program</th>
<th>Heap size</th>
<th>Binary heap(s)</th>
<th>Fibonacci heap(s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annihilation</td>
<td>20</td>
<td>2.03</td>
<td>3.12</td>
<td>0.65</td>
</tr>
<tr>
<td>Annihilation2D</td>
<td>993</td>
<td>8.44</td>
<td>37.12</td>
<td>0.23</td>
</tr>
<tr>
<td>Morphogenesis2D</td>
<td>2015</td>
<td>361.25</td>
<td>2398</td>
<td>0.15</td>
</tr>
<tr>
<td>Mincde</td>
<td>9761</td>
<td>677.43</td>
<td>11479</td>
<td>0.059</td>
</tr>
</tbody>
</table>

As a consequence of the Fibonacci heap consisting of doubly linked lists, it requires four extra pointers and one extra integer per node compared to the binary heap. An extra integer in each node was used to store the index of the data, which was required for the consolidation function.

For a heap of size \( N \), the binary heap allocates \( 3N \) doubles, while the Fibonacci heap allocates \( 3N \) doubles, \( 6N \) pointers and \( 3N \) integers for the same heap size. This is almost four times as much memory usage.

4.1.1 Modified Fibonacci heap

<table>
<thead>
<tr>
<th>Program</th>
<th>Heap size</th>
<th>Binary heap(s)</th>
<th>Fibonacci heap(s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annihilation</td>
<td>20</td>
<td>2.03</td>
<td>3.17</td>
<td>0.64</td>
</tr>
<tr>
<td>Annihilation2D</td>
<td>993</td>
<td>8.44</td>
<td>30.88</td>
<td>0.27</td>
</tr>
<tr>
<td>Morphogenesis2D</td>
<td>2015</td>
<td>361.25</td>
<td>1858.93</td>
<td>0.19</td>
</tr>
<tr>
<td>Mincde</td>
<td>9761</td>
<td>677.43</td>
<td>9753.5</td>
<td>0.069</td>
</tr>
</tbody>
</table>

The modified implementation requires the same amount of memory as the implementation in 4.1, with \( N \) additional doubles for the key value copies in each node.
5 Discussion

With no speedup and a major slowdown, the Fibonacci heap performs poorly compared to the binary heap. Sedgewick, Sleator and the Fibonacci heap creators Fredman and Tarjan state that "[a]lthough theoretically efficient, Fibonacci heaps are complicated to implement and not as fast in practice as other kinds of heaps." [6] Implementation choices can decrease run times, but the extra memory accesses that the Fibonacci heap generates, compared to the binary heap, are difficult to reduce without changing the structure and usage of the heap itself. The most probable reason to the slowdown is the structure of the heap itself when used in URDME.

The promising feature of the Fibonacci heap is the time complexity, as seen in Table 1. With most of the procedures operating in amortized constant time, it is clear to see that a program’s performance would benefit from utilizing the heap’s constant operations more frequently than the non-constant ones. Unfortunately, the solver in URDME makes use of the slower operations more or equally often as the constant operations.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Fibonacci heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make-heap</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Minimum</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Extract-min</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Union</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Decrease-key</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Table 1: Time complexity of the binary and Fibonacci heap [4].
The four used procedures are `make-heap`, `insertion`, `delete` and `extract-min`. The heap will be created once per run, wherein all nodes are inserted at once. The solver calls the `update` function to let the heap know what index in the external data array has been updated. This makes it difficult to know if the data has been increased, decreased or not modified at all in the implementation where nodes only have a pointer to the data and not a copy of the previous data. The result is that the constant `decrease-key` function is not being utilized. Therefore, the corresponding node to the updated data is always moved from its place in the heap into the root list in each `update` function call. The `update` function works as a combination of `delete` and `insert` if the node is not the minimum, alternatively `extract-min` and `insert` if the updated node is the minimum node. This is done by re-linking the updated node, and any possible children of said node, into to the root list. If the minimum node is the updated node then the program will perform unnecessary costly operations.

However, in the modified implementation where a copy of the previous data is stored, the run time differences between the implementations are minimal, as seen in section 4. This implies that the solver increases the data value in the minimum node more often than it decreases, and does not utilize at least one of the constant functions as much as it would have to, in order for it to be more efficient than the binary heap.

Linked lists are not efficient for the processor to handle in general, and frequently invoke cache misses when nodes are placed randomly in memory. In both implementations, nodes are allocated in a block, which increases the chance of reading multiple nodes into the same cache line. However, the size of a node in the Fibonacci heap is so big that only one or two nodes would fit in a cache line, depending on the processor. This can be compared to the binary heap, which requires no memory overhead and can fit multiple nodes in a single cache line. The binary heap also has the advantage of being able to utilize the prefetcher in the processor to save some time on reading memory, as the jumps in memory become predictable. Compared to the Fibonacci heap that only has the pointers that lead to unknown memory according to the processor.
6 Conclusions

There are implementation choices that could improve the performance of the Fibonacci heap when used in URDME. For example, improving the integration of the heap and the solver, and modifying the solver for the Fibonacci heap, could speed up the program. However, improving the implementation will not make the Fibonacci heap more efficient than the binary heap. For a program to efficiently utilize the Fibonacci heap, the majority of procedure calls must be to the constant operations, which is not possible in URDME, where each node update is a combination of extract-min/delete and insert. The Fibonacci heap will spend a lot of time accessing memory, since moving a node to another linked list often requires ten writes in five different nodes. If an updated node has children, the pointer to the parent node must be removed from all child nodes, resulting in additional memory accesses equivalent to the number of children. While the binary heap also need to process a lot of nodes in each update call, this procedure can be performed more efficiently than in the Fibonacci heap.

To sum up, utilizing the Fibonacci heap in URDME instead of the currently used binary heap will not result in greater efficiency.

References


