Advanced Data Structures
Project 1 - Heaps

Peter Gabrielsen 20114179
Christoffer Hansen 20114637

October 27, 2015

Abstract

This project aims to verify the theoretical bounds of the *Fibonacci* and *Binary* heaps through experimentation. We will conduct experiments that tests the experimental performance of each of the basic priority queue operations: `Insert`, `DeleteMin`, `FindMin`, and `DecreaseKey`. The experiments will test the data structures in the average and in the worst case situation. The two heaps will then be tested in use with Dijkstra's single source shortest path algorithm where we will measure the performance in regard to difference families of graphs.

The results of the heap operations align very good with the theoretical bounds in the worst case. In the average case the binary heap seems to be able to do inserts and decrease key operations in constant time. We explain this in the number of bubble up operations in the average case which seem to be constant.

The experiments with using the heaps with Dijkstra's algorithm showed that the binary heap consistently outperforms the Fibonacci heap except for a single case which were a random connected graph with vertex degree of 100.

The code can be found at:
`https://dl.dropboxusercontent.com/u/8990890/2015Q1Q2_ADS_20114179_20114637_Project1.zip`.

Contents

Contents **1**

1 Introduction **2**

2 Implementation **2**

3 Worst case time bounds of the heaps **3**

4 Experimental setup **4**

5 Heap experiments **5**

5.1 `FindMin` 5

5.2 `Insert` 6

5.3 `DeleteMin` 8

5.4 `DecreaseKey` 11

5.5 Conclusion 13

6 Dijkstra's algorithm **13**

6.1 Few `DecreaseKey` operations 14

6.2 Many `DecreaseKey` operations 14

7 Experiments on Dijkstra's **16**

7.1 Methodology 16

7.2 Expectations 17

7.3 Results 18

7.4 Discussion 20

7.5 Conclusion 21

Bibliography **22**

1 Introduction

In this project, we compare the practical performance for *Fibonacci Heaps* and *Binary Heaps* throughout experimentation and evaluation of the actual implementations. This report documents our theoretical, implementational and experimental design considerations and present our experimental results.

In section 2 we present an overview of our implementations and continue to argue about the worst-case time bounds for each of the operations in section 3. We describe our overall experimental setup in section 4 and design and perform experiments where we measure the running times and number of comparisons of the operations for both priority queue implementations in section 5. Continuing our studies, we implement Dijkstra's algorithms for the single-source shortest path problem for weighted graphs with non-negative edge-weights, such that it can switch between the two priority queue implementations. Finally we describe families of graphs where Dijkstra's algorithm performs many, respectively few, `DecreaseKey` operations in section 6 and perform experiments focusing on how the amortized $\mathcal{O}(1)$ `DecreaseKey` operation performs in our experimental setup in section 7.

2 Implementation

The implementations of the Heaps are done in `c++` and supports the operations `MakeHeap`, `FindMin`, `Insert`, `DeleteMin` and `DecreaseKey`. We will in this section describe and discuss the design choices of our implementations of the binary heap and the Fibonacci heap.

Binary heap

The binary heap by Williams [2] was implemented as a minimum heap using the pseudo code in CLRS [1, p. 151-170]. Using this implementation we get the following time bounds for the heap operations:

Operation	Bound
`MakeHeap`	$\Theta(1)$
`Insert`	$\Theta(\log n)$
`FindMin`	$\Theta(1)$
`DeleteMin`	$\Theta(\log n)$
`DecreaseKey`	$\Theta(\log n)$

Figure 1: Time bounds of the binary heap.

In order to implement the decrease key operation we need some way to find the node in the binary heap corresponding to a node in our graph. To do this we have a table which given a node id from our graph gives us the index in the binary heap of that node id. We can then call decrease key. This way of implementing it gives an overhead in space usage of n, where n is the number of vertices in the graph.

Fibonacci heap

The Fibonacci heaps of Fredman and Tarjan [3] was implemented using the pseudo code in CLRS [1, p. 505-522]. Using this implementation we get the following amortized time bounds for the heap operations:

Operation	Bound
`MakeHeap`	$\Theta(1)$
`Insert`	$\Theta(1)$
`FindMin`	$\Theta(1)$
`DeleteMin`	$\mathcal{O}(\log n)$
`DecreaseKey`	$\Theta(1)$

Figure 2: Time bounds of the Fibonacci heap.

In order to implement the decrease key operation we include a table which maps node id from our graph to a pointer to the node within the heap. The mapping was implemented as a $\mathcal{O}(1)$ lookup such that it does not affect our asymptotic time bound. Following the same reasoning as for our binary heap we add an extra n in space usage by using this strategy.

3 Worst case time bounds of the heaps

In this section we will argue about how the algorithms perform in the worst case as opposed to the amortized time bounds. We will for each operation on the heaps argue how the worst cases are obtained.

`FindMin`

Finding the minimum element is in both the Fibonacci heap and the binary heap a constant time operation. We keep a pointer to the minimum element in the Fibonacci heap and in the binary heap we just return the first element in the array which holds the heap.

`Insert`

Inserting into the binary heap is simple. We insert the element at the last position in the binary tree, i.e. the heap, and bubble the element up to its correct position. In the worst case we need to bubble the element to the top, which means bubbling through $\mathcal{O}(\log n)$ layers which gives a logarithmic complexity in the worst case for this operation.

Inserting in the Fibonacci heap is even simpler. We simply just add the element to the root list which just involves manipulating a doubly linked list, i.e. some pointer calculations in $\mathcal{O}(1)$ work.

`DeleteMin`

Deleting the minimum element is $\mathcal{O}(\log n)$ for the binary heap and $\mathcal{O}(\log n)$ amortized for the Fibonacci heap. This upper bound is trivially obtained in the binary heap by having the heap ordered as a sorted array. This way, whenever we delete the minimum we would move the largest element to the top and bubble it $\log n$ levels down again to the bottom.

It can however be extremely expensive to delete the minimum in the Fibonacci heap. When we delete the minimum from a Fibonacci heap we need to consolidate the heap. This operation is linear in the size of the root list. The root list can grow linearly in the worst case if we do n inserts in a row, then the root list will contain at least n nodes, which needs to be consolidated in time $\mathcal{O}(|root\, list|)$.

`DecreaseKey`

Decreasing a key in the binary heap is done by giving the element a new priority and then bubble it up to its new position. The worst case is that the element decreased is the last element and its new priority is the lowest in the tree, which means that it needs to bubble to the top of the heap. This costs $\mathcal{O}(\log n)$.

The worst case for the Fibonacci heap would be if we have a tree of height $\mathcal{O}(\log n)$ where every internal node on the path to the lowest leaf, which is decreased, is marked. This will cause a chain of cuts and cascading cuts all the way up to the root, which will take $\mathcal{O}(\log n)$ time.

4 Experimental setup

The experiments were performed on a machine with a Intel i5-3210M @ 2.5GHz (Ivy Bridge) with 128K bytes of L1 cache, 512K bytes of L2 and 3072K bytes of L3 cache. The machine had 4.2GB ram and ran Ubuntu 14.04 with kernel version 3.16.0-50.

The running time was measured using the built in `high_resolution_clock` in the `chrono` library. This measures the wall clock. It is the clock in `c++` with the highest precision, i.e. the shortest tick period.

The code is compiled with `g++ 4.8.4` with the `c++11` standard enabled and no optimization level.

The elements in our data structures were 32 bit integers. Random elements were generated uniformly in the 32 bit integer range using the Mersenne Twister 19937 from the `random` library.

CPU measurements were collected using `PAPI 5.3.0.0`

We used `perf` to measure page faults.

5 Heap experiments

In this section we will design and perform experiments where we try to measure the running time and number of comparisons of the operations for both our priority queue implementations. We will give a summary of the design of each experiment, results, and discussion of the found results for each operation.

5.1 `FindMin`

Since finding the minimum element of both queues are constant time operations and basically just dereferencing a pointer we did not think it made sense to compare the running times in terms of wall clock. What we did instead is we measured the number of cycles for performing a single FindMin operation and the number of conditional branching operations needed. We would highly expect that the results do not depend on the size of the heap, why we made an experiment where the size of the heap were powers of two in the range from $\left[2^0, 2^{22}\right]$. The experiment were repeated 10000 times and averaged. The results can be found in figure 3 and 4. Our hypothesis were fortunately true and we do not think that there is much to discuss about the results.

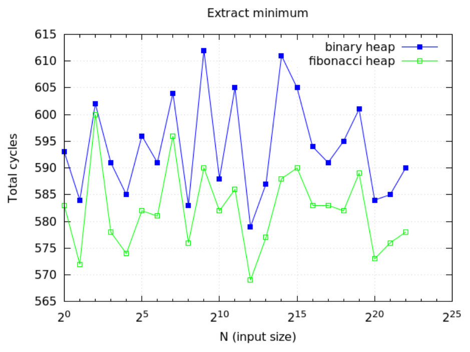

Figure 3: The number of cycles fluctuates a little but this is to be expected.

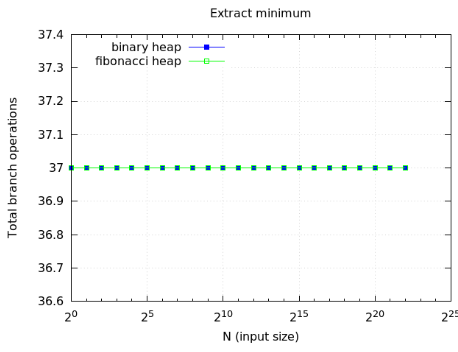

Figure 4: It took exactly 37 branch operations to extract the minimum element.

5.2 Insert

Inserting in the Fibonacci heap is a constant time operation while inserting into the binary heap is logarithmic. We would like to see that the binary heap performs worse. We will test both priority queues in both the average case and the worst case. In the average case we select a random priority in the integer range and insert it into the heap. In the worst case we insert elements into the heap in reverse order. This will cause the elements to bubble all the way to the top on every insert.

The experiments are repeated 100 times and the results are averaged. We plot the y-axis to depict the amount of time a single insert takes or how many branch operations are performed per insert.

We expect to see that the Fibonacci heap is constant time in any case and that the binary heap is logarithmic in both cases. The number of comparisons performed in the Fibonacci heap should therefore also be constant on each insert while the number of comparisons in the binary heap should increase logarithmically in the size of the input and increase even more drastically in the worse case.

The results can be found in figure 5, 6, 7, and 8.

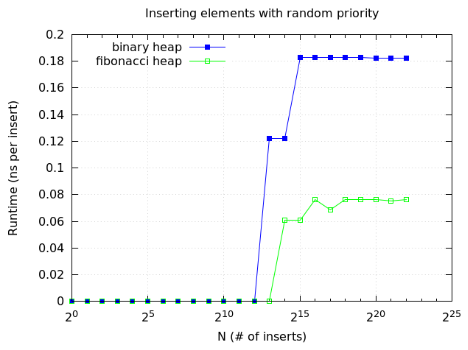

Figure 5: Inserting n elements with random priority and the number of nanoseconds per insert.

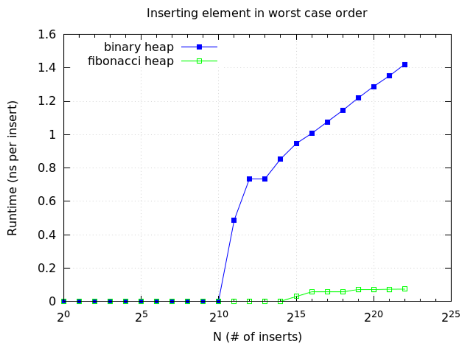

Figure 6: Inserting n elements in reversed order and the number of nanoseconds per insert.

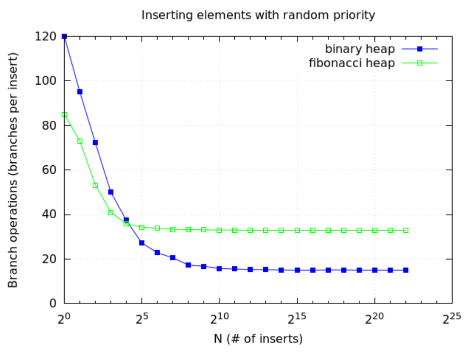

Figure 7: Inserting n elements with random priority and the number of branch operations per insert.

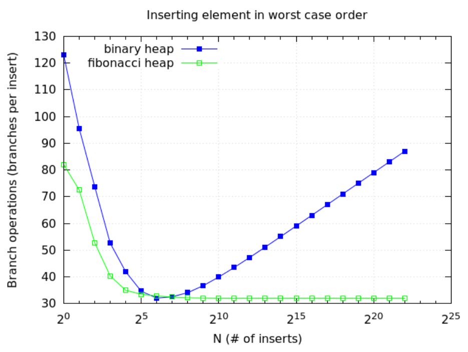

Figure 8: Inserting n elements in the reversed order and the number of branch operations per insert.

Figure 5 shows that inserting randomly into a Fibonacci heap does not depend on the input size beyond a little spike at 2^{12} which is the size at which we exceed the page size. It is interesting to see that inserting random elements into a binary heap is also constant time. This might be explained by the fact that we insert elements in the full integer range, which then results in very few bubble ups. To test this hypothesis we conducted an experiment where we did not use the entire integer range but limited the priority to take random values from $[0, \text{input size}]$. The results of this experiment can be found in figure 9 and 10. The results are very similar to before and we conclude that number of bubble ups must be constant in the random case. To be fully sure about this we measure the number of bubble ups that each insert makes in the random case in the full integer range. The results of this experiment can be found in figure 11. This results shows that indeed the number of bubble up operations is constant in the random case and in the worst case it is logarithmic.

Figure 6 supports that inserting in the binary heap in the worst case is logarithmic and that it does not matter for the Fibonacci heap.

Finally figure 7 and 8 also shows that inserting in the random case is constant for both the binary heap and the Fibonacci heap which were explained in the number of bubble up operations, and in the worst case the binary heap performs logarithmically.

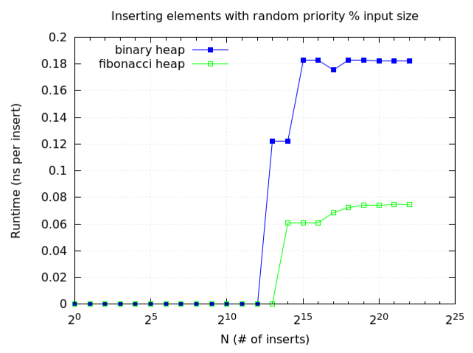

Figure 9: Runtime for the heaps when inserting elements only below the heap size

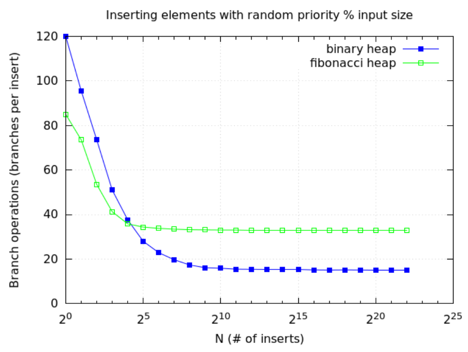

Figure 10: Branch operations for the heaps when inserting elements only below the heap size

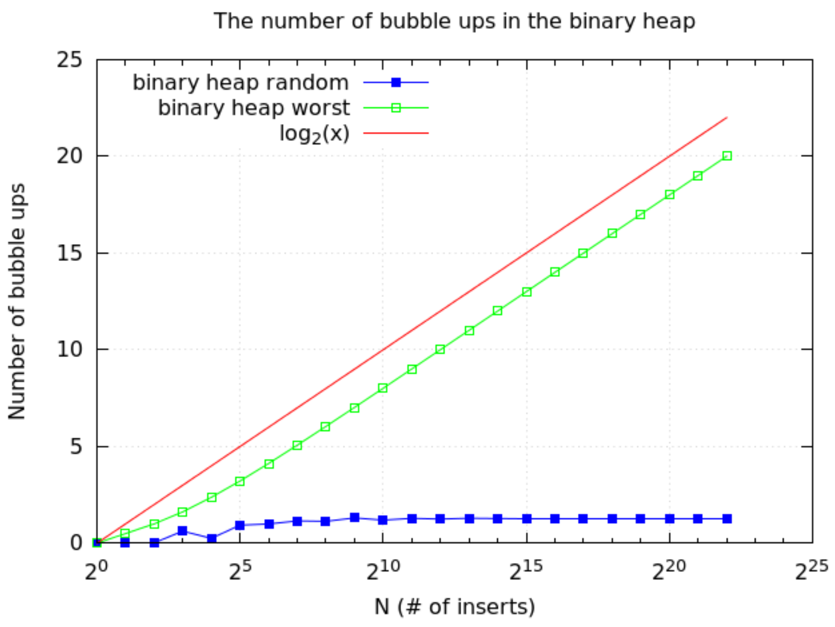

Figure 11: Number of bubble ups in the binary heap in the average and worst case.

5.3 `DeleteMin`

Deleting the minimum element takes logarithmic time in both priority queues. It can however take linear time in the worst case for the Fibonacci heap if a delete minimum operation is performed right after n insert operations. We measure deleting the minimum in the average case where we randomly insert n elements and then first pop one element from both queues such that the Fibonacci queue has consolidated. We then measure deleting one element for different heap sizes. We expect to see both have a logarithmic running time and logarithmic number of comparisons. Testing in the worst case we will do the same but not consolidate the Fibonacci heap first and the binary heap should be inserted in sorted order.

We will also make an experiment where we fill the heap with n elements and then pop them all again. This experiment is repeated 100 times and averaged where as the other with a single and two pops will be repeated 1000 times and averaged.

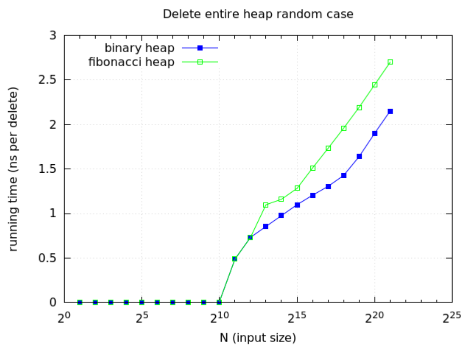

Figure 12

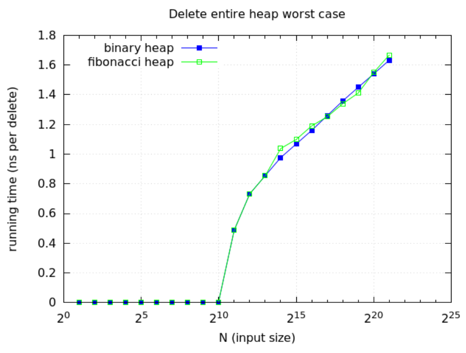

Figure 13

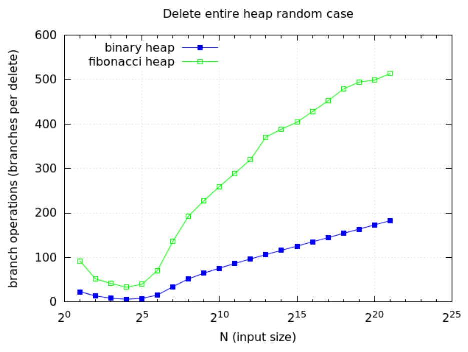

Figure 14

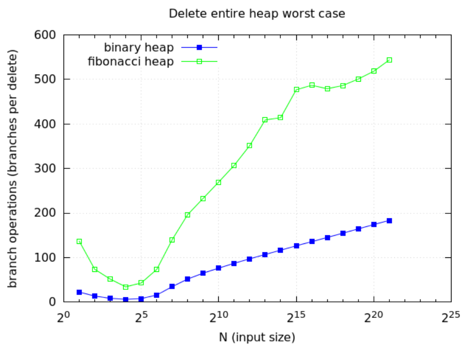

Figure 15

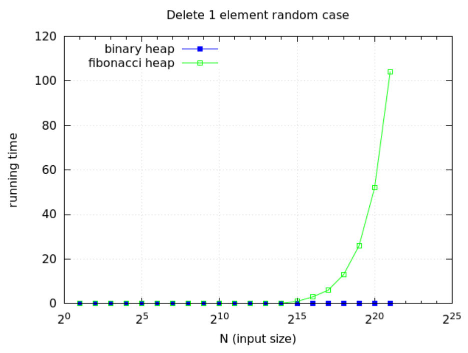

Figure 16

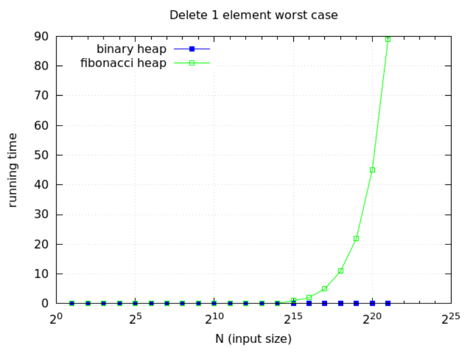

Figure 17

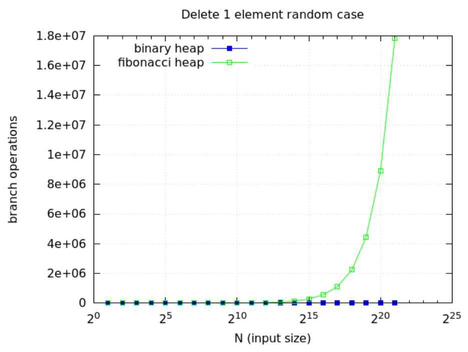

Figure 18

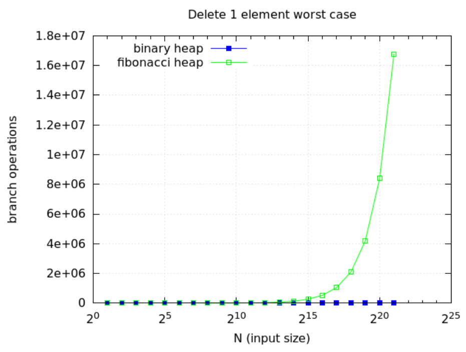

Figure 19

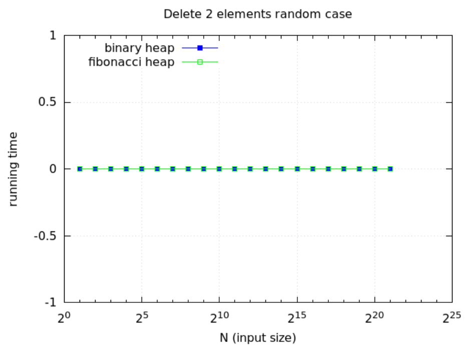

Figure 20

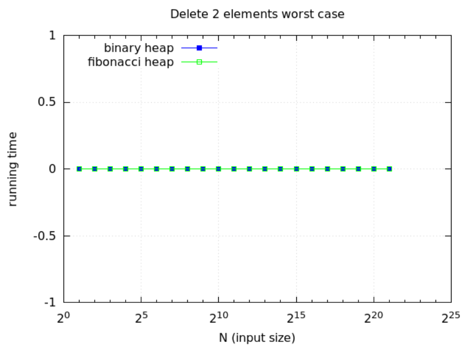

Figure 21

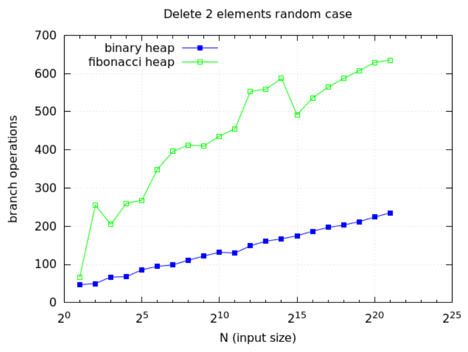

Figure 22

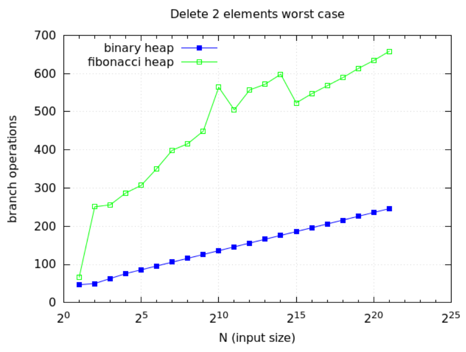

Figure 23

Figure 12 and 13 shows that deleting the minimum takes logarithmic time in the size of the input. The Fibonacci heap does however use a good deal more comparisons in both the random and the worst case as figure 14 and 15 shows.

Deleting one element from the Fibonacci heap after n inserts is really expensive as figure 16 and 18 shows. This is also what we expected due to the expensive linear time consolidate that has to be run. We have the same picture in the worst case in figures 17 and 19.

However if we consolidate the Fibonacci heap by first performing a delete minimum which we do not measure and then run a delete minimum operation, we get the same running time as for the binary heap. The results of this are shown in figures 20, 21, 22, and 23. It can be seen from the number of comparisons that the operations runs in logarithmic number of operations which is what we expected.

5.4 `DecreaseKey`

Decreasing the key in the Fibonacci heap should be amortized constant time as opposed to the logarithmic time in the binary heap. We would like to see this so we conduct an experiment where we first randomly insert elements into the heap, and then perform some decrease key operations decreasing the key with some random number for a random element.

After that we will perform a worst case experiment. Here we will always decrease the key such that it will bubble to the top in the case of the binary heap. The worst case for the Fibonacci heap is really tricky to make a test case for, why the test will be the same as for the binary heap in the worst case.

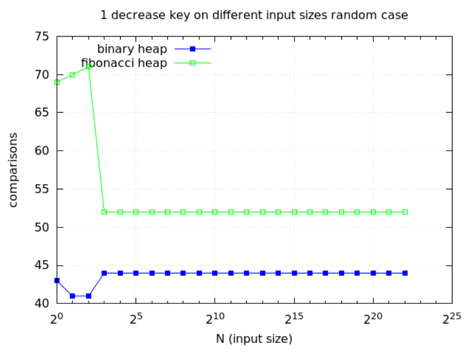

Figure 24

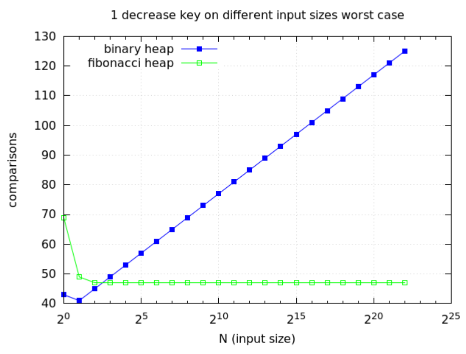

Figure 25

Figures 24 and figure 25 shows the number of comparisons performed when doing decrease key in both the random case and in the worst case for the two heaps. The Fibonacci heap is clearly constant in both the random and worst case which is what we expected. The binary heap is logarithmic in the worst case and as previously seen when using random priorities we get a constant decrease key operation since the number of bubble ups are constant.

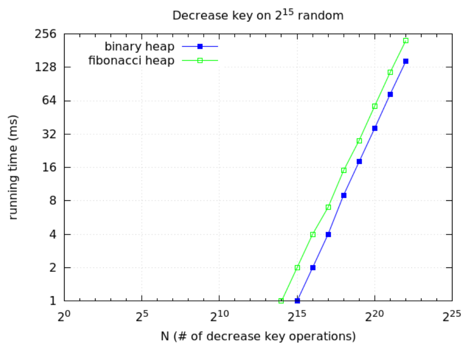

Figure 26

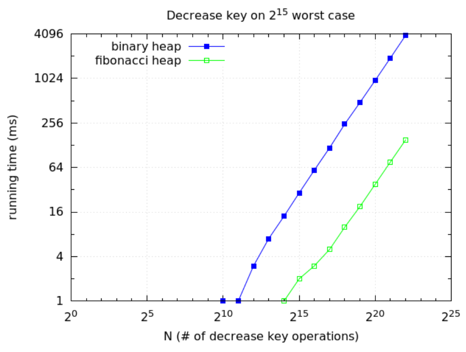

Figure 27

Figure 26 and 27 shows that the operations grows linearly in the number of DecreaseKey operations. In the random case the binary heap outperforms the Fibonacci heap but in the worst case the Fibonacci heap takes the lead by far.

Figure 28 and 29 gives the same picture.

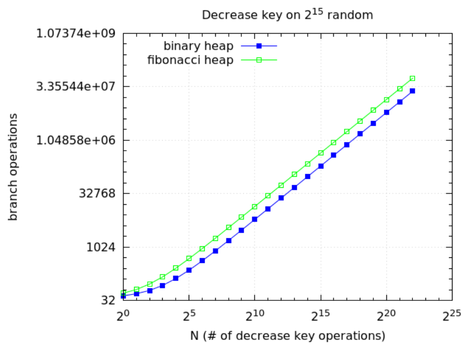

Figure 28

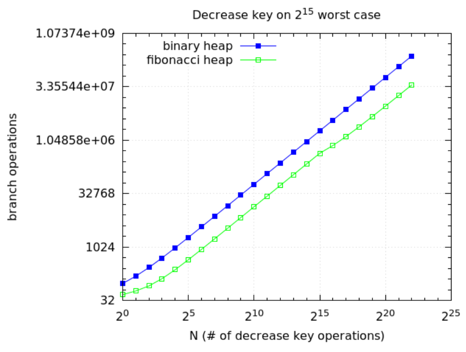

Figure 29

5.5 Conclusion

Finding the minimum element was found to be a constant time operation as expected.

Inserting into the binary heap and Fibonacci heap were very similar in the random case due to the constant number of bubble ups that the binary heap needed to perform in the average case. In the worst case the Fibonacci heap remained constant while the binary heap took logarithmic time per insert as expected.

Deleting the minimum element in a Fibonacci heap can in the worst case become a linear time operation which was shown by inserting n elements and then performing one delete minimum operation causing the consolidation to take linear time. Other than that the Fibonacci heap and binary heap performed very similar as expected.

Decreasing the key in the binary heap is similar to inserting since we need to perform bubble ups and as such the results of this experiment is very similar to those of inserting. In the worst case we get logarithmic running time of decreasing the key in the binary heap, while in the average we perform similar to the Fibonacci heap due to the low number of bubble ups performed.

6 Dijkstra's algorithm

In this section we introduce families of graphs that we believe make Dijkstra's algorithm perform many, respectively few, `DecreaseKey` operations. It is obvious that the number of `DecreaseKey` operations is an exact measure of the number of edges we have to *relax*.

6.1 Few `DecreaseKey` operations

Since Dijkstra's algorithm finds shortest paths from the source s to all vertices in the graph, we have to relax at least one ingoing edge for all other vertices. We therefore conclude that the lower bound for relaxing edges must be $|V| - 1$. If not, the graph would contain unvisited vertices. Having a chain of vertices with *weight* 0 on all edges causes Dijkstra to *relax* each edge exactly once, giving us a graph with the desired property. It is clear that no additional (positive weighted) edges will be relaxed. For a graphical representation of the described class of graphs, please to figure 30. For future reference we denote this family of graphs `ChainGraphs`.

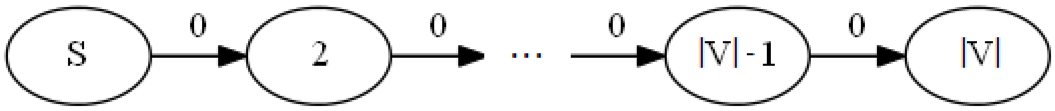

Figure 30: Chain of $|V|$ nodes with edge weight 0.

6.2 Many `DecreaseKey` operations

We consider an analysis of Dijkstra's algorithm using the fact that our implementation makes use of a priority queue Q. Since each vertex is removed from Q exactly once, and the adjacency list of each vertex is scanned exactly once, it is clear that we can at most relax all edges exactly once. In other words we need to consider graphs that makes Dijkstra's algorithm perform $|E|$ `DecreaseKey` operations in order to reach the upper bound.

We present two candidates of classes of graphs with the above mentioned property. The first proposal makes use of negative weights. Dijkstra's algorithm would still be able to run since we are not introducing negative weight cycles, but as we allow for previously marked nodes to be reinserted into the priority queue, we cannot rely on the asymptotic time bound. The second proposal uses only positive edge weights, and can therefore be compared against the asymptotic analysis.

Candidate 1

In the first candidate we split n vertices into two sets S_1, S_2 of sizes $\lceil (n-1)/2 \rceil$ and $\lfloor (n-1)/2 \rfloor$ respectively. We now construct a chain of the vertices of S_1. For vertex v_i in the chain we have outgoing edge weight equal to i for all $i \in [1, \ldots, |S_1|-1]$. The last vertex on the chain has no outgoing edges for now. We now connect each vertex v_i on the chain to all vertices v'_i in S_2 with edge weight equal to $-2 \cdot i$. Finally we connect the source s to v_1 in the chain with edge weight n and to all $v'_i \in S_2$ with edge weight $n - 1$. Please refer to figure 31 for an example with $n = 6$. It is clear the example will make

Dijkstra's algorithm relax all edges, and that this fact generalizes to graphs of arbitrary sizes.

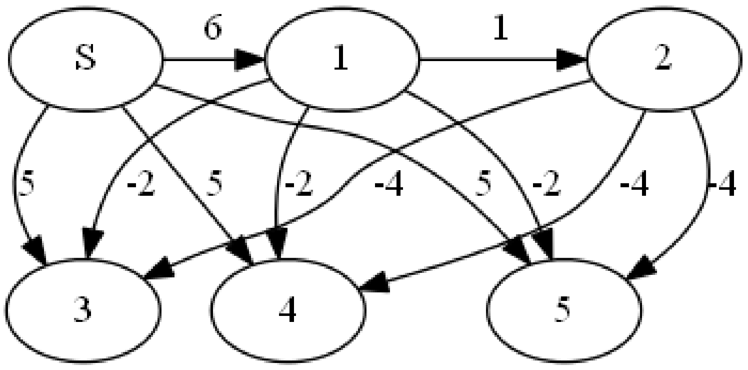

Figure 31: Dijkstra's algorithm will have to relax all edges but we make us of negative edges

Candidate 2

In the second candidate we split n vertices into three sets S_1, S_2, and S_3 of sizes $\lceil (n-1)/3 \rceil$, $\lfloor (n-1)/3 \rfloor$ and $(n-1) - |S_1| - |S_2|$ respectively. We now construct a chain of vertices from S_1 with edge weight 0. Connect each vertex v_i on the chain to all vertices $v''_i \in S_3$ with edge weight equal to $n - 2i$. Now connect each vertex $v'_i \in S_2$ to all vertices $v''_i \in S_3$ with edge weight $n^2 + 2n - 2i$ (we could have used n^3 for smaller n, but in order to avoid overflow we use this edge weight). Finally connect the source s to the first vertex on the chain in S_1 with edge weight n^2 and to all vertices $v'_i \in S_2$ with edge weight $n + i$. Please refer to figure 32 for an example with $n = 9$. It is clear the example will make Dijkstra's algorithm relax all edges, and that this fact generalizes to graphs of arbitrary sizes. For future reference we denote this family of graphs `HeavyGraphs`

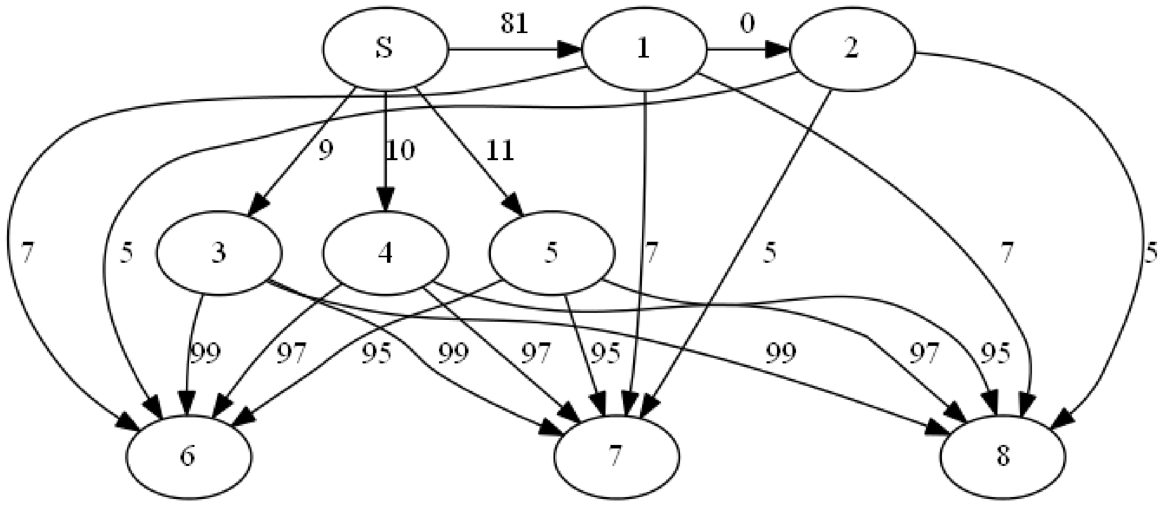

Figure 32: Dijkstra's algorithm will have to relax all edges and we use only positive edges

7 Experiments on Dijkstra's

In this section we document the experiments done on Dijkstra's algorithm using both heap implementations. In particular we focus our studies on observing if the version using Fibonacci heaps achieves an improved performance because of the amortized $\mathcal{O}(1)$ time `DecreaseKey` operation. In the following we set $n = |V|$.

7.1 Methodology

We perform experiments using three families of graphs, namely `ChainGraphs` introduced in section 6.1, `HeavyGraphs` introduced in section 6.2 and `RandomGraphs`. We will introduce the latter family in a short moment. Since we aim at comparing Dijkstra's algorithm on the different families, we started out by adding edges to the graphs such that they have equal number of edges on equal number of vertices. It is intuitively obvious that `HeavyGraphs` have more edges than `ChainGraphs`. A careful analysis tells us that graphs in `HeavyGraphs` with $|V| = n$ will have exactly $2(n/3)^2 + (n/3) - 1$ edges. This gives reason for the following extensions of the constructions of each family, that will make sure that all graphs have equal number of edges on equal number of vertices.

`ChainGraphs`

We construct each graph as described in section 6.1. Furthermore we add $2(n/3)^2 + (n/3) - n - 2$ edges uniformly distributed over all vertices with edge weight 0. This will make the graph have the desired total number of edges equal to graphs in `HeavyGraphs`. The extension will not make the graphs loose the property of making Dijkstra's algorithm perform few `DecreaseKey` operations.

`HeavyGraphs`

We construct each graph following the algorithm presented as *Candidate 2* in section 6.2. No further extensions are needed.

`RandomGraphs`

We construct each graph with n vertices having $\frac{2(n/3)^2 + (n/3) - 1}{n}$ outgoing edges. Each edge is connected uniformly random to another vertex. We avoid having edges connecting a vertex to itself and we avoid to have multiple outgoing edges to the same vertex. Edge weights are taken uniformly random in the interval $[0, .., 100]$. After construction we make sure that all components are connected. We do this by colouring all components and afterwards connecting different coloured components with a random edge. This scheme have a very limited overhead in number of extra edges.

Experimental setup

Please refer to chapter 4 for a general overview of the experimental setup. We performed experiments on Dijkstra's algorithm using both heap implementations and using graphs from `ChainGraph`, `RandomGraph` and `HeavyGraph`. Each experiment was performed 5 times and the average was used.

7.2 Expectations

In general we expect Dijkstra's algorithm to achieve better running times, when using the Fibonacci heap, because of the amortized $\mathcal{O}(1)$ `DecreaseKey` operation. We expect to see the best running times on graphs from `ChainGraph` and the worst running times on graphs from `HeavyGraph` as argued in section 6.1 and section 6.2. When running on graphs in `RandomGraph` we expect to see a slightly worse running time compared with runs on graphs from `ChainGraph`, but we expect to be far better than the runs on the worst case graphs from `HeavyGraph`.

7.3 Results

In figure 33, 34 and 35 we present plots of the total running time with Dijkstra on graphs with $|V| = N \in [2^2, 2^3, .., 2^{16}]$ on `ChainGraph`, `RandomGraph` and `HeavyGraph` respectively. In figure 36, 37 and 38 we present plots of the total running time divided by N. In figure 39, 40 and 41 we present plots of the total number of `DecreaseKey` operations initiated by Dijkstra's algorithm. In figure 42, 43 and 44 we present the total number of conditional branches caused when running on `ChainGraph`, `RandomGraph` and `HeavyGraph` respectively.

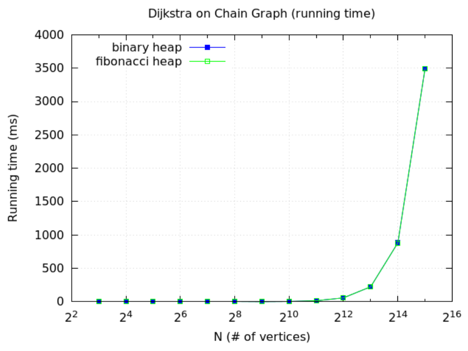

Figure 33: Dijkstra running time on ChainGraph

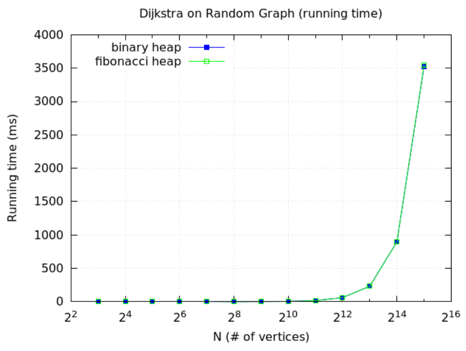

Figure 34: Dijkstra running time on RandomGraph

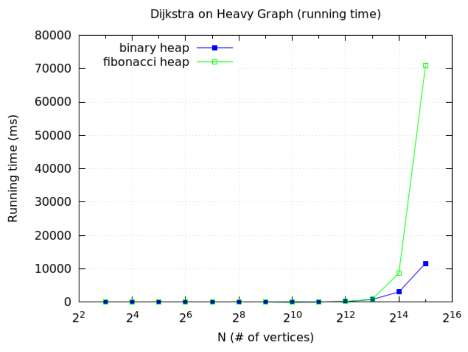

Figure 35: Dijkstra running time on HeavyGraph

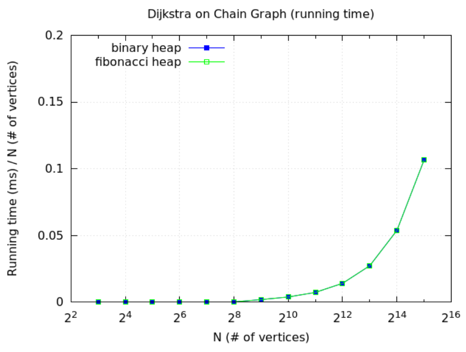

Figure 36: Dijkstra running time per vertex on ChainGraph

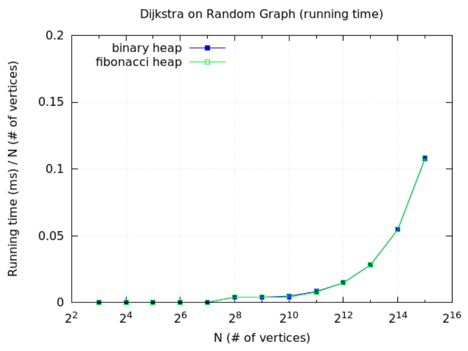

Figure 37: Dijkstra running time per vertex on RandomGraph

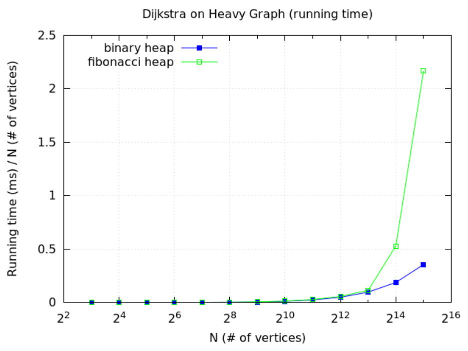

Figure 38: Dijkstra running time per vertex on HeavyGraph

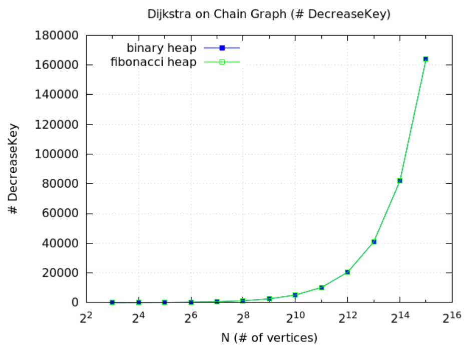

Figure 39: DecreaseKey operations on ChainGraph

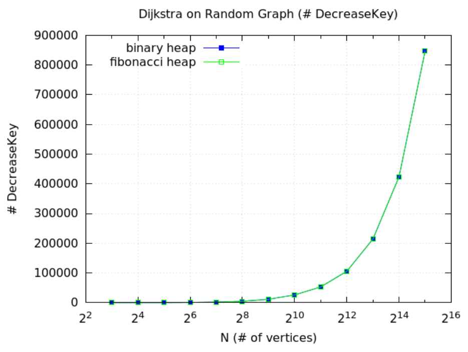

Figure 40: DecreaseKey operations on RandomGraph

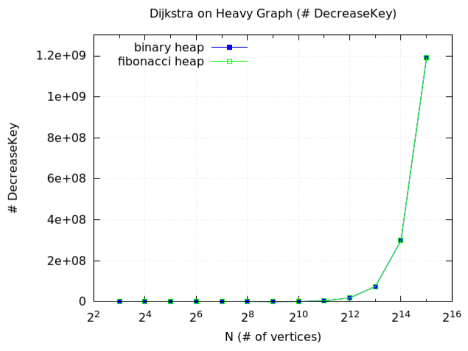

Figure 41: DecreaseKey operations on HeavyGraph

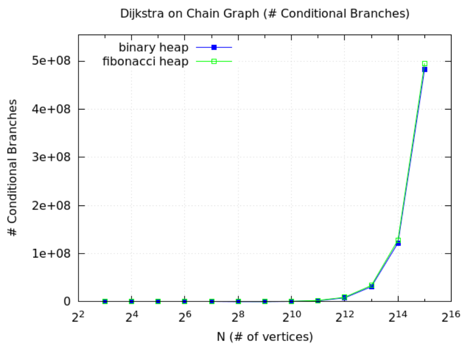

Figure 42: Conditional Branches on ChainGraph

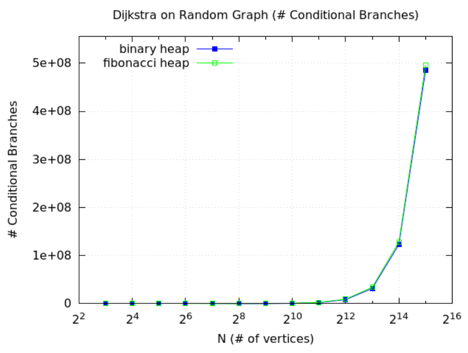

Figure 43: Conditional Branches on RandomGraph

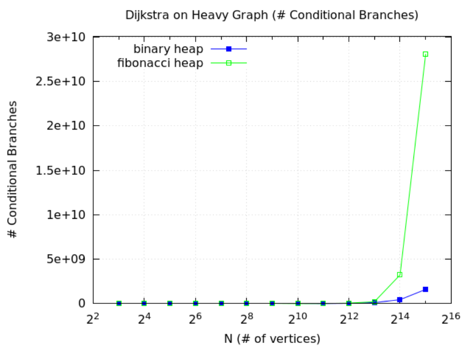

Figure 44: Conditional Branches on HeavyGraph

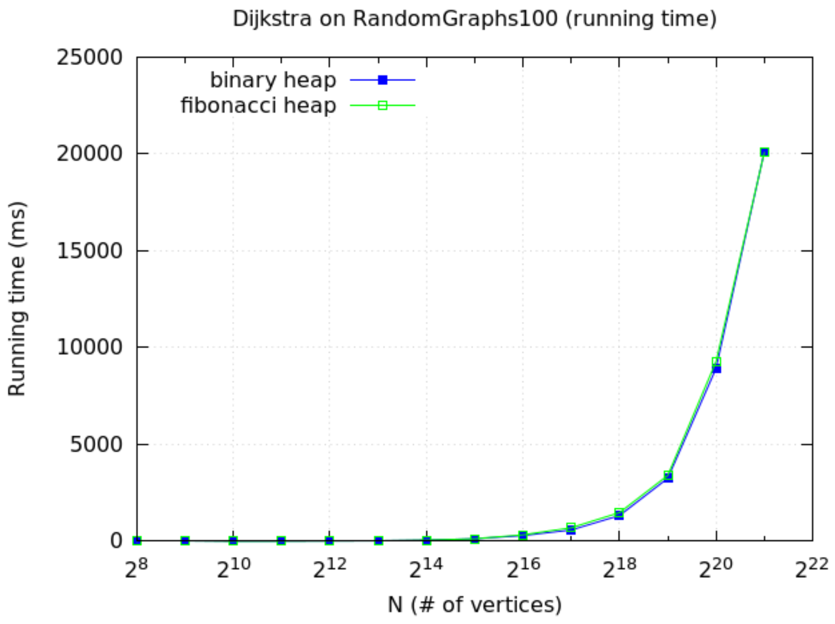

Figure 45: Running time of Dijkstra's algorithm on graphs from RandGraph_100

7.4 Discussion

Comparing the running times on `ChainGraph` respectively `RandomGraph` in figure 33 and 34 we see that our hypothesis of seeing close to the same running times on graphs from `ChainGraph` and `RandomGraph` holds. We are surprised to see that the running times is almost the exact same for both heap implementations. We believe we see this behaviour because we have chosen graphs with relatively few vertices. Even including the $\log_2(n)$ cost of performing an `DecreaseKey` operation on the Binary Heap, we are only penalised with a relatively small overhead of at most $\log_2(2^{16}) = 16$.

It could be that the the constant work we have to do when cutting out a node on a `DecreaseKey` operation on the Fibonacci Heap is so high that we see no difference in terms of actual running time on the test data we have chosen. In figure 35 we are pleased to see that the overall running time has increased. This is what we would expect because of the topology of the graphs in `HeavyGraph`. What surprises us is that the Binary Heap seem to outperform the Fibonacci Heap as graphs grows bigger. We would have expected to see the opposite behaviour. In figures 42, 43 and 44 we measure conditional branches. We believe the figures supports our hypothesis of seeing the heavier constant work done by the Fibonacci Heap in terms of more branches and ultimately in slower actual running time than the asymptotic bound would have suggested.

In order to gain further insight into the validity of our claim, we decided to add another class of graphs to our test set, namely RandomGraphs_100 where we lower the number of edges such that each node has exactly 100 outgoing edges. It is clear that we are able to run test on higher numbers of $n = |V|$ on graphs in RandomGraphs_100 than those introduced in section 7.1. The result of this experiment is repeated in 45. We are pleased to see that Dijkstra's algorithm using the Fibonacci heap achieves slightly better running times that using the binary heap for high values of n.

The fact that the binary heap is so much faster in many cases may also be explained by the fact that in the random case it seems that the binary heap does decrease key in constant time due to the constant number of bubble ups.

7.5 Conclusion

We conducted experiments resulting in data that showed a small tendency of Dijkstra's algorithm achieving better running times, when using the Fibonacci heap, most likely because of the amortized $\mathcal{O}(1)$ `DecreaseKey` operation. Furthermore we got data showing that the best running times was found on graphs from `ChainGraph` and the worst running times was found on graphs from `HeavyGraph` which is what we expected. Furthermore we recorded data that backed the claim of slightly worse running time from graphs in `RandomGraph` compared with runs on graphs from `ChainGraph`. The results indicates that the binary heap performs better than the Fibonacci heap on smaller values of n, which is surprising taking the amortized analysis in consideration.

Bibliography

[1] Thomas H Cormen and Charles E Leiserson. *Introduction to algorithms, 3rd edition*. MIT Press, 2009.

[2] G. E. Forsythe. Algorithms. *Communications of the ACM*, 7(6), 1964.

[3] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.