Binary Heaps, Fibonacci Heaps and Dijkstra's shortest path

Kristoffer Just Andersen, 20103316

Troels Leth Jensen, 20095039

Morten Krogh-Jespersen, 20022362

Project 1, Advanced Data Structures 2013, Computer Science
October 2013
Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Contents

Chapter 1

Introduction

This report details our investigation into the performance characteristics of the heap data structure. The heap can be used as a minimum priority queue, a use applicable in a wide variety of problem domains: process scheduling, sorting and graph algorithms all have a place for heaps.

This paper will explore two general approaches, the binary and Fibonacci heaps, and demonstrate two ways of implementing each, for a total of four different data structures. We argue for their worst case time behavior.

We then put them to the empirical test, stressing them under load from Dijkstra's Single Source Shortest Path algorithm, which we use to demonstrate the performance characteristics of the data structures. Dijkstra's can be implemented in two different ways, and we investigate the consequence of one method over the other.

Chapter 2

Binary heaps

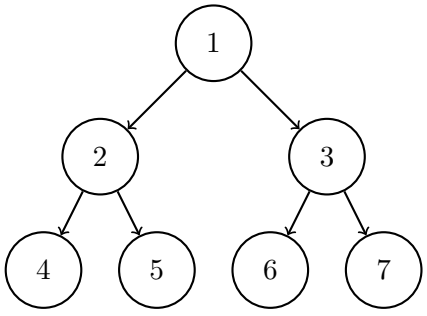
This chapter explores binary heaps as presented in [3]. A binary heap gets its name from the balanced, binary search tree over its elements that it maintains. We shall see two ways to achieve this, one maintaining an explicit search tree, the other relying on the isomorphism between arrays and balanced trees.

The key to using a binary tree for the purpose of a priority queue is the "heap invariant": that the priority of each element in the tree is smaller than its children. Maintaining this invariant is the focus of each operation on binary heaps, and the notion of "bubbling" elements up and down through the tree, swapping children and parents, appears again and again, as we shall see. We begin with an overview of the running times of each of the two implementations we investigate.

Operation	Array Implementation	Pointer Based
MakeHeap	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$\Theta(1)$
Insert	$\mathrm{O}(\log n)$	$\Theta(\log n)$
DeleteMin	$\Theta(\log n)$	$\Theta(\log n)$
DecreaseKey	$\Theta(\log n)$	$\Theta(\log n)$

2.1 Binary Heaps with Arrays

The original paper from 1964 utilizes the balanced binary tree implied by the order of elements in an array. If we label the elements in the array from 1, we see that they correspond uniquely to a position in the following binary tree:



The parent-child relationships in the tree can be seen in the array by observing that the numbers of node is children are $2i$ and $2i + 1$. Similarly, the parent of node i is node $\left\lfloor \frac{i}{2} \right\rfloor$. Therefore, if the implementation language supports arrays, there is no need to maintain an explicit representation of a binary tree, likely saving space and time.

It is of course entirely possible to take the alternate approach, which we investigate in the next section.

2.2 Implementation Decisions

The algorithms used in our C implementation are faithful to the paper, save for slight modernizations such as removal of `goto` statements in favor of familiar looping control flow. This does not alter the asymptotic running times or space consumption of the algorithm.

We choose to in-line the SWOPHEAP operation as used by [3], as we believe it makes the intent clearer. SWOPHEAP has the effect of "bubbling" an element down through the heap until the heap invariant is reestablished. This is used for DELETEMIN, where the root of the tree is removed and replaced by the last leaf of the tree, which is then bubbled downwards.

For the DECREASEKEY operation, we refer to Cormen *et al.* [1], which demonstrates an implementation of maximum priority queues supporting an INCREASEKEY operation. It is easy to transfer the algorithm to our implementation. The basic algorithm decreases they key of an element, and then proceeds to bubble the element up through the heap until the heap invariant is reestablished, similar to how the original implementation achieves insertion.

2.3 Time-complexity for Binary Heaps with Arrays

We here present arguments for the worst-case running times of the following operations: MAKEHEAP, FINDMIN, INSERT, DELETEMINand DECREASEKEY- the operations we need to implement Dijkstra's Algorithm in later sections.

MakeHeap Runs in constant time: $\mathrm{O}(1)$. All we do is a single allocation of contiguous memory to store the elements of the heap.

FindMin Runs in constant time: $\mathrm{O}(1)$. We do a single array indexing, as, according to our heap invariant, the element of lowest key is always in root position - the first element in our underlying array.

Insert Runs linearly in the number of elements in the heap: $\mathrm{O}(n)$. We insert the incoming node into the last position of the tree, which we know by maintaining the current element count. We then bubble the element upwards, performing a comparison and a swap per level it moves. Thus we achieve a logarithmic running time as the depth of a balanced binary tree is proportional to $\log n$. However, due to the workings of C, we cannot dynamically alter the

size of the underlying array, and so, worst-case, we have to reallocate the entire underlying array. We use the same approach as the standard Java ArrayList of doubling the underlying array each time the limit is reached. This requires copying the entire array, byte by byte, into a new array, an operation linear in the number of elements, which dominates the logarithmic cost of maintaining the heap invariant. While this is a worst-case analysis, it is worth noting that the amortized cost is $\mathrm{O}(1)$, where we pay for the cost of reallocation by paying more per insertion.

DeleteMin Runs logarithmically in the number of elements inserted: $O(\log n)$. We Remove the minimal element according to FindMin and replace it as root with the last element in the tree (always found at position Size in the array). We then bubble the element down through the tree, at most two comparisons and a swap per level in the tree. The depth of a balanced binary tree is $O(\log n)$. So we do a constant amount of work per level the node moves, resulting in the logarithmic running time.

DecreaseKey Runs logarithmically in the number of elements inserted: $\mathrm{O}(\log n)$. We decrease the key of the specified element, and proceed to bubble it upwards until the invariant is reestablished, using a single comparison and a swap per level moved. The argument is similar to the one used for insertion and deletion.

2.4 Binary Heaps with Pointers

While the original implementation used an underlying array, it is not strictly necessary to achieve the benefit of a binary heap. We can model the balanced binary tree explicitly using pointers.

The algorithms follow the same procedures operationally as described above, but with slight variations since we loose the direct access to arbitrary positions in the tree facilitated by an underlying array. Instead we must "count" our way into the tree.

The "trick" to achieve the same, if not better, running times as the original implementation, is observing that position of a node i in the tree is given uniquely by the binary representation of $i - 1$. Read from most significant bit, the ones and zeroes gives "directions" from the root to the particular node. We defer to the implementation for the bloody details.

2.5 Time-complexity for Binary Heaps with Pointers

We here detail the worst case running times for the key operations of a minimum priority queue: MakeHeap, FindMin, Insert, DeleteMinand DecreaseKey.

MakeHeap Runs in constant time: $\mathrm{O}(1)$. Allocation of the necessary variables for keeping count of the elements inserted and a pointer to the root node, once inserted.

FindMin Runs in constant time: $\mathrm{O}(1)$. By maintaining a pointer to the root node, we have constant time access to it.

Insert Runs logarithmically in the number of elements: $\mathrm{O}(\log n)$. By maintaining the number of elements inserted, we can calculate directions to the position of the next element. This involves reading bits and following pointers. We follow one pointer per bit of "direction". The binary representation of n uses $\log n$ bits, so we follow $\mathrm{O}(\log n)$ pointers. Upon insertion, we then bubble the element up to reestablish the heap invariant, swapping appropriate pointers as we go. We do this a number of times proportional to the depth of the tree. Thus we run in time $\mathrm{O}(2 \log n) = \mathrm{O}(\log n)$.

DeleteMin Runs logarithmically in the number of elements: $\mathrm{O}(\log n)$. Similarly to INSERT, we can find the last element in the heap by tracing a path down from the root. This is logarithmic in the number of elements for the same reasons. Then, we swap the root and the last element, and bubble the last element down until the heap invariant is restored. This is also logarithmic in the number of elements.

DecreaseKey Runs logarithmically in the number of elements: $\mathrm{O}(\log n)$. We decrease the key of the specified element, and bubble upwards until we reestablish heap order. This is proportional to the depth of the tree and thus logarithmic in the number of elements.

2.6 Correctness of Binary Heaps

We opted for a visual inspection rather than unit testing. We implement a `to_dot` procedure that serializes a heap in the form of a GraphViz dot file, from which we can render an image of the associated binary tree. The procedure traverses the actual tree structure recursively in the case of the pointer based implementation, while calculating the implicit binary tree in the case of the array based implementation.

This gives us quick and easy visual feedback, which is invaluable particularly in the case of the pointer based implementation. Between two nodes, we have a total of 4 children pointers and 6 parent pointers that needs to be updated, which is a lot for any programmer to manage at a single time. Visualizing the pointers gave us immediate feedback when on a by-need basis testing proved discrepancies in behavior.

Chapter 3

Fibonacci heaps

In this chapter we focus on Fibonacci heaps, which is a data structure that has a forest of rooted trees as opposed to a binary heap that only has one tree [2]. The data structure was invented by Michael L. Fredman and Robert Endre Tarjan and was published in the Journal of ACM in 1987. It has it name because the size of any subtree in a Fibonacci heap will be lower bounded by F_{k+2} where k is the degree of the root in that subtree and F_k is the kth Fibonacci number. Below is the time-complexities of each of the heap operations listed:

Operation	Binary heap	Fibonacci heap v1 (amortized)	Fibonacci heap v2 (amortized)
MakeHeap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$\Theta(1)$	$O\left(l(\log(\frac{n}{l})+1)\right)$
Insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
DeleteMin	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$O(1)$	$O(1)$
Delete	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
Meld	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

3.1 Properties of the Fibonacci heap

The Fibonacci heap is a heap that has better amortized bounds than binary heaps, and one of the reasons to this is that some of the operations are lazy. We will later see that this has a big impact on worst-case time complexities for the operations that do the heavy lifting.

The heap maintains a collection of root nodes in a doubly linked circular list that supports inserting, joining and single deletions in constant time. Each node has a left and a right sibling pointer that facilitates the circular doubly linked list, a pointer to the parent and a pointer to an arbitrary child. If there is no parent or child the pointers point to **null**. A node is marked if it has had a child removed. If a child is removed from a parent that is already marked the parent will be moved to the root and the parents parent will be marked or moved. A mark on a node is removed when it is added as a child to another node.

Items added to the heap are added as single-item trees rooted with a node for the corresponding item. It is only when performing a linking step that trees grow. Therefore, the degree of nodes can only change when linking, removing or decreasing the key of an item, since decreasing the key of an item cuts the node from its parent and move the node up to be joined with the roots.

The upper bound $D(n)$ on the degree of any node of an n-node Fibonacci is $\mathrm{O}(\log n)$ [1, p. 523] [2, p. 604]. This can be shown by first observing that the degree of any node y in the Fibonacci heap is bounded by when it was inserted into the list of its parent x child, where $degree(x) = k$. When y_i was linked to x, where i declares when y was added to the children list of x, x and y_i must have had the same degree which results in $degree(y_i) \leq k-1$. y could have lost at most one child before it would be cut from x resulting in $degree(y_i) \geq i-2$ where $i = 1, 2, \cdots, k$.

By three different induction proofs and the lemma described above, it can be shown that if $k = degree(x)$ for any node in a Fibonacci heap then $size(x) \geq F_{k+2} \geq \phi^k$, where $\phi = (1+\sqrt{5})/2$ also known as the golden ratio. Hereafter, showing the bound of $D(n)$ is straight forward:

$$\begin{gathered} n \geq size(x) \geq \phi^k \\ \Downarrow \\ k \leq \lfloor \log_\phi n \rfloor \end{gathered}$$

3.2 The potential method

We will analyze the amortized running times of the Fibonacci heaps by using the potential function [2, p. 215]. A potential function maps a state of a data structure to a real number representing the potential contained in that data structure [1, p. 459]. The amortized cost of an operation c_i that transforms a data structure in state D_{i-1} to D_i is:

$$\hat{c} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The amortized cost is therefore the cost of the operation c_i plus the change in the potential. If the operation releases potential, the released potential gets subtracted from the cost of the operation. For the Fibonacci heaps we define the potential function as:

$$\Phi(H) = trees(H) + 2marked(H)$$

Where trees define the number of trees/roots in the forest and marked is the number of marked nodes. The reason to why marked nodes contains two units of potential is that one unit pays for the cut and the other pays for the node to become a root and thereby form a new tree.

3.3 Fibonacci heap version 1

The first Fibonacci heap variant we present is the original version proposed in FT87. A potential function is used to analyze the performance, thus the above stated time-complexities are amortized.

Our implementation pretty much follows from the article, with few minor exceptions. The article do not specify exactly how a node is found from an item in constant time, so we decided to place a pointer on the item. Also, melding is not totally destructable since we join the heap in of the two existing heaps and return an arbitrary one.

The article mentions that DELETE takes $\mathrm{O}(1)$ if the node to remove is not the min-node and without cascading deletes. The children of the node to delete must be moved up onto the root which can only be done in constant time if every children has a pointer to a parent pointer. In this way, we only have to change one pointer to update all parent pointers for the children of the aforementioned node. Since the running time of the DELETE operation is amortized $\mathrm{O}(\log n)$ we chose a simpler version where we just update all parent pointers one by one. As stated above $size(node) \leq \mathrm{O}(\log n)$ so this only takes $\Theta(\log n)$ time.

MAKEHEAP constructs a new empty Fibonacci heap, which means there are not roots and no marked nodes. Therefore, the potential of the empty heap is 0 and constructing the heap can be done in constant time. FINDMIN does not alter the potential and is only a pointer lookup which clearly is also a constant time operation. INSERT inserts a new root which increments the potential by one. Combine that with the actual cost which is $\mathrm{O}(1)$ the INSERT operation is $\mathrm{O}(1)$. MELD can be carried out in $\mathrm{O}(1)$ actual time and since the change is:

$$\Delta = \Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$$

the MELD operation has time $\mathrm{O}(1)$.

When analyzing the amortized cost of DELETEMIN, let us say that the minimum node to delete is actually the node with degree $D(n)$ which is the upper bound on the maximum degree for any node. Setting the parent pointer to **null**and concatenating each node with the root list takes constant time for each of the $D(n)$ children. The linking phase work on at most $D(n)+tress(H)-1$ trees, since we have removed the minimum node from the root list and at most will be adding $D(n)$. We also join trees of same degree, but when two trees are joined only one will remain in the root list, so this can happen at most the number of roots in the root list. Therefore, the actual working time is $\mathrm{O}(D(n) + trees(H))$.

The potential before the DELETEMIN operation executes is $trees(H) + 2marked(H)$. After, at most $D(n)+1$ roots will exist because all others would be removed during the linking phase and there is no change to marked nodes. Therefore, we have that:

$$\begin{aligned}
\text{DELETEMIN} &= \mathrm{O}(D(n) + trees(H)) + ((D(n)+1) + 2marked(H)) \\
&\quad - (trees(H) + 2marked(H)) \\
&= \mathrm{O}(D(n)) + \mathrm{O}(trees(H)) - trees(H) \\
&= \mathrm{O}(D(n))
\end{aligned}$$

Actually, a slight subtlety is happening above. We can turn up the units that we pay to cover the hidden constant in $\mathrm{O}(trees(H))$, which allows us to cancel those two terms out, resulting in an amortized running time of $\mathrm{O}(D(n)) \leq \mathrm{O}(\log n)$.

DecreaseKey takes constant time for moving the node for the item with the decreased key to the root, but then c cascading deletes can occur. Therefore, the actual time is $\mathrm{O}(c)$. The change is potential is:

$$\begin{aligned}\Delta &= \mathrm{O}(trees(H) + (c-1) + 1) + 2(marked(H) - (c-1) + 1) \\ &\quad - (trees(H) + 2marked(H) \\ &= \mathrm{O}(trees(H) + c) + 2(marked(H) - c + 2) - (trees(H) + 2marked(H)) \\ &= 4 - c\end{aligned}$$

Therefore, the amortized running time of DecreaseKey is $\mathrm{O}(c) + 4 - c = \mathrm{O}(1)$

Last, we have to cover the running time of Delete. First, we cut the node and move the children to the root, which is at most $D(n)$ operations. Next, a chain of cascading deletes can occur where the length is c, making the actual time $\mathrm{O}(D(n) + c)$. The change in potential is as follows:

$$\begin{aligned}\Delta &= \mathrm{O}(trees(H) + D(n) + c) + 2(marked(H) - c + 2) \\ &\quad - (trees(H) + 2marked(H) \\ &= D(n) + 4 - c\end{aligned}$$

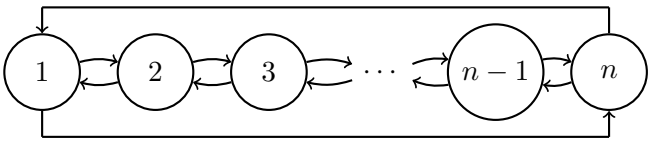
But as we showed for DecreaseKey, the released potential pays for the cascading deletes, so we only have to pay the $D(n)$ for the actual time and $D(n)$ for the change in potential, which is $\mathrm{O}(D(n)) \leq \mathrm{O}(\log n)$.

3.4 Worst case time-complexity for Fib heap v1

There are three operations where changes to the potential occurs, and thus, the stated times are amortized for DeleteMin, DecreaseKey and Delete. Below we illustrate the worst-case for each of these operations by showing a configuration and how that configuration can be obtained from a sequence of operations :

3.4.1 Worst case time-complexity for DeleteMin

As we showed in the previous section the actual time for performing DeleteMin is $\mathrm{O}(D(n) + trees(H))$. When linking trees the change in potential pays for $trees(H)$, therefore, it is in the amount of trees the worst case configuration for DeleteMin can be found. It is easy to see that the worst case is when all the nodes in the heap is at the root in a linked list:



This configuration can be achieved by just calling insert n times. For simplicity, let us assume that n is odd, and a call to DeleteMin happens. In the above example 1 will be removed and we are left with $n-1$ root nodes to link. This results in $n-1$ key comparisons, but all the trees of rank 1 will be joined too and this will continue until no trees of duplicate size is found.

$$\#\text{ of operations} = \mathrm{O}\left(n-1\right) + \mathrm{O}\left(\frac{n-1}{2}\right) + \mathrm{O}\left(\frac{n-1}{2^2}\right) + \cdots + \mathrm{O}\left(\frac{n-1}{2^{\log(n-1)-1}}\right)$$

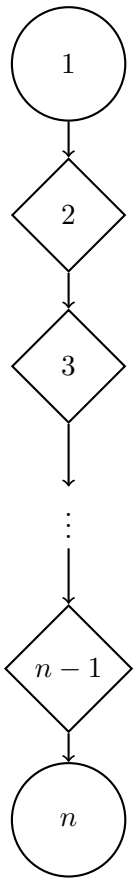
which is $\Theta\left(n\right)$.

3.4.2 Worst case time-complexity for Delete and DecreaseKey

As we shoved in the previous section, the release in potential for cascading deletes pays for most of the work, therefore, we can find a worst-case configuration that will perform considerably worse in actual time.

If Delete is invoked with the min-node as argument then Delete calls DeleteMin, therefore, the worst case for Delete is $\Theta\left(n\right)$, but we will show that without the min-node as argument, we still end up with $\Theta\left(n\right)$. The following observation holds true for DecreaseKey as well.

If we delete a child to an arbitrary node x we mark x if is not marked and if it is marked, we cut x from its parent, move the subtree formed by x to the root and try to mark the previous parent of x. This could result in cascading deletes. Therefore, the worst situation would be the following configuration:

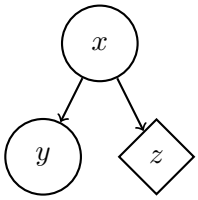


where a diamond is modeling a marked node. If either DecreaseKey or Delete is called with an item corresponding to node n a cascading delete will begin and will not stop until it reaches 2 in this example. The amount of operations is therefore the entire chain:

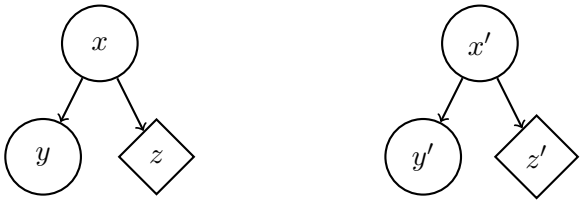
$$\text{length of chain} = n - 1$$

which is $\Theta(n)$.

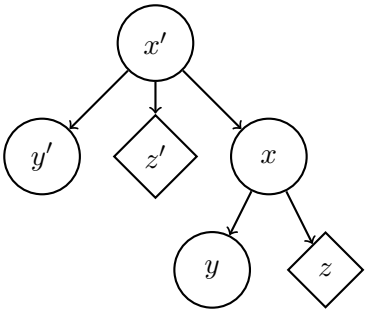
Such a configuration can be obtained by constructing triangles, chaining and deleting (as a side note, this is exercise 19.4-1 in [1]). In order for us to create the first triangle, we insert five nodes, call DeleteMin and then remove the bottom most:



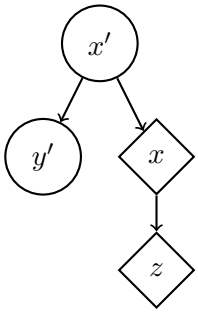
We can then create another triangle, having x' being a lesser key than x. The picture below is just before joining the two rank two trees.



The result is then:



Now, we can delete y and z' and we will have the wanted structure.



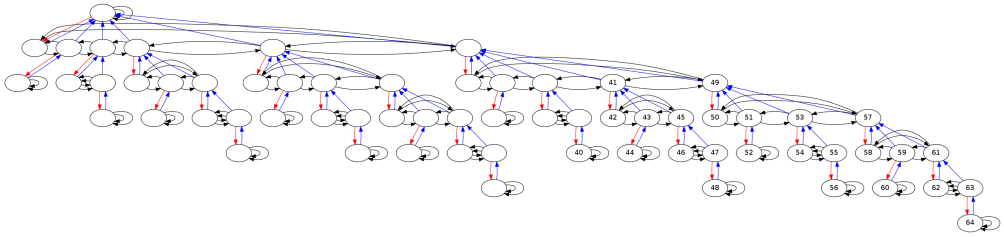
To create a chain of height n, we have to create n triangles.

3.5 Fibonacci heap version 2

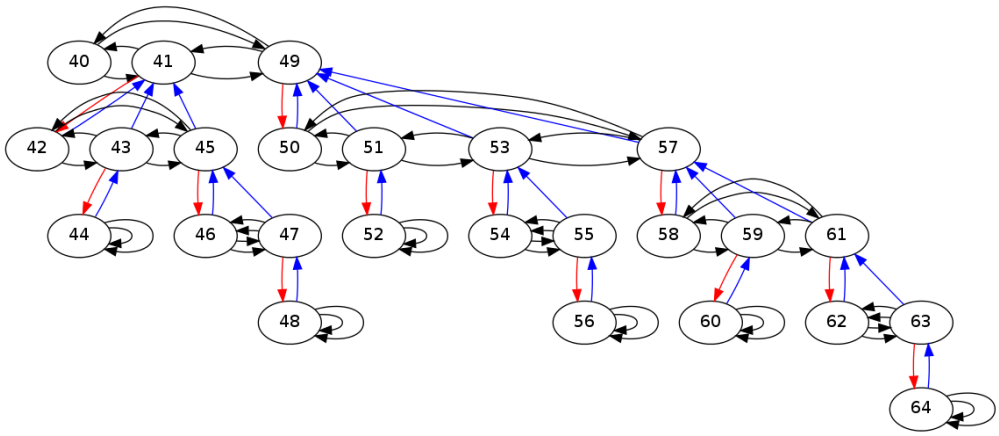
In this second version of a Fibonacci heap we try to be more lazy, by doing as little work as possible for DELETE and DELETEMIN. When a call to either one of the operations is called, we mark the node as vacant but then halt to do any additional work. This makes DELETE and DELETEMIN run in $\mathrm{O}(1)$.

We have to do some work at one point which happens in FINDMIN. Now, FINDMIN works like DELETEMIN in version 1, but also has to delete every vacant node it meets on the path. MELD also requires little work, such that when melding two heaps, if a root is marked as vacant it should be the new root of the resulting heap. There is no change in actual time for MELD.

Below is an image from our test-framework showing how a tree looks like with vacant nodes:



After a call to FINDMIN the heap looks like:



We defer the analysis of the time complexcity of the FINDMINoperation to [2], Lemma 2, but mention the final result of $\mathrm{O}\left(l \cdot (\log \frac{n}{l} + 1)\right)$, where l is the number of vancant nodes in the heap.

3.6 Worst case time-complexity for Fib heap v2

The worst case example is extremely easy. Since l and n are disjoint, the can be filled with vacant nodes. The worst-case would then be calling FINDMIN, just to find out that no such node exist.

3.7 Testing correctness of Fibonacci Heaps

Implementing the Fibonacci heaps in C required a lot of work and since we are working with that many pointers, we decided very early in the process that we needed some kind of test-framework to assist us.

First, we implemented a consistency checker, that checks the following properties for each node x: That the nodes the sibling pointers of x points to points to x with the corresponding sibling pointer. That the parent pointer is correct if the node x is a child of any node and that the key is larger than its parent. Still, this tests the implementation more than the properties of the Fibonacci heaps, so we needed one more tool.

We build a pretty printer that could convert any subtree (or heap) to a graphical representation, such that we could see how the heap looked like. We then made test-cases, that for each operation printed the output, and then we could manually check if DeleteMin, Delete and DecreaseKey behaved as it should. We manually checked test-instances of size less than or equal to 100 operations.

Attached in the zipped-file accompanying the report are generated images we have used to check correctness.

Lastly, since we now have four data structures, we could check the output for each and compare it with the others, to make sure, that all our heaps answer correctly (or all could answer wrong the same way, which is highly unlikely).

Chapter 4

Dijkstra's algorithm

Dijkstra's algorithm is an graph search algorithm that solves the single-source shortest path problem. Without a min-priority queue the algorithm runs in $\mathrm{O}(V^2) \geq \mathrm{O}(E)$ for a graph $G = (V, E)$[3]. But we have two heaps that can actually be used as priority queues, so this will give us better asymptotic running times depending on the connectivity of the graph. Normally, Dijkstra is implemented by using the DecreaseKey operation, but if we disregard space, we can actually implement Dijkstra without DecreaseKey and just insert new nodes every time we find a shorter path. Below is a summary of the running time and space complexities:

Dijkstra	Binary heaps	Fibonacci heap v1	Fibonacci heap v2
with	$\mathrm{O}((n+m)\log nn)$	$\mathrm{O}(n\log nn+m)$	$\mathrm{O}(n\log nn+m)$
without	$\mathrm{O}((n+m)\log nn)$	$\mathrm{O}(n\log nn+m)$	$\mathrm{O}(n\log nn+m)$

We represent graphs by a linked list so running Dijkstra with inserts will make a drastic increase in memory consumption for sparsely connected graphs, but only be a constant factor larger for a fully connected graph.

4.1 Running times for Dijkstra with heaps

We can show an upper bound on the time complexity for Dijkstra as a function taking the number of nodes $n = |V|$ and the numbers of edges $m = |E|$ as such: $\mathrm{O}(n \cdot dm + m \cdot dk)$ where dm denotes the number of DeleteMin operations and dk denotes the number of DecreaseKey operations.

First, let us consider Binary heaps and the case for Dijkstra with DecreaseKey operations. A node can never be re-added once it has been deleted by DeleteMin, therefore there can only be n dm operations. For each node, there can be up to m dk operations, one for each adjacent node in the graph. Since Insert, DeleteMin and DecreaseKey all take logarithmic time, the total time is $\mathrm{O}((n+m)\log nn)$. For the Dijkstra version with Inserts only, the m previous calls to DecreaseKey will now call Insert instead but the time will remain the same.

We are able to reduce the time complexity asymptotically when we use a Fibonacci heap. For Fibonacci heaps, DecreaseKey takes amortized time

$\mathrm{O}(1)$ and DeleteMin takes amortized time $\mathrm{O}(\log nn)$ resulting in the time-complexity $\mathrm{O}(m + n \log nn)$ for Dijkstra with Fibonacci heap version 1. Because the time-complexity for Insert is also $\mathrm{O}(1)$, using Dijkstra without DecreaseKey has the same total time complexity.

So what happens when we are lazy with deletions, and for DeleteMin mark a node as vacant in Fibonacci version 2? There are no random deletions in the Fibonacci heap when running Dijkstra, therefore l, which denotes the number of vacant nodes, has an upper bound of 1. Therefore, the amortized time for FindMin is $\mathrm{O}(\log nn + 1)$ which results in the same running times for Dijkstra as Fibonacci heap version 1.

4.2 Connectivity and generating graphs

Above we argued for the running times of the Dijkstra algorithms, and as we can see, the running times are dominated by the number of edges for the graph. This suggest, that sparsely connected graphs can run quicker than highly connected graphs.

If the graph is sufficiently sparse it is an effective speed-up to implement Dijkstra using a min-priority queue of some sort. For Binary heaps, the threshold is when $m = o(n^2/\log nn)$:

$$\mathrm{O}((n+m)\log nn) = \mathrm{O}\left((n^2/\log nn)\log nn\right) = \mathrm{O}\left(n^2\right)\sum_{i=1}^{l}\phi^{k_j-2} \leq n$$

which is the same running time as without a min-priority queue.

For Fibonacci heaps the time $\mathrm{O}(n \log nn + m)$ suggest, since m dominates $n \log nn$, that if the graphs is not totally connected the Fibonacci heap will run quicker or in the same time as Dijkstra without min-priority queues. We are very confident, that the graph should be strictly sparser since the running time in O-notation for Fibonacci heaps hide away a very large constant.

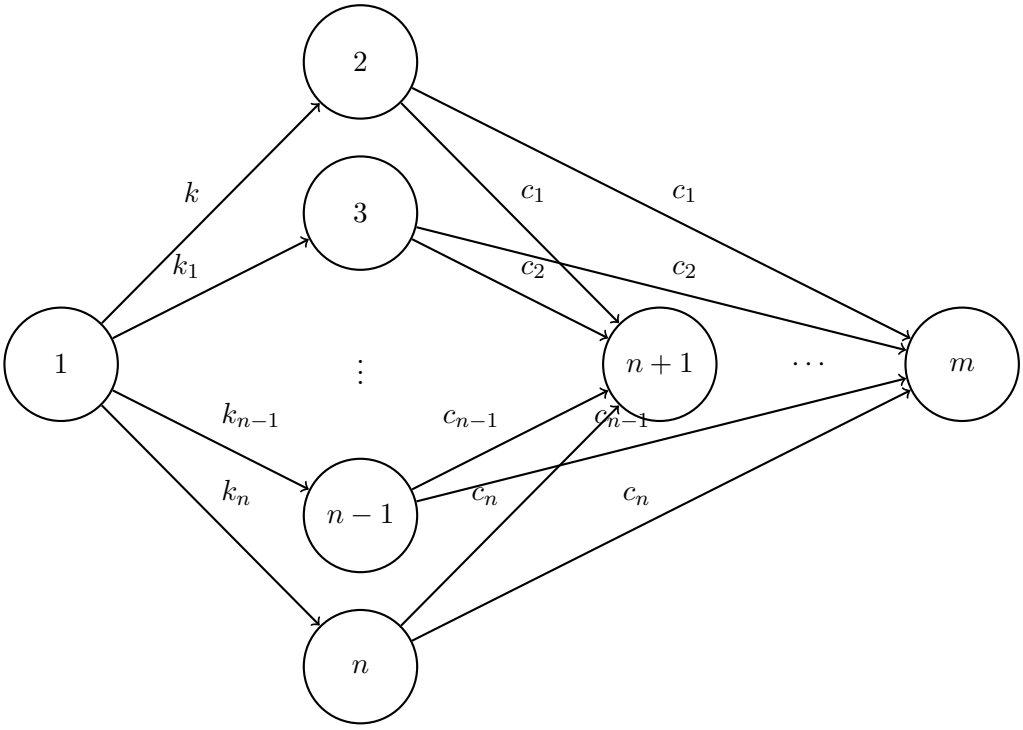
Because connectivity is a factor when running Dijkstra we are, when testing the running times for the Dijkstra algorithms, generating graphs by supplying size and connectivity probability as parameters to our graph-creation program. This is based on the pseudo-random number generator in C but we will use the resulting graphs as if they were completely randomly generated graphs with an expected probability on the number of edges in the graph.

4.3 Pathological Cases

Overall, the previous section provides a general description of when Dijkstra will perform a lot of operations, but one could be interested in graphs that has many or few Insert- or DecreaseKey operations.

If the first path that Dijkstra computes to each node is the optimal path only n DecreaseKey operations will be called. A trivial case is when the graph is a tree.

A more interesting case that can result in many or few calls to DecreaseKey has the following structure:



For suitable k_i and c_j we can make DIJKSTRA call DECREASEKEY for every node $n+1$ to m, for each of the nodes in the vertical list 2 to n. If $n = m = |V|$ DIJKSTRA will make $\mathrm{O}\left(n^2\right)$ calls to DECREASEKEY.

Chapter 5

Empirical Testing

In this chapter we present out findings gathered through empirical testing of the heap implementations discussed in this report. We run two rounds of testing. One a general stress-testing round where we try to force corner-cases and worst-case behavior, in order to verify our implementations and the time complexities we claim in the preceding chapters.

Secondly, we investigate the performance of each of the 4 implementations discussed so far, when plugged into the two variations Dijkstra's Algorithm found above.

5.1 Stress-Testing

We have devised a series of 11 tests in order to exhibit the various time complexities of each operation facilitated by our implementations under different special cases. These tests were also used to control correctness of the implementations. The first six tests listed below are run on the order of 600.000 elements, for all 4 implementations.

- A series of INSERTs, elements inserted in increasing key order. We expect linear behaviour, as the keys are inserted immediately in the right location. Our results prove us correct in 6.1 and 6.12.
- As before, followed by a single DELETEMIN. This forces re-balancing of Fibonacci heaps. We point to 6.2 and 6.13.
- As the previous test, but with elements inserted in decreasing key order, in order to force bubbling and tree traversals on each insert. This behaves as expected as 6.3 and 6.14 show.
- As the first test, followed by as many DELETEMIN operations, testing correctness of repeated bubbling actions. This is surely something that is expected to be $n \log n$ operations as we also show in 6.4 and 6.15.
- Mixed insertions and deletions, to simulate some degree of normal use, i.e. inserts may or may not introduce new minimums. Any chain of operations will be $\mathrm{O}(\log n)$ per operation for Binary heaps and for Fibonacci heap we

can establish the same upper bound due to Theorem 1 [2, p.604]. Please see 6.5 and 6.16 for running time and comparison measurements.

- As the first, followed by as many DecreaseKey operations; each making the currently largest element the smallest. Showcasing the worst case for the binary heaps, as each DecreaseKey operation bubbles the element from a leaf to the root. Again, this is expected to be $n \log n$ and can bee seen in 6.6 and 6.17.

The following five test cases are run only for the Fibonacci heaps, as a they use the Delete operation, which we have only defined for these. The last test is especially interesting as it is designed to force worst-case behaviour as opposed to the amortized bounds of remove. These tests were also run on an order of 600.000 elements.

- A series of inserts, followed by as many Delete operations. Removing the elements in decreasing order. 6.7 and 6.18.
- Same as the first, but with a DeleteMin operation before the Delete (one less now) operations. The point being that the DeleteMin operation will force the Fibonacci heap to make a linking step. 6.8 and 6.19
- Same as above, but removing in reverse(increasing) order. 6.9 and 6.20
- Same as above, but removing from the middle of the range of keys, in alternatingly decreasing and increasing order. 6.10 and 6.21
- A series of deletions and insertions to force the phenomenon of cascading deletions. 6.11 and 6.22

We mention them here for completeness as they were part of our verification/testing approach, but not strictly asked of the report. We refer to Appendix A.3 for plots of the running times.

5.2 Dijkstra's Algorithm Tests

We have 4 different data structures and 2 variations, for a total of 8 different combinations.

We generate graphs of input sizes in the interval of 100 to 20.000 in increments of 500 nodes. A graph with 20.000 nodes is represented by adjacency matrix where all indices are integers is by our calculation around 1.5 GB of space, which by all considerations should be a "fairly large" graph. This should reveal asymptotic discrepancies if any.

In order to exercise worst- and best-case cases, we run each configuration at each input size with 10 different graphs, distributed evenly on the parameter of connectivity. We range from 10% to 100% connectivity in steps of 10%.

Finally, we run the algorithm-data structure-input size-connectivity configuration 3 times to account for effects out of our hands such as the operating system of the test machine.

We measure running times through the use of the GNU `time` command, from which we gather the actual CPU time spent on running the test, not the wall clock time. This accounts for various, unforeseen effects on the running time such as preemption by the operating system or other disturbances.

Instrumentation that allows us to tally the number of comparisons are implemented through the use of C macros. By doing it this way, we can selectively compile the instrumented comparison functions out when we are not counting comparisons, removing any overhead imposed by such.

5.3 Raw Data

We here present 10 plots of running times. The 10 plots are for each of the degrees of connectivity, in decreasing order. That is, Figure 5.1 displays the running times of each algorithm / data structure pair as a function of input size, on completely connected graphs.

We observe no discernible difference in the choice of data structure. The plots group into 2 significant divisions, one using implementation of Dijkstra's Algorithm that uses DecreaseKey, the other using Insert.

In general, we observe shorter absolute running times in inverse proportion to the connectivity: the denser the graph, the longer the running time. This is to be expected, as the run time complexity of Dijkstra's is a function of the number of edges in the graph.

However, as the connectivity decreases, we observe that the implementation of Dijkstra's Algorithm using Insertpulls away from the other combinations - that is, given a choice between the two implementations of the algorithm, the Insertbased is expected to perform better asymptotically, regardless of the choice of underlying data structure.

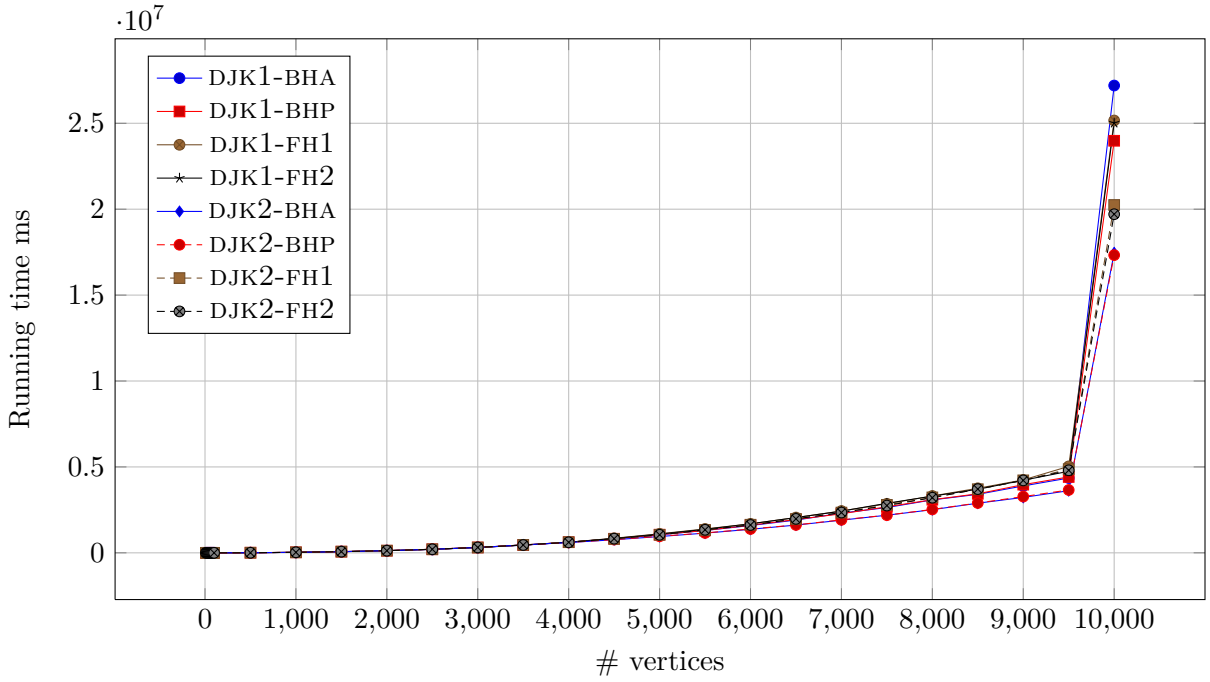


Figure 5.1: Average time of running Dijkstra1 and Dijkstra2 on 100 % connected graph

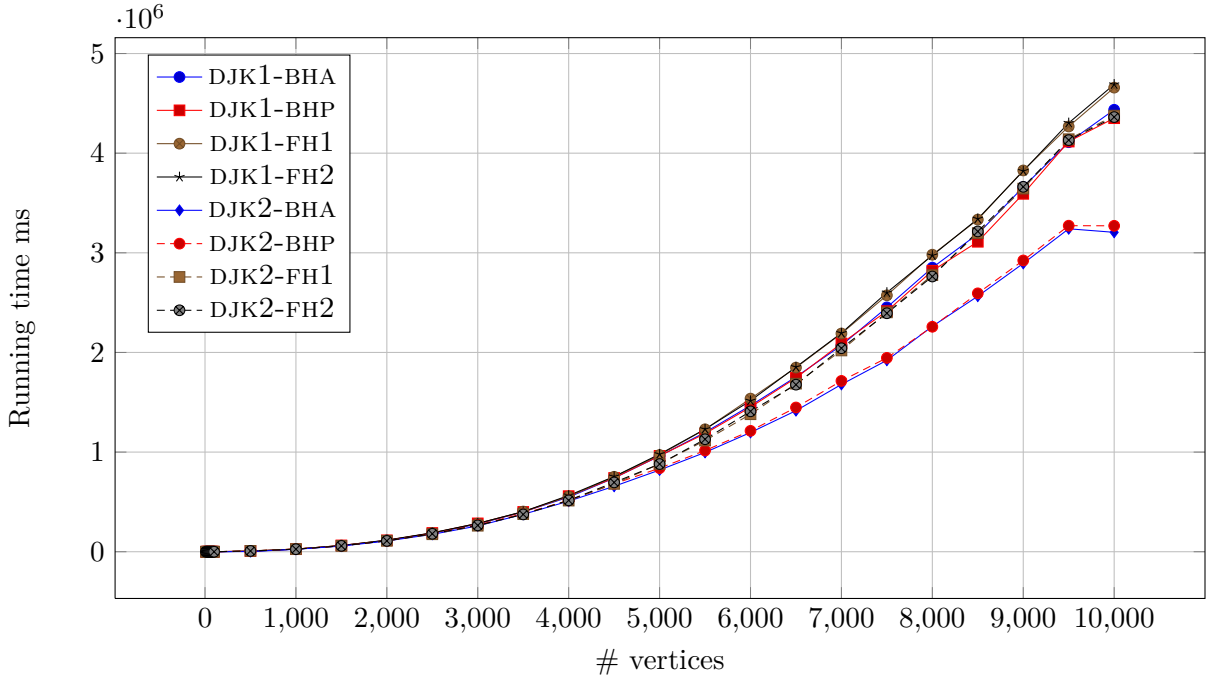


Figure 5.2: Average time of running Dijkstra1 and Dijkstra2 on 90 % connected graph

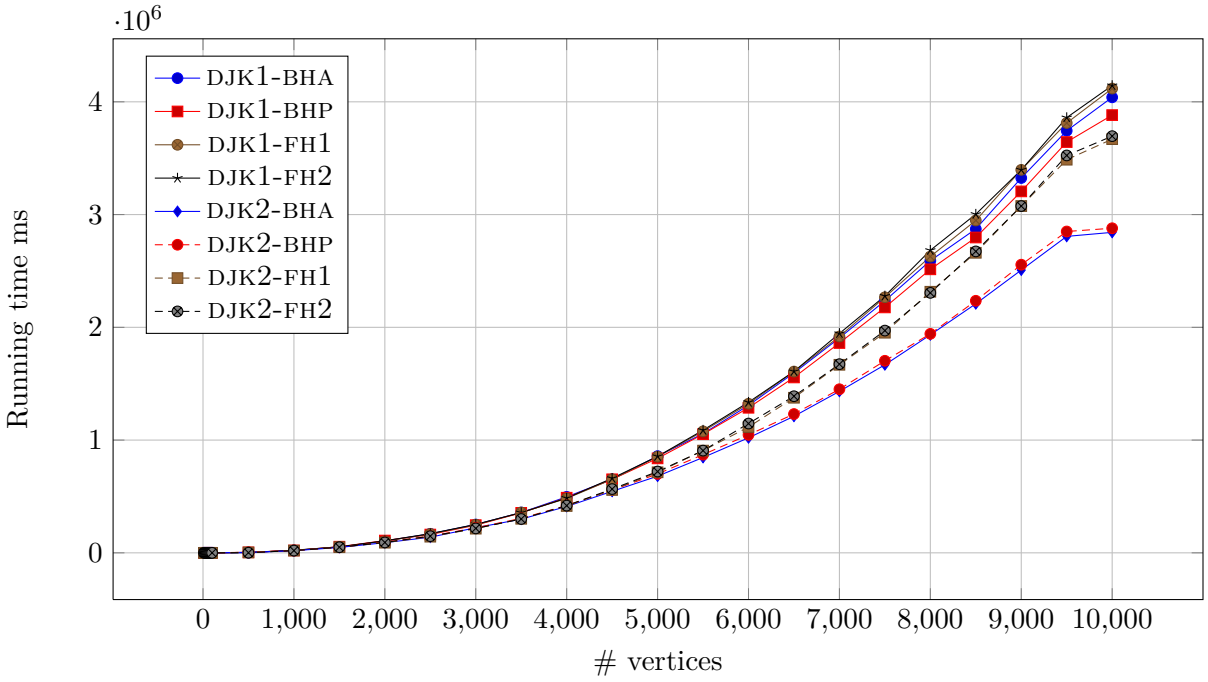


Figure 5.3: Average time of running Dijkstra1 and Dijkstra2 on 80 % connected graph

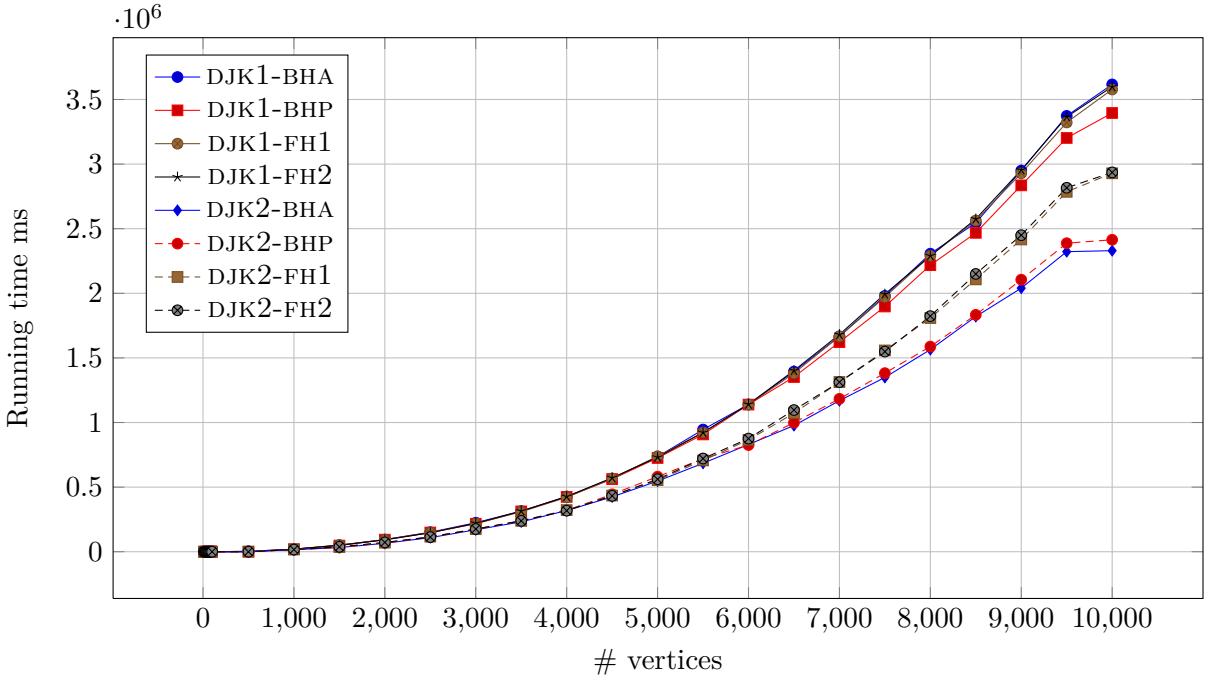


Figure 5.4: Average time of running Dijkstra1 and Dijkstra2 on 70 % connected graph

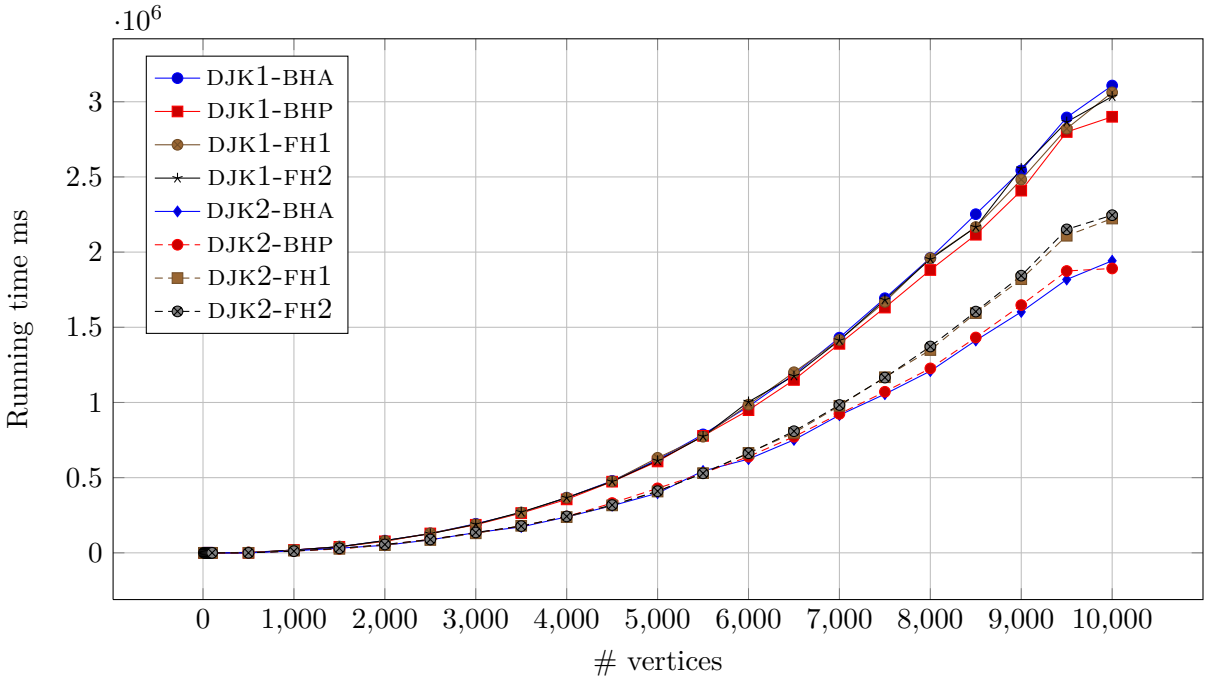


Figure 5.5: Average time of running Dijkstra1 and Dijkstra2 on 60 % connected graph

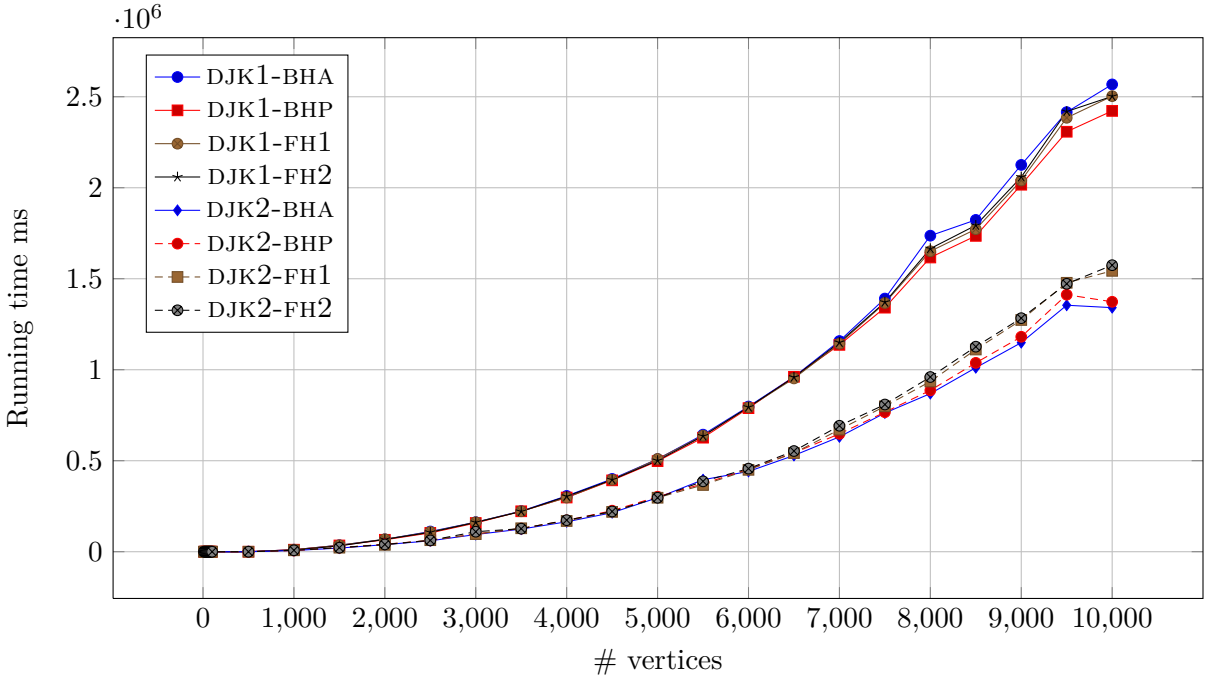


Figure 5.6: Average time of running Dijkstra1 and Dijkstra2 on 50 % connected graph

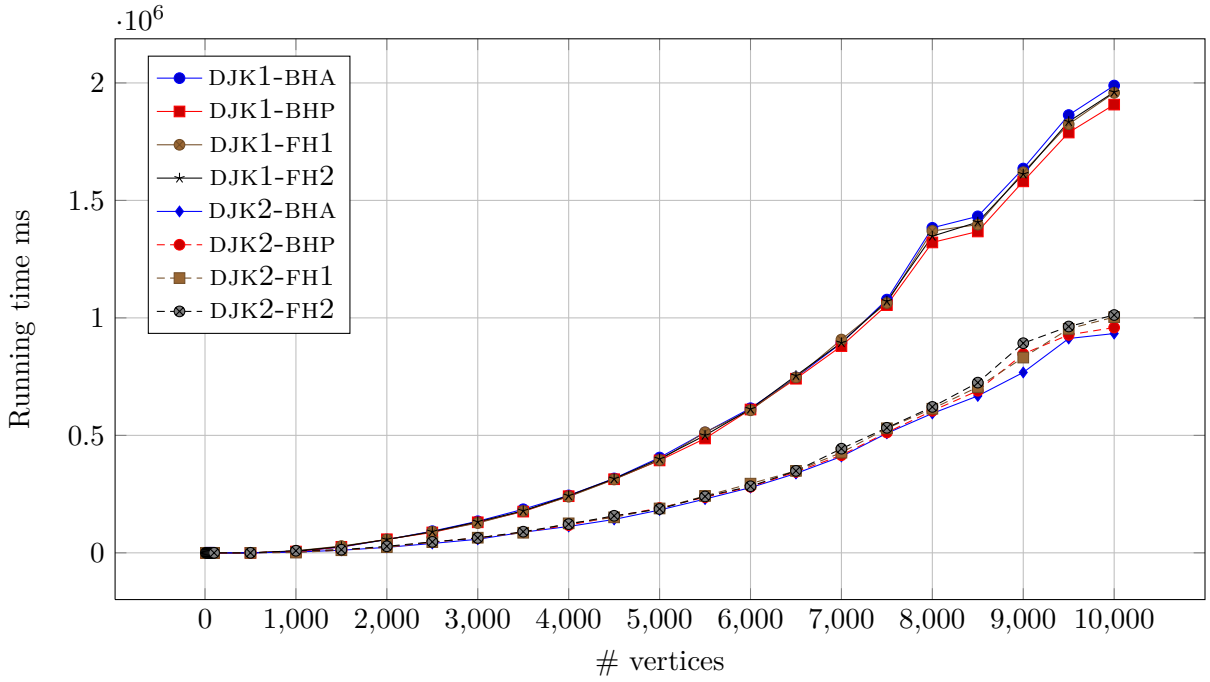


Figure 5.7: Average time of running Dijkstra1 and Dijkstra2 on 40 % connected graph

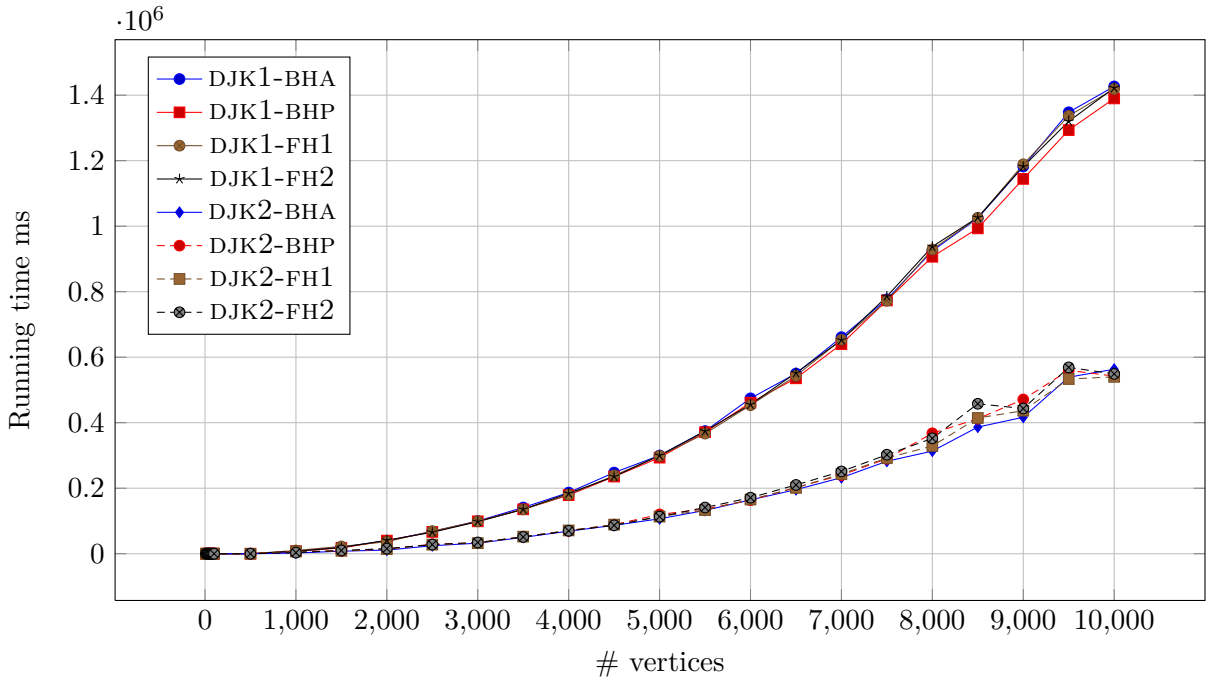


Figure 5.8: Average time of running Dijkstra1 and Dijkstra2 on 30 % connected graph

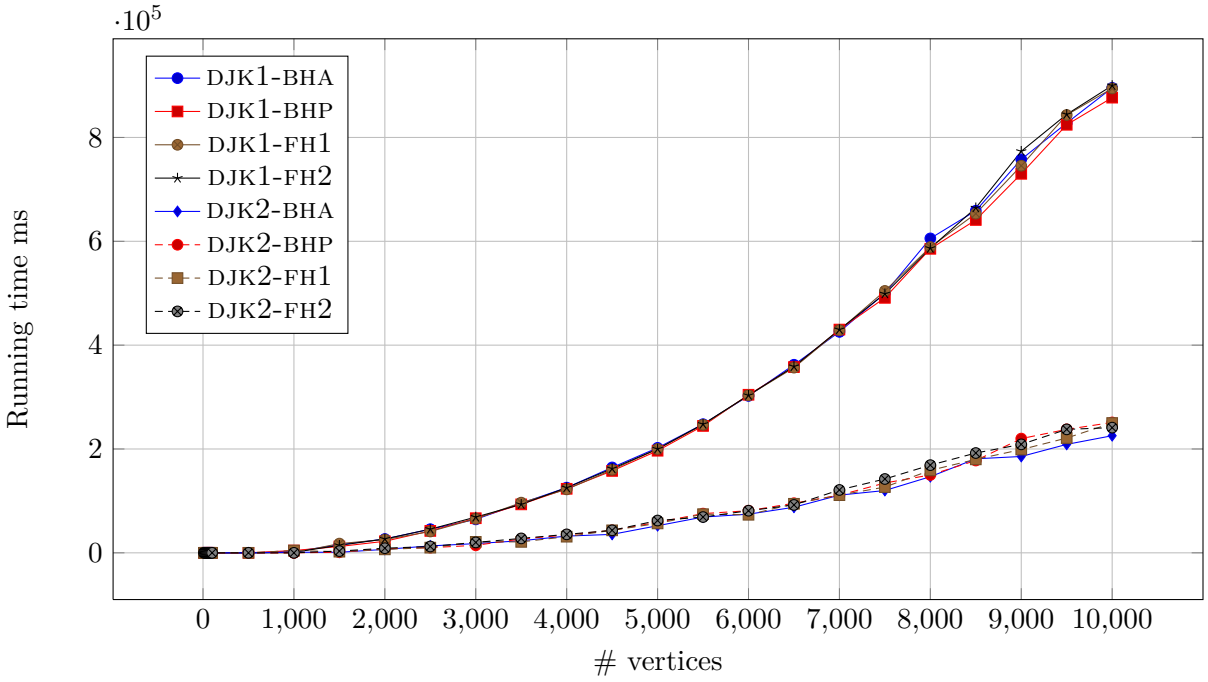


Figure 5.9: Average time of running Dijkstra1 and Dijkstra2 on 20 % connected graph

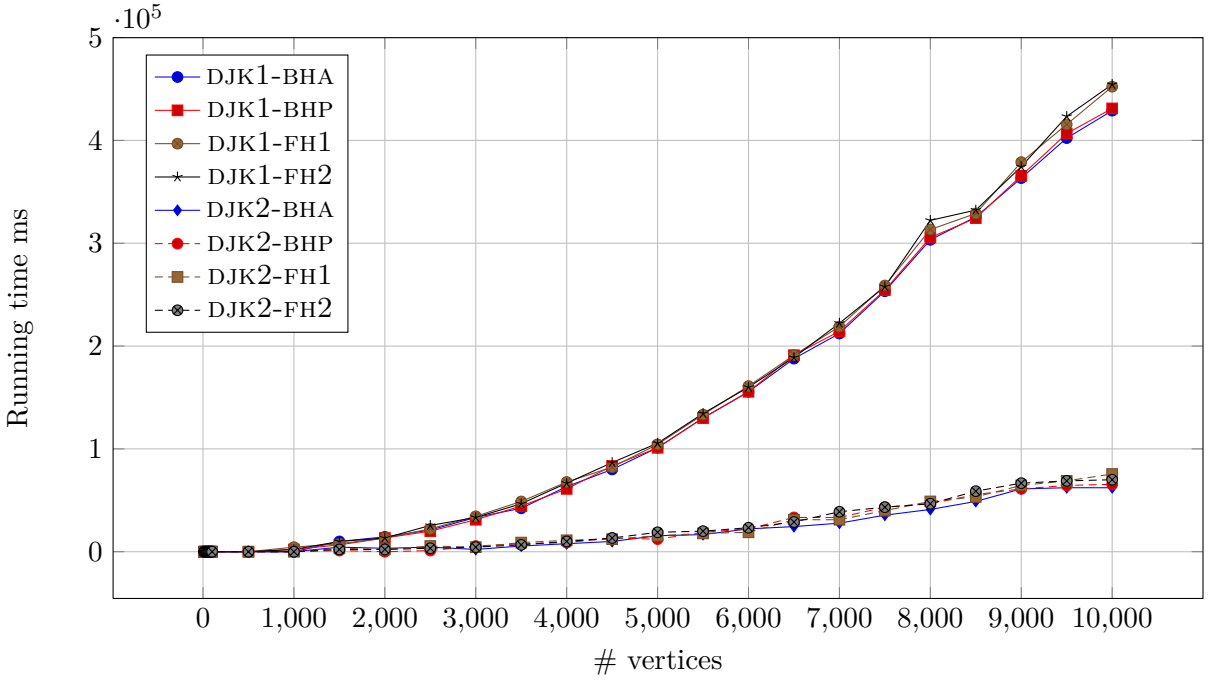


Figure 5.10: Average time of running Dijkstra1 and Dijkstra2 on 10 % connected graph

Chapter 6

Conclusion

The conclusion of the paper is one we draw with mixed feelings. From the bounds analysed in the first few chapters we expected to empirically observe a clear advantage of the Fibonacci family of heaps over the classical binary heaps (there are after all almost 3 decades of research between them), at the least on sparse graphs.

However, our empirical data leads us to the conclusion, that Fibonacci heaps are not worth the implementation effort. That is, faced with the forced choice of implementing a heap structure for any given project, choose the simplest possible structure of the 4 – the classical, array based binary heap – *until* other signs indicate that the choice of heap is a performance bottle neck. In our case study, the implementation of Dijkstra's algorithm proved to be far more influential than the choice of heap.

While not figuring in the above report, we experienced first hand the implementation effort involved in all 4 data structures, and the array based implementation is by far the simplest, simply by virtue of not using pointers, and had a remarkably shorter development time.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT Press, third edition, 2009.

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

[3] J. W. J. Williams. Algorithm 232, heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

Appendix A: Plots of Running Time

A.1: Plots of Stress Test Running Times

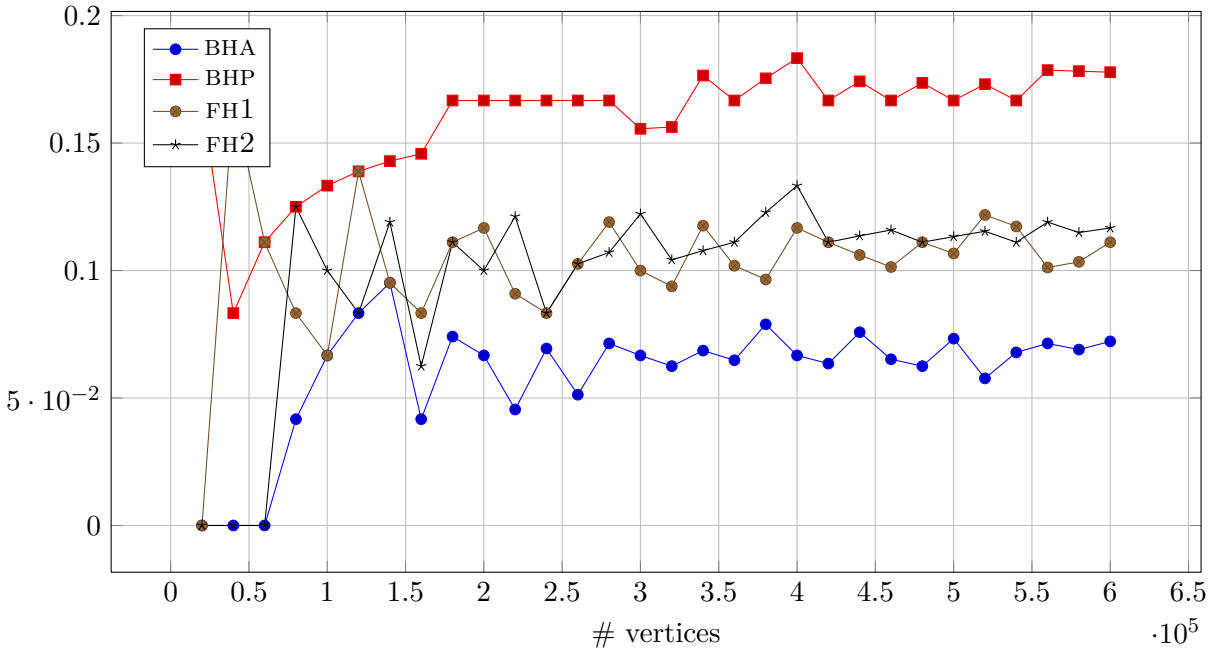


Figure 6.1: Running time divided by n

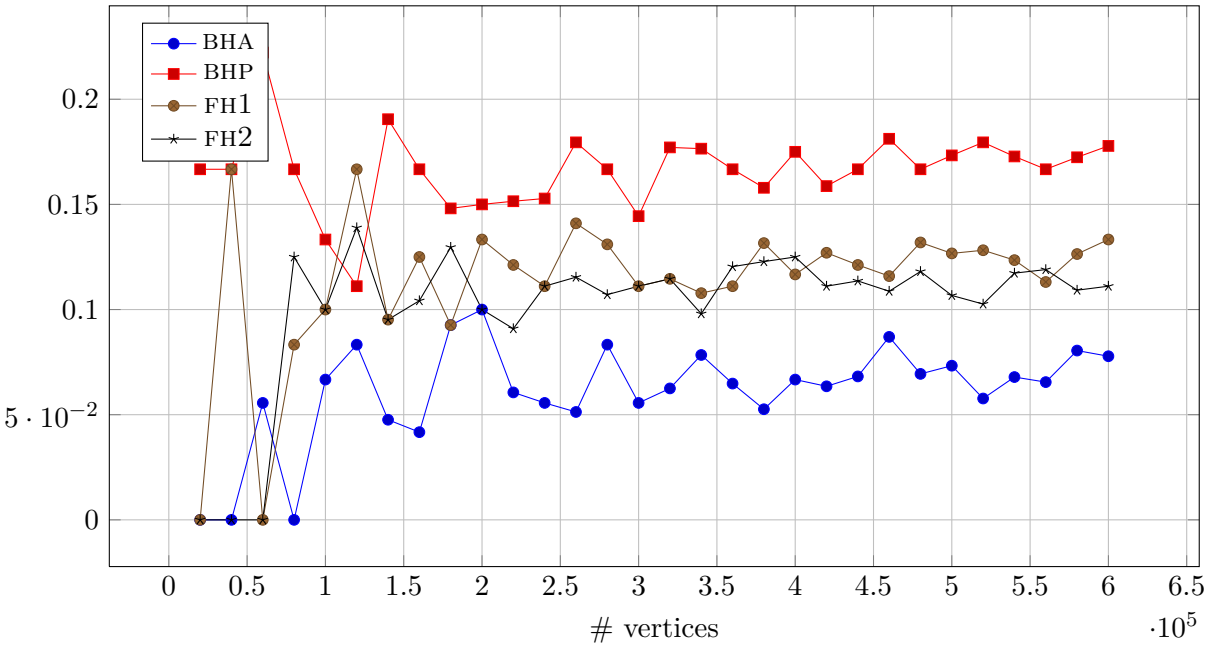


Figure 6.2: Running time divided by n

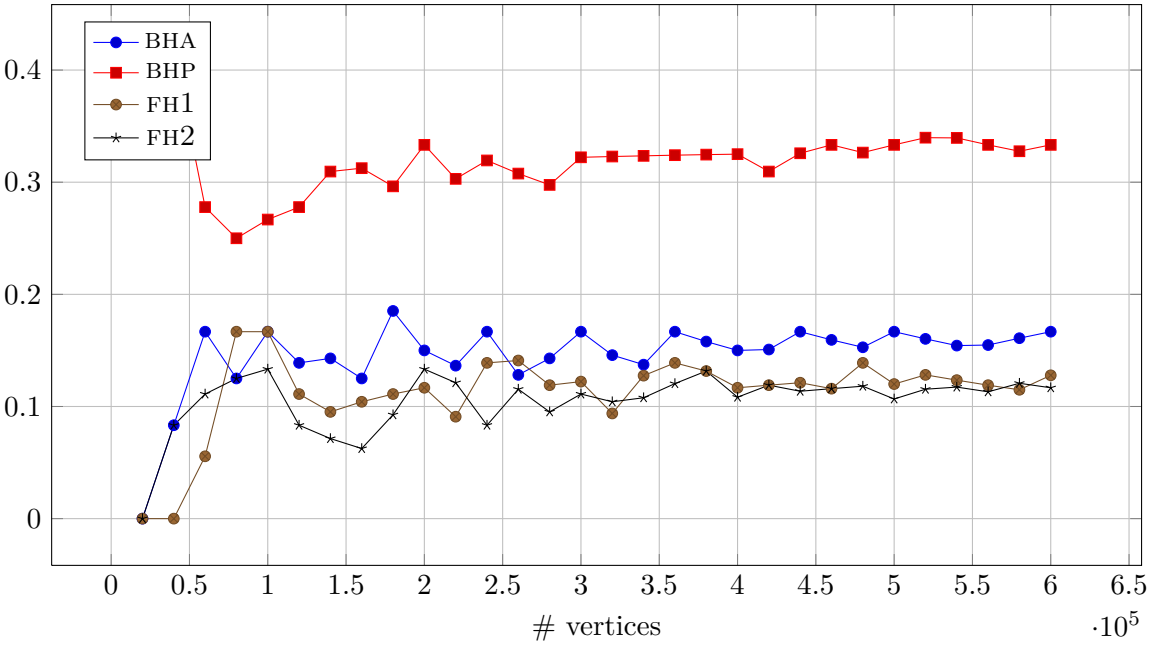


Figure 6.3: Running time divided by n

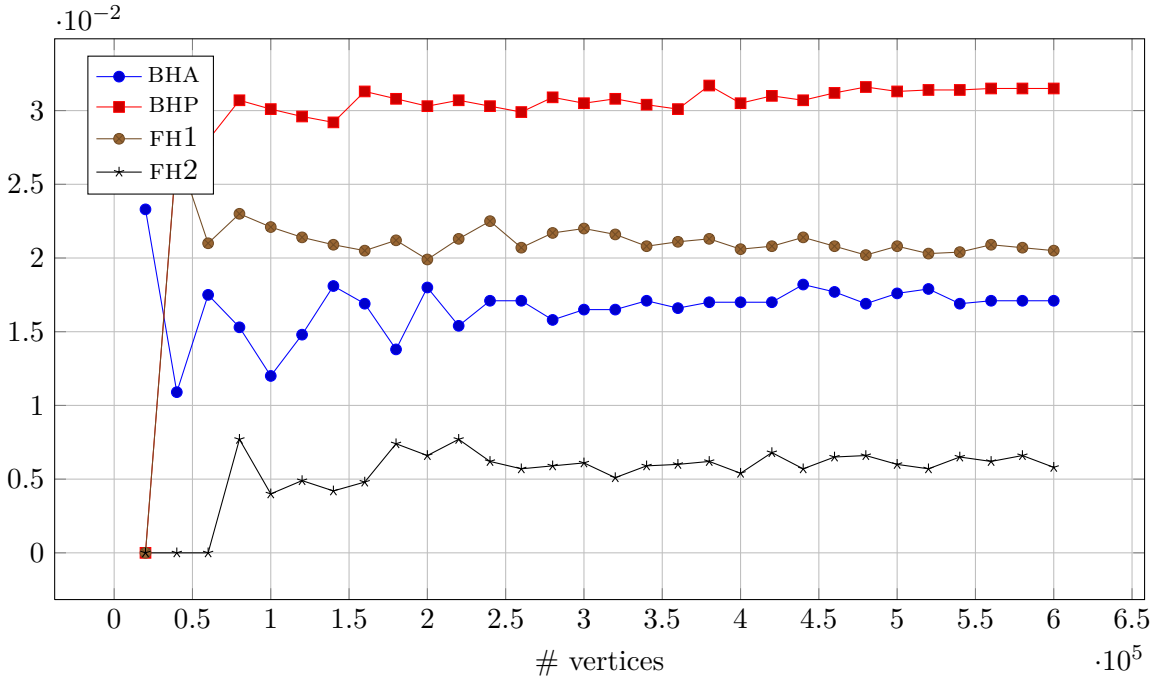


Figure 6.4: Running time divided by *nlogn*

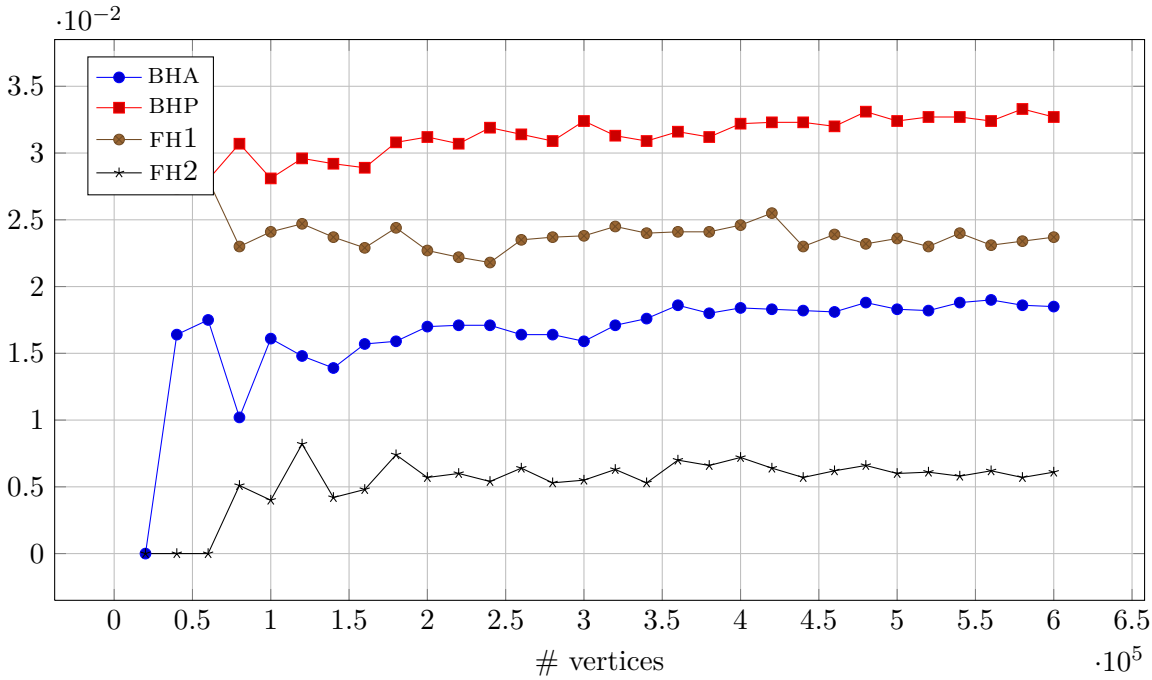


Figure 6.5: Running time divided by *nlogn*

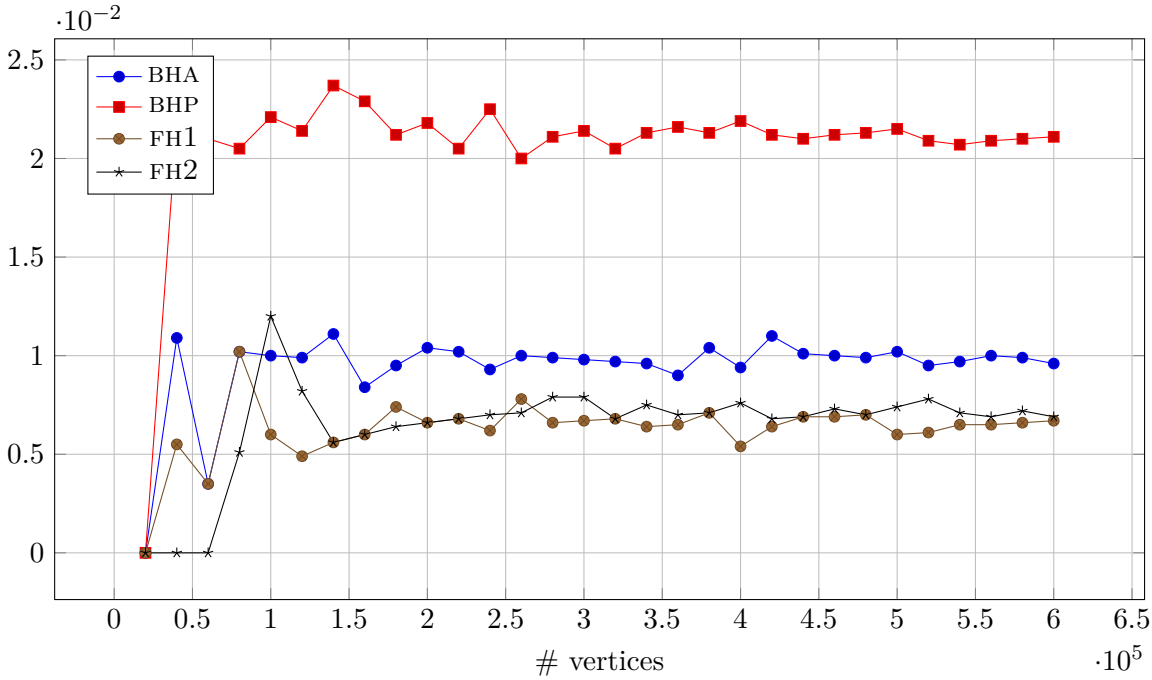


Figure 6.6: Running time divided by *nlogn*

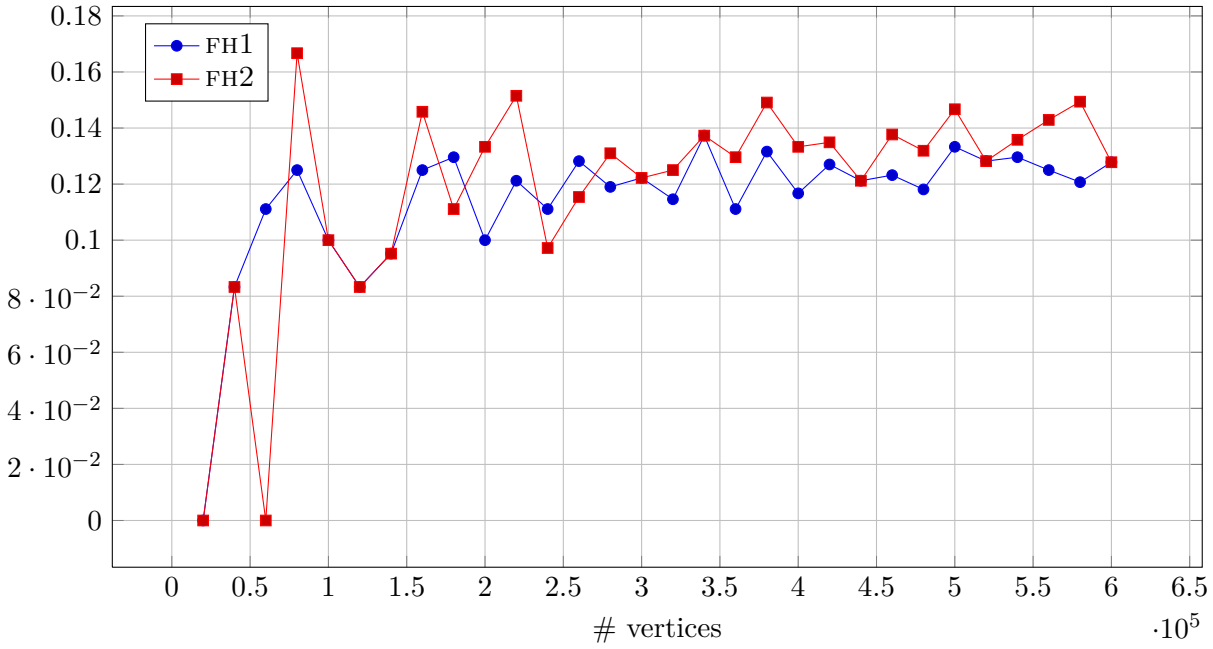


Figure 6.7: Running time divided by n

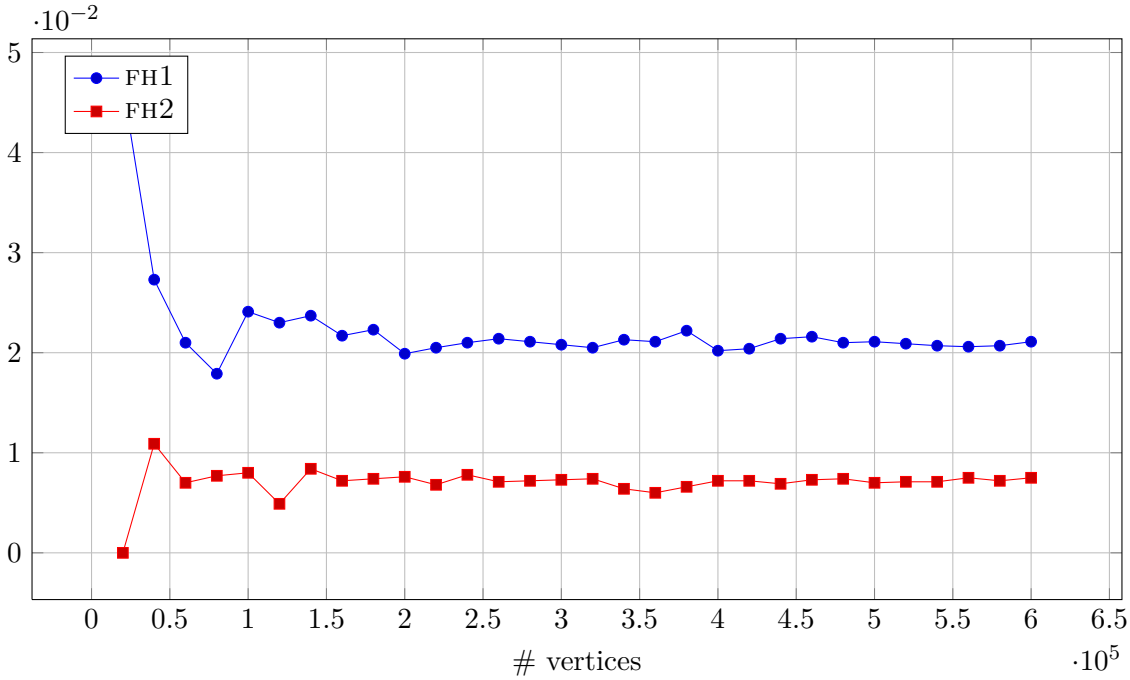


Figure 6.8: Running time divided by *nlogn*

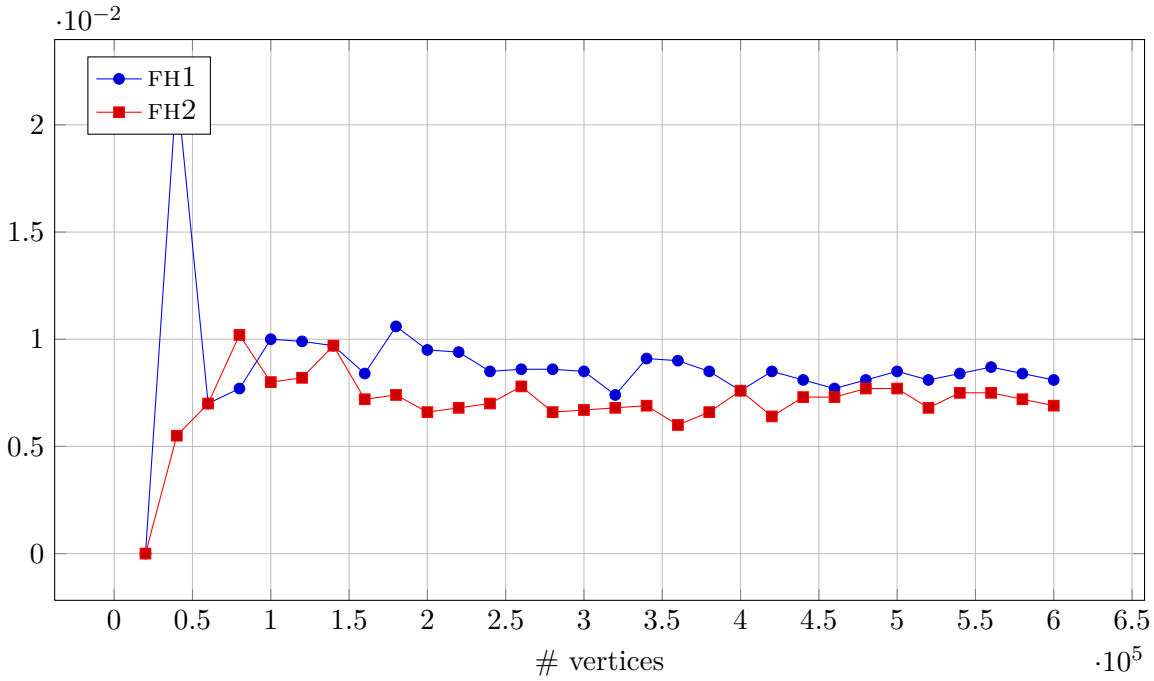


Figure 6.9: Running time divided by *nlogn*

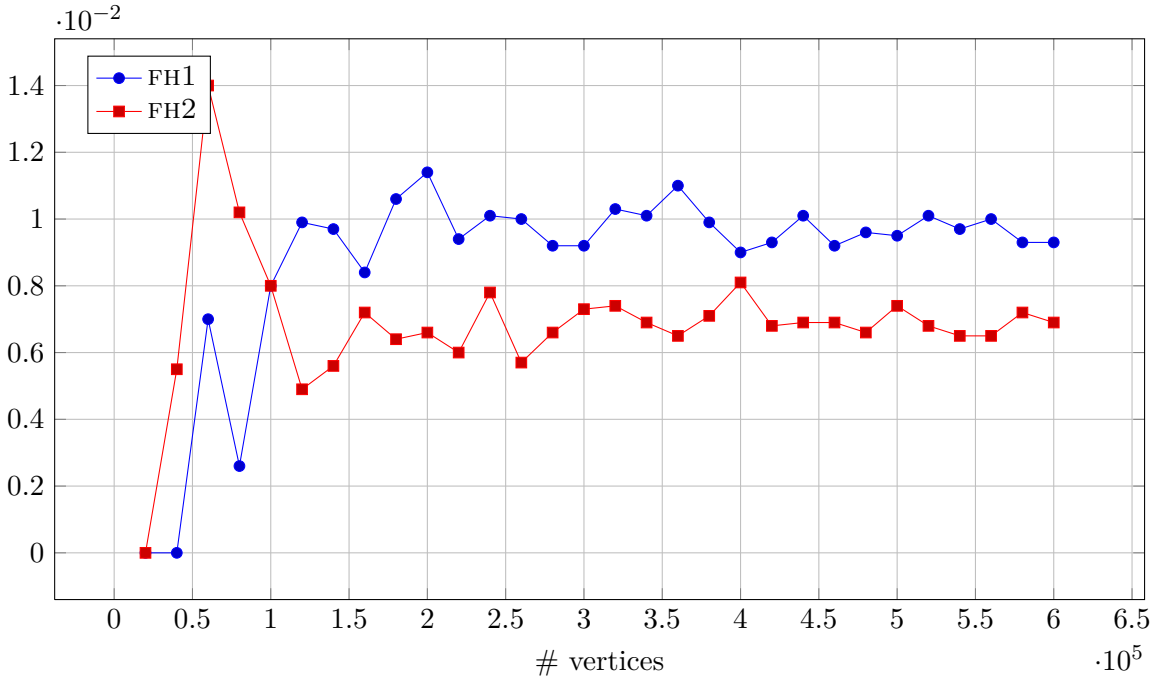


Figure 6.10: Running time divided by *nlogn*

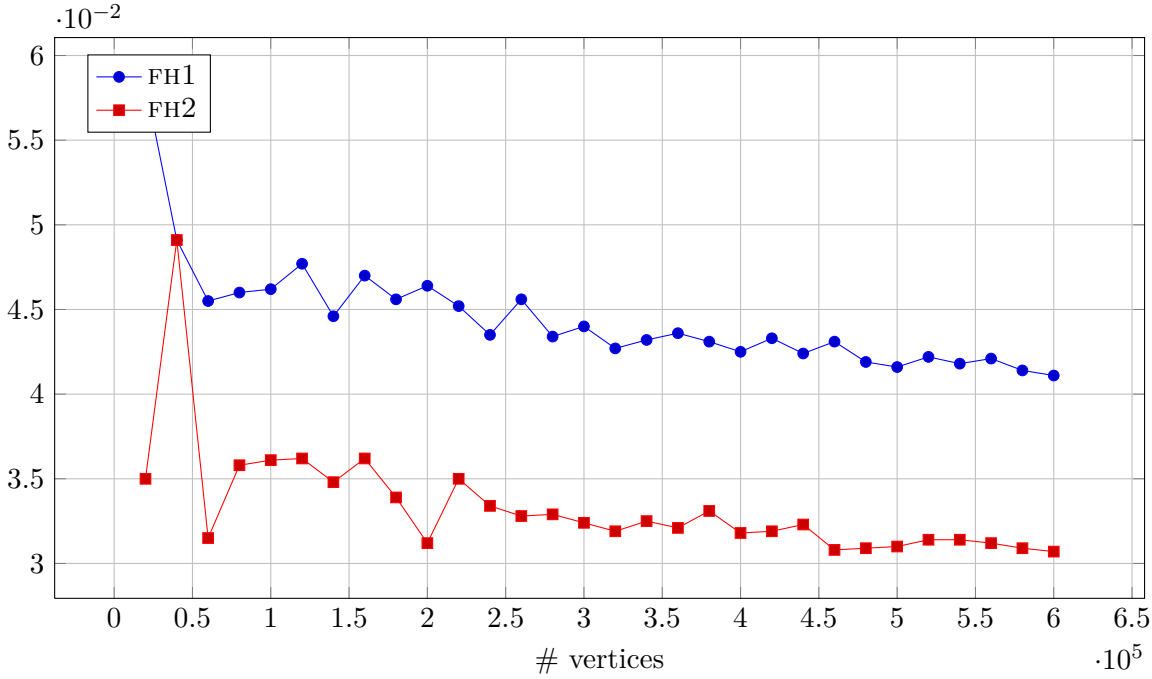


Figure 6.11: Running time divided by *nlogn*

A.2: Plots of Stress Test # of comparisions

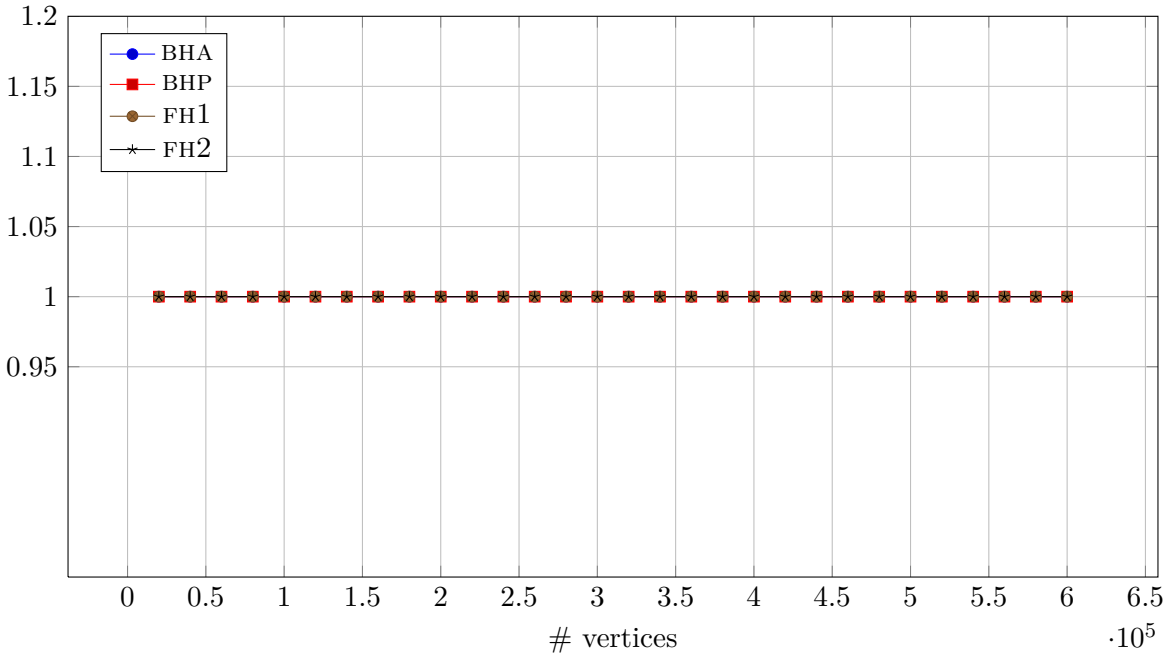


Figure 6.12: # comparisons on inserts divided by test size

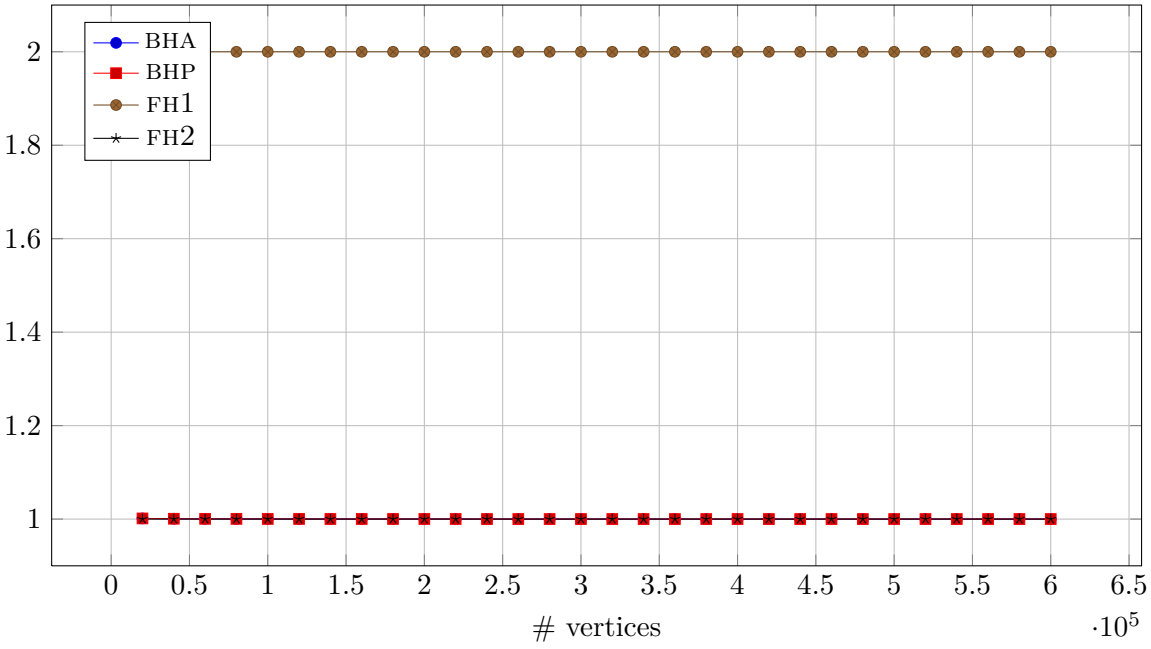


Figure 6.13: # comparisons on inserts divided by test size

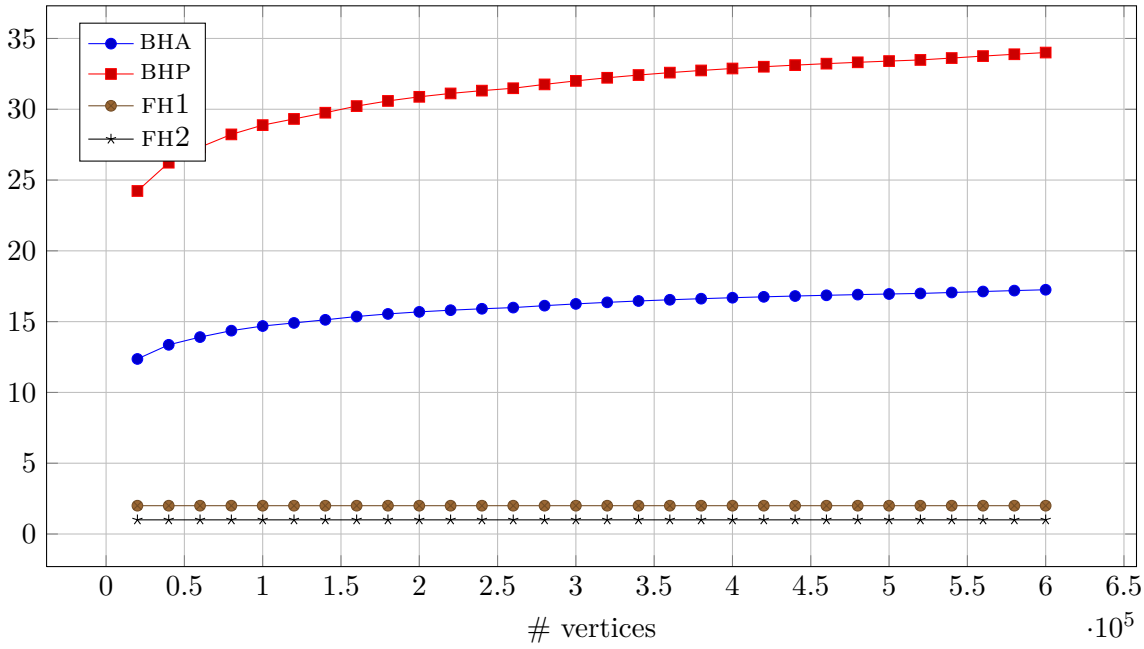


Figure 6.14: # comparisons on inserts divided by test size

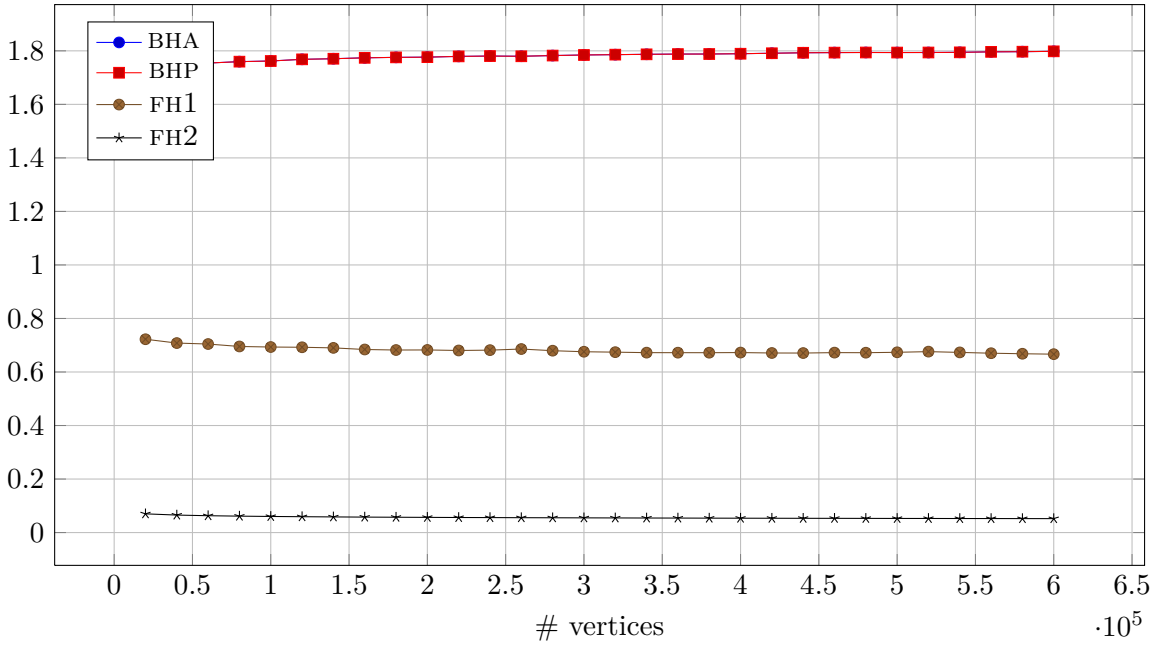


Figure 6.15: # comparisons on inserts divided by *nlogn*

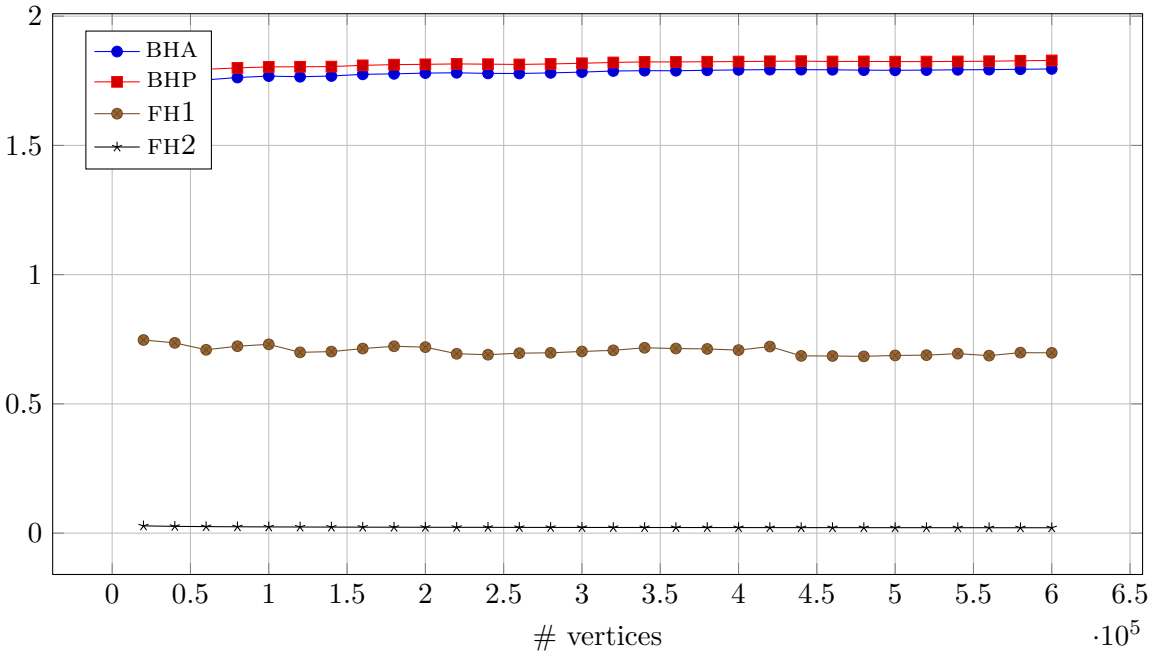


Figure 6.16: # comparisons on inserts divided by *nlogn*

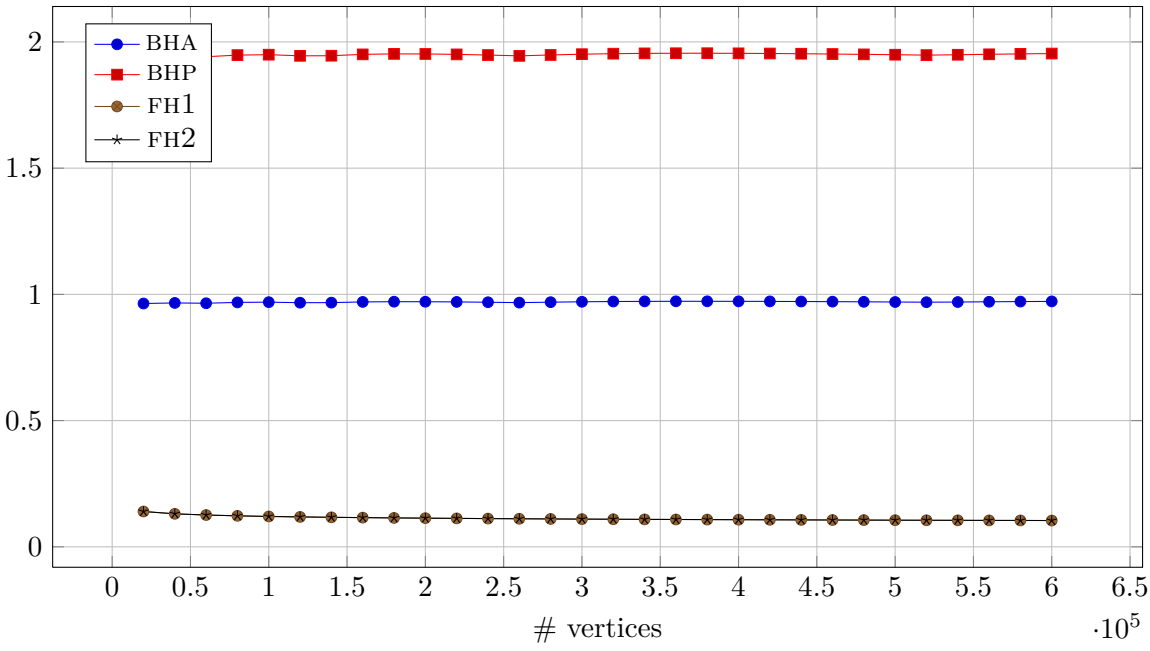


Figure 6.17: # comparisons on inserts divided by *nlogn*

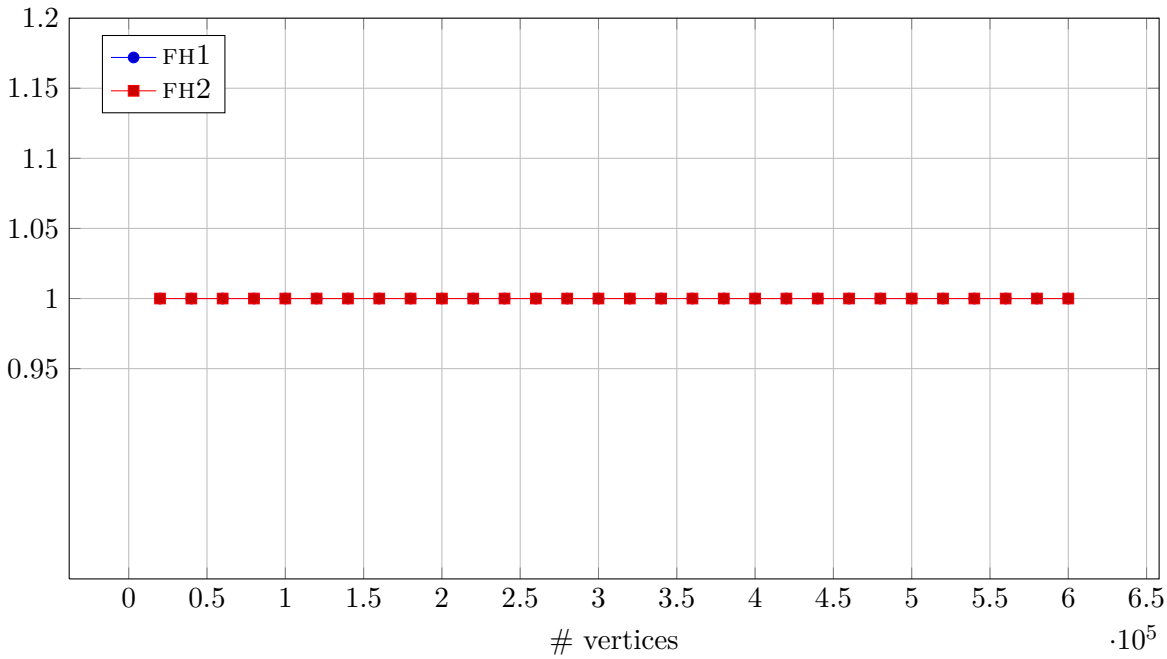


Figure 6.18: # comparisons on inserts divided by n

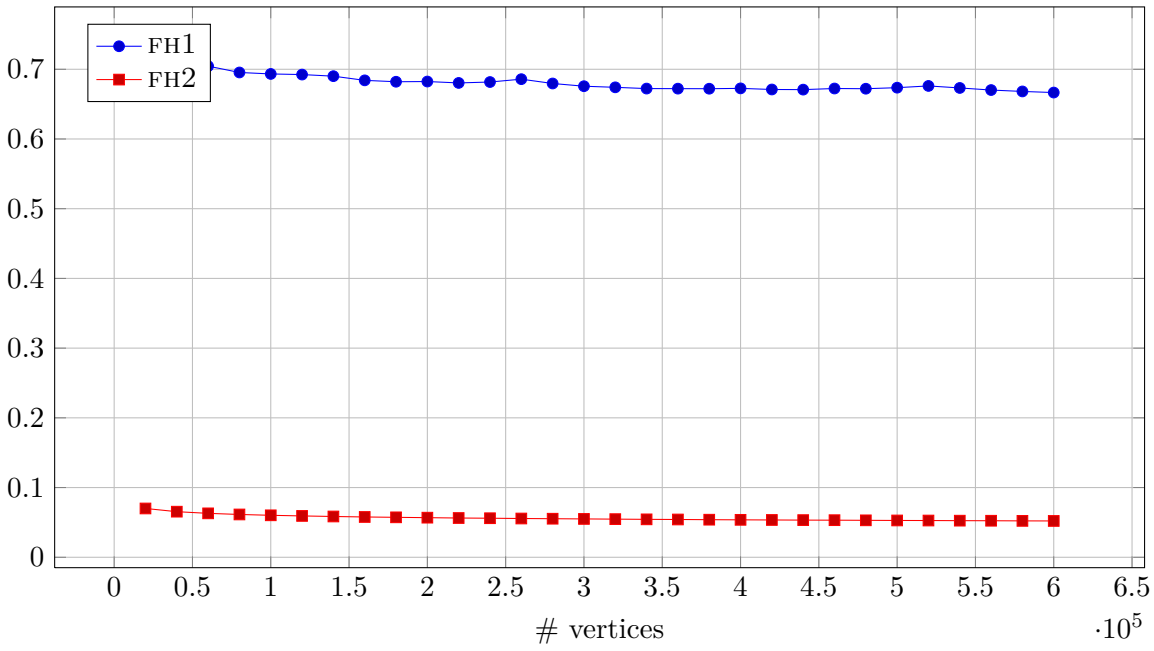


Figure 6.19: # comparisons on inserts divided by $nlogn$

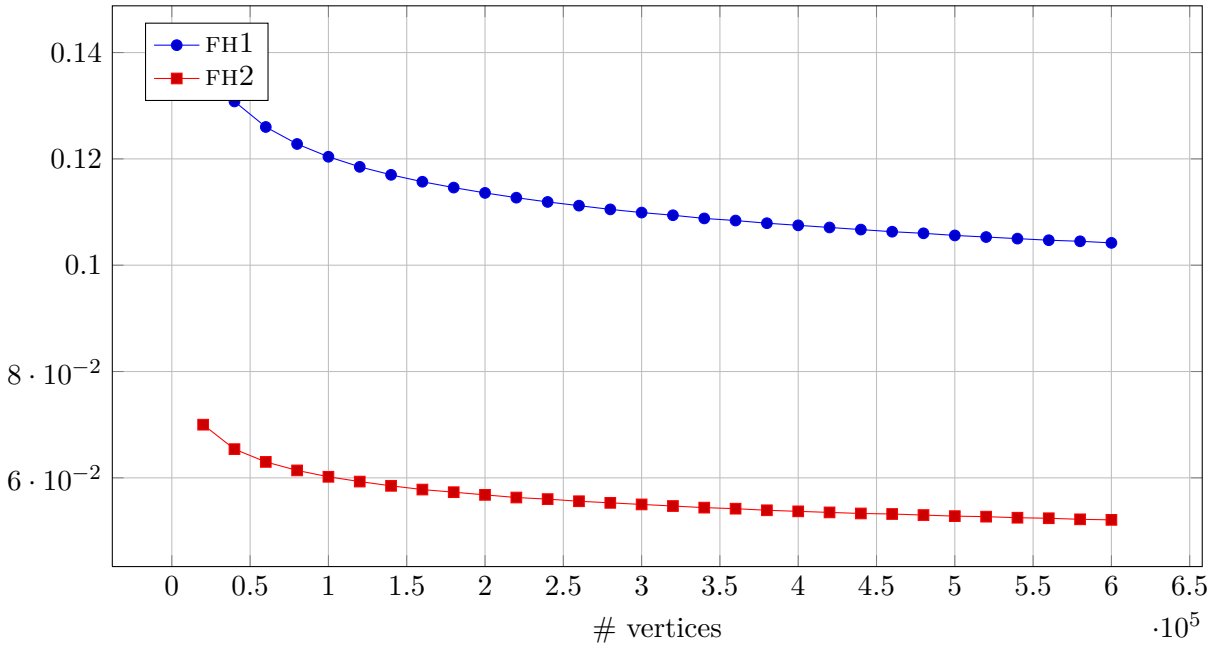


Figure 6.20: # comparisons on inserts divided by *nlogn*

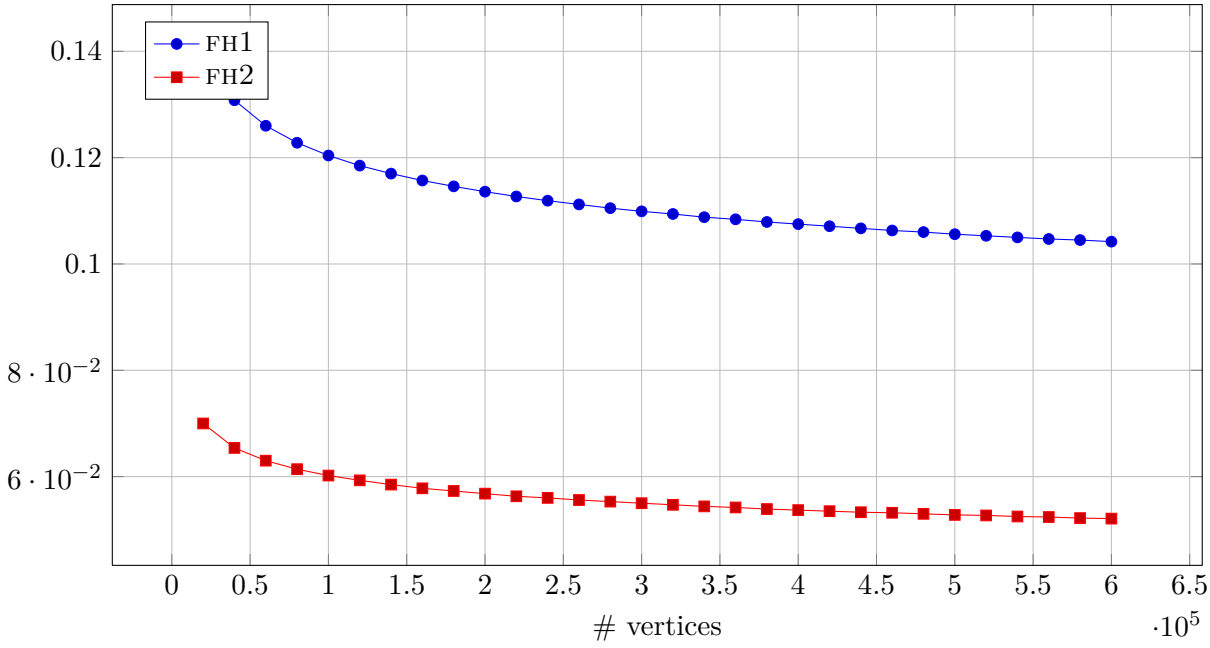


Figure 6.21: # comparisons on inserts divided by *nlogn*

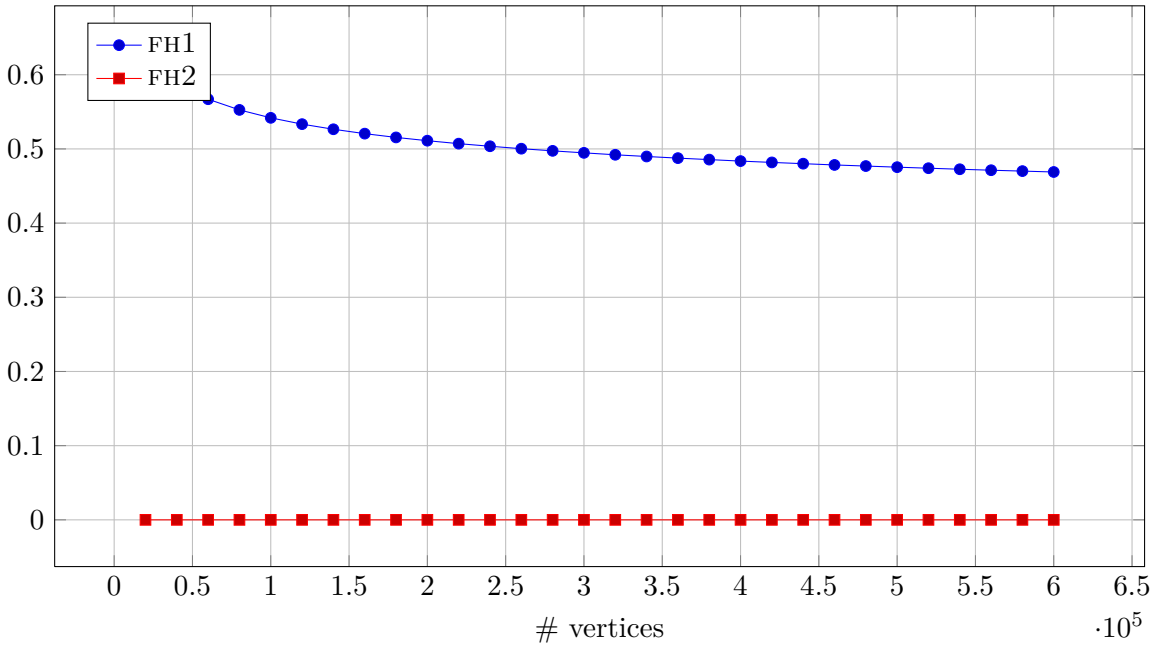


Figure 6.22: # comparisons on inserts divided by *nlogn*

A.3: Plots of Running Times divided by Big-Oh times

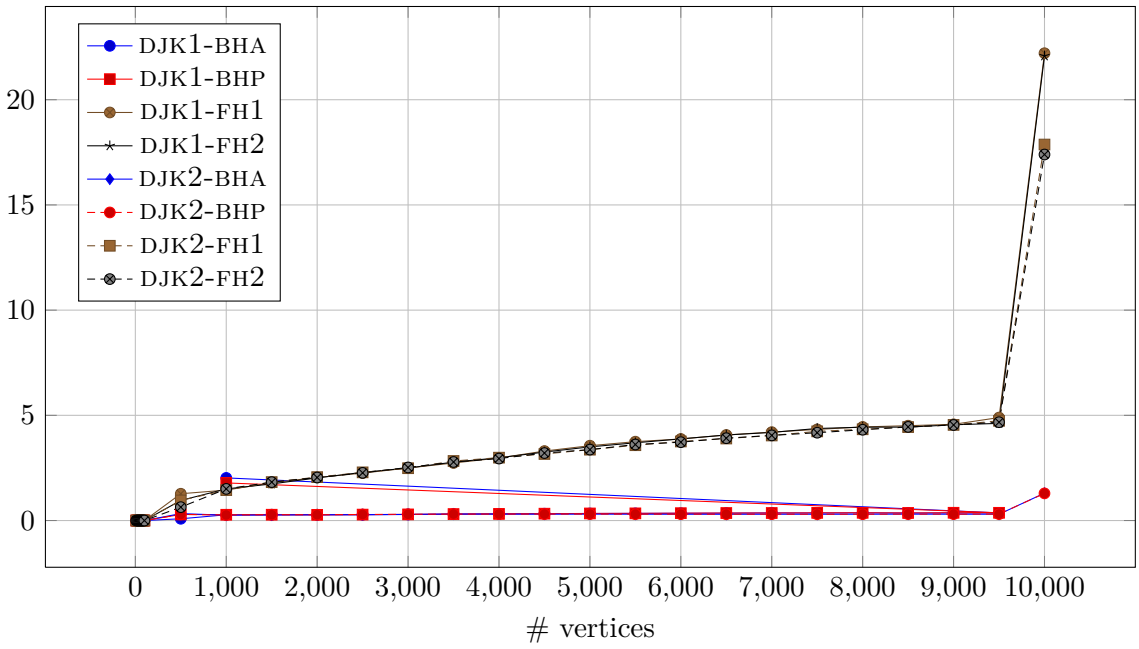


Figure 6.23: Average time divided Big-Oh response of running DIJKSTRA1 and DIJKSTRA2, 100% connected

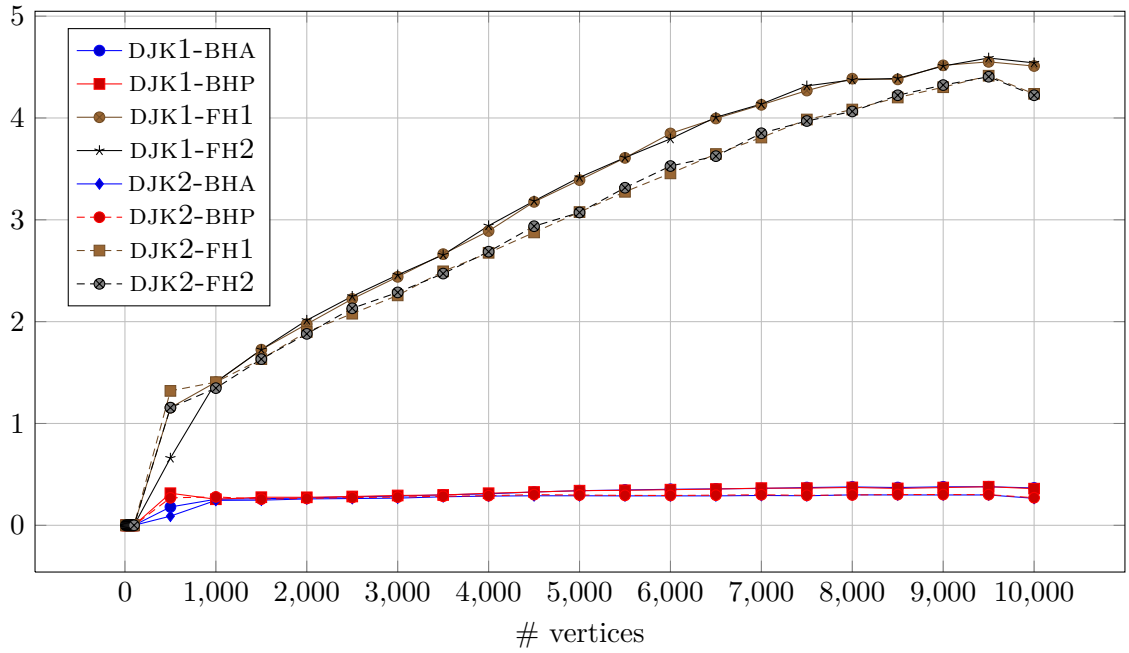


Figure 6.24: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 90% connected.

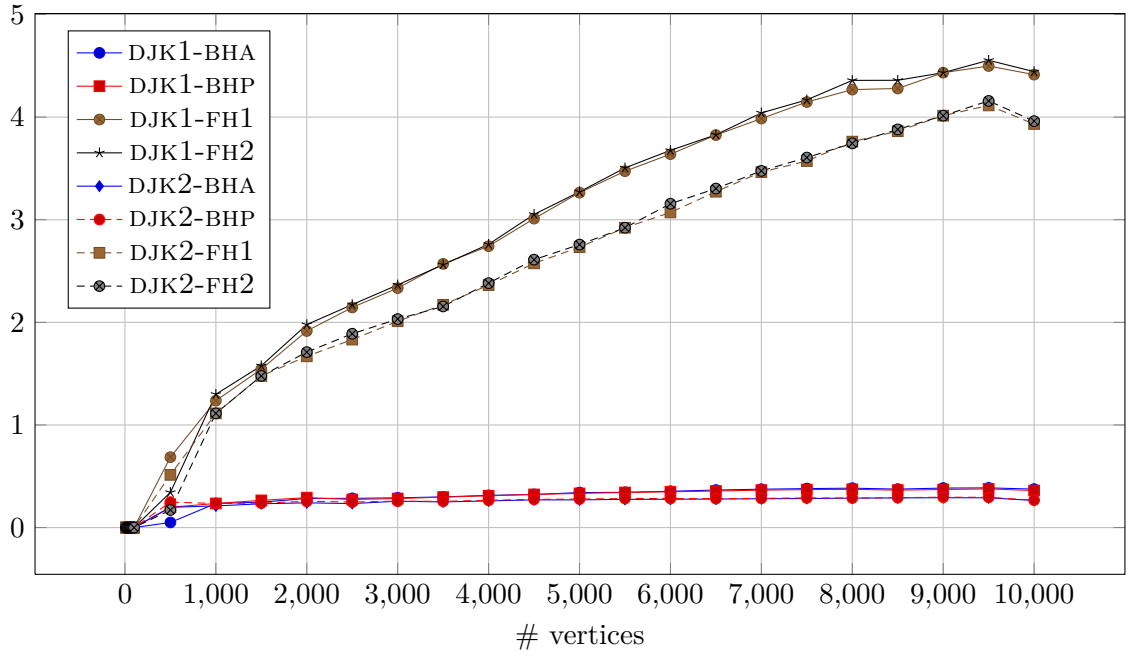


Figure 6.25: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 80% connected.

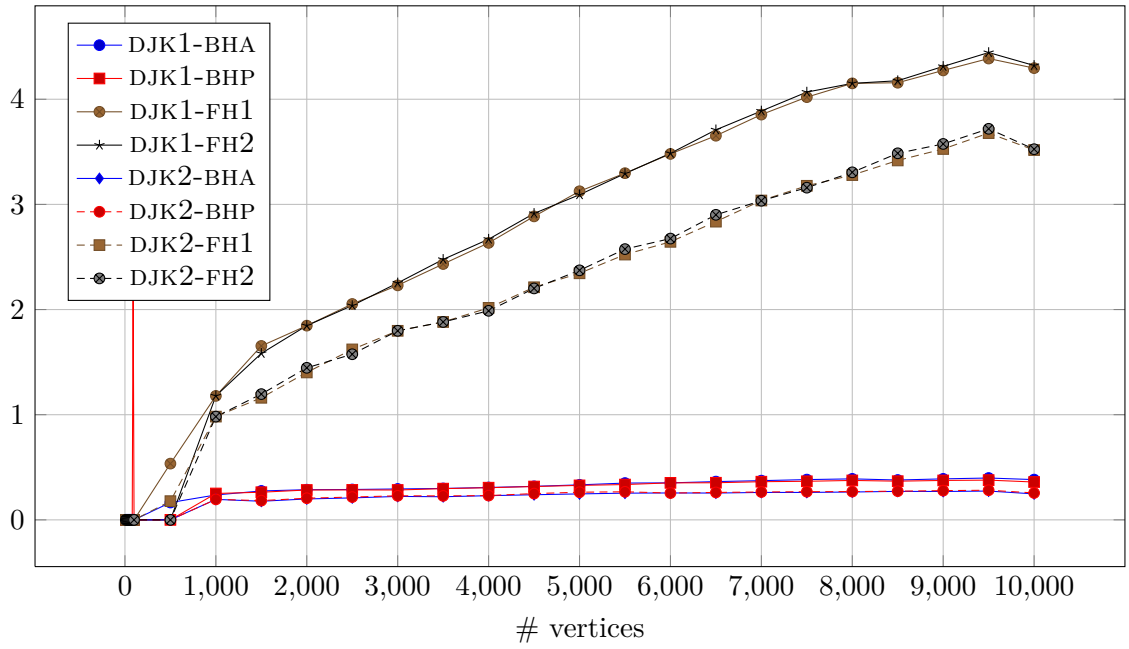


Figure 6.26: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 70% connected.

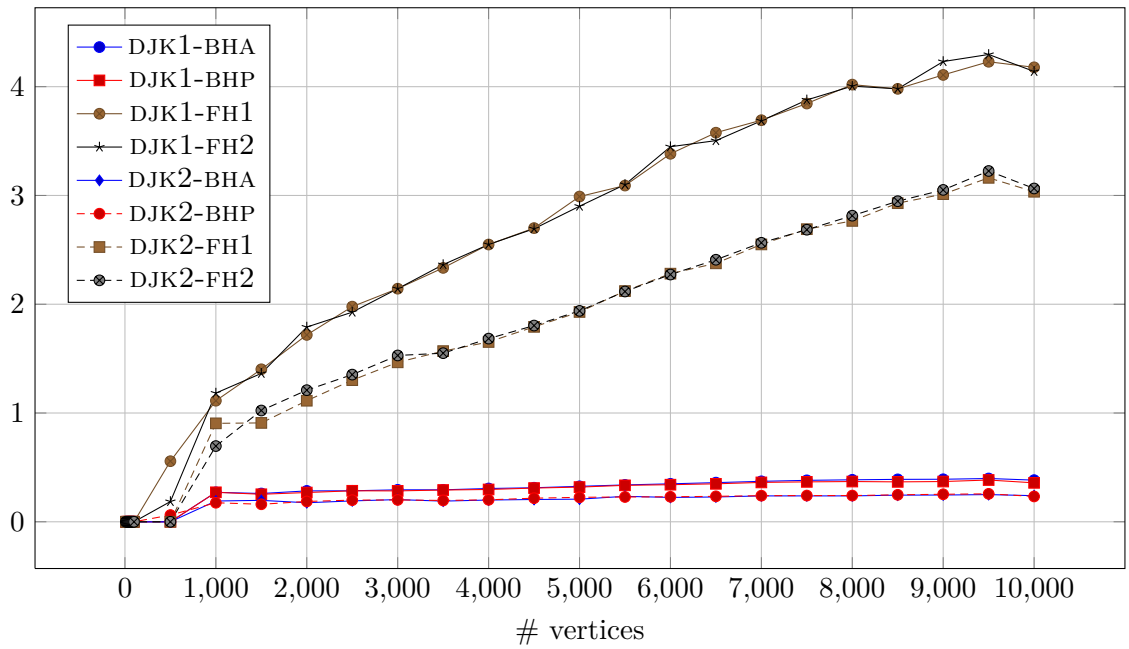


Figure 6.27: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 60% connected.

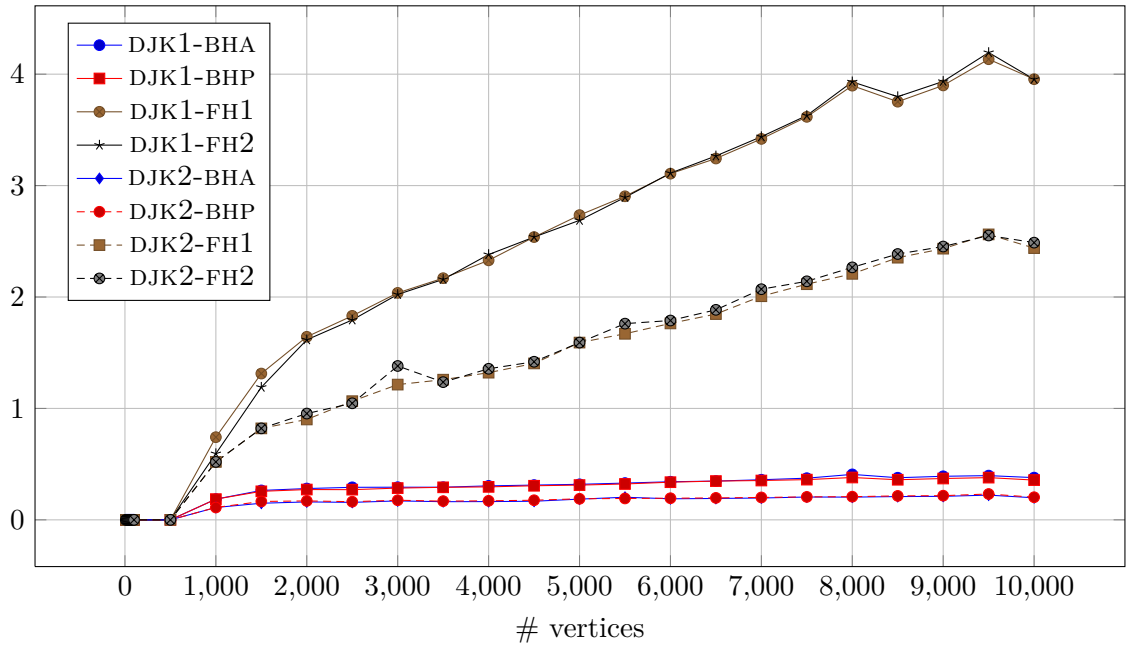


Figure 6.28: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 50% connected.

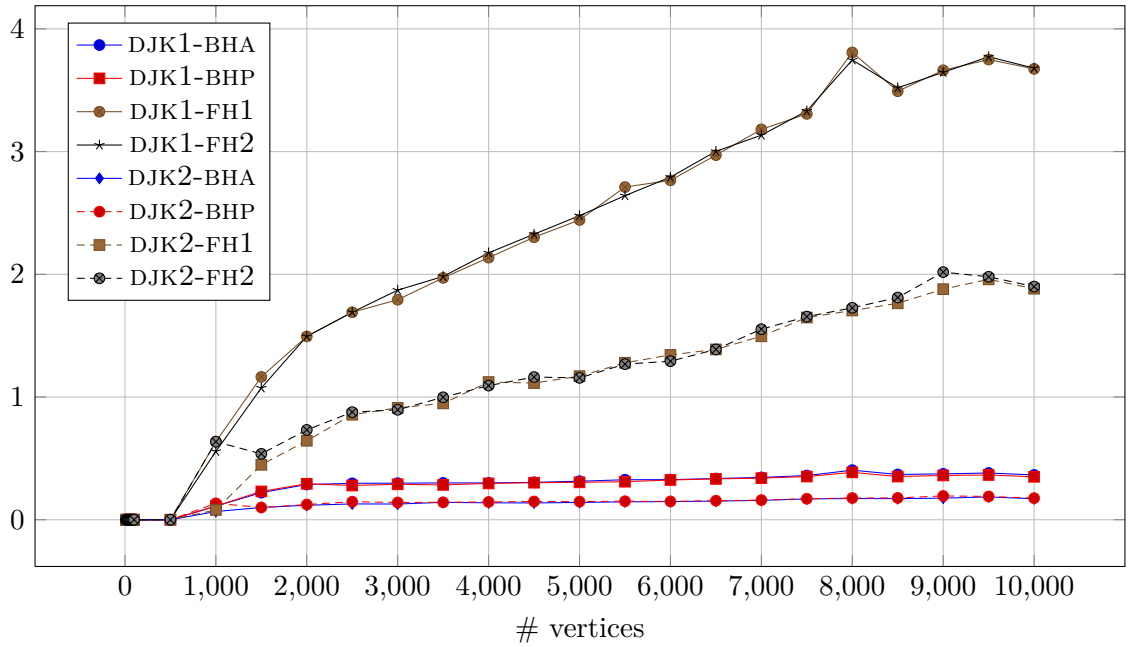


Figure 6.29: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 40% connected.

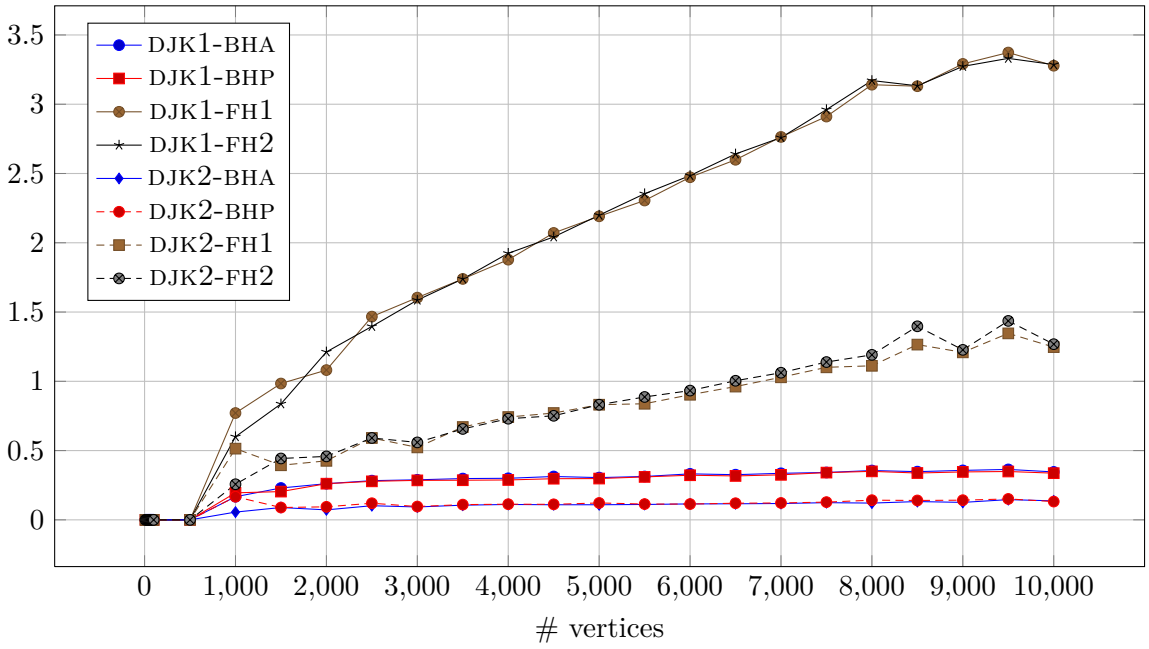


Figure 6.30: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 30% connected.

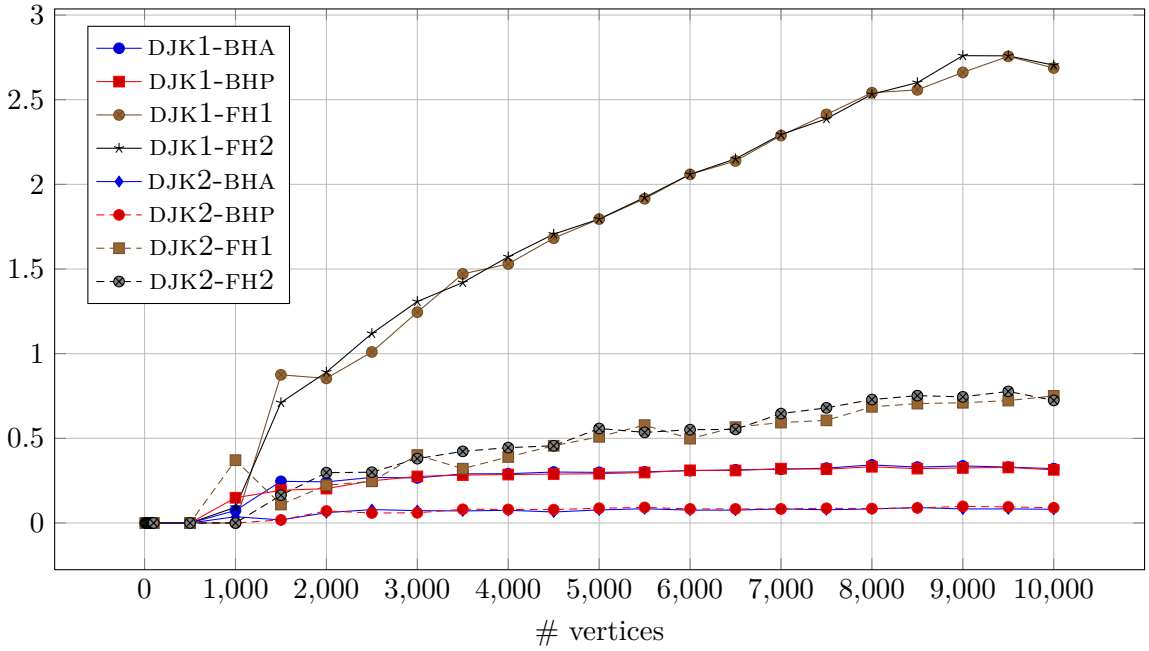


Figure 6.31: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 20% connected.

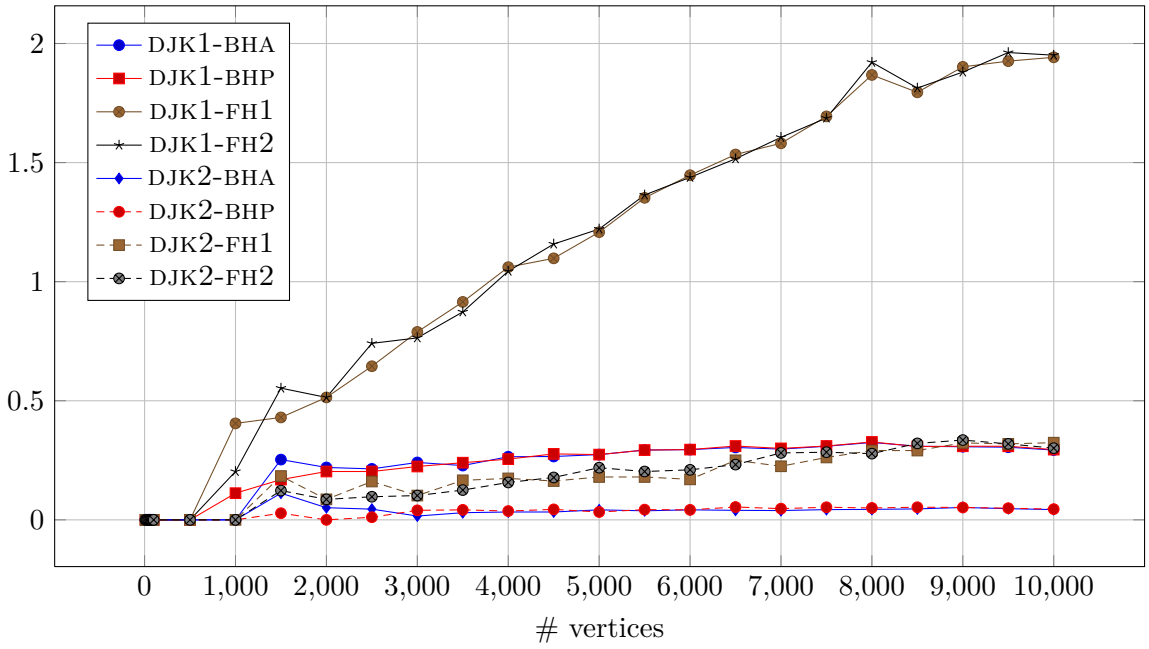


Figure 6.32: Average time divided by Big-Oh response of running Dijkstra1 and Dijkstra2, 10% connected.