Binary Heaps, Fibonacci Heaps and Dijkstras shortest path

Kristoffer Just Andersen, 20051234

Troels Leth Jensen, 20051234

Morten Krogh-Jespersen, 20022362

Project 1, Advanced Data Structures 2013, Computer Science
February 2014

Advisor: Gerth StÃ¸lting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Contents

Chapter 1

Introduction

needs content

Chapter 2

Binary heaps

needs content

2.1 Binary heap with array

needs content

2.2 Implementing decrease key

needs content

2.3 Time-complexity for binary heap with array

2.4 Binary heap with pointers

needs content

2.5 Implementing decrease key

needs content

2.6 Time-complexity for binary heap with pointers

2.7 Testing correctness of Binary Heaps

needs content

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	$\Theta(1)$	$\Theta(1)$	$O\left(l(\lg(\frac{n}{l})+1)\right)$
Insert	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$
DeleteMin	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
DecreaseKey	$\Theta(\lg n)$	$O(1)$	$O(1)$
Delete	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
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 $\mathrm{O}(\lg n)$ [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 childs, 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 analyse 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 relased 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 perfomance, 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 beforementioned node. Since the running time of the DELETE operation is amortized $\mathrm{O}(\lg n)$ we chose a simpler version where we just update all parent pointers one by one. As stated above $size(node) \leq \mathrm{O}(\lg n)$ so this only takes $\Theta(\lg n)$ 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 concatinating 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 executues 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:

$$\begin{aligned}\textsc{DeleteMin} &= \mathrm{O}\left(D(n) + trees(H)\right) + \left((D(n)+1) + 2marked(H)\right) \\ &\quad - \left(trees(H) + 2marked(H)\right) \\ &= \mathrm{O}\left(D(n)\right) + \mathrm{O}\left(trees(H)\right) - trees(H) \\ &= \mathrm{O}\left(D(n)\right)\end{aligned}$$

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

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 $\mathrm{O}\left(c\right)$. The change is potential is:

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

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

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 $\mathrm{O}\left(D(n) + c\right)$. The change in potential is as follows:

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

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 $\mathrm{O}\left(D(n)\right) \leq \mathrm{O}\left(\lg n\right)$.

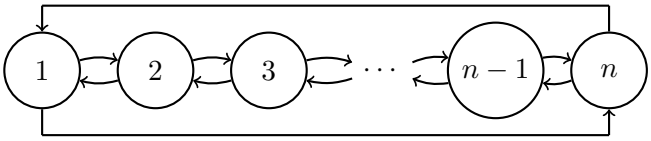
3.4 Worst case time-complexity for Fib heap v1

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 $\mathrm{O}\left(D(n) + trees(H)\right)$. 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.

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

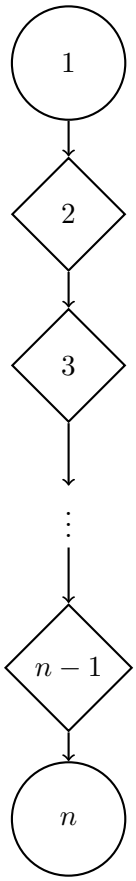
which is $\Theta\left(n\right)$.

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:

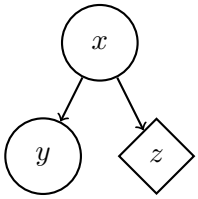


where a diamond is modelling a marked node. If either DecreaseKey or Delete is called with an item corresponding to node n a cascading delete will begin and will not stop until it reaches 2 in this example. The amount of operations is therefore the entire chain:

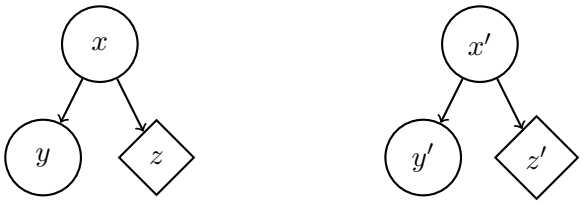
$$\text{length of chain} = n - 1$$

which is $\Theta(n)$.

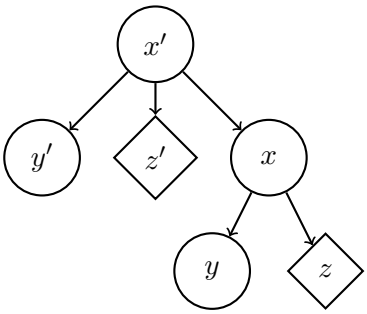
Such a configuration can be obtained by constructring triangles, chaining and deleting (as a side note, this is exercise 19.4-1 in [1]). In order for us to create the first triangle, we insert five nodes, call DeleteMin and then remove the bottom most:



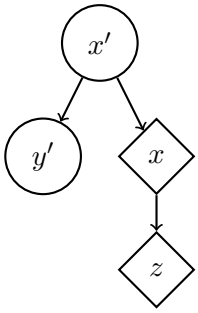
We can then create another triangle, having x' being a lesser key than x. The picture below is just before joining the two rank two trees.



The result is then:



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



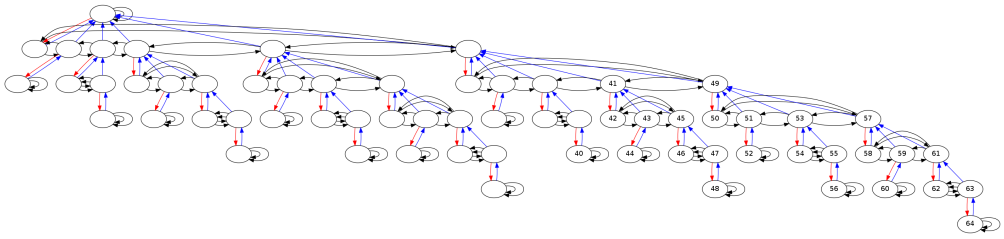
To create a chain of height n, we have to create n triangles.

3.5 Fibonacci heap version 2

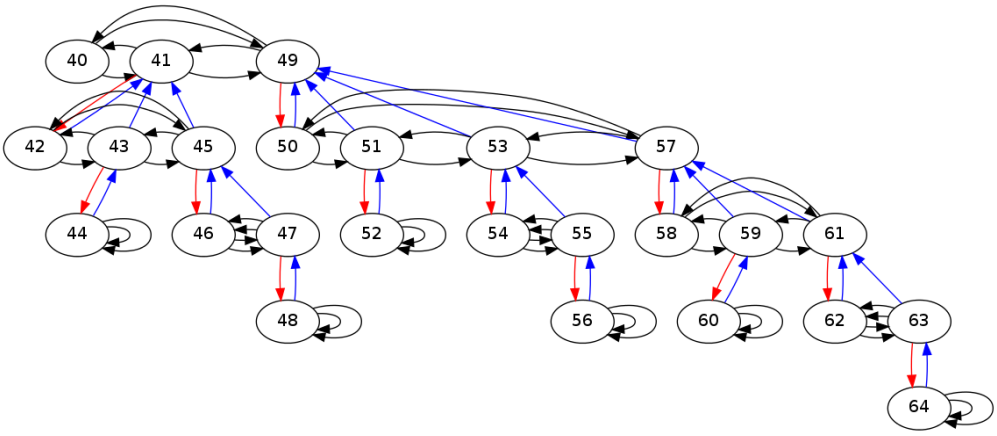
In this second version of a Fibonacci heap we try to be more lazy, by doing as little work as possible for DELETE and DELETEMIN. When a call to either one of the operations is called, we mark the node as vacant but then halt to do any additional work. This makes DELETE and DELETEMIN run in $\mathrm{O}(1)$.

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-framwork showing how a tree looks like with vacant nodes:



After a call to FindMin thee heap looks like:



Let us analyze the time for the FindMin operation. Let l mark the number of vacant nodes. If $l = 0$ then FindMin points to the node representing the item with the minimum key, and the work is constant. If $l \geq 1$ then the amortized running time is the new roots created from destroying vacant nodes and moving the non-vacant child nodes to the root plus the linking of the trees which is at most $\mathrm{O}(\lg n)$. We argued previously that a node y have $degree(y_i) \geq i - 2$ for $i = 1, 2, \cdots, k$ where k is the degree of its parent. We use this to say that at least one of the vacant nodes will have a rank of $k - 2$. We also know that the size of subtree rootet at a node with degree $k-2$ will be at least be of size ϕ^{k-2}. If all children of the vacant nodes we destroy are nonvacant, they will increase the number of trees and thereby the potential by:

$$\sum_{i=1}^{l} \phi^{k_j - 2} \leq n$$

where $k_1, k_2, \cdots, k_i$ are nonvacant nodes. The sum is maximized for k_i. MAYBE SOME MORE??

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 proces 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 companying the report are generated images we have used to check correctness.

Lastly, since we now have four datastructures, 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

Test-results

needs content

Chapter 5

Dijkstra's algorithm

Dijkstra's algorithm is an graph search algorithm that computes 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 summery of the running time and space complexities:

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

We represent graphs with an 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.

5.1 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 readded 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 $\mathrm{O}((n+m)\lg n)$. For the Dijkstra version with Inserts only, the m previus 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 $O(\lg n)$ resulting in the time-complexity $O(m + n \lg n)$ for Dijkstra with Fibonacci heap version 1. Because the time-complexity for Insert is also $O(1)$, 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 $O(\lg n + 1)$ which results in the same running times for Dijkstra as Fibonacci heap version 1.

5.2 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 $m = o(n^2/\lg n)$:

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

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

For Fibonacci heaps the time $O(n \lg n + m)$ suggest, since m dominates $n \lg n$, 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 stricly sparser since the running time in O-notation for Fibonacci heaps hide away a very large constant.

Because connectivety 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 probality on the number of edges in the graph.

5.3 Testing time-complexities for Dijkstra

Because we have four different heaps and two different implementations of Dijkstra we have a lot of test to perform. We are generating graphs in the interval from 100 to 20.000 and increments of 100 nodes. A graph with 20.000 nodes represented by an adjacency matrix where all indicies are int is by our calculation around 1.5 GB, which should give us enough information about running times for large graphs.

For each level in our node-size interval we have 10 different graph connectivity-probabilities, 10 % to 100 % in steps of 10 %, where we do following three times:

generate a random graph with beforementioned probability and size and run all heaps and dijkstra algorithms on this instance three times to find an average. Thus:

$$\#\text{ of tests} = 200 * 10 * 3 * 8 * 3 = 144.000$$

Chapter 6

Binary heap vs Fibonacci heap

needs content

Chapter 7

Test-results

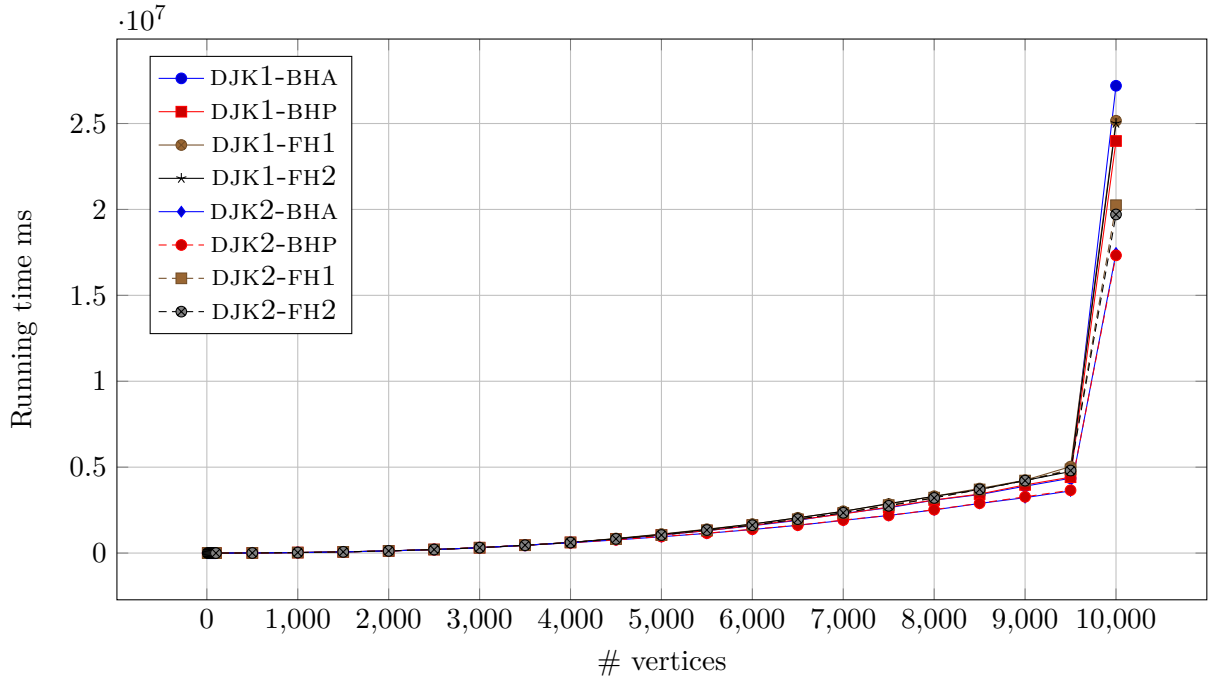


Figure 7.1: Average time of running Dijkstra1 and Dijkstra2 on 100 % connected graph

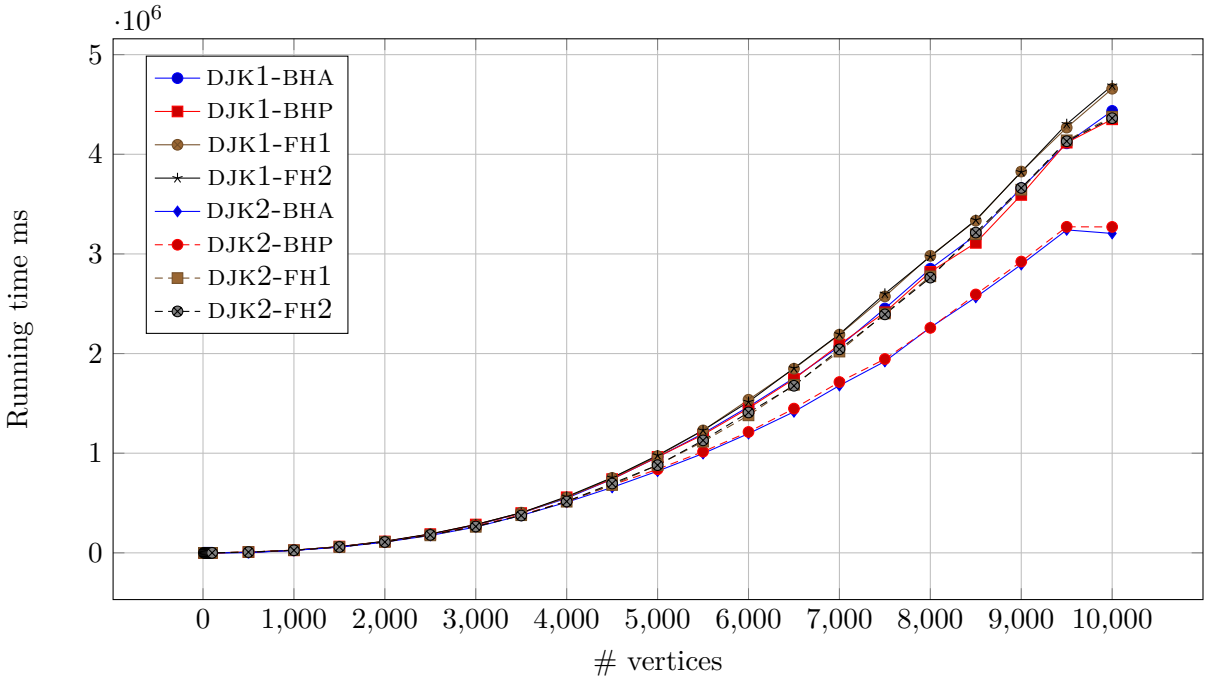


Figure 7.2: Average time of running DIJKSTRA1 and DIJKSTRA2 on 90 % connected graph

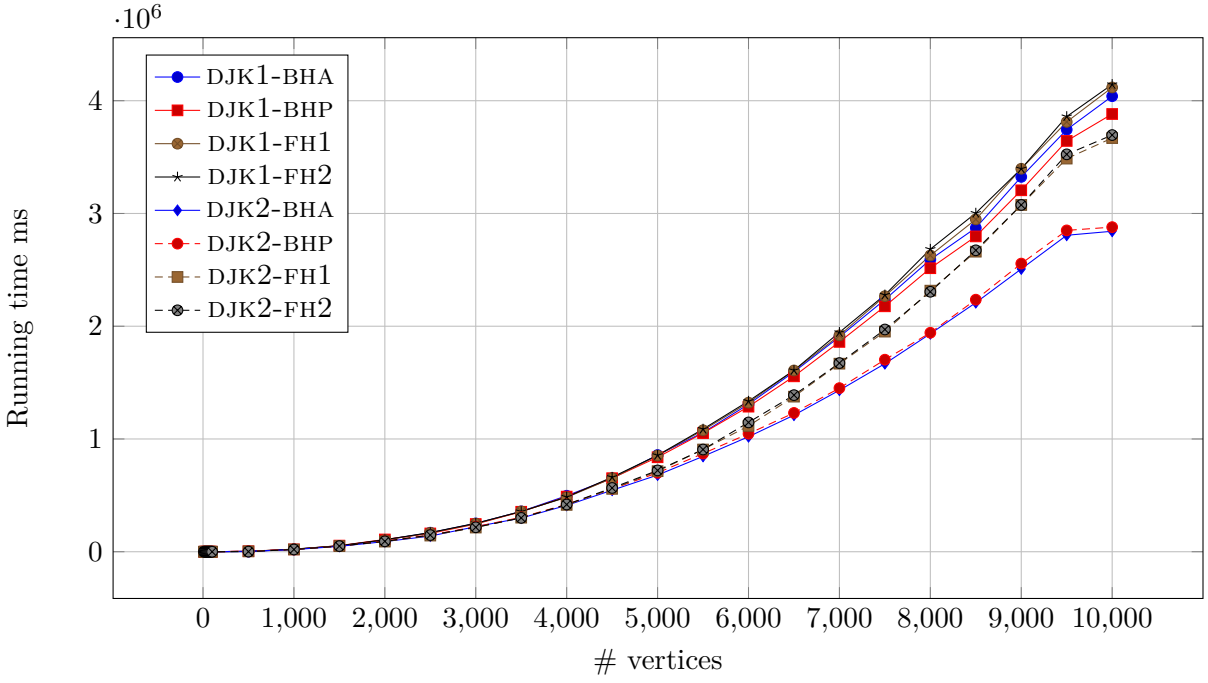


Figure 7.3: Average time of running DIJKSTRA1 and DIJKSTRA2 on 80 % connected graph

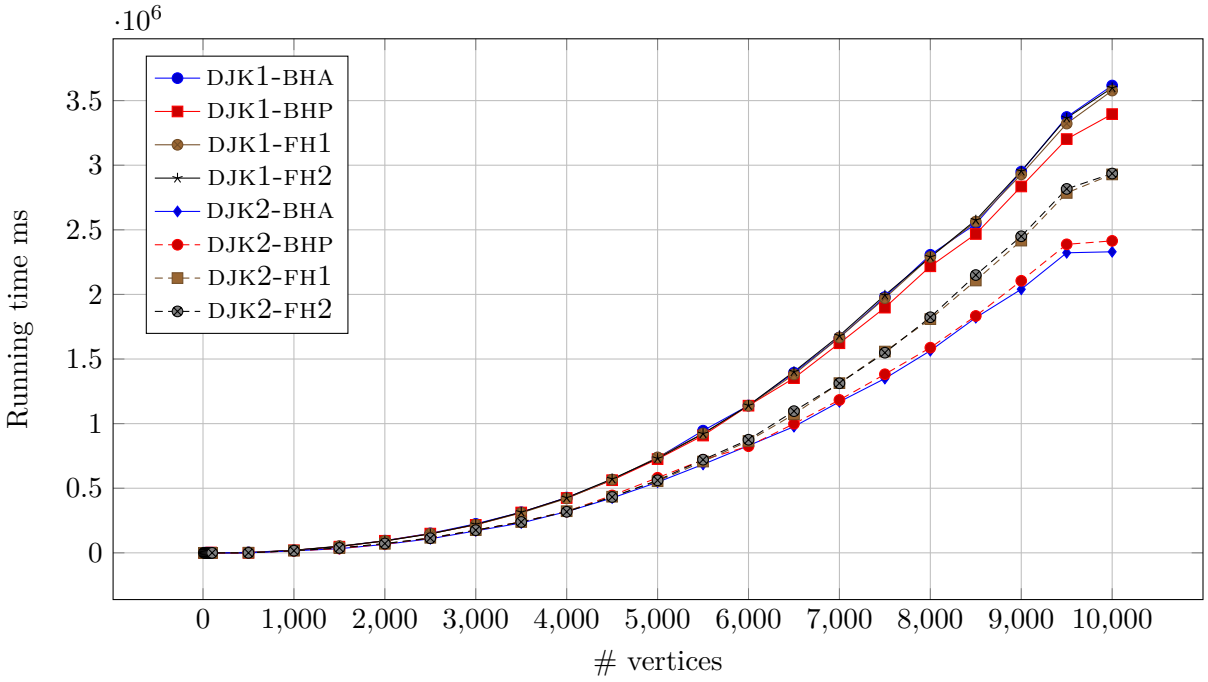


Figure 7.4: Average time of running DIJKSTRA1 and DIJKSTRA2 on 70 % connected graph

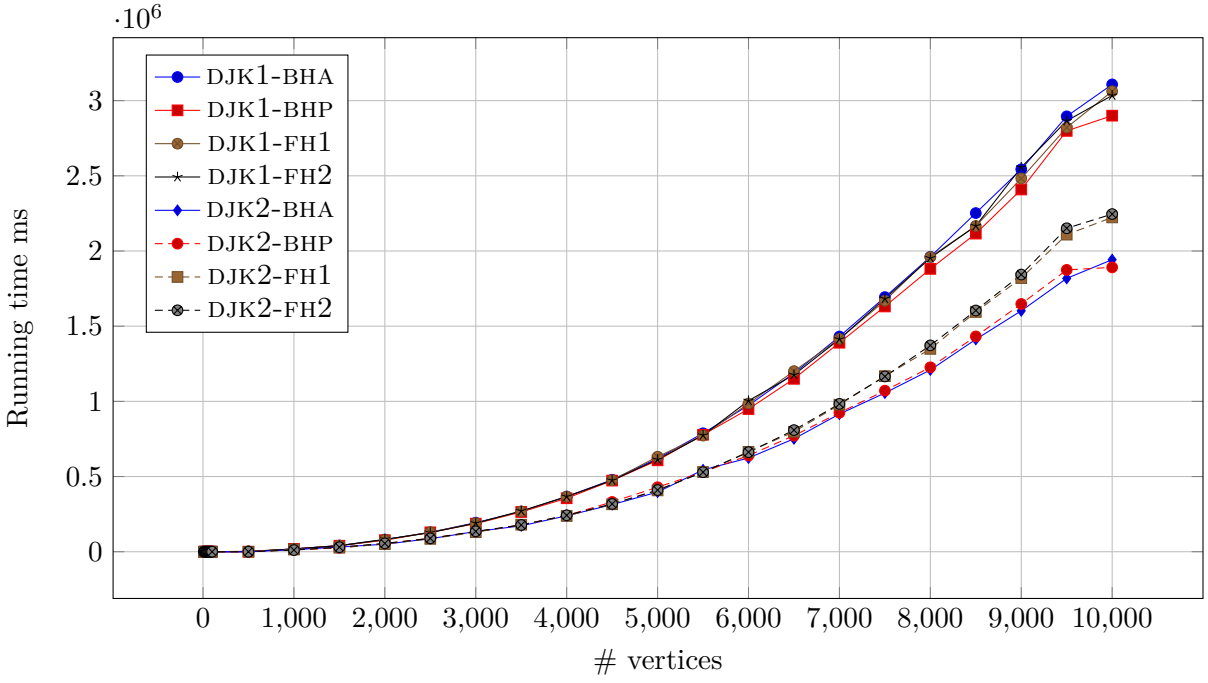


Figure 7.5: Average time of running DIJKSTRA1 and DIJKSTRA2 on 60 % connected graph

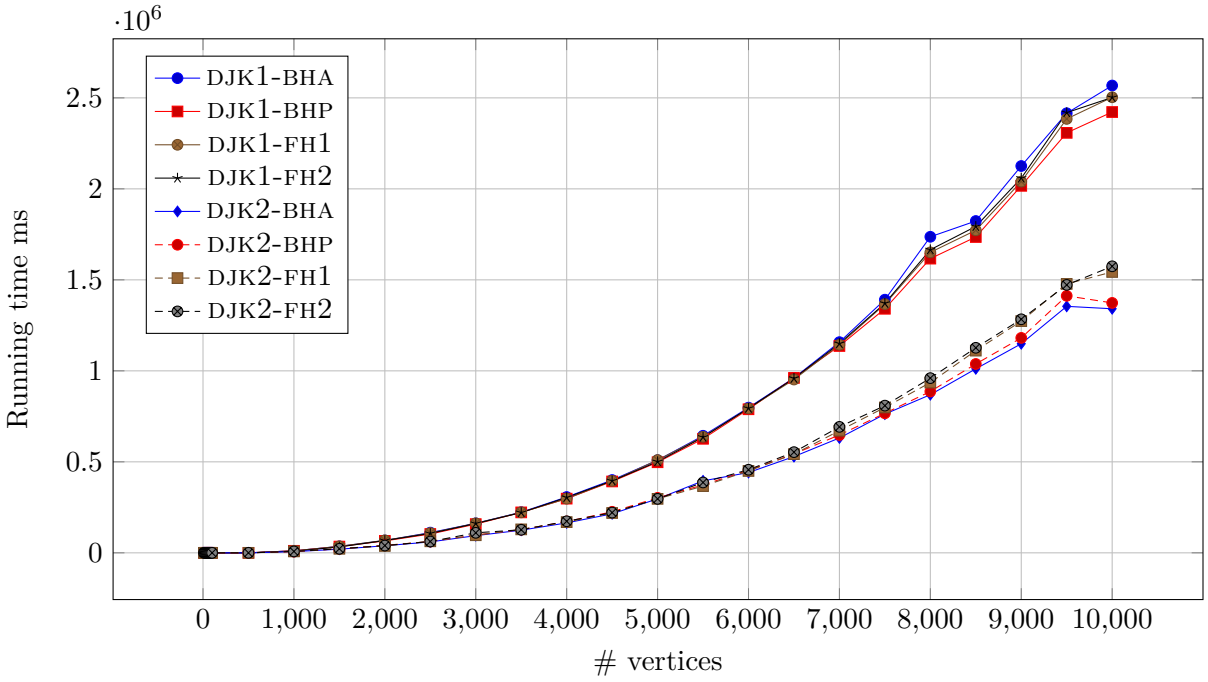


Figure 7.6: Average time of running Dijkstra1 and Dijkstra2 on 50 % connected graph

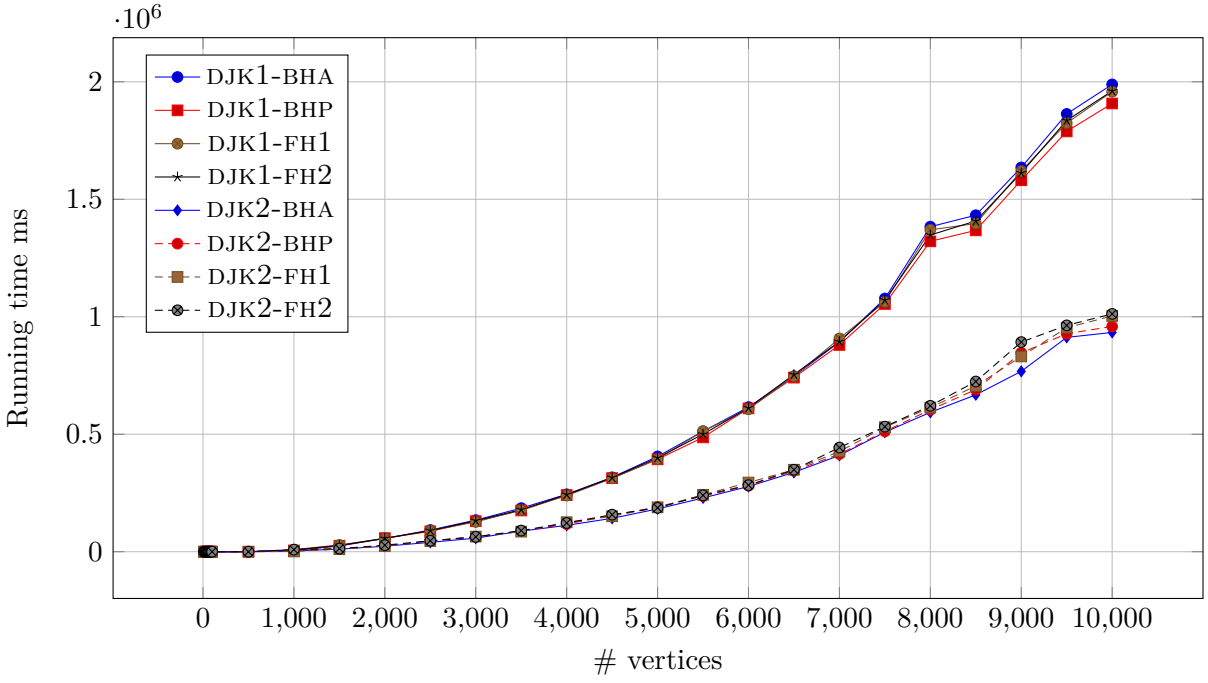


Figure 7.7: Average time of running Dijkstra1 and Dijkstra2 on 40 % connected graph

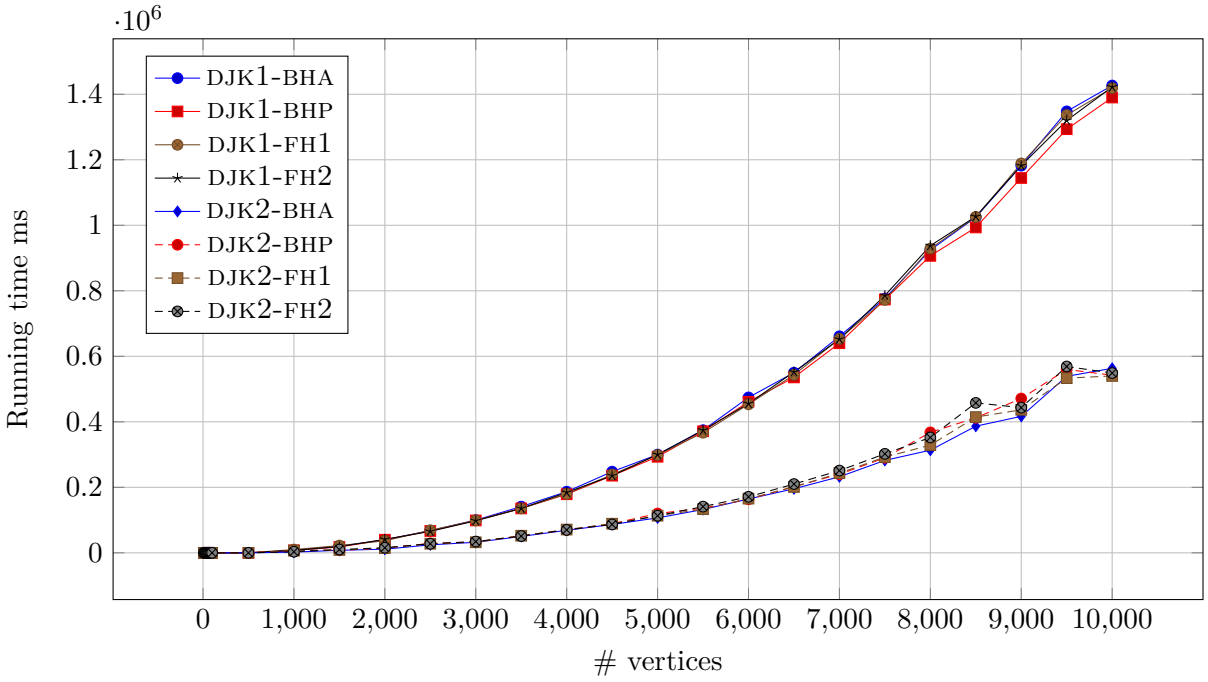


Figure 7.8: Average time of running Dijkstra1 and Dijkstra2 on 30 % connected graph

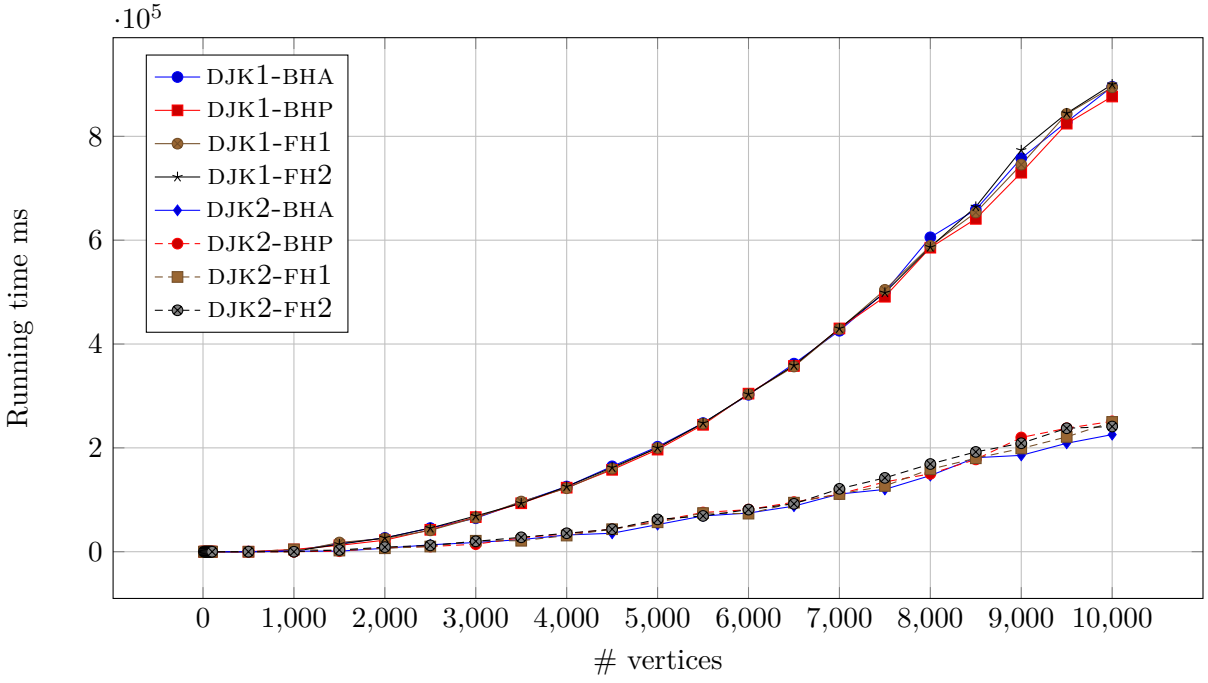


Figure 7.9: Average time of running Dijkstra1 and Dijkstra2 on 20 % connected graph

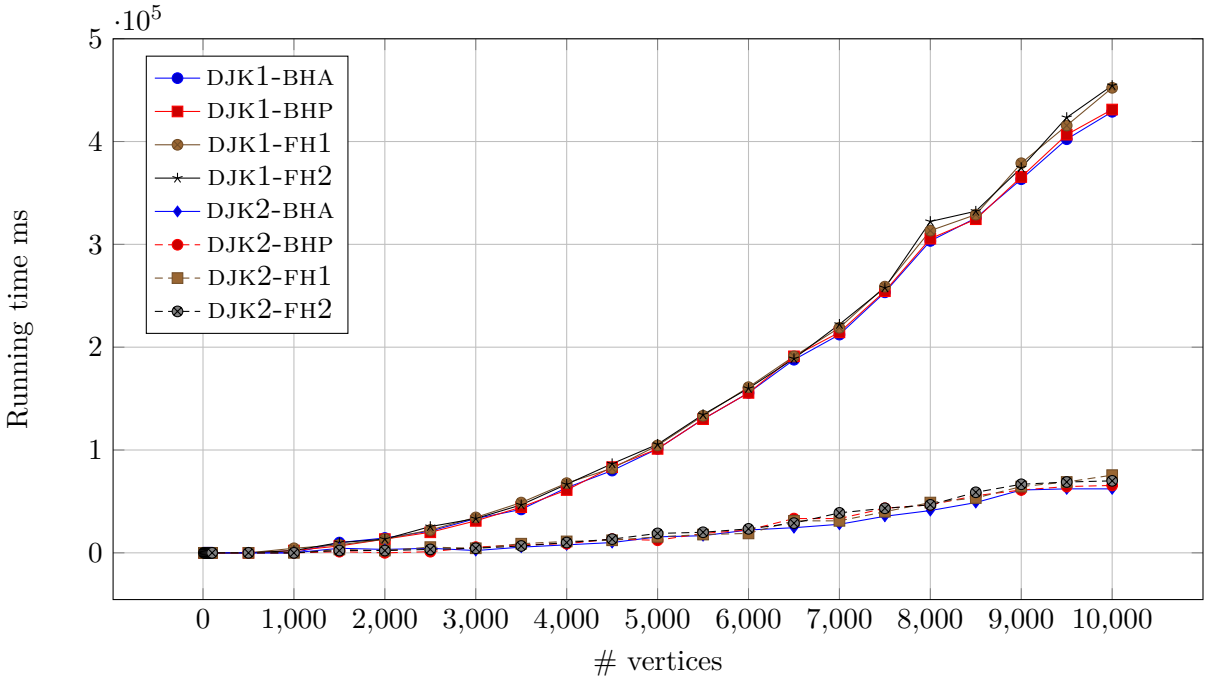


Figure 7.10: Average time of running Dijkstra1 and Dijkstra2 on 10 % connected graph

Chapter 8

Conlusion

needs contents

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.