LULEÅ UNIVERSITY OF TECHNOLOGY

Algorithms and Data Structures - Lab 3

Course: D0012E

Marcus Lund (amuulo-4)
Edvin Åkerfeldt (edvker-4)
Samuel Karlsson (samkar-4)
February 1, 2016

Second submission.

Abstract

In this report the runtime of Dijkstra's algorithm is analysed with different number of connections on on each node and different "D" sizes ("D" is the D in D-arr heap). The trend shows that a bigger graph results in a longer runtime. The best runtime were when "D" was about 13 thousand.

Generally the tests we ran were very case sensitive. The runtime differs a lot between worst and best case. The time we used was an average of multiple testes to minimize the effect from best and worst case tests.

CONTENTS

1 Introduction

In this report a lab on Dijkstra's algorithm is presented. The lab was done in the course "algorithms and computer structure". The goal for the lab was to see how the runtime is effected on different sizes of "D" and different number of connections (edges) to each node.

1.1 Language

We chose to use Java as our programming language of choice over other languages, because it is a language the we are comfortable to work with. Java is also one of the better optimized languages.

1.2 Implementation

Implementing Dijkstra's algorithm is a challenge because the graphs representation have a huge influence on Dijkstra's algorithms runtime.

We chose to use a class representation of the graph. The class implementaion is in some regard straight forward and logical but has a clear disadvantages when it comes to memory consumption. When the graph grows its memory usage grows very fast. and method cols are heavy sow it is not the fastest way to implement it.

The hardest part of implementating the Dijkstra's algorithm is to find a way to add the new nods to the added ones without adding the same node twice. To solved that problem we used an array where we stored the added nodes, distance to node and the preceding node.

2 Test procedure

The tests were ran several times with different sizes "D" and edges. We ran the testes multiple times and took the averages time.
We ran two tests. The first test the value of D was in the range $1000-10000$ while setting the nodes to 10000 and edges to 40000. In the other test we let the edge number be in the range of $10000-40000$, 10000 nodes and D at 1000. We incremented the edges by 500 each run and the value of D increased by 500.

3 Result

In the figure 3.1, we see that the time is big at the beginning and decreases to a optimum value of D before the time increases again. The total runtime is on the Y axes and the x axes is the number of runs. Between each run, there is an increment of 500 for each D. Sow D has a minimum runtime value at 5000.

In figure 3.2, we can see that an increase of the edges result in longer runtime. The starting amount is 10000 and the increasement is 500 edges per run to a maximum of 40000 edges.

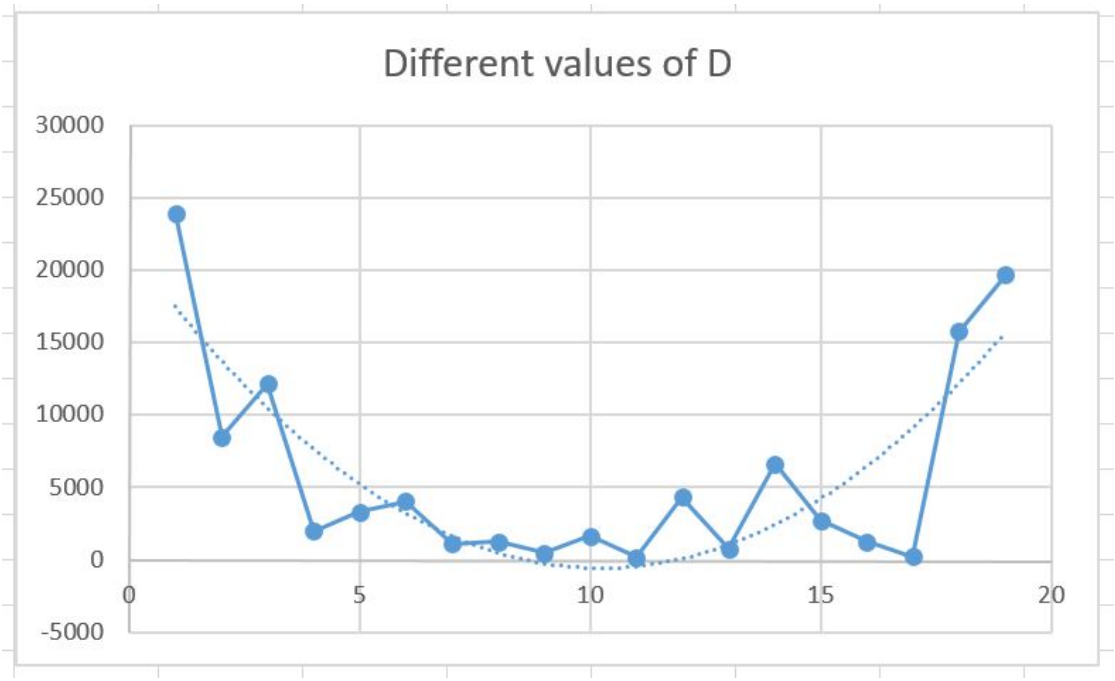

Figure 3.1: Different values of D.

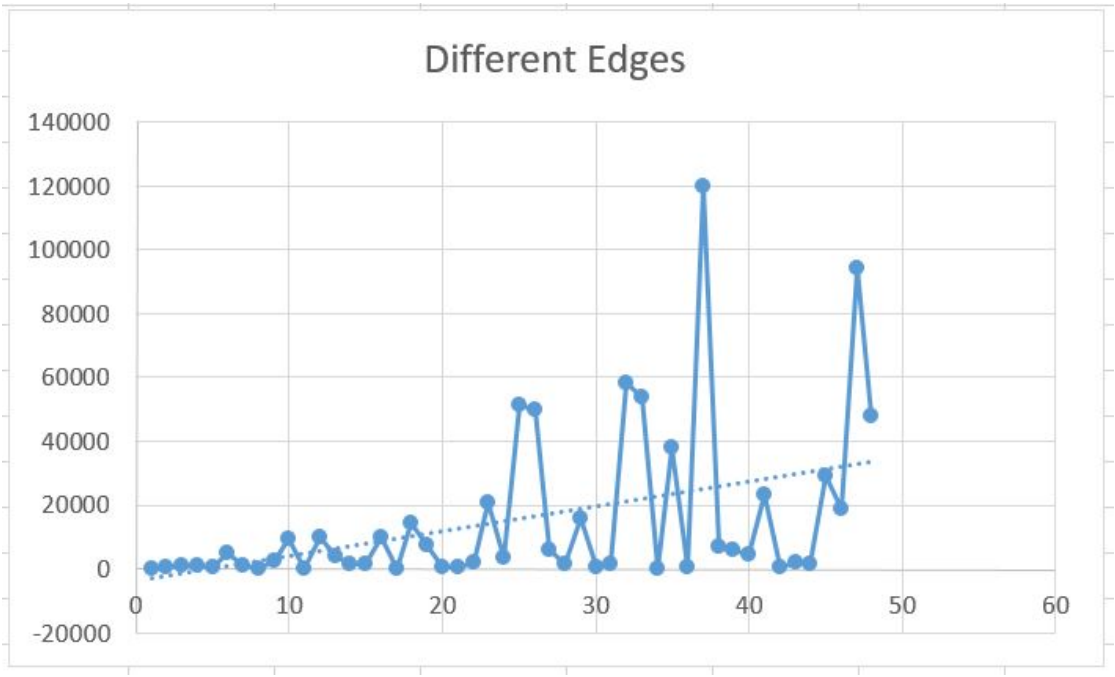

Figure 3.2: Different sizes of edges.

4 Discussion

When we compare the two figures 3.2 and 3.1, we see that increasing the edges will increase the runtime while the value of D may worsen or improve the runtime.

We notice that the value of D has an optimum value for a given graph. While the edges only increase the runtime, the value of D may improve the runtime instead. D has an optimal runtime at approximately 5000 (see figure 3.1). We can see that the value of D has a greater effect of the runtime when passed the optimum state. If we increase D from its optimal value, the increase of D will affect the runtime much more than the increase of the total edges in the graph. Increase the total numbers of edges in the graph only generate in a slow grows of runtime (see 3.2).

Analyse the runtime in Dijkstra's algorithm is tricky becks the runtime depend on the representation of the graph. We choose to use a objective representation of the graph, which is simple to understand but is bad for de runtime. In theory is the worst case runtime $O(E \cdot log_D(N))$ where E is number of edges. N is number of nods. With our graph representation is the runtime not ass god ass it cod had been. By using a fast accessible representation (for example a matrix) of the graph can the a descent amount of time be sawed.

Ass there can be seen in our graphs is there some variation in the runtime. These is because we use a random generated graph, where the stop nod cod be one of the first nods or the last nod. The way we implemented the D-arr heap leaves some rum for random output of time. In our implementation is there a risk for "double work" bat it can trade of and win time. In these case is it probably better to take a slightly slower implementation in avarices-case but more consistent and a better worst-case. These to get graphs with a clear trends which make it easer to analyse.

The implementation is crucial in an algorithm such as this one. This due to the fact that a faulty implementation, thinking about datastructsurs and functions, may and will affect the algorithms runtime to a degree where the result will be computed at a rate that is unusable. We noticed this behaviour from our previous implementations of this algorithm.