Data Structures and Algorithms Graded Assignment Report

Péter Gulyás

December 4, 2023

1 Introduction

In this paper I will be presenting my measurements on 5 sorting and 3 search algorithms. My choosen sorting algorithms were Selection, Bubble, Insertion, Merge and Quick sort.

Durning my measurements I performed all the sortings 10 times on a sample size of 10, 50, 100, 500, 1000 and 5000 elements and I randomly populated my data structures.

The searches are Breadth-first, Depth-first and A*. These I will be performing on a graph structure we implemented during the lectures.

All my measurements can be found on the following google sheets.

2 Sorting Algorithms

2.1 Chosen Data Structures

This part of the assignment was written in C++. I decided to use vectors for this assignment for their convinience(many built in functions, manages memory automatically).

2.2 Bubble sort

Based on my measurements Bubble sort seemed the slowest algorithm of the 5. The main reason is that the algorithm iterates through the data set multiple times, comparing the adjecent elements. On the next figure you are able to see my measurements.

Bubblesort:	10 elements	50 elements	100 elements	500 elements	1000 elements	5000 elements
1	0,0000010	0,0000310	0,0001340	0,0032880	0,0169910	0,3253000
2	0,0000010	0,0000350	0,0001300	0,0044950	0,0127890	0,2996500
3	0,0000010	0,0000290	0,0001260	0,0035450	0,0140650	0,2861000
4	0,0000010	0,0000300	0,0001250	0,0032160	0,0137100	0,2879800
5	0,0000010	0,0000310	0,0001270	0,0032350	0,0127900	0,2888400
6	0,0000010	0,0000340	0,0001210	0,0031690	0,0121530	0,3068700
7	0,0000010	0,0000310	0,0001230	0,0031320	0,0120930	0,2919000
8	0,0000010	0,0000310	0,0001200	0,0035940	0,0120610	0,2985400
9	0,0000020	0,0000310	0,0001280	0,0032790	0,0120940	0,2891500
10	0,0000020	0,0000320	0,0001460	0,0031780	0,0133980	0,2892500
Avg:	**0,0000012**	**0,0000315**	**0,0001280**	**0,0034131**	**0,0132144**	**0,2963580**
Median:	**0,0000010**	**0,0000310**	**0,0001265**	**0,0032570**	**0,0127895**	**0,2905750**
Min:	**0,0000010**	**0,0000290**	**0,0001200**	**0,0031320**	**0,0120610**	**0,2861000**
Max:	**0,0000020**	**0,0000350**	**0,0001460**	**0,0044950**	**0,0169910**	**0,3253000**

Figure 1: Bubble sort.

2.3 Selection Sort

Selection Sort performes better that bubble sort(in most cases), but it is still slow compared to the more advanced algorithms. It makes fewer swaps usually than bubble sort. But bubble sort can be more efficient on a data set that is close to sorted.

Selection sort:	10 elements	50 elements	100 elements	500 elements	1000 elements	5000 elements
1	0,0000010	0,0000140	0,0000800	0,0019090	0,0050340	0,1418890
2	0,0000010	0,0000140	0,0000790	0,0019610	0,0052350	0,1136400
3	0,0000000	0,0000150	0,0000820	0,0013540	0,0063350	0,1208990
4	0,0000000	0,0000120	0,0000830	0,0012100	0,0077710	0,1070660
5	0,0000000	0,0000210	0,0000810	0,0012220	0,0076000	0,1228140
6	0,0000000	0,0000210	0,0000810	0,0012150	0,0076720	0,1226380
7	0,0000010	0,0000200	0,0000810	0,0017990	0,0058870	0,1193360
8	0,0000000	0,0000200	0,0000780	0,0011800	0,0044660	0,1113510
9	0,0000000	0,0000200	0,0000820	0,0013480	0,0044190	0,1106180
10	0,0000000	0,0000210	0,0000810	0,0012570	0,0053760	0,1192800
Avg:	**0,0000003**	**0,0000178**	**0,0000808**	**0,0014455**	**0,0059795**	**0,1189531**
Median:	**0,0000000**	**0,0000200**	**0,0000810**	**0,0013025**	**0,0056315**	**0,1193080**
Min:	**0,0000000**	**0,0000120**	**0,0000780**	**0,0011800**	**0,0044190**	**0,1070660**
Max:	**0,0000010**	**0,0000210**	**0,0000830**	**0,0019610**	**0,0077710**	**0,1418890**

Figure 2: Selection sort.

2.4 Insertion sort

I was expecting insertion sort to perform worse than selection since it tends to do more comparisons and more swaps, yet it wasn't the

case based on my measurements. They have very similar performance, with insertion sort having a slight advantage.

Insertion sort:	10 elements	50 elements	100 elements	500 elements	1000 elements	5000 elements
1	0,0000010	0,0000090	0,0000470	0,0009830	0,0042890	0,0998100
2	0,0000000	0,0000100	0,0000390	0,0009730	0,0042250	0,0995080
3	0,0000000	0,0000120	0,0000390	0,0010450	0,0044190	0,0975430
4	0,0000000	0,0000110	0,0000430	0,0010400	0,0042590	0,0970040
5	0,0000000	0,0000080	0,0000370	0,0011190	0,0044580	0,0982920
6	0,0000000	0,0000120	0,0000450	0,0010750	0,0039190	0,0967770
7	0,0000000	0,0000100	0,0000430	0,0012090	0,0043570	0,1036710
8	0,0000000	0,0000100	0,0000440	0,0010450	0,0040860	0,1083190
9	0,0000000	0,0000110	0,0000370	0,0010530	0,0037960	0,1036600
10	0,0000000	0,0000080	0,0000440	0,0010550	0,0039570	0,1031780
Avg:	**0,0000001**	**0,0000101**	**0,0000418**	**0,0010597**	**0,0041765**	**0,1007762**
Median:	**0,0000000**	**0,0000100**	**0,0000430**	**0,0010490**	**0,0042420**	**0,0996590**
Min:	**0,0000000**	**0,0000080**	**0,0000370**	**0,0009730**	**0,0037960**	**0,0967770**
Max:	**0,0000010**	**0,0000120**	**0,0000470**	**0,0012090**	**0,0044580**	**0,1083190**

Figure 3: Insertion sort.

2.5 Quick sort

From my tests, Quick Sort performed the best by a lot. The larger the group of data, the better Quick Sort was. It worked so well because it needed to compare and swap things less, and it used a smart divide-and-conquer strategy.

Quick sort:	10 elements	50 elements	100 elements	500 elements	1000 elements	5000 elements
1	0,0000010	0,0000000	0,0000100	0,0000700	0,0001400	0,0008900
2	0,0000000	0,0000100	0,0000100	0,0000700	0,0001400	0,0009700
3	0,0000010	0,0000000	0,0000100	0,0000700	0,0001500	0,0009500
4	0,0000010	0,0000000	0,0000100	0,0000700	0,0001400	0,0009200
5	0,0000000	0,0000000	0,0000100	0,0000800	0,0001500	0,0009000
6	0,0000000	0,0000000	0,0000100	0,0000700	0,0001600	0,0012400
7	0,0000000	0,0000000	0,0000100	0,0000700	0,0001600	0,0014600
8	0,0000000	0,0000000	0,0000100	0,0000700	0,0001400	0,0014500
9	0,0000000	0,0000100	0,0000100	0,0000700	0,0001500	0,0015000
10	0,0000000	0,0000000	0,0000100	0,0000600	0,0001500	0,0014100
Avg:	**0,0000003**	**0,0000020**	**0,0000100**	**0,0000700**	**0,0001480**	**0,0011690**
Median:	**0,0000000**	**0,0000000**	**0,0000100**	**0,0000700**	**0,0001500**	**0,0011050**
Min:	**0,0000000**	**0,0000000**	**0,0000100**	**0,0000600**	**0,0001400**	**0,0008900**
Max:	**0,0000010**	**0,0000100**	**0,0000100**	**0,0000800**	**0,0001600**	**0,0015000**

Figure 4: Quick sort.

2.6 Merge Sort

2.6.1 Vector

I found Merge Sort to be quite interesting. It didn't do well with smaller datasets (up to 100 elements), performing worse than other sorting methods. At 500 elements, it outperformed Bubble Sort but

still lagged behind Selection and Insertion Sorts. However, as we reached larger datasets in the thousands, Merge Sort caught up, yet it remained noticeably slower than Quick Sort.

Merge sort:	10 elements	50 elements	100 elements	500 elements	1000 elements	5000 elements
1	0,0000150	0,0000820	0,0001710	0,0015690	0,0032170	0,0101630
2	0,0000160	0,0000840	0,0001730	0,0015720	0,0033540	0,0103480
3	0,0000130	0,0000810	0,0001720	0,0015720	0,0032410	0,0099340
4	0,0000170	0,0000810	0,0002620	0,0015770	0,0026740	0,0103490
5	0,0000130	0,0000810	0,0002940	0,0016660	0,0018190	0,0104120
6	0,0000130	0,0000810	0,0003060	0,0015640	0,0018430	0,0125130
7	0,0000140	0,0000880	0,0003110	0,0016070	0,0019210	0,0109180
8	0,0000130	0,0000810	0,0003060	0,0016510	0,0019180	0,0106720
9	0,0000130	0,0000810	0,0002980	0,0016470	0,0018340	0,0125990
10	0,0000130	0,0000830	0,0003620	0,0015370	0,0019230	0,0115680
Avg:	**0,0000140**	**0,0000823**	**0,0002655**	**0,0015962**	**0,0023744**	**0,0109476**
Median:	**0,0000130**	**0,0000810**	**0,0002960**	**0,0015745**	**0,0019220**	**0,0105420**
Min:	**0,0000130**	**0,0000810**	**0,0001710**	**0,0015370**	**0,0018190**	**0,0099340**
Max:	**0,0000170**	**0,0000880**	**0,0003620**	**0,0016660**	**0,0033540**	**0,0125990**

Figure 5: Merge sort.

2.6.2 Array

As a slight improvment I decided that I am going to implement the Merge sort with arrays instead of vectors. I thought this could improve on it's performance since in C++ accessing an element of an array is more efficient than accessing an element of a vector. The algorithm performed numbers comparable to Quick sort when applied to vectors.

Merge sort array:	10 elements	50 elements	100 elements	500 elements	1000 elements	5000 elements
1	0,0000020	0,0000150	0,0000250	0,0001060	0,0001980	0,0012670
2	0,0000010	0,0000110	0,0000170	0,0001020	0,0002360	0,0012030
3	0,0000010	0,0000110	0,0000210	0,0001010	0,0002200	0,0011470
4	0,0000010	0,0000110	0,0000230	0,0001180	0,0002310	0,0011690
5	0,0000060	0,0000100	0,0000160	0,0001200	0,0001860	0,0010000
6	0,0000010	0,0000110	0,0000270	0,0001330	0,0002300	0,0010660
7	0,0000010	0,0000110	0,0000180	0,0001020	0,0001970	0,0011660
8	0,0000010	0,0000140	0,0000160	0,0001010	0,0001900	0,0010740
9	0,0000010	0,0000110	0,0000240	0,0001020	0,0002090	0,0010580
10	0,0000010	0,0000070	0,0000160	0,0000800	0,0002200	0,0010060
Avg:	**0,0000016**	**0,0000112**	**0,0000203**	**0,0001065**	**0,0002117**	**0,0011156**
Median:	**0,0000010**	**0,0000110**	**0,0000195**	**0,0001020**	**0,0002145**	**0,0011105**
Min:	**0,0000010**	**0,0000070**	**0,0000160**	**0,0000800**	**0,0001860**	**0,0010000**
Max:	**0,0000060**	**0,0000150**	**0,0000270**	**0,0001330**	**0,0002360**	**0,0012670**

Figure 6: Merge sort implemented with array.

3 Search Algorithms

For this task, I chose to use C# because I'm more comfortable with that language. I mainly followed the pseudocodes given in class, which helped me decide on the types of data structures to use. I ended up using various ones like C# lists, Queue, HashSet, and C# dictionaries (HashMaps).

During my measurement I found that A* required way more time to find the solution, althought I found the optimal one, unlike breadth first and depth first which managed the reach the goal way faster but took a really unoptimal path.

4 Conclusion

In summary, after looking at different ways to sorting and finding path, we found that Quick Sort is the best at putting things in order, especially when there's a large data set. It's good because it doesn't have to compare and swap things as much and uses a smart strategy. Merge Sort, especially when I changed the type of data structer, did pretty well too, but not as fast as Quick Sort. When it comes to finding path, A* is the best at getting the best solution, but it takes more time. Breadth-first and Depth-first are quicker, but they might not find the best solution.