Computability Report

Table of Contents

[Homework Exercise 2](#_Toc183765919)

[The Program 2](#_Toc183765920)

[Results 3](#_Toc183765921)

[Analysis 5](#_Toc183765922)

[Bubble sort 6](#_Toc183765923)

[Insertion sort 6](#_Toc183765924)

[Merge sort 6](#_Toc183765925)

[Conclusion 6](#_Toc183765926)

[Extension Exercise 6](#_Toc183765927)

[Introduction 6](#_Toc183765928)

[Challenges 7](#_Toc183765929)

[Results 7](#_Toc183765930)

[Brute Force 9](#_Toc183765931)

[Nearest Neighbour 9](#_Toc183765932)

[Minimum Spanning Tree 10](#_Toc183765933)

[Christofides’ Algorithm (Nearest Neighbour) 12](#_Toc183765934)

[Christofides’ Algorithm (Hungarian Algorithm) 13](#_Toc183765935)

[Conclusion 15](#_Toc183765936)

[Bibliography 15](#_Toc183765937)

# Homework Exercise

## The Program

I chose to write all three sorting algorithms (Bubble, Insertion, and Merge) and compare them. The application uses an extremely simple CLI and was written in plain C using the Visual Studio 2022 IDE. The program can run in two modes depending on the input provided:

* If the word gen and no other input is provided the program will generate 20 files of increasing size up to the maximum containing just over 1 million random signed 64-bit values encoded as ASCII as to be readable.
* If a filename is given, the program will load this file and try to interpret it as a list of randomly arranged, signed, 64-bit integers delimited by newline characters ignoring any other characters. The program will then make 3 copies of this list and use one algorithm to sort each of these lists respectively, it will then make a second pass of the algorithms on the already sorted lists, and then make a final pass re-sorting the lists in reverse order. Finally, the program will display statistics gathered from these operations (Figure 1 shows a typical output).

A screenshot of a computer

Description automatically generated

Figure 1. Typical output or sorting application.

## Results

The program was run on all 20 generated data files 3 times with an average time in seconds taken from these 3 runs and recorded in a spreadsheet as can be seen in Table 1. Additionally, several graphs were created showing:

* a comparison between the average case for each algorithm (Figure 2).
* a comparison of the different passes for each algorithm respectively (Figures 3, 4, and 5).

A screenshot of a spreadsheet

Description automatically generated

Table 1. Average case comparison.

Figure 2. Time complexity comparison.

Figure 3. Bubble sort time complexity.

Figure 4, Insertion sort time complexity.

Figure 5. Merge sort time complexity.

## Analysis

As we can clearly see from Figure 2 bubble sort is the slowest algorithm with insertion sort being in the middle and merge sort being the fastest. Bubble sort is the slowest algorithm in all cases but by looking at the spreadsheet we can see that insertion sort beats merge sort on small datasets up to 64 integers in size, most likely due to the overhead of merge sort.

### Bubble sort

The time complexity of all cases is as expected with the random order and reverse order scaling quadratically, while the sorted order scales linearly. What is interesting however is the reverse order being consistently faster than the average (random order) case. It is likely that this is because the branch predictor is incorrectly guessing the direction of our branches less often when compared to a random order and thus reducing the number of pipeline stalls within the CPU.

### Insertion sort

The time complexity of all cases is as expected with the random order and reverse order scaling quadratically, while the sorted order scales linearly.

### Merge sort

The time complexity is linear in all cases as expected. While all cases are expected to take the same amount of time random order is the slowest and this is most likely due to the branch prediction of the CPU again. The difference between reverse order and sorted order is likely an implementation detail and could be due to reverse order staying within the inner loop more often and causing fewer scope changes.

## Conclusion

The time complexities of all algorithms were largely as expected with the differences being largely attributable to hardware optimizations. This is quite likely a lot more noticeable in a language like C where we are close to the hardware as compared to higher level languages where significant work is done between the instructions requested by the programmer.

# Extension Exercise

## Introduction

The travelling salesman problem was chosen for the extension exercise. It is reasonably simple to deduce how to calculate the solution to the travelling salesman problem using a brute force method, but this quickly becomes unusable as the number of nodes increases due to it having a time complexity of O(n!). No known polynomial time solution exists for this problem, so heuristics are usually employed such as nearest neighbour. A minimum spanning tree can be used to generate a solution with a length no more than twice of the optimal solution, we can take this further with Christofides’ algorithm (Christofides, 2022) which uses the Hungarian algorithm (Kuhn, 1955) to calculate the set of odd degree nodes with the minimum total length, combining this with the minimum spanning tree gives us a solution to the problem that is at most 1.5 times longer than the optimal path. A screenshot of the help message for the complete application is provided (Figure 6) which gives a complete list of features implemented.

A screenshot of a computer

Description automatically generated

Figure 6. Application help message.

## Challenges

The major challenge was decoding the very mathematical papers into something that could be understood and implemented by someone with limited mathematical ability. Additionally debugging of the program becomes significantly more challenging as the size of the matrices being operated on increase.

Supporting the full specification of the TSPLIB was a challenge and it was discovered that some edge cases are not covered with EXPLICIT matrix data files that contain partial matrices not being supported, however, it was also very useful to be able to feed these datasets into the algorithm for testing purposes and so worth the effort.

## Results

The average time taken of all algorithms (excluding brute force) on randomly generated nodes is shown in Figure 7, brute force is shown separately in Figure 8 as it is infeasible to generate a solution with even 32 nodes this way.

Figure 7. Algorithm time taken comparison.

Figure 8. Brute force time complexity.

### Brute Force

Brute force works fine for a handful of nodes but becomes infeasible to compute very quickly with a time complexity of O(n!) (Figure 8). It guarantees the minimum path length and so where this is required an optimized version of this algorithm is the only choice.

### Nearest Neighbour

The nearest neighbour heuristic is the fastest algorithm by far with a time complexity between O(n log(n)) and O(n2) (Figure 9) it however does not provide any guarantees of its accuracy making it less useful where a solution with determinable accuracy is required (Figure 10).

Figure 9. Nearest Neighbour time complexity.

Figure 10. Nearest Neighbour accuracy.

### Minimum Spanning Tree

Using a minimum spanning tree to calculate the path gives an accuracy of no more than 2 times the optimal path length (Figure 11) with a time complexity between O(n2.5) and O(n2.7) (Figure 12) which gives some guarantee of accuracy in exchange for slower computation.

Figure 11Minimum Spanning Tree time complexity.

Figure 12. Minimum Spanning Tree accuracy.

### Christofides’ Algorithm (Nearest Neighbour)

Using a nearest neighbour heuristic to calculate the matching odd degree node pairs means that the accuracy can no longer be guaranteed as being no more than 1.5 times the maximum path. The time complexity is the same as the minimum spanning tree (Figure 13) however and adding the heuristic can only improve the path generated as can be seen in Figure 14, however, it is not possible to say by how much.

Figure 13. Christofides' (Nearest Neighbour) time complexity.

Figure 14. Christofides' (Nearest Neighbour) accuracy.

### Christofides’ Algorithm (Hungarian Algorithm)

There is a flaw in the implementation of the Hungarian algorithm which makes the algorithm take much longer to process under certain (unknown) conditions that become more common at higher node numbers, this yields some instability in the graphs. The implemented algorithm when working performs exceptionally well, providing a solution that is no more than 1.5 times longer than the optimal path (Figure 15) due to the instability and reduced dataset it is difficult to measure but it seems to be between O(n log(n)) and O(n2) (Figure 16).

Figure 15. Christofides' (Hungarian) time complexity.

Figure 16. Christofides' (Hungarian) accuracy.

## Conclusion

Where accuracy is required a brute force method must be used although for large problems this becomes infeasible, this makes heuristics important for making good enough solutions in reasonable amounts of time. Due to its speed of computation, it always makes sense to perform a nearest neighbour calculation first, and only if this is not good enough then using a fully functional implementation of the Christofides’ Algorithm is the best that can be achieved within polynomial time.

# Bibliography

Christofides, N. (2022, March). Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem. *SN Operations Research Forum, 3*(10), 1-5. Retrieved November 15, 2024, from https://www.researchgate.net/publication/235067784\_Worst-Case\_Analysis\_of\_a\_New\_Heuristic\_for\_the\_Traveling\_Salesman\_Problem

Kuhn, H. (1955, March). The Hungarian Method for the Assignment Problem. *Naval Research Logistics, 2*, 83-97. Retrieved November 16, 2024, from https://web.eecs.umich.edu/~pettie/matching/Kuhn-hungarian-assignment.pdf