Jospeh Hulcher
Dr. Yessick
CS 470
02/22/24

# An Exploration of TSP approximation algorithms

## 1. Preliminary Algorithms

The first algorithm created and tested was the Nearest Neighbor Algorithm described in class. The algorithm starts with the specified node, then finds the next closest node in the graph until a complete tour is found. It works in O(n^2) time, and it is quite fast. The algorithm was made by sketching it up then writing it out in C#. The Nearest Neighbor algorithm provided a good base for comparison with other algorithms.

The next algorithm was Nearest Neighbor All. This algorithm is a straightforward extension of the Nearest Neighbor Algorithm, simply running the previous algorithm starting on each node, then taking the lowest result of them all. It runs in O(n^3) time.

The last preliminary algorithm was the brute force algorithm. It also was sketched up without research and then programmed. Following is the pseudocode for the version used in this paper:

```
Brute Force Algorithm:
    Add node 0 to the path
    Call the recursive solver

    Recursive Solver:
        If only one node left:
            Add the last node to the path
            Check if the current path is shorter than the shortest found
                If so, store current path as shortest
            Remove last node from current path and return
        If n > 1 nodes left:
            For each node left i:
                add i to the current path
                recursively call the solver
                remove i
```

This algorithm featured a few small optimizations. First, it always starts with node 0, as any optimal tour can be rotated so that the first node is node 0. With this, a recursion was saved, making it possible to run on graphs one larger. Further, the length of the current path is tracked so that checking the length is an O(1) operation instead of an O(n). This again allows for at most 1 additional node, as the time of the algorithm is still O(n!).

This project was written and tested in C#. In the live testing of this program, the Brute Force could solve graphs of 14 nodes in about 6 minutes. Larger graphs were not tested. With compiling the project and allowing for C# testing, it is suspected that the algorithm would have been able to run a few more nodes.

## 2. Explored Algorithms

### Bubble Path

The next algorithm developed was bubble path, or also known as random insertion. Bubble Path works by creating a subtour consisting of the first node, then inserting the next next node in the spot of the current tour that minimizes the next subtour's length. It inserted the nodes in order. The name came from the idea of the tour "bubbling" from a few nodes to a complete graph. The idea was that by inserting, the next node could be chosen in a way that optimizes the tour already created. Hence, particularly bad edges chosen earlier in the algorithm could be split up by nodes inserted later. This would hopefully avoid particularly long edges sometimes chosen by the Nearest Neighbor algorithm, leading to a better solution.

After further exploration of solutions made by others, it was discovered that this algorithm is very similar to Random Insertion. The only difference between them is that in Bubble Path, the order of nodes inserted is the order of nodes presented while in random insertion, the order of insertion is random. However, as the nodes are randomly generated, this suggests that Bubble Path should act very similar to Random Insertion. However, Bubble Path eliminates the possibility of running a Nearest Insertion Algorithm multiple times to choose a better base for further optimization. Though, one could try an algorithm that optimizes the order of the nodes given to Bubble Path by rearranging the order of the nodes given to Bubble Path.

### Nearest Insertion

The next algorithm explored was Nearest Insertion. This algorithm works the same as the bubble path algorithm explained above, except the next node to be inserted into the graph is the remaining node nearest to the current subtour. This is supposed to improve the worst case compared to optimal path, making it at max two [2]. However, in the test runs, it performed the worst of all the algorithms. I am not sure why. First I suspected something was

programmed badly. However, the solutions produced are valid after checking, and after reviewing the code nothing appeared to be amiss. Possibly by choosing the nearest neighbor each time, later optimizations gained by bubble sort are missed.

### 2-Opt

In preparation for the competition, the last algorithm written was 2-opt. The 2-opt algorithm consists of switching each pair of two edges in the graph, and checking to see if the resulting graph is shorter. It repeats this process until no optimizations are found. After running this algorithm, a solution should be 2-optima [1]l, which means no 2-edge swap would improve the current solution. This algorithm improved the solutions generated by other algorithms, but interestingly enough it did not improve them equally, as insertion algorithms saw less benefit after applying the 2-opt algorithm.

## 3. Testing and Comparison

### Data

The following table shows the test data like this:

On the top row, which algorithm the column displays results for is shown.

On the leftmost row, three pieces of data is presented about each test case:

1: the size of the graph

2: the number of nodes in the graph

3: the number of graphs tested

In the middle boxes of the graph, two pieces of data are presented:

1: the average length of the solution for that specific test case

2: the total running time of the test case in milliseconds

| Test Case | Nearest Neighbor | Nearest Neighbor All | Bubble Path | Nearest Insertion | Nearest Neighbor +2Opt | Nearest Neighbor All+2Opt | Bubble Path +2Opt | Nearest Insertion +2Opt |
|---|---|---|---|---|---|---|---|---|
| 100^2 n=100 Times:25 | 965.78<br>0ms | 879.54<br>78ms | 845.37<br>2ms | 958.55<br>3ms | 827.14<br>13ms | 812.31<br>87ms | 824.78<br>8ms | 846.13<br>16ms |
| 1000^2 n=100 Times: 25 | 9658.55<br>1ms | 8904.06<br>86ms | 8519.25<br>1ms | 9705.84<br>3ms | 8291.39<br>14ms | 8206.11<br>83ms | 8367.77<br>8ms | 8643.58<br>16ms |
| 10000^2 n=100 Times: 25 | 95770.32<br>0ms | 87858.33<br>75ms | 83763.19<br>2ms | 96857.96<br>4ms | 82422.12<br>13ms | 81188.59<br>83ms | 82337.28<br>9ms | 86432.16<br>17ms |
| 100^2 n=1000 Times: 5 | 2862.31<br>25ms | 2746.12<br>14182ms | 2600.06<br>38ms | 3026.04<br>74ms | 2498.84<br>474ms | 2476.01<br>13534ms | 2541.29<br>253ms | 2625.38<br>437ms |
| 1000^2 n=1000 Times: 5 | 29206.44<br>23ms | 27554.45<br>15477ms | 26263.4<br>39ms | 30571.08<br>88ms | 25007.91<br>370ms | 24996.98<br>15950ms | 25602.74<br>388ms | 26532.51<br>588ms |
| 10000^2 n=1000 Times: 5 | 288439.65<br>21ms | 279472.31<br>15339ms | 259208.87<br>36ms | 306271.79<br>83ms | 249170.12<br>308ms | 249677.21<br>15015ms | 253499.32<br>360ms | 263499.72<br>639ms |

## Discussion

As demonstrated by the data, of the 4 initial algorithms, Bubble path created the best solutions, performing about 2-5% better than Nearest Neighbor all, which in turn performed almost 10% better than Nearest Neighbor 1. Unexpectedly, Nearest Insertion performed the worst in terms of length, performing on average slightly worse than Nearest Neighbor. Even after reviewing, Nearest Insertion did give correct solutions, and upon debugging appeared to perform as expected, so the algorithm is thought to be correct.

In terms of runtime, Nearest Neighbor was the fastest. Then BubblePath next, and Nearest Insertion third, which were the O(n^2) algorithms. Then Nearest Neighbor All executed about n times slower than the other three algorithms, which is in line as it was a O(n^3) algorithm.

When combining these algorithms with 2-opt, their performance got much closer, all creating graph lengths on average within 5% of each other. However, now the best performer was Nearest Neighbor All with about 2% shorter paths than Nearest Neighbor and Bubble Path, while again Nearest Insertion performed the worst. It seems that the insertion algorithms benefited less from the 2-Opt adition, which makes sense as insertion algorithms might clean up the places where the solution crosses itself, exactly what 2-opt cleans up later on. This heuristic is supported by the fact that Bubble Path only saw a 1-2% improvement on average compared to a greater than 10% improvement for Nearest

Neighbor algorithm. Further, the Bubble Path+Opt-2 ran faster than any other combination with 2-opt, suggesting Bubble Path solutions were already closer to 2-optimal.

Again, Nearest insertion ran the slowest of the O(n^2) algorithms, and gave the worst solutions on average.

## 4. Conclusion

It was quite informative exploring and testing a few approximation algorithms. The most interesting results were the comparisons between the insertion and non-insertion algorithms before and after 2-opt, because it indicated Bubble Path performed better than Nearest Neighbor Algorithms because it left fewer crosses. For a fast and reasonable O(n^2) approximation, I would choose the bubble path. However, I would like to explore other algorithms in the future to better understand different heuristics and approaches for this problem

## 5. Appendix

### What I would like to continue exploring

If I were to redo the project, I would do two things differently. First, I would make a formalized tester sooner, and one that I could choose what algorithm to test with. This way, I would have seen sooner how the different algorithms interacted, in particular that 2-opt tended to perform better with neighbor algorithms than for insertion algorithms.

Further, I would explore more algorithms online, and write out sudo code every time I programmed an algorithm, to solidify my understanding of the algorithm.

Last, I want to try to explore other k-opt programs and see how they compare for different graphs to gain more insight on the deficiencies of insertion and neighbor based algorithms.

### The Programming Environment

These algorithms were programmed in C# and ran in the “run” mode meaning the code was not optimized by the C# engine. A short summary of the program structure can be found at the top of “Program.cs”. The test data displayed in the Data section can be found at “TextFiles/Tests.txt”. If the project is wanted to be run, it must be done in the dotnet sdk with “dotnet run”, or by building the code.

Citations in the paper are of the form [n], with n being the number of the source in the Sources section

## Sources


[1] Kuo, M. (2023, November 8). *Algorithms for the travelling salesman problem*. RSS. https://www.routific.com/blog/travelling-salesman-problem

[2] Rosenkrantz, D. J., Stearns, R. E., & Lewis, P. M. (1970, January 1). *An analysis of several heuristics for the traveling salesman problem*. SpringerLink. https://link.springer.com/chapter/10.1007/978-1-4020-9688-4_3