Traveling Salesman Experiment

Team JS: Jonathan Irvin Gunawan, Khor Shi-Jie
November 14, 2014

1 INTRODUCTION

The study of traveling salesman problem (TSP) is a natural generalization of the Hamiltonian cycle problem. We are interested to identify a cycle within a weighted graph such that the cycle includes all vertices in the graph and each vertex is only visited once in the cycle. In addition, we would to identify the Hamiltonian cycle with the minimum sum of edge weight. A close analogue to this problem is the Chinese postman problem which seeks to identify a tour of minimum weight that traverses each edge in the graph at least once. While the exact solution to the Chinese postman problem can be obtained in polynomial time, it is surprising that a subtle change in the requirement of the problem makes the Travelling Salesman Problem unsolvable in polynomial time. In particular, we note that

Theorem 1. *TSP is NP-Hard.*

Proof. We will reduce a directed Hamiltonian Cycle Problem into TSP. Given a directed graph G, we construct a weighted complete directed graph G' which has the same vertices as G. In addition, we assign a weight of 1 to the edges in G which were originally in G', and a weight of ∞ otherwise. Suppose we found a solution to the TSP using graph G' that has finite weight. Then, each of the edges must have weight 1 and hence must be present in the original graph G. As such, the solution to the TSP in G' is also a solution to the Hamiltonian Cycle Problem in G. On the other hand, if the solution to TSP in G' has infinite weight, then there is no Hamiltonian cycle which only consist of edge weight 1. Hence, there is no Hamiltonian cycle in G. As such, the directed Hamiltonian Cycle Problem reduces to TSP. Since the directed Hamiltonian Cycle problem is NP-Hard, TSP is also NP-Hard. □

There has been considerable research done in the past in determining the exact solution for TSP in a graph. A naive algorithm that checks all possible Hamiltonian cycle will incur $O(n!)$ time. It is possible to improve the runtime to $O(n^2 2^n)$ using dynamic programming (i.e. Held-Karp algorithm), but even such algorithm will take significant time on graphs with small number of vertices[1]. Researchers have explored other approaches such as branch-and-bound algorithms and linear programming. Currently, the record for the highest number of vertices in a graph in which a TSP solution is found

[1] On a modern computer, it is possible to obtain the optimal solution using Held-Karp algorithm within 1 second for a graph with less than 18 nodes. [5] Any more vertices will incur a much higher runtime.

is approximately 85,900 nodes[2]. Nevertheless, this is a small quantity compared to many real world instances of TSP.

Due to the impossibility of find a polynomial-time algorithm to solve TSP and the relevance of TSP to real life problems, there has been a lot of research that seeks to find an efficient approximation algorithm with a small approximation ratio. Within our course, we have explored several approximation algorithm to solve the TSP, including a 2-approximation algorithm[3] based on constructing a minimum spanning tree (MST), and a 1.5-approximation algorithm that include the application of a minimum weight perfect matching algorithm. However, the effectiveness of these algorithms are only theoretical in nature. While the worst-case bounds for these algorithms are irrefutable, the optimality of these algorithm may differ from our expectations when we apply them to real world data (i.e. a 2-approximation algorithm may perform better on average than a 1.5-approximation algorithm). As such, this project aims to compare the suitability of various approximation algorithm when applied to graphs obtained from real world data or other random means.

2 Problem Formulation

Generally, there are four variants of TSP that one can seek to solve. We can either have a graph in which the distance function is a metric, or a graph with a generic distance function. In addition, we can also choose to allow repeated vertices within our solution. The four variants are summarized in the table below:

	Repeats	No Repeats
Metric	M-R TSP	M-NR TSP
General	G-R TSP	G-NR TSP

We have shown in class that M-R TSP, M-NR TSP and G-R TSP are equivalent. For our project, we are more interested in M-NR TSP as we are using data obtained from the real world. In addition, we will assume that the graph is non-directional, and the distance between any two vertices is well-defined.

For this project, we will implement four different approximation algorithms and profile the performances on a set of random graphs and real world data. The four approximation algorithms are:

1. *Nearest neighbour heuristics.* We start from a cycle with only one vertex in the cycle. Then, while there are vertices which are not included within our cycle, we choose a vertex which is closest our cycle. Then, we add the vertex to the cycle at a position whereby the increase in the length of the cycle is minimal. Once all the vertices are added to the cycle, we will obtain a $O(logn)$-approximate solution to the TSP (to be proven in the next section).

2. *Minimum spanning tree.* We will build a minimum spanning tree for the graph and conduct a DFS traversal on the graph. Within the DFS traversal, we will skip any vertices which has already been visited. This will produce a 2-approximate solution to the TSP (proven in class).

3. *1.5-approximate algorithm.* After finding the minimum spanning tree of the graph, instead of conducting a DFS, we add edges to the graph until all vertices in the graph have even degree. Then, we can find an Eulerian tour for the graph. The edges to be added will be selecting using a minimum weight perfect matching algorithm such as a modified Edmond's Blossom algorithm.[6]

[2] Found by Applegate et al. using Concorde TSP Solver. [1]

[3] This algorithm only applies for a metric TSP. The same algorithm can be applied to obtain approximate solutions to M-R TSP and G-R TSP as these variants are equivalent.

4. *2-opt algorithm.* This algorithm makes use of the fact that a cycle without crossing is shorter than a cycle with crossing under the assumption of triangle inequality. We will refine any random cycle until there can be no more improvement to the weights. This algorithm is an $O(\sqrt{n})$-approximate algorithm on average case for a metric space. [4] In a normed vector space (which applies for the Euclidean metric), the algorithm is an $O(\log n)$-approximate algorithm.

We are interested in the optimality of the solution produced by the above four algorithms, as well as the runtime in producing the solutions. In particular, we want to know how well these algorithms will perform on random graphs and real world data.

In our project, we will first implement methods to generate random graphs and prepare 5 such graphs for experimentation. Then, we will parse real world data from (to be confirmed) and (to be confirmed) and store them in a weighted graph. We will apply each approximation algorithm on our generated graphs and obtain the weight of TSP solution produced by each algorithm. Each algorithm will be executed thrice and the average runtime taken for the algorithms will be calculated. Upon collecting these data, we will compare the results and conclude on the effectiveness of each approximation algorithm.

3 Algorithms and Theory

In this section, we will describe the implementation of the four approximation algorithms proposed in the previous section. We will assume that the graph is stored as an adjacency list.

3.1 Nearest Neighbour Heuristic

We will implement the version of nearest neighbour heuristics described in the pseudocode below:

Algorithm 1 Nearest Neighbour Heursitics

1: **procedure** NEAREST-NEIGHBOUR-HEURISTICS($G = (V, E)$) ▷ Returns the TSP approximation
2: Set $visited[v] \leftarrow$ false, for all $v \in V$
3: $TSP \leftarrow \emptyset$
4: $u_0 \leftarrow 1$ ▷ Choose vertex 1 as the starting point
5: $u \leftarrow u_0$
6: $visited[u] \leftarrow$ true
7: **while** there are unvisited vertices **do**
8: $v \leftarrow$ unvisited vertex that is closest to u
9: $TSP \leftarrow TSP \cup (u, v)$
10: $visited[v] \leftarrow$ true
11: $u \leftarrow v$
12: **return** $TSP \cup (u, u_0)$

For any walk P, define $cost(P)$ to be the total weight of edges included in walk P. We also define $d(u, v)$ to be the weight of the edge between u and v (which exists based on our assumption that our graph is complete). In addition, let NNH be the tour returned by the algorithm given above. We will prove that:

Theorem 2. *Algorithm 1 is an $O(\log n)$-approximation algorithm.*

Proof. The proof is adapted from [7]. For every vertex $u \in V$, let l_u be the length of the edge leaving vertex u to vertex v, which is the nearest unvisited vertex from u from the tour generated by the nearest neighbour heuristic.

Let $(x_1, x_2, \cdots, x_n)$ be a permutation of $\{i \mid 1 \leq i \leq n\}$ such that $l_{x_i} \geq l_{x_{i+1}}$ for any i satisfying $1 \leq i \leq n$. We first prove the following lemmas.

Lemma 1. *For any pair of vertices* (u, v), $d(u, v) \geq \min(l_u, l_v)$

Proof. Consider the following two cases:

Case 1: If vertex u was visited in NNH before vertex v, then v was a candidate for the closest unvisited vertex from u. Therefore, edge (u, v) is no shorter than the edge leaving vertex u in NNH. Since the edge that is chosen has weight l_u due to the greedy property of the algorithm, we conclude that $d(u, v) \geq l_u \geq \min(l_u, l_v)$.

Case 2: If vertex u was visited in NNH after vertex v, then u was a candidate for the closest unvisited vertex from v. Therefore, edge (u, v) is no shorter than the edge leaving vertex v in NNH. Since the edge that is chosen has weight l_v due to the greedy property of the algorithm, we conclude that $d(u, v) \geq l_v \geq \min(l_u, l_v)$.

Either case 1 or case 2 must be true. Therefore, $d(u, v) \geq \min(l_u, l_v)$ □

Lemma 2. *For any vertex* u, $2l_u \leq OPT$

Proof. Consider breaking the optimal tour into two disjoint paths P and Q, where P starts from u and ends at v, and Q starts from v and ends at u. Note that,

$$OPT = cost(P) + cost(Q)$$

Due to the triangle inequality, we note that $d(u, v) \leq cost(P)$ and $d(u, v) \leq cost(Q)$. Therefore,

$$2d(u, v) \leq OPT$$

Hence,

$$2l_u \leq OPT$$

□

Lemma 3. *For any* k *satisfying* $1 \leq k \leq n$, $OPT \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_{x_i}$

Proof. Let H be the subgraph defined on the set of vertices $\{x_i \mid 1 \leq i \leq \min(2k, n)\}$. Let T be the tour in H which visits the vertices in the same order as the optimal tour. By triangle inequality, every edge (u, v) in T have a length of less than or equal to the length of any path from u to v in the original graph. Therefore,

$$\begin{aligned} OPT &\geq COST(T) \quad \text{since } T \text{ is a part of } OPT \\ &\geq \sum_{i=1}^{\min(2k,n)} \min(l_{x_i}, l_{x_{i+1}}) \quad \text{from Lemma 1. We assume that } x_{\min(2k,n)+1} = x_1. \end{aligned}$$

Note that each l_{x_i} can appear at most twice in the summation above, since a vertex can only be the endpoint of at most 2 edges. We can attain the lower bound of the inequality by choosing the nodes with the smallest value for l_{x_i}. Using the fact that $l_{x_i} \geq l_{x_{i+1}}$ and since each node can only be chosen twice, we have

$$OPT \geq \sum_{i=1}^{\min(2k,n)} \min(l_{x_i}, l_{x_{i+1}}) \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_{x_i}$$

□

By Lemma 3

$$OPT \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_{x_i}$$

$$\sum_{k=0}^{\lceil \log n \rceil - 1} OPT \geq \sum_{k=0}^{\lceil \log n \rceil - 1} \left(2 \sum_{i=2^k+1}^{\min(2^{k+1},n)} l_{x_i} \right)$$

$$\geq 2 \sum_{k=0}^{\lceil \log n \rceil - 1} \left(\sum_{i=2^k+1}^{\min(2^{k+1},n)} l_{x_i} \right)$$

$$(\lceil \log n \rceil) OPT \geq 2 \sum_{i=2}^{n} l_{x_i}$$

By Lemma 2

$$OPT \geq 2 l_{x_1}$$

Therefore,

$$(\lceil \log n \rceil + 1) OPT \geq 2 \sum_{i=1}^{n} l_{x_i}$$

Therefore,

$$\frac{1}{2} (\lceil \log n \rceil + 1) OPT \geq \sum_{i=1}^{n} l_{x_i}$$

$$\frac{1}{2} (\lceil \log n \rceil + 1) OPT \geq COST(NNH)$$

Therefore, NNH is $O(\log n)$-approximation algorithm □

3.2 Minimum Spanning Tree

The construction of a 2-approximate tour using minimum spanning tree is exactly the same as the one introduced in lecture. As such, we will not restate the algorithm in this report. For this project, we will implement Kruskal's algorithm which generates a minimum spanning tree in $O(m \log n)$ time. This is sufficiently fast for a graph of no more than 10,000 vertices.

To generate the MST for a graph of size more than 10,000 vertices, we need to make use of the fact that the graphs which we will be analysing are complete, metric graphs. In this case, it can be proven that the minimum spanning tree of the graph is the minimum spanning tree of every Delaunay triangulation of the graph.[8] It is possible to generate the Delaunay triangulation of a graph in $O(n \log n)$ time. Since the triangulated graph is planar, it only has $O(n)$ edges. Running Kruskal's algorithm on the new graph will yield a runtime of $O(n \log n)$. The total runtime for the algorithm will be $O(n \log n)$, which can be used to process graphs with size up to 1,000,000 vertices comfortably.

3.3 2-opt Algorithm

The 2-opt algorithm is an approximation algorithm that seeks to improve on the weight of a given tour. Suppose we have already found a tour $x_1 x_2 \cdots x_n x_1$ for the given graph. For every pair of numbers i, j that satisfy $1 < i+1 < j \leq n$, we can construct a new tour $x_1 x_2 \cdots x_i x_j x_{j-1} \cdots x_{i+1} x_{j+1} x_{j+2} \cdots x_n$. This new cycle is formed by deleting the edges (x_i, x_{i+1}) and (x_j, x_{j+1}), and adding the edges (x_i, x_j) and (x_{i+1}, x_{j+1}). As such, there will be an improvement on the length of the tour if

$$d(x_i, x_j) + d(x_{i+1}, x_{j+1}) < d(x_i, x_{i+1}) + d(x_j, x_{j+1}).$$

The 2-opt algorithm is described in the following pseudocode:

Algorithm 2 2-opt Algorithm

```
1:  procedure 2-OPT-ALGORITHM(G = (V, E))                    ▷ Returns the TSP approximation
2:      TSP ← Random Tour in G
3:      while TSP was improved do
4:          i ← 1
5:          j ← i + 2
6:          while i ≤ n − 2 do
7:              while j ≤ n do
8:                  newTSP ← x_1 x_2 ··· x_i x_j x_{j−1} ··· x_{i+1} x_{j+1} x_{j+2} ··· x_n
9:                  change ← d(x_i, x_{i+1}) + d(x_j, x_{j+1}) − d(x_i, x_j) + d(x_{i+1}, x_{j+1})
10:                 if change > 0 then
11:                     TSP ← newTSP
12:                 j ← j + 1
13:             i ← i + 1
14:     return TSP
```

To improve the runtime of the algorithm, we can start with a TSP derived from the nearest neighbour heuristics. Note that each improvement on the TSP incurs a runtime of $O(n^2)$. Since the nearest neighbour heuristic is already an $O(\log n)$-approximation algorithm, we expect that there will not be a large number of improvements conducted on the initial tour.

The solution to the 2-opt algorithm can be further improved using local search algorithms. The global optimum to the TSP is known to exhibit a "big valley" structure, in which the global optimum is located in the middle of a cluster of local optima.[2] With this property, a local search algorithm allows us to escape from the local optimum in a gradient that leads to the global optimum. For this project, we will not conduct a gradient search to identify the best modification to a local optimum to move towards a global optimum. Instead, we will mutate a local optimum by conducting a 2-opt modification twice on random pairs of edges. Upon experimenting on small graphs (in particular, graphs with 20 vertices in which an exact solution can be found using dynamic programming), we have found that our local search approach converges to the global optimum within 0.3s[4]. The pseudocode for the improved algorithm is presented in the next page.

[4]The same algorithm is used to solve a challenge posted in CS2010R in which students are required to submit a code that returns the exact solution to the TSP for 30 vertices. The solution got accepted within 7 seconds of running time for 20 test cases.

Algorithm 3 2-opt Algorithm with Local Search

procedure 2-OPT-ALGORITHM-WITH-LOCAL-SEARCH($G = (V, E)$)
    $TSP \leftarrow$ Random Tour in G
    while Time elapsed is less than 10s **do**
        $currentTour \leftarrow$ MUTATE-TSP(TSP)
        $i \leftarrow 1$
        $j \leftarrow i + 2$
        while $i \leq n - 2$ **do**
            while $j \leq n$ **do**
                $newTSP \leftarrow x_1 x_2 \cdots x_i x_j x_{j-1} \cdots x_{i+1} x_{j+1} x_{j+2} \cdots x_n$ ▷ Modified from $currentTour$
                $change \leftarrow d(x_i, x_{i+1}) + d(x_j, x_{j+1}) - d(x_i, x_j) + d(x_{i+1}, x_{j+1})$
                if $change > 0$ **then**
                    $TSP \leftarrow newTSP$
                $j \leftarrow j + 1$
            $i \leftarrow i + 1$
    return TSP

procedure MUTATE-TSP(T) ▷ Returns a tour which undergoes two random 2-opt operations
    $i \leftarrow$ Random Vertex
    $j \leftarrow$ Random Vertex
    $k \leftarrow$ Random Vertex
    $l \leftarrow$ Random Vertex
    $T \leftarrow x_1 x_2 \cdots x_i x_j x_{j-1} \cdots x_{i+1} x_{j+1} x_{j+2} \cdots x_n$
    $T \leftarrow x_1 x_2 \cdots x_k x_l x_{l-1} \cdots x_{k+1} x_{l+1} x_{l+2} \cdots x_n$
    return T

3.4 1.5 Approximation Algorithm

The construction of a 1.5-approximate tour using minimum spanning tree and minimum weight perfect matching is exactly the same as the one introduced in lecture. As such, we will not restate the algorithm in this report. We will use the $BlossomV$ algorithm [6] which finds the minimum weight perfect matching for a graph within $O(mn\log(n))$ time, which is equivalent to $O(n^2\log(n))$ for a tree.

4 Implementations and Experiments

In our experiments, we will not implement the 1.5 approximation algorithm as the implementation was too complicated. As such, we will only compare the performance of the minimum spanning tree algorithm, nearest neighbour heuristics and 2-opt algorithm.

Firstly, we will generate 30 random graphs using our random graph generator. The first 10 graphs have size 15, the next 10 graphs have size 1,000, and the last 10 graphs have size 10,000. Each node in the graph represents a latitudinal and longitudinal coordinate of a point. This is done to maintain consistency with the graph generated by real life data.

We will then conduct the following four experiments:

- Experiment 1: We will check if the approximation ratio for the minimum spanning tree algorithm and the nearest neighbour heuristic is accurate. This is done by conducting the above two algorithms on 10 small graphs and comparing the length of tour generated with the optimal solution. The optimal solution will be generated using the Held-Karp dynamic programming solution which runs in $O(n^2 2^n)$. We will then conduct a t-test to conclude that the approximation ratio is observed.
- Experiment 2: We will run the 2-opt algorithm with local search on one small-sized graph, one medium-sized graph and one big-sized graph for 500 seconds respectively. Then, we will plot the cost of tour generated by the algorithm against the running time of the algorithm. This is done so that we can estimate a good running time for the 2-opt algorithm with local search to be used in experiment 3.
- Experiment 3: We will run all 3 algorithms on all 30 random graphs generated. Then, we will tabulate the cost of tour and the runtime for each algorithm.
- Experiment 4: We will conduct experiment 3 on two graphs obtained from real life data.

Our program was implemented in C++ and can be found attached with this submission. We will use a 2013 Macbook Air which runs on a 1.3 GHz Intel Core i5 with 8 GB 1600 MHz DDR3 RAM to conduct the experiments.

4.1 Experiment 1

The following table compares the cost of tour generated by the minimum spanning tree algorithm and the nearest neighbour heuristics with the optimal cost.

	Minimum Spanning Tree	Nearest Neighbour Heuristics	Optimal Cost
Graph 1	4928315.180	3866649.612	3866649.612
Graph 2	5425169.500	4066528.191	3891322.038
Graph 3	5133344.897	5465683.472	4246799.680
Graph 4	3854912.065	4221832.474	3580902.145
Graph 5	4011702.855	4221044.957	3627174.778
Graph 6	5167080.062	4266919.237	3620991.298
Graph 7	4362921.176	4082391.416	3601182.450
Graph 8	5172227.620	4421372.309	3830178.789
Graph 9	5008799.294	4395268.315	4037190.385
Graph 10	5260510.214	5173134.690	4026714.264

The following table depicts the ratio of the cost of the generated tour to the optimal cost.

	Ratio of Minimum Spanning Tree	Ratio of Nearest Neighbour Heuristics
Graph 1	1.275	1.000
Graph 2	1.394	1.045
Graph 3	1.209	1.287
Graph 4	1.077	1.179
Graph 5	1.106	1.164
Graph 6	1.427	1.178
Graph 7	1.212	1.134
Graph 8	1.350	1.154
Graph 9	1.241	1.089
Graph 10	1.306	1.285

Let μ_{MST} be the average approximation ratio for the minimum spanning tree algorithm. Let H_0 be the hypothesis that $\mu_{MST} > 2$, and let H_1 be the hypothesis that $\mu_{MST} \leq 2$. We assume that the approximation ratio for the minimum spanning tree follows a normal distribution. The t-statistic,

$$t = \frac{2 - 1.2597}{0.11487/\sqrt{10}} = 20.38$$

and hence we reject the null hypothesis at 99% confidence interval.

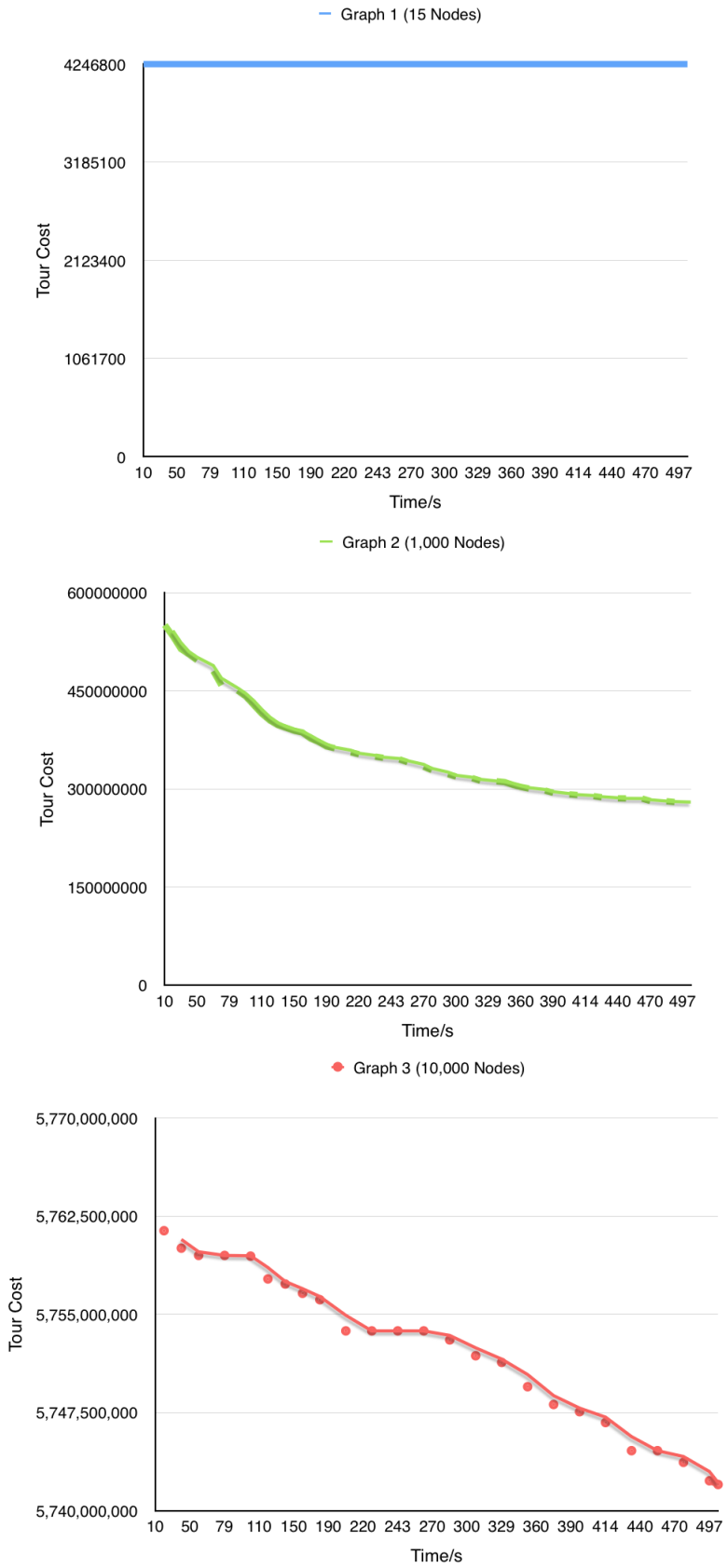
Next, let μ_{NNH} be the average approximation ratio for the nearest neighbour heuristics. Let H_0 be the hypothesis that $\mu_{NNH} > \log 15 = 3.9$, and let H_1 be the hypothesis that $\mu_{NNH} \leq 3.9$. We assume that the approximation ratio for the nearest neighbour heuristics follows a normal distribution. The t-statistic,

$$t = \frac{2 - 1.1515}{0.092081/\sqrt{10}} = 29.14$$

and hence we reject the null hypothesis at 99% confidence interval. Both results above suggest strongly that our implementation of the minimum spanning tree algorithm and nearest neighbour heuristics follow the theoratical ratio of approximation.

4.2 Experiment 2

The following charts depicts the cost of tour returned by the 2-opt algorithm with local search when it is being ran for 500 seconds.



There are several observations to be made from the graphs:

1. For a small-sized graph, the 2-opt algorithm with local search returns the optimal solution immediately. This solution is compared with the optimal solution produced by the Held-Karp

algorithm.

2. For the medium-sized graph, as time increases, the cost of tour returned by the algorithm decreases at a decreasing rate. The cost returned by the algorithm starts to plateau at around 330s.

3. For the large-sized graph, the algorithm has to be ran much longer before it approaches the optimal solution.

5 Conclusions

6 Bibliography

References

[1] D. L. Applegate, R. M. Bixby, V. Chavatal, and W. J. Cook. *The Traveling Salesman Problem*. Princeton University Press, 2007.

[2] Kenneth D. Boese, Andrew B. Kahng, and Sudhakar Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Oper. Res. Lett.*, 16(2):101–113, September 1994.

[3] J. Clark and D. Holton. *A First Look at Graph Theory*. World Scientific, 2005.

[4] C. Engels and B. Manthey. Average-Case Approximation Ratio of the 2-Opt Algorithm for the TSP. *Operations Research Letters*, 2009. Retrieved from `http://wwwhome.math.utwente.nl/~mantheyb/journals/ORL_EngelsManthey_2Opt.pdf`.

[5] S. Halim and F. Halim. *Competitve Programming 3*. lulu, 2013.

[6] V. Kolmogorov. Blossom V: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 2009. Retrieved from `http://pub.ist.ac.at/~vnk/papers/blossom5.pdf`.

[7] S. S. Ravi. *Fundamental problems in computing essays in honor of Professor Daniel J. Rosenkrantz*. Springer, Dordrecht, 2009.

[8] Robert Sedgewick and Kevin Wayne. Minimum spanning tree lecture notes, Spring 2007. Retrieved from `http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/19MST.pdf`.