An evaluation of solution quality among various Traveling Salesman Problem algorithms

Asra Yousuf
School of Computational Science & Engineering
Georgia Tech
Atlanta, Georgia
ayousuf@gatech.edu

Courtney Wong
Wallace H. Coulter Department of Biomedical Engineering
Georgia Tech
Atlanta, Georgia
wong.courtney@gatech.edu

Puting Yu
School of Computational Science & Engineering
Georgia Tech
Atlanta, Georgia
pyu73@gatech.edu

Yu-Ho Hsieh
School of Computational Science & Engineering
Georgia Tech
Atlanta, Georgia
yhhsieh@gatech.edu

ACM Reference Format:
Asra Yousuf, Courtney Wong, Puting Yu, and Yu-Ho Hsieh. 2018. An evaluation of solution quality among various Traveling Salesman Problem algorithms. In , .

1 INTRODUCTION

The Traveling Salesman Problem (TSP) has been a topic of immense interest in the field of theoretical computer science since its formalization in the 1930s due to its versatility in solving everyday route optimization problems and its unfailingly elusive optimal solution. Similar to other NP-hard problems, various heuristics and approximation algorithms are known to obtain viable minimal-cost routes between a set of points, each with their own trade-offs and caveats with respect to computation time and solution quality. The purpose of this study is to implement and evaluate the performance of four different algorithms using three classic approaches: branch-and-bound, approximation, and local search. Runtime and solution error were both considered in the empirical analysis. The results of this study indicate that 3-opt local search provides the most accurate solutions within a 10-minute window and quickest runtime, suggesting its candidacy as the best algorithm among the four under investigation for estimating the optimal TSP tour.

2 PROBLEM DEFINITION

Given the coordinates of n cities and a pairwise distance function* $d(u, v)$ defined for every pair of cities, find the shortest simple cycle that visits all n cities exactly once.

*In this study, both Euclidean and geographical distance calculations are considered.

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

3 RELATED WORKS

The traveling salesman problem was first conceived in 1832 but was not formalized mathematically until 1930 [1]. In the 1950s and 1960s, the TSP problem became popular in the math, computer science, physics and chemistry fields. The original solution proposed in the 1950s was capable of solving an instance with a size of 49 cities. For decades, several different approaches were proposed, from the branch-and-bound algorithm [7] which bounds the problem using multiple optimal lengths and partial solution construction cost, to using a minimum spanning tree to achieve a 2-times approximation. At the same time, TSP also can be formulated as Integer Linear Programming problem. In 1976, Christofides algorithm gave an approximation guarantee of 1.5 times the optimal solution. *TSPLIB* was published online in 1991 provides instances of TSP that serves as the benchmark of the most current research related to this problem.

Exact Algorithm

The brute force algorithm has time complexity in $O(n!)$, which is highly impractical even for cases where the number of cities is as small as 20. However, the branch-and-bound algorithm is workable for size 40 to 60, reminiscent of linear programming, which works well for up to 200 cities. An instance of up to 85,900 was solved using branch-and-cut (cutting plane method) algorithm by Applegate et al in 2006 [1].

Heuristic and Approximation Algorithms

Modern approximations can find a solution with 2-3% error within reasonable time.[8] The Nearest Neighbor algorithm, known as greedy algorithm, has approximation factor $\theta(\log V)$ where V is the number of cities. [4] Christofides' algorithm is based on the original minimum spanning tree algorithm and achieves a solution that is at most 1.5 times worse than the known optimal [2]. Christofides' algorithm is aimed at improving the lower bound of TSP and applies the concepts of an Eulerian graph and Eulerian tour within $O(n)$ time to achieve the approximation.

Iterative Improvement (Local Search)

In addition to the exact algorithms and approximation algorithms, local search techniques have been extensively applied to solve the traveling salesman problem. This class of algorithms iteratively improve the current solution by searching for a better one in its predefined neighborhood. The algorithm stops when there is no better solution in the given neighborhood or if a certain number of iterations has been reached. The pairwise exchange, known as the 2-opt technique, can also be applied to find the local optimal solution of TSP. The exchange can be generalized to k-opt, giving rise to 3-opt, which is the most popular approach introduced by Shen Lin of Bell Labs in 1965. Stepping away from fix-opt, the variable-opt method was proposed in 1972 and was one of the most reliable heuristic algorithm to solve TSP problem. [9] These iterative method can be optimized through randomization and the obtained best solutions are extremely close to optimal solutions under for problem sizes of 700 to 800 cities.

4 ALGORITHMS

Four different algorithms were selected and implemented to compute the best solution to TSP: branch-and-bound for an exact solution, MST-Approx for an approximate solution, and 2-opt and 3-opt for local search heuristic solutions.

Branch-and-Bound

Overview. The idea behind the branch-and-bound algorithm contains two parts. First, we construct a tree to enumerate all the possible solutions. Second, we prune the sub-tree from a certain node if it is known that the solution (the leaves) from the node cannot be best solution. The tree construction process is divided into four steps. The first is to reduce the adjacency matrix to compute the lower bound of the current node, then expand the current node by either selecting one candidate edge into TSP solution or not select that edge. Third is to update the adjacent matrix as well as add up additional lower bound to the expansion (sub-nodes) of current node. Finally, we pick the lowest cost node and repeat the first step again until every city is visited.

Implementation. The branch-and-bound algorithm is similar to the brute force method. Hence, an effective and efficient implementation is extremely important. The discussion is divided into 2 parts. The first part is to develop the algorithm details and the second part is the implementation tricks.

Matrix Cost Reduction. Matrix cost reduction is the process of obtaining the lower bound of each subproblem. For each row and column, the minimum value can be subtracted, as it does not change the optimal route. The sum of all subtracted numbers becomes the lower bound of the sub problem. After the reduction, the partial solution is expanded through edge selection.
Tricks: During the implementation, take infinity as zero, meaning if all the numbers in a row/column equal to infinity, subtract zero from the row/column.

Edge selection. The next step is to expand the subproblem. An edge selection means an expansion. We have two different methods to expand: multi-select or binary-select. For multi-select, all edges that are connected to the current node index are selected and added into the frontier queue. The upside of this method is simplicity, and it can obtain the best solution faster. However,it will end up storing too many unused nodes in the frontier queue. For binary select, the node is expanded by considering two cases: add one sub-node picking a certain edge or do not add it. The advantage of this method is space usage, as only two nodes are appended into the frontier queue each time. However, more time may be required to achieve best solution because of fewer candidates in the frontier queue. In this implementation, the binary select method was chosen because the multi-select method exceeded the heap memory capacity of the computer.
Tricks: It is unnecessary to select the edge which connects to its parent node index. Rather, select any of the edges that maximize the lower bound difference between the two children nodes. This method allows the algorithm to "concentrate" on digging into one partial solution and may find a new solution more quickly. The solution may update the best cost so far and prune more branches.

Update Matrix. After edge selection, the matrix is updated accordingly. In order to avoid visiting same city twice, corresponding row and column of the selected edge are updated to be infinity. At the same time, cycle formation is avoided by changing the $(dest, ori)$ entry to infinity. For the not-select edge sub-node, the $(ori, destination)$ entry is set to infinity in order to remove the edge and perform the cost reduction again to update the lower bound.
Tricks: Consecutive edges are no longer selected, and hence the selected edge must be connected with the edges in the solution to relax both ends and update the correct row and column.

Sub-node Construction. After the edge selection, the sub-node is constructed through the lower bound, differentiated accordingly, and placed into the frontier queue before the algorithm returns to the matrix cost reduction step.
Tricks: During the node construction, it may be necessary to store a copy of matrix in each node because the state must be memorized. However, the original matrix may be reused if the node will be discarded immediately to save the space usage.

Computation Tricks. If the above design is followed, there is a higher chance to encounter $OutOfMemory$ problem because the growth of frontier queue is still too fast. Hence, a buffer idea was introduced into the implementation. A node is expanded only when its' depth exceeds a certain threshold or the frontier queue size is below a certain threshold. Otherwise, the non-select expansion sub-node is stored into the buffer and will not be expanded until most of the candidate nodes in frontier queue are processed.

Time & Space Complexity. The branch-and-bound algorithm is an optimized brute force algorithm. Hence in worse case, it requires exponential time to compute the solution. At the same time, even after reducing the space usage by pruning the tree, the worst case space complexity is still exponential.

Pseudo-code. The following section outlines the pseudo-code for the branch-and-bound algorithm.

Algorithm 1 branch-and-bound algorithm

Input: *Graph of Cities, cutoff Time*
Output: *Minimum cost of tour, Optimal Tour*

```
matrix ← graph.getMatrix()
FQ ← frontier queue()// lower bound as priority
best ← max integer
Root ← node()
while (size of FQ != 0 and time < cut time) do
    node ← pop from FQ
    children ← node.genChilds()
    if children = null then:
        if node.getCost() ≤ best then:
            best = node
    edgeSelect ← children[0]
    edgeNotSelect ← children[1]
    if cost of edgeSelect ≤ best then:
        FQ ← edgeSelect
    if cost of edgeNotSelect ≤ best then:
        FQ ← edgeNotSelect
procedure genChilds()
    if all cities are visited then
        return (null)
    //Select edge node
    copy matrix ← copy(matrix)
    Reduce and update copy matrix
    Select edge
    Update new lower bound
    child[0] ← new node(new lower bound, copy matrix)
    //Not select edge node
    Remove edge
    Update original matrix
    Update lower bound
    child[1] ← new node(lower bound, original matrix)
    return (child[0], child[1])
```

MST-Approx

Overview. The MST-Approximation algorithm investigated in this study is characteristic of typical approximation algorithms: it strives to produce an answer quickly at the cost of accuracy, within a defined guarantee. The implementation is based on Christofides' algorithm and relies on the concept of a minimum spanning tree (MST) to obtain a minimal-cost simple cycle. Following MST construction, an Eulerian tour is constructed and shortcuts are taken to obtain the resulting Hamiltonion TSP tour with a guaranteed cost and reasonable runtime. The general steps taken is outlined in the list below and explained in detail in the following subsections. The role of randomization that is introduced at several parts in the implementation is further explored by repeating the algorithm with different random values in order to obtain improved solutions at the cost of slightly increased runtime.

(1) Compute the minimum spanning tree M of a given graph G
(2) Find the subset of nodes O that have an odd degree
(3) Find the minimum weight perfect matching P by pairing nodes in O with an edge
(4) Create a connected multigraph whose nodes all have an even degree (required for Eulerian traversal) by overlaying P onto M
(5) Compute the Eulerian circuit of the multigraph and exclude duplicate nodes to obtain the Hamiltonian TSP tour

MST Construction. A minimum spanning tree for a given connected undirected graph $G = (V, E)$ with n nodes and real-valued edge weights c_e is defined as the subset of edges $T \subseteq E$ that form a tree among every V whose sum of edge weights is minimized. In this study, Kruskal's algorithm was chosen for its simplistic implementation. While this approach is normally inferior compared to Prim's algorithm for dense graphs (in TSP, the given graph is always maximally dense as all nodes are interconnected), its runtime complexity is significantly reduced when edges are pre-sorted by weight, as sorting dominates the usual $O(ElogE)$ time complexity in Kruskal's. This advantage is made possible by the Priority Queue structure used to store edges during the file parsing phase. Additional efficiency is introduced in the Union-Find structure during tree formation, where union-by-rank and -compression are implemented to achieve a well-balanced, shortened tree that reduces traversal time when determining whether two different edges in T belong in the same disjoint set. The result of these runtime improvements is a total amortized cost of $O(E\alpha(V))$ for this portion of the algorithm.

Minimum Weight Perfect Matching. Each node in an Eulerian tour constructed from a graph will have an even degree–one outlet and one inlet. For a constructed MST M, this can be achieved by connecting all odd-degree nodes in M. The number of nodes in this subset, $|O|$, must be even because the total sum of degrees of every node in M itself must be even, as it is equal to the number of (integer) edges ($|T|$) times two. A perfect pairing of nodes in O must therefore exist. Edmond's Blossom algorithm and its many variations seen in literature provides a polynomial-time solution to obtain this optimal pairing. However, due to its runtime complexity of $O(V^3)$ and complicated implementation, a Greedy-approximation approach was taken instead: a random node is selected and paired to the nearest available neighbor iteratively to form an edge until all nodes in O have been paired. These pairings, P, are then unioned with T to create a new graph, $mstMatched$.

Eulerian Traversal. An Eulerian tour for a given graph is one that begins and ends at the same node and traverses every edge exactly once. This tour can then be converted into a Hamiltonion cycle for the same graph by excluding any duplicate nodes that are visited. For the purposes of TSP, this process is identical to performing a depth-first-search (DFS) on $mstMatched$. It is important to note that, due to the addition of edges in the matching step, cycles are likely to exist in the graph. Therefore, when implementing DFS, there must be a condition prior to exploring the next neighbor that ensures that it has not been visited *and* that it is not already in the LIFO queue.

Role of Randomization in Iterative Improvements. Due to the highly efficient nature of this approximation algorithm, an extra step was explored to find improved solutions without immensely damaging runtime. Randomization is introduced during minimum weight matching (selecting a random node to greedily choose its nearest

available neighbor) and DFS traversal (selecting the root and the next neighbor among all possible neighbors). Random combinations are tested within an inputted cut-off time, and improved solutions are recorded along with its corresponding runtime. This results in highly luck-based solutions in terms of runtime with a cost within (and likely much better than) the original accuracy guarantee. To remain in the true spirit of approximation algorithms where accuracy is sacrificed (to a guaranteed extent) for speed, however, an additional feature was implemented to exit out of the algorithm once a "good enough" solution is obtained: an improvement tolerance (TOL) that forces code termination if the solution has not been improved in the past TOL iterations. Decreasing the value of this tolerance results in better runtime and less accurate results. Increasing its value produces the inverse effect, but past a certain extent, runtime becomes no longer reasonable and a local search algorithm would be preferred. This idea is further explored in the results section.

Approximation Guarantee. This algorithm is a variation of Christofides' algorithm, in which total cost of the MST edges ($1 * MST$) plus the cost of the pairings ($MST/2$) is guaranteed ($1.5 * MST$). Since MST is the lower bound of the optimal TSP tour, Christofides' algorithm provides a 1.5-approximation guarantee. The algorithm in this study, however, is slightly different, as it uses a Greedy approximation pairing. Findings in literature [6] have proven empirically that the Greedy approach implemented has a tight approximation guarantee of $\theta(log|O|)$ of the optimal "perfect" pairing, where the subset $O \in V$ contains the odd-degree nodes in the constructed MST. In the absolute worse case, $|O| = |V|$, such as in a star-shaped MST with an odd-degree central node. The approximation guarantee of the implemented algorithm is therefore $1 + \frac{logV}{2}$, but this guarantee is not tight.

Time and Space Complexity. The overall complexity of this algorithm is $O(V^2 logV)$, where V is the number of nodes in the given graph. To derive this, the individual time complexity of core contributors of the MST-approximation algorithm is outlined below, along with the corresponding line numbers in the pseudo-code in the next subsection.

(1) computeMST: $O(E\alpha(V))$, lines 16-26
(2) findOdds: $O(V)$, $O(V)$, lines 27-32
(3) minWeightPairing:$O(V^2 logV)$, lines 33-39
(4) eulerTraversal: $O(V + E)$, lines 40-52
(5) shuffleNeighbors: $O(V * E)$, line 12

Lastly, both an adjacency matrix and an adjacency list are used to store the graph due to efficiency in different applications: the matrix form excels in quick retrieval of edge weights, while the adjacency list has the upper-hand during DFS traversal. The space complexity of this algorithm is therefore $O(V^2)$, dominated by the adjacency matrix structure.

Pseudo-code. The following section outlines the pseudo-code for the MST-Approximation algorithm.

Algorithm 2 MST-approximation algorithm

Input: *Graph of Cities, cutoff Time, seed*
Output: *Minimum cost of tour, Optimal Tour*

```
1:  MST ← computeMST(G)
2:  odds ← findOdds(mst)
3:  while runtime < cutoffTime do
4:      mstMatched ← minWeightPairing(g, mst, odds, seed)
5:      tour ← eulerTraversal(mstMatched)
6:      cost = 0
7:      for each edge ∈ tour do
8:          cost+ = edge.getCost()
9:      if cost < G.getCurrentBestCost() then
10:         G.addApproxResult(cost, runtime)
11:         G.setCurrentBestCost(cost, tour)
12:     MST.shuffleNeighbors(seed)
13:     if no improvements made in last TOL iterations then
14:         break
15: return G.getCurrentBestCost()
16: procedure computeMST(G)
17:     edgeList ← G.getEdgeList()
18:     MST = ∅
19:     for each v ∈ G do
20:         makeSet(v)
21:     while MST.numEdges() < G.getSize() − 1 do
22:         (node1, node2) = edgeList.poll()
23:         if findSet(node1)! = findSet(node2) then
24:             MST.addEdge(node1, node2)
25:             union(node1, node2)
26:     return MST
27: procedure findOdds(MST)
28:     odds = ∅
29:     for each v ∈ MST do
30:         if mst.getEdgeList(v).size()%2 == 1 then
31:             odds ∪ v
32:     return odds
33: procedure minWeightPairing(MST, odds, seed)
34:     while !odds.isEmpty() do
35:         v ← a random node in odds using seed
36:         w ← nearest neighbor of v in MST
37:         MST.addEdge(v, w) if not already connected
38:         odds \ (v, w)
39:     return MST
40: procedure eulerTraversal(MST, seed)
41:     root ← random node in MST using seed
42:     LIFO.push(root)
43:     visited[root] = 1
44:     while !LIFO.isEmpty() do
45:         cur = LIFO.pop()
46:         tour ∪ cur
47:         visited[cur] = 1
48:         for each neighbor of cur do
49:             if visited[nbr] == 0 and nbr not in LIFO then
50:                 LIFO.push(neighbor)
51:     tour ∪ root
52:     return tour
```

2-Opt

Overview. The 2-opt algorithm belongs to the family of local search methods designed to solve the traveling Salesman problem. The main idea behind the algorithm is to consider two edges (i.e. two pair of connected nodes) and replace them by connecting nodes forming the opposite edge. This results in a new tour who's cost may be different from the current tour, while still ensuring that all nodes are visited exactly once. The algorithm then compares whether the cost of the new tour is lesser than the cost of the current tour. If yes, it takes the new tour and repeats the process. The algorithm stops when no changes are observed in the tour cost beyond a certain point in time and the solution converges. Hence, it is built on the principle of local search wherein it starts with an initial random solution and explores the immediate neighbourhood of the solution to check if a better solution can be found. In our current approach, we have a set a cut-off time along with a threshold of 1000. If for a set of 1000 continuous runs, the best cost of the tour does not change for the better, we stop the algorithm and report the best cost found at that point in time. If an improvement is detected in any of the iterations, we reset the threshold and restart the algorithm. Potential improvements of the approach could involve restarting the algorithm when the local minima is reached and no change is detected after a set of iteration. Another technique that can be applied is the concept of Tabu Search. We can maintain a list of swaps that have happened recently and prevent the same swaps to happen in the next set of iterations. This would ensure that the algorithm does not keep searching for the optimal solution within the same neighbourhood.

Pseudo-code. The following section outlines the pseudo-code for the 2-opt algorithm

Algorithm 3 2-opt algorithm

Input: *Graph of Cities, cutoff Time, seed*
Output: *Minimum cost of tour, Optimal Tour*

```
1:  G ← Adjacency Matrix of distance between nodes
2:  initialRoute ← GenerateInitialRandomSolution
3:  while (tour cost decreases ) do
4:      for j ← 1 to numberOfCities: do:
5:          for k ← j+1 to numberOfCities do:
6:              alternateRoute ← twoOptSwap(G, j, k)
7:              if cost(alternateRoute) ≥ cost(currentRoute): then
8:                  setCurrentBestResult(alternateRoute,
                        cost(alternateRoute)
9:  procedure twoOptSwap(graph, j, k)
10:     currentRoute ← graph.getCurrentBestRoute
11:     alternateRoute ← EmptyList
12:     Add to alternateRoute all cities between currentRoute[1]
            and currentRoute[j] in order
13:     Add to alternateRoute all cities between currentRoute[j]
            and currentRoute[k] in reverse order
14:     Add to alternateRoute all remaining cities in currentRoute
            post index k in order
15:     return (alternateRoute)
```

Time & Space Complexity. For the purpose of estimating the time complexity of the 2-opt algorithm, two steps of the approach must be considered. The first step involves finding a neighbourhood solution and the second step deals with improving the solution until a local optima is reached. Since the algorithm for calculating the next best tour in the neighborhood considers all combinations of nodes in the graph, the complexity of executing each round of the algorithm is $O(n^2)$ where n is the number of nodes. The number of rounds executed is dependent on the problem set and it is difficult to predict the time that the algorithm may take to converge. The space complexity of the algorithm is $O(n^2)$ since a 2D array is used to store the adjacency matrix of the cities and a list data structure is being used to store the alternate routes calculated.

Choice of Algorithm. 2-opt is known to be one of the most popular local search algorithms for TSP. Historically, 2-opt [3] was one of the first successful algorithms to solve larger TSP instances. It is a local search algorithm whose neighbourhood is defined by the removal of two edges from the current tour and reconnecting two other edges to form a tour. One of the primary motivations behind selecting the 2-opt approach was due to the promising results generated by the algorithm over the decades [5] and the fact that it forms the essential building blocks for other local search techniques. Another motivation behind the selection was to also compare the performance of 2-opt and 3-opt techniques on the same dataset in terms of the quality of the solution and the time taken by each. A potential weakness of the algorithm, as with most local search algorithm, is that it may converge at the local optima which might not be the same as the global optima. Additionally, 2-opt works better with smaller datasets as compared to larger ones.

3-Opt

Overview. In this part, we aim to apply another local search method "Three-Opt" to approximate optimal solution by "local minimum solution". Furthermore, we aim to compare two local searches with same idea but different neighborhood. In the process, we will randomly select a route as our initial condition and start search possible candidate optimal solution around "neighborhood" of initial condition.
In the whole process, we will set an time spec as an ending point, and see how close and how far we are from the optimal solution when this spec is met.

Pseudo-code. The following section outlines the pseudo-code for the 3-opt algorithm.

Algorithm 4 3-opt algorithm

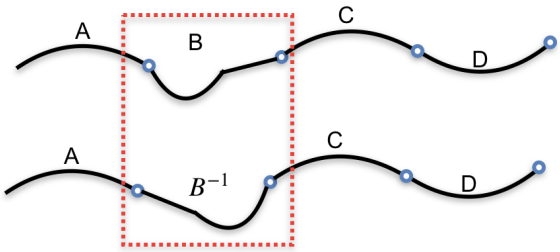
Input: *Graph of Cities, cutoff Time, seed*
Output: *Minimum cost of tour, Optimal Tour*

```
G ← Adjacency Matrix of distance between nodes
initialRoute ← GenerateInitialRandomSolution
initialize Best_result ← INT_MAX
while (all neighbors have not been visited ) do
    alternateRoute ← 3 − OptSwap(G, i, j, k)
    if cost(alternateRoute) ≤ Best_result: then
        store result into BestRoute and Best_result
procedure 3-OptSwap(graph,i, j, k)
    currentRoute ← graph.getCurrentBestRoute
    alternateRoute ← EmptyList
    Select three edges of CurrentRoute and split tour into four segments
    Find best combination between four segments
    If result is better than best_known result, update alternateRoute.
    Otherwise, return empty list.
    return (alternateRoute)
```

The core idea of local search is to "explore" neighborhood of a given initial condition, and try to approximate optimal solution by local optimum. However, this idea seems quite vague in terms of program execution, for example, "What is this neighborhood?" and "How to actually explore this neighborhood?". Three-Opt adopts the following strategies.
1. Given an initial route, delete three edges randomly, which splitting the initial tour into four parts, denote them by A,B,C,D according to its order in initial route. Thus, original route can be expressed as ABCD.
2. Define the neighborhood of initial route to be all possible combinations of A$\{B, B^{-1}, C, C^{-1}\}$ D.
(we only take two sets from $\{B, B^{-1}, C, C^{-1}\}$, and The positions of A,D are fixed. Moreover, B^{-1} represent the reversion of route B.)
3. Find minimum cost of neighborhood defined above, which we called as "local minimum".

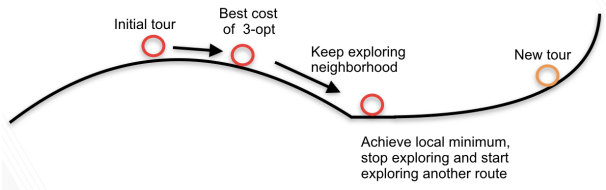
Time & Space Complexity. As we mentioned, we randomly deleted three edges in our tour and calculate the cost of new tour. Therefore, there are total n*(n-1)*(n-2) combinations of edges we should delete and calculate the resulting cost. Thus, the time complexity would be $O(n^3)$. For space complexity, the space for storing optimal tour dominates the space complexity, so the space complexity would be $O(n)$

Algorithm Tricks . As one can see in our pseudo-code, 3-opt basically involves permutation on segments of tour. That is, when we partition a tour into four segments, there are 8 possible combinations we have to calculate its cost and check if there is a better route. However, assuming we have 100 cities, if we calculate the cost of each combination, the time complexity would become $O(8 * n^3)$, which is far more larger than $O(n^3)$. However, we can reduce $O(8 * n^3)$ to $O(n^3)$ by following tricks. Assume we Partition tour into four segments A,B,C,D, since adjacency matrix for weight is symmetric (an undirected graph), the cost of B is the same as B^{-1}, C is the same as C^{-1}. For example, we can get the cost of tour AB^{-1}CD by A+B+C+D+"cost from A's ending point to B's starting point" + "cost from B^{-1}'s starting point to C's starting point", other things remains the same as ABCD. Therefore, once we get the cost of A,B,C,D, we can calculate each possible combination by calculating A,B,C,D.



B and B^{-1} have same cost, the only difference occurs at two endpoints

Another strategy we use here is "keep exploring neighborhood until local minimum is achieved". That is, given an initial tour, we start exploring its neighborhood and find a best neighbor, use this neighbor as initial tour and execute 3-opt, if cost is improved, keep doing this, otherwise, we assume the local minimum is achieved, select another tour and execute again.



5 METHOD

All benchmark instances were run in the terminal on a computer with no other tasks running in the background to ensure consistency and to minimize any interference with runtime. The impact on runtime due to any differences between physical computers used is assumed to be negligible. The following list summarizes the platform used to obtain the empirical results in the results section.

- OS: macOS
- CPU: i5
- RAM: 2GB
- Language: Java
- Compiler: javac

The evaluation criteria utilized to judge the relative solution quality among the four TSP algorithms studied was relative error. Relative error is defined as $\frac{(ALG-OPT)}{OPT}$, where ALG is the cost of the shortest route of the given data set found within a 10 minute runtime window, and OPT is the optimal, minimum cost across all four algorithms and values found in literature. runtime is also taken into consideration.

6 RESULTS

A comprehensive table of results across all four algorithms is shown in Table 1.

Branch-and-Bound

For branch-and-bound, at first we try out the multi-select expansion. The multi-selection expansion is able to finish instance with size below 10. For the size over 10, the algorithm ran out of heap memory. Hence, we choose from multi-select expansion to binary-select expansion. The binary selection is able to finish the instance size over 10 but it sticks with instance which size is over 16. The pruning helps reduce space usage, so We decide change the the edge selection method. The speed of computing 16 instance drops from 6 seconds to less than 1 second and it can solve instance over 20. However, with instance more than 40, the algorithm fails. As a result, we come up with the "batch" computation method which stores most candidates in the buffer without expanding them immediately. Finally, the branch-and-bound algorithm is able to work on arbitrary size of instance. In addition, the algorithm can find one solution of every instance within a second which helps prune the tree early. Depends on the distribution of adjacent matrix, our branch-and-bound implementation may finish instance with cities of 40 within 2 minutes (Boston.tsp) but also may not finish instance with cities of 30 (Philadelphia.tsp) before the cut off time (20 minutes). Usually we may not finish instance larger than 50 within 10 minutes.

MST-Approx

The empirical results produced by the MST-Approx algorithm is consistent with the general goal of approximation algorithms: efficient runtimes at the bounded expense of solution quality. It should be noted that the results for Approx summarized in **Table 1** are the best solutions in terms of cost found within a 10-minute window. Further analysis was performed to determine whether an earlier cutoff time should be used.

Figure 1 was generated by setting $TOL = 1$ to suppress the random Greedy improvements portion of the algorithm in order to obtain the average computation time ($n = 10$ runs per instance) of a single run in relation to instance size. The relationship confirms the $V^2 log(V)$ runtime complexity discussed in a previous section, and indicates that for even instances as large as $|V| = 230$, it is worthwhile to run this algorithm multiple times in the hopes of improving the solution by chance.

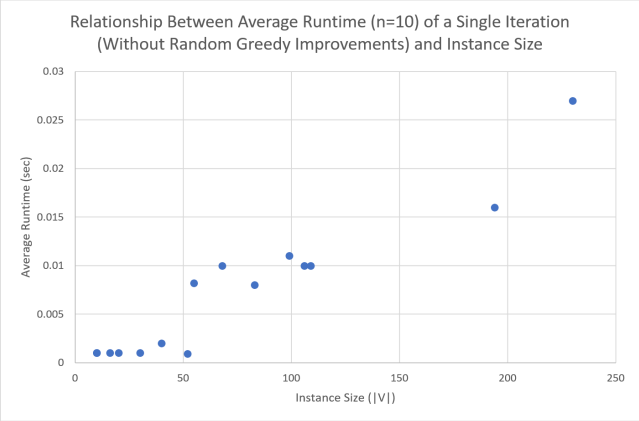


Figure 1: MST-Approx algorithm runtime vs instance size

Figure 2 shifts focus to a single, extremely large dataset of size $|V| = 4663$ whose optimal solution is known. This plot was generated by running ca4663 using $n = 10$ different seeds using a cutoff time of 300 seconds (to reduce computation time) and $TOL = -1$ to allow the program to run for the full time even if no improvements are made. These results indicate that the majority of random improvements made for even an extremely large dataset is obtained in the first 60 seconds and any improvements past this point can be dismissed as diminishing returns in realistic situations. Thus, a cutoff time of 60 seconds is recommended for this algorithm. In other words, if a time limit of one minute or less is enforced for solving TSP, MST-approx is an excellent choice.

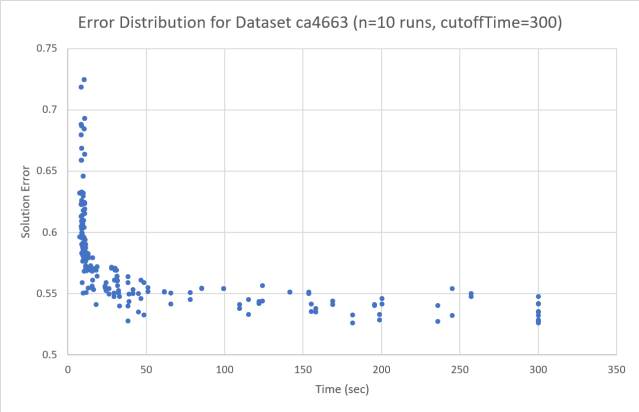


Figure 2: MST-Approx algorithm error distribution from iterative random Greedy improvements for dataset ca4663

2-opt

We performed a detailed empirical analysis on four of the 14 instances in **Table 1**: ulysses16, boston40, and berlin52, roanoke230. While we observed that for some runs of converged at the optimal solution in cases of Cincinnati, Berlin, ukansasState and Ulysses, a slight relative error is observed in the case of Roanoke, Berlin and Boston dataset. Hence, it can be observed that 2-opt performs

Instance	Branch-and-Bound			MST-Approx			2-Opt			3-Opt		
	Time (s)	Best ALG	Rel Err	Time (s)	Best ALG	Rel Err	Time (s)	Best ALG	Rel Err	Time (s)	Best ALG	Rel Err
cincinnati10	0.10	277952	0.0000	0.01	279537	0.0057	0.06	278206	0.0009	0.01	277952	0.0000
ukansasstate10	0.14	62962	0.0000	0.03	65518	0.0406	0.07	62962	0.0000	0.01	62962	0.0000
ulysses16	0.30	6859	0.0000	0.01	6980	0.0176	0.1498	6872	0.0019	0.01	6859	0.0000
atlanta20	0.20	2003763	0.0000	0.28	2003763	0.0000	0.15	2055406	0.0258	0.20	2003763	0.0000
philadelphia30	720.56	1395981	0.0000	128.02	1447566	0.0370	0.33	1429504	0.0240	0.30	1395981	0.0000
boston40	7.63	893536	0.0000	9.55	927078	0.0375	0.6684	928028	0.0386	2.30	893536	0.0000
berlin52	350.64	7542	0.0000	7.85	7932	0.0517	1.705	8150	0.0806	1.80	7542	0.0000
champaign55	331.96	52865	0.0042	95.76	53135	0.0093	1.81	54302	0.0315	0.20	52643	0.0000
nyc68	125.43	1616016	0.0392	150.61	1613855	0.0378	3.06	1626892	0.0432	13.20	1555060	0.0000
denver83	76.37	115658	0.1516	403.71	104461	0.0401	5.25	105486	0.0380	28.20	100431	0.0000
sanfrancisco99	9.99	946560	0.1614	102.93	907023	0.1129	7.65	841757	0.0328	27.20	815028	0.0000
umissouri106	207.69	149730	0.1183	258.84	145608	0.0875	11.25	139642	0.0430	31.30	133891	0.0000
toronto109	161.52	1594527	0.3535	529.78	1293571	0.0980	13.50	1246801	0.0583	67.20	1178095	0.0000
roanoke230	271.12	818588	0.2466	561.26	755753	0.1509	120.07	693664	0.0582	115.20	656670	0.0000

Table 1: Comprehensive table of results across four algorithms

better in term of the result on smaller datasets as compared to larger datasets with more nodes.

There were variations in solution quality that were also observed across these instances. As observed in **Figure 3** for Ulysses, the relative error was extremely low hence the variation in quality was in the range of 0 to 0.004 where as Berlin showed the highest variation in quality from 0.03 to 0.09. The figures for solution quality distribution in **Figure 4** depict the variation in quality as the running time is fixed. We observe that for longer time periods, the program is more likely to find an optimal solution. In addition, when the relative error is low, the number of 'solved' instances are few and the number of 'solved' instances increase linearly, in some cases, as the threshold for relative error is increased.

Lastly, the box plot in **Figure 5** provides a visualization of the run-time distribution of the various instances in case of 2-opt. Since there are huge variations in run-time across the different instances due to their size (Roanoke has 230 cities while Ulysses has only 16 points to visit), we have normalised the run-time to a scale of $0-1$ using min-max normalisation. As can be observed from the plot, the runtime for Boston and Berlin is fairly evenly distributed where as the runtime for Roanoke and Ulysses have a higher 3^{rd} quartile.

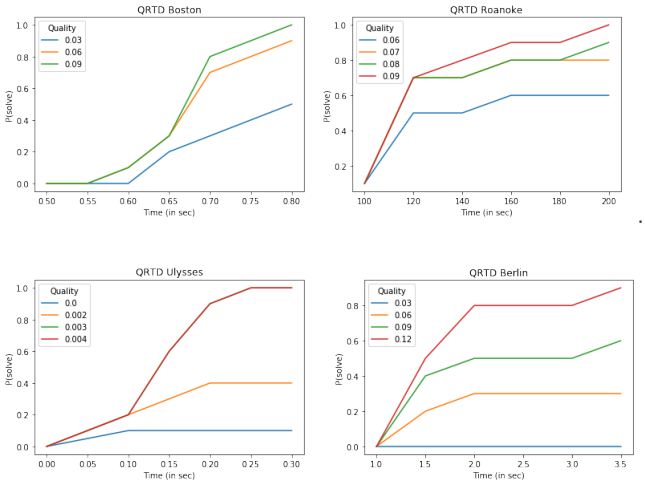


Figure 3: QRTD plots for 2-opt

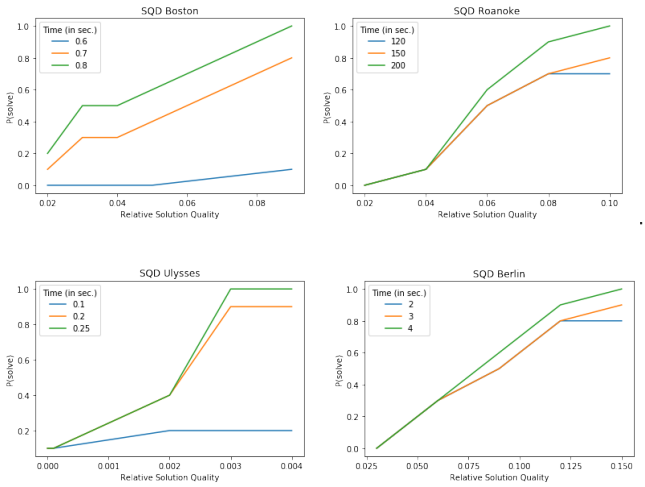


Figure 4: SQD plots for 2-opt

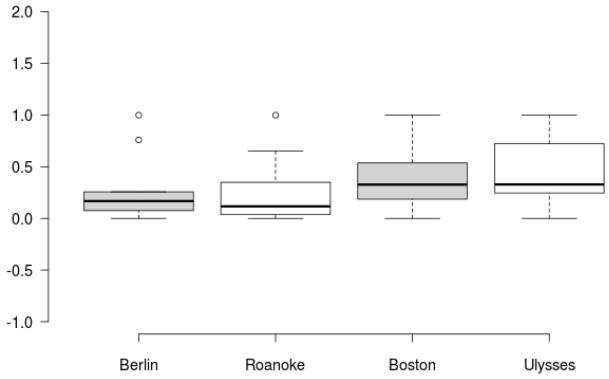


Figure 5: Box Plot of running time for 2-opt

3-opt

The following plots show the performance of the 3-opt algorithm across 4 different datasets. 3-opt quickly achieve optimun on Ulysses, Boston,Berlin dataset, so we can see all curves are overlapping in QRTD plot. In Roanoke dataset, the complexity of tour grows sharply, most executions can not approximate optimum closely within 100 seconds.

In RTD plots, same situation happens on Ulysses,Boston,Berlin dataset, as 3-opt achives optimal so quickly, while in Roanoke dataset, the curve with larger time value better approximate optimal value, which matches result in QRTD.

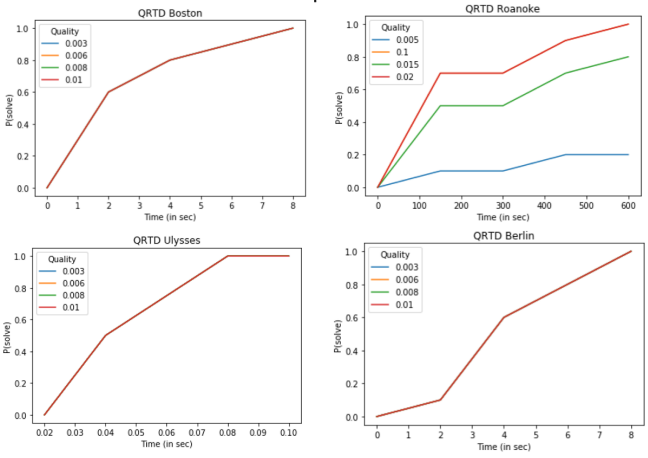


Figure 6: QRTD plots

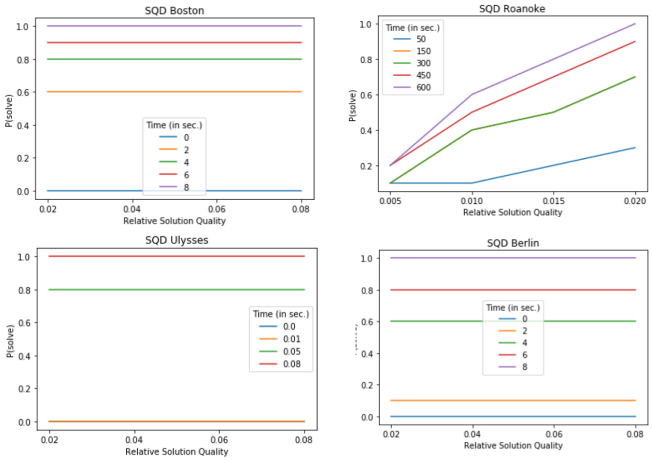


Figure 7: SQD plots

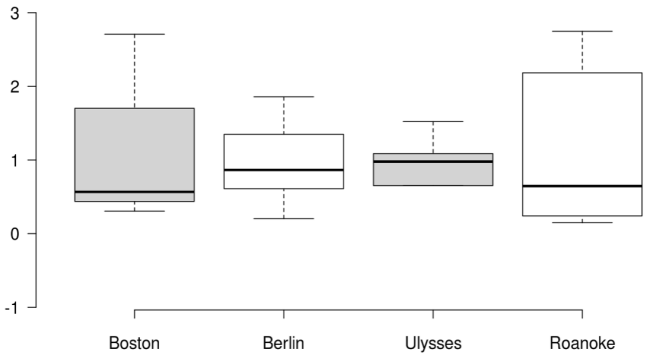


Figure 8: Box plot of running time of 3-opt

7 DISCUSSION

In this study, a total of four algorithms were developed and analyzed: branch-and-bound for an exact solution, MST-approx for an approximation, and two local search heuristics. For small instances (city count < 40), all four algorithms found optimal result within

10s. However, the runtime of branch-and-bound explodes when city count exceeds 50, and MST-approx faces a trade-off between accuracy and runtime beyond this threshold as well. One can view this problem from "global sense" and "local sense". In BnB, every time the sub-tree is expanded, we have to copy reduced cost matrix for following reference. Therefore, when instances exceeds certain number, the number of sub-trees and matrices we need to copy explodes and causes run-time to increase sharply. In terms of MST-approximation, every time we create a tour we resort to DFS, which is required to explore the entire adjacency list. These two algorithms are forced to use "global" information, such as the reduced cost matrix in BB. However, in two local search algorithms, our focus is restricted on few points, i.e. we only deal with our tour in "local" sense. This could be the main reason causing such difference on run-time performance.

Another significant insight can be derived from the comparison of the two local search algorithms. Since, 3-opt is built upon the general idea of 2-opt, it is interesting to compare the performance of one over another on the same set of instances. We observe that in terms of time complexity, 2-opt is better than 3-opt. As a result, we can see that 2-opt find its local optimal solution much faster than 3-opt. However, since 3-opt explores a much larger neighborhood, we can see that 3-opt yields better result in terms of quality, returning the lowest relative error of all the four algorithms. This somehow reflects a trade-off. If we are seeking for an quick approximation with a reasonable quality, the 2-opt approach can work well. On the other hand, if quality is our first priority and it is required to find a solution as optimal as possible, then the 3-opt approach would be the better choice.

8 CONCLUSION

This evaluation of four different approaches to the well-known Traveling Salesman Problem has the following implications. Despite TSP falling under the NP-hard class of problems, it is not difficult to implement algorithms that are reasonable in terms of implementation difficulty, runtime, and accuracy. From a real-world perspective, slight inaccuracies from an "optimal" solution can often be sacrificed in favor of highly reasonable runtimes, as seen in the results. For example, rather than spending hours daily running a branch-and-bound algorithm to determine the perfect delivery route between a set of drop-off locations, a package delivery company can opt to save time to run a local search algorithm for a slightly sub-optimal route, or an approximation algorithm for a route of guaranteed quality. In situations where the exact solution is required, however, branch-and-bound remains the sole option among these four choices. This theme of trade-offs is commonly present in the field of optimization and brings realism to virtual results. Future work may build on the findings in this study and move towards more modern solutions such as algorithm parallelization.

REFERENCES

[1] R. M.; ChvÃątal V.; Cook W. J. Applegate, D. L.; Bixby. 2006. *The Traveling Salesman Problem.*

[2] N Christofides. 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. (1976).

[3] G.A Croes. 1958. A method for solving traveling-salesman problems. *Operations Research* 6(6) (1958), 791âĂŞ812.

[4] Gregory; McGeoch Lyle; Yeo Anders; Zhang Weixiong; Zverovitch Alexei Johnson, David; Gutin. 2007. Experimental Analysis of Heuristics for the ATSP. The Traveling Salesman Problem and its Variations. Combinatorial Optimization. (2007).

[5] McGeoch L.A Johnson, D.S. 1997. The traveling salesman problem: A case study in local optimization. *Local Search in Combinatorial Optimization* (1997).

[6] Meyerson A; Nanavati A; Poplawski L. 2006. Randomized Online Algorithms for Minimum Metric Bipartite Matching. *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm* (Jan 2006), 954–959. DOI:http://dx.doi.org/1109557.1109662

[7] E.L. Lawler. 1985. *The Traveling salesman problem : a guided tour of combinatorial optimization (Repr. with corrections. ed.).*

[8] Dorabela; Glover Fred; Osterman Colin Rego, CÃĭsar; Gamboa. 2011. Traveling salesman problem heuristics: leading methods, implementations and latest advances. *European Journal of Operational Research* (2011).

[9] Brian W. Kernighan Sen Lin. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. (1973).