Exact and Approximate Solutions to the Traveling Salesperson Problem

JIAYING HE, XIAO YANG, XUNCHENG ZHOU, SANDESH ADHIKARY

1 THE TRAVELING SALESPERSON PROBLEM

The Traveling Salesperson (TSP) Problem is a popular example of an NP-complete problem, the exact solution of which involves searching through the space of all permutations of a set of objects[4, 9, 15]. The problem is generally defined as one of finding the optimal route for a salesman who needs to visit each one of N cities, without making multiple stops at any city. In our experiments, we work with a specific instance of the TSP problem called the Metric TSP, which is defined as follows:

Definition 1 (The Metric Traveling Salesperson (TSP) Problem). *Given the $x - y$ co-ordinates of N points (vertices) in a plane, and a cost function $d(u, v)$ defined for each pair of points such that $d(u, v) = d(v, u)$ and $d(u, v) \leq d(u, w) + d(w, v)$ for all vertices u, v, and w, find the shortest simple cycle that visits all N points.*

The first condition $d(u, v) = d(v, u)$ in Definition 1 restricts us to symmetric cost functions, which is a natural constraint when dealing with distances between geographic points. In our experiments, the points are always assumed to either be on a flat $2D$ surface or on the surface of a sphere (the Earth)[12]. In the former case, we use the Euclidean distance defined as

$$d(u, v) = \sqrt{[u(x) - v(x)]^2 + [u(y) - v(y)]^2}. \tag{1}$$

When dealing with points on the Earth with a radius R of approximately 6378.388 km, we use the Geographical distance defined as

$$d(u, v) = R \arccos\left[\frac{1}{2}\left[[1 + q_1(u, v)]q_2(u, v) - [1 - q_1(u, v)]q_3(u, v)\right]\right], \tag{2}$$

where

$$q_i(u, v) = \begin{cases} \cos(\text{longitude}[u] - \text{longitude}[v]) \text{ for i = 1} \\ \cos(\text{latitude}[u] - \text{latitude}[v]) \text{ for i = 2} \\ \cos(\text{latitude}[u] + \text{latitude}[v]) \text{ for i = 3.} \end{cases}$$

The latitude and longitude values in the above equations are assumed to be in radians.

The second condition $d(u, v) \leq d(u, w) + d(w, v)$ in Definition 1 is the triangle inequality, which can be interpreted to mean that taking the direct path between two cities is always shorter than if one were to visit another city in between.

2 RELATED WORK

The Traveling salesman problem (TSP) was first mentioned in 1800s, and was formally defined in the 1920s. Since then, it has been widely studied and significant progress has been made in solving them. In particular, TSP LIB Library published by [12], provided researchers across the world with a repository of interesting TSP problems on which they could try their new methods. Since it is an NP-hard problem, researchers have focused on two alternatives to solve this problem: looking for heuristics and developing approximations. Given the history and wide-spread applications of TSP, there exists a wealth of diverse algorithms aimed at tackling it including Genetic Algorithms [11], Ant Colony Optimization [5][13], Particle Swam Optimization [14], and many more. Even today, the TSP problem serves as an interesting benchmark for new algorithms attempting to solve combinatorial optimization problems.

A survey of prior research reveals that that heuristics algorithms usually obtain solutions of better quality with decent time performance [6]. These heuristics starts from the basic local search algorithm and by various improvements, keep obtaining better results. Most of the classic improvements include pair-wise exchange, k-opt heuristic, tabu search. In particular, Lin-Kernighan [7] algorithm is a famous heuristic algorithm that combines several improvements mentioned above. Besides iterative improvement, there also exists randomized improvement.

In this paper, we employ both smart exhaustive search (Branch-and-Bound) algorithms, approximiation algorithms (MST) with relaxed optimality guarantees, as well as local heuristic algorithms (Iterative Local Search, Simulated Annealing) to the age-old TSP problem. We compare these algorithms across their run-times and solution qualities, and attempt to identify elements of their algorithm design that yield their respective results. We apply our algorithms on 14 instances of the metric TSP problem from the TSPLIB library [12].

3 BRANCH AND BOUND ALGORITHM

3.1 Description

Branch-and-bound is an algorithm improved from backtracking. Just like backtracking, branch-and-bound systematically searches through the whole solution space and provides a solution. Instead of verifying the existence of a valid solution as what backtracking does, branch-and-bound is applicable to optimization, which means that it can present us the exact optimal solution. Meanwhile, branch-and-bound algorithm, by using computed bounds of partial solutions to prune branches of search space, can achieve better performance.

3.1.1 Search Space.

For branch-and-bound algorithm, its search space is originally all possible solutions of a certain problem. For Traveling Salesman problem, the searching space is all permutation of cities (vertices in given graph). According to the searching logic, the search space can be structured as a tree where a path from root to a node at certain level represents a partial solution. A node at level k corresponds to the city which appears to be the k-th one in the partial solution. In practice, the search space usually are pruned by the comparison between a computed lower bound and the best result we have so far, in order to improve performance.

Author's address: Jiaying He, Xiao Yang, Xuncheng Zhou, Sandesh Adhikary.

3.1.2 Choose and Expand Partial Solution.

To choose and expand a partial solution is a part of the search. At each step of search, we need to decide which city we should consider next. In the tree search space, the problem is when we at a parent node, which child node we should consider first. Branch-and-bound algorithm applies Best-first Search to this problem. Best-first search defines that the search order should be: always first choosing the city (vertex, or say child node in the tree) that gives the best result, which we can call it the best city.
After choosing a city, we need to expand the partial solution based on that city so that our algorithm can keep searching in the remaining searching space. In TSP, the expansion can be either including the chosen city into our partial solution or excluding it. Specifically, we modify our adjacent matrix and compute a lower bound based on inclusion or exclusion. If we decide to include the chosen city in our partial solution, then based on the updated partial solution, we keep on searching for the next city; otherwise we exclude the chosen city, move the city out of candidate set, and choose the best city among the remaining candidate cities.

3.1.3 Solution Validation.

Solution validation consists of two parts: completeness and validness.
Completeness is intuitively about whether the solution covers all the cities. If the solution fail to cover all the cities, then it is not complete. Since obviously in metric TSP problem, there definitely exists a city adjacent to the last city in our partial solution that can be included. Checking the completeness enables us to know whether we have a possible complete solution, not partial (we reach a leaf in the tree).
Validness indicates whether the solution can form a cycle in the graph going through all the cities. Undoubtedly, a valid solution must be a complete one. We only check validness after we verify that the solution is complete. And if a solution is valid, we compare it to the best solution we have already found. If it is even better, we claim it as our best solution.

3.1.4 Pruning branches.

To avoid redundant search and thus reduce running time, we need to prune the branches (part of the search space) once we know that the partial solution is already worse than the best solution we have so far. In TSP, all edges have positive weights, and if a partial solution has its cost already larger than the best solution, no matter how we further expand this partial solution, the weight will not decrease and become better than best solution. Consequently, future search based on this partial solution is redundant and need to be pruned.
Usually, pruning only happens when we try to expand partial solution by including a city. Because when excluding a city, we do not increase the cost of partial solution. Actually if we exclude a city, we then consider the next best city and try to include it. And if including that city gives us a solution worse than best solution, we prune the following search on that branch. So in a nutshell, we only prune after we including a city.
The way to prune use the concept of backtracking. Suppose we now at city v and we choose the best city u among all the candidate cities to expand our partial solution. First we try to include u and if we find out that after inclusion, the lower bound of cost now is worse than the cost of our best solution. Then we prune this branch by remove u out and search in the branch of excluding u.
In order to prune, we need to compute a lower bound of our partial solution and compare it with the weight of best solution. The most common way to compute lower bound is by computing the total cost of edges in our solution, which can be done by using reduced matrix. Another way is by computing minimum spanning tree.

3.1.5 Lower Bound: Reduced Matrix.

In this approach, we compute the lower bound for a partial problem by considering the lowest weight costs of entering and leaving a city.
We begin with the adjacent matrix of a given graph, in which i-th row represents the weights of edges leaving i-th city to every other cities (weights of i's leaving edges), and similarly j-th column represents the weights of edges entering j-th city from every other cities (weights of j's entering edges). Before we start searching, we need to first reduced the matrix.
We reduce the matrix by first reducing all the rows and then all the columns, and we also try to record a reduced weight. To reduce all the rows, we first find the minimum weight in each row. Then we subtract it from all weights in that row, and also add it to the reduced weight. Based on the matrix with reduced rows, we then reduce all the columns, which is basically the same as reducing rows except we subtract the minimum weight from its column. After reducing the matrix, each row or column should at least has one zero weight. The reduced weight we have now is the lower bound for all the possible permutation of cities (or say the lower bound for an empty partial solution). The main idea behind the reduced matrix method is that even if the instance, where we enter and leave each city with the lowest cost, does not lead to a valid solution, let alone a solution, we know that it is the best we could have hoped for even without any constraints. Therefore, the actual solution to our problem cannot have a lower cost that this lower bound.
During search, by the rule of best-first, suppose we now at city v, we need to find an edge that has zero weight in the row that corresponding to the leaving edge of city v, say an edge (v, u). Then this edge is an edge leaving v and entering u with cost 0. And u should be the best city we should consider for expansion.
Now we have chosen a city u to expand, first we try the solution that includes u. We then delete the row corresponding to v and the column corresponding to u from the reduced matrix and reduce the matrix again. We delete the row and the column because when we have edge (v, u), we do not need to consider any other edges leaving v and any other edges entering u. After reducing the matrix, if the lower bound is not larger than best cost, we can move on searching.
Another possible expansion is by excluding u. By this we need to set the weight of edge (v, u) to infinity (so that the weight will be too large for the algorithm to choose u again before we finish the search on current branch) and reduce the matrix again.

As we keep expanding our solution, the reduced weight will increase and become tighter as we expand our solution and eventually when we have a valid solution, the reduced weight is actually the exact cost (total weight) of that solution. This is because the process of reduction can be regarded as taking the minimum weight edges into our solution. And if a valid solution chooses another edge instead of the one with minimum weight (by exclusion), then we have no zero value in a certain row and column (the zero value was set to infinity). By reducing the matrix again, we simply find another edge with minimum weight, which is actually the minimum weight difference from the previous minimum edge's weight. We subtract that difference from the row or column and add it to reduced weight. Adding the difference makes the reduced weight the same as we have chosen that edge at the first place. Therefore, when we find a complete solution, the reduced weight is exactly the total cost of that solution.
Whenever we find a complete solution, the lower bound (reduced weight) should be the cost of that solution. So if the solution gives us a full tour in the graph (valid), we can simply compare the lower bound with our best cost to see which solution is better.

3.2 Pseudocode

The pseudocode for the Branch and Bound algorithm, as well as the various helper functions used in it are presented in Algorithms 1,2,5,4 and 5.

Algorithm 1: Branch-and-Bound Algorithm

Input: Graph G
Output: $bestSolution[]$
$partialSolution[] \leftarrow -1, bestSolution[] \leftarrow -1$
$bestCost \leftarrow \infty$
$bestSolution \leftarrow$ SearchHelper($G, source, partialSolution, bestSolution$)
return $bestSolution, bestCost$

3.3 Time and Space Complexity

Branch-and-Bound Algorithm iterates all possible solution in the search space, so the time complexity increases intensely as the scale of given graph increasing. The space complexity of branch-and-bound algorithm is also terrible, since we need to somehow store the current subproblem so that once we finish search in that branch we can backtrack to that subproblem and switch to a different branch and keep on searching.

3.3.1 *Time Complexity.*

Suppose given a graph with n vertices, we need to find a solution in the graph, which is a list of vertices with certain order. And a valid solution is a list of vertices with an order that can form a cycle in the graph. The branch-and-bound solution iterates all possible solutions (either valid or invalid solution), which means all permutation of this list of vertices are considered once by the algorithm. The number of all permutation should be $n!$.When implementing the algorithm, we also need other operation to compute

Algorithm 2: SearchHelper

Input: Graph $G, source, partialSolution, bestSolution$
Output: $bestSolution[]$, Array of city indexes (denote a full tour)
$bestDst \leftarrow$ best city adjacent to $source$
if *No bestDst* **then**
  return $bestSolution, bestCost$
end
else
  Include $bestDst$ into $partialSolution$
  $LB \leftarrow LowerBound(partialSolution)$
  if *partialSolution is not complete AND* $LB < bestCost$ **then**
    $bestSolution \leftarrow$ SearchHelper($G, source, partialSolution, bestSolution$)
  end
  if *partialSolution is complete and valid AND* $LB < bestCost$ **then**
    $bestCost \leftarrow LB, bestSolution \leftarrow partialSoltion$
  end
  Exclude $bestDst$ into $partialSolution$
  $bestSolution \leftarrow$ SearchHelper($G, source, partialSolution, bestSolution$)
  return $bestSolution, bestCost$
end

Algorithm 3: Reduction-Include

Input: Matrix M ($int[][]$), $source, destination, reducedWeight$
Output: Matrix $ReducedM$ ($int[][]$), $reducedWeight$
for $i = 0 \leftarrow M.size$ **do**
  $M[source][i] \leftarrow \infty$
end
for $j = 0 \leftarrow M.size$ **do**
  $M[j][destination] \leftarrow \infty$
end
$ReducedM \leftarrow$ Reduction($M, reducedWeight$)
return $ReducedM, reducedWeight$

Algorithm 4: Reduction-Exclude

Input: Matrix M ($int[][]$), $source, destination, reducedWeight$
Output: Matrix $ReducedM$ ($int[][]$)
$M[source][destination] \leftarrow \infty$
$ReducedM \leftarrow$ Reduction($M, reducedWeight$)
return $ReducedM, reducedWeight$

the lower bound, and also set up the backtrack point. According to the pesudocode, there are few operations that take more than constant time. First, finding the best city, which is the adjacent city promising the least cost at current step, takes $O(n)$ to go through the corresponding row in adjacent matrix. Computing the lower bound, or say reducing the matrix, takes $O(n^2)$, since for each one of the total n rows, we need to first go through it to find the minimum

Algorithm 5: Reduction

Input: Matrix M ($int[][]$), $reducedWeight$
Output: Matrix $ReducedM$ ($int[][]$), $reducedWeight$

```
for i = 0 ← M.size do
    minWeight ← minimum weight in i-th row
    if minWeight ≠ 0 AND minWeight ≠ ∞ then
        reducedWeight ← reducedWeight + minWeight
        for j = 0 ← M.size do
            if M[i][j] ≠ ∞ then
                M[i][j] ← M[i][j] − minWeight
            end
        end
    end
end
for i = 0 ← M.size do
    minWeight ← minimum weight in i − th column
    if minWeight ≠ 0 AND minWeight ≠ ∞ then
        reducedWeight ← reducedWeight + minWeight
        for j = 0 ← M.size do
            if M[j][i] ≠ ∞ then
                M[j][i] ← M[j][i] − minWeight
            end
        end
    end
end
return M, reducedWeight
```

weight and then go through it again to do the subtraction, both of which takes $O(n)$. Reducing columns takes the same amount of time. So totally reducing matrix takes $O(n^2)$.As for storing the backtrack point, there are multiple ways to do it. An intuitive way is to simply make a copy of the whole reduced matrix, which will take $O(n^2)$ for copying and restore respectively. The second way, which is more efficient, is by tracking the changes of reduced matrix. For inclusion, we need two arrays to store the values of the row and column that should be set to infinity. For exclusion, we need to record the index of endpoints (cities) of the edge that should be set to infinity. Finally, for the reduced matrix, we need to store the minimum weight at every column and row. At the time when we need to backtrack, we add the minimum weight back and recover the row and column that are set to infinity during inclusion, or the value that is set to infinity during exclusion. All this operations takes $O(n^2)$. In all, the total time complexity for branch-and-bound algorithm should be $O(n!n^2)$

3.3.2 Space Complexity.

Obviously, we first need a space to store the adjacent matrix, which takes $O(n^2)$. Other space the algorithm taking up is due to backtracking points. At each node we need to store reduced matrix so that we can recover it when we go back. As discussed above in the section time complexity, there are two ways to store the reduced matrix.The first way, storing the whole reduced matrix, totally takes $O(n^4)$ in worst case. First, a reduced matrix takes n^2 space. Then we consider a complete solution. Since to obtain a complete solution, we need to do inclusion for n times (1 city at each level, totally n levels), and during searching, in the worst case, we may do exclusion $n-1$ times which means we excluded all the other cities except the one we finally included. In this case, we store $n-1$ reduced matrix for exclusion at each level. Therefore, at each level, we store n reduced matrix and after n levels, we stored $O(n^2)$ reduced matrix, which takes $O(n^4)$ in space. Also notice that space cost reaches the maximum because the worst case is the last possible case branch-and-bound algorithm searches. The second way, storing the changes of reduced matrix, saves much more space. We also consider the worst case discussed above. At each level, we have $n-1$ exclusion with constant space cost each time, 1 inclusion with $2n$ space cost each time and n reduction with $2n$ space cost each time. All of these sum up to $O(2n^2+3n-1) = O(n^2)$ space cost at each level. And totally for n levels, it takes $O(n^3)$ space.

3.4 Strength and Weakness

The strength of branch-and-bound algorithm is that given sufficient time, it can guarantee the optimal solution since it iterate through all potential solution. Especially when the problem size is not large, the algorithm, though still slower than other algorithm, but the time cost is acceptable and it is always good to be guaranteed of a optimal solution. The performance of branch-and-bound algorithm can also be improved by tighter lower bound which can prune more search space and avoid more redundant search.
The weakness of branch-and-bound is that its time and space are both unacceptable when the problem size is huge. Let alone the huge amount of time it consumes, it may cause memory overflow since it takes up much space for backtracking. Besides, the implementation is also much more complicated than other algorithms because of recursive logic.

4 APPROXIMATION ALGORITHMS

When looking for an exact solution takes a long time, we can try to find an approximate solution that is not too worse than the optimal solution but takes much shorter time. These methods are called approximation methods. The key idea is to find a solution with some quality guarantee in a short time. For TSP, the target would be to find a path whose cost is guaranteed to not exceed some relation error comparing to the optimal solution. For this project, since we are solving a euclidean TSP, we implemented the MST approximation method to solve the problem.

4.1 MST Approximation

The MST approximation method works for all euclidean TSP and is guaranteed to give a solution whose cost is no more than twice the cost of the optimal solution. It works by getting a minimum spanning tree of the whole graph. After getting the MST, randomly select a node as root and starting point of paths and travel though the tree in depth first order. Add a new node to the path whenever a node is visited, no matter what direction to visit. Denote the path we have now as S''. In S'', some nodes are visited more than once. To make a valid solution for TSP, we go through S'' and remove a node when it has already been visited, except the last node. The

new cycle is denoted as S' and is the final output of this algorithm.

4.1.1 Approximation Guarantee.

If we randomly remove one edge from the optimal solution, the remaining path becomes a spanning tree of the original graph and the cost is at least as large as the cost of the minimum spanning tree. Let S stands for the optimal solution, we have:

$$Cost(MST) \leq Cost(S). \qquad (1)$$

For an edge $e_{u,v}$ in S', if u and v are directly connected in the minimum spanning tree, then

$$d^{S'}_{u,v} = d^{MST}_{u,v} = d^{S''}_{u,v} = w_{u,v}. \qquad (2)$$

.

If if u and v are not directly connected in the minimum spanning tree and there's a path $u, p_1, p_2, ..., p_n, v$ between them. For a euclidean graph:

$$d^{S'}_{u,v} = w_{u,v} \leq w_{u,p_1} + w_{p_1,p_2} + ... + w_{p_n,v} = d^{MST}_{u,v} = d^{S''}_{u,v}. \quad (3)$$

.

Combine (2) and (3), we have:

$$Cost(S') = \sum d^{S'}_{u,v} \leq \sum d^{S''}_{u,v} = 2(\sum w_{u,v}) = 2Cost(MST). \quad (4)$$

.

Combine (1) and (4):

$$Cost(S') \leq 2Cost(MST) \leq 2Cost(S). \qquad (5)$$

.

So we can prove that the cost of the solution get by using the MST approximation method is at most twice the cost of the optimal solutions.

4.1.2 Pseudocode.

The pseudocode for this method is shown in Algorithm6, 7 and 8.

Algorithm 6: MST_Approximation

Input: A graph g
Output: A list of integers representing the path $c_1, c_2, c_3, ..., c_n, c_1$

```
mst ← compute_MST(g)
path ← ∅
distance ← ∞
for city ∈ g do
    visited ← ∅
    newPath ← getPath(mst, city, visited, newPath)
    newDistance ← getDistance(newPath)
    if newDistance < distance then
        distance ← newDistance
        path ← newPath
    end
end
return path
```

Algorithm 7: compute_MST

Input: A graph g
Output: A graph mst

```
pq ← g.edges()
pq.sort()
mst.vertices ← g.vertices()
mst.edges ← ∅
while mst not connected do
    edge ← pq.pop()
    while two ends of edge are connected do
        edge ← pq.pop()
    end
    mst.edges ← mst.edges ⋂ {edge}
end
return mst
```

Algorithm 8: getPath

Input: A graph g, A starting position p, A set $visited$, A list of integers $path$

```
visited ← visited ⋂ {p}
path ← path ⋂ {p}
for neighbors n of g do
    if n ∉ visited then
        getPath(g, n, visited, path)
    end
end
```

4.1.3 Complexity Analysis.

Denote the number of cities as n, then the total number of edges is $m = n(n-1)/2 = n^2/2 - n/2$.

The whole program is divided into 2 parts, computing the MST and constructing path.

The MST is computed by using the Kruskal's algorithm.

A balanced unionf structure is used to compute MST. It is used to check if two nodes are connected or not. To run Kruskal's to compute the MST, the step to collect info of all the edges is of time complexity $O(1 \times m) = O(n^2)$. Sorting and polling the edges into and from a priority queue will use time $O(m \log m) = O(n^2 \log n)$. For each edge from the queue, the time to check whether the two nodes are already connected or not and combine two nodes in unionf is of $O(\log n)$ and the total time for this step is of $O(m \log n) = O(n^2 \log n)$. Adding new edge to the MST is of time $O(1 \times (n-1)) = O(n)$. So the time complexity of computing an MST is $O(n^2) + O(n^2 \log n) + O(n^2 \log n) + O(n \times n) = O(n^2 \log n)$.

The graph and the mst are stored in two $n \times n$ matrices, the priority queue stored info of all the edges, so the space complexity for this step is $O(n^2)$.

To construct a path, we select one city as root and do depth first search from the root, and we add a city to the path at the first time the city is visited. This will have the same output as to save all cities to the path and then do shortcuts because it directly skips repeated cities. This step is of time $O(n)$. Since starting from different roots

can result in different paths, we do this for all cities and select the one with minimum cost. Then the time complexity of this step becomes $O(n^2)$.

For this step, we only need $O(n)$ space to store the order of cities in paths.

In conclusion, the whole method is of time complexity $O(n^2 \log n) + O(n^2) = O(n^2 \log n)$. The space complexity is $O(n^2) + O(n) = O(n^2)$. Since the time complexity is not exponential, we didn't include the time of this method in the comprehensive table.

5 LOCAL SEARCH ALGORITHMS

Since the optimal solution of TSP requires an enumeration over $N!$ permutations, we seek solutions that may not be optimal but would likely work well in practice. For cases where we have a very large number of nodes, the salesperson might be content with picking a 'good-enough' tour instead of having to wait around for the exact algorithm to find the optimal one. Local search algorithms fall under such a category of heuristic search algorithms which do not provide any guarantees of optimality (or closeness to optimality) , but are constructed in a manner that makes it likely that we will obtain a high quality solution.

The central idea behind a local search algorithm is to begin with some plausible solution to the problem, and search for better solutions in its neighborhood. For TSP, a possible initial solution could be any tour of all nodes. The neighborhood around such a solution is generally defined as the set of solutions that can me reached by making minor changes to the existing one. For instance, we could swap a few of the edges of the solution while still ensuring that we still have tour. If the new neighbor solution yields a lower total tour cost, we accept it then begin searching around its neighborhood.

In the most naïve version of local search, we would perform a simple hill climbing search, where we move towards the neighbors that decrease our overall cost, and stop searching when none of the neighbors yields a lower cost. However, such an algorithm would be highly sensitive to our choice of the initial solution, and also on the presence of local maxima. It is quite likely that if we start our search close to a local maxima, the hill climbing algorithm gets stuck there without exploring the rest of the space. We can employ various improvements to this hill climbing approach to try to escape such local optima. We summarize such methods that we implemented in Table 2 and describe them in detail below.

5.1 Iterative Local Search

As in Hill Climbing, we generate an initial tour and perform a local search around its neighbors. We define the neighborhood around a tour as all the other tours that can be obtained by performing a '2-swap'. Here, we simply find two random edges in our solution (u_1, u_2) and (u_3, u_4) and disconnect them. We then create new edges to connect the nodes (u_1, u_3) and (u_2, u_4). This swap is done in such a way that we the resulting permutation of nodes is also a valid tour, as illustrated in Fig. 1.

To avoid getting stuck in a local optimal, we also consider the neighbors of a slightly perturbed version of our initial tour. In particular, we perform a 'double-bridge' perturbation where we disconnect four random edges and connect the nodes back up to create a double bridge between the tour as shown in the Fig. 2. We now perform a 2-swap on this perturbed tour and check if it provides a lower tour cost than what we had achieved earlier. We select as our next tour the one with the lower cost.

Fig. 1. The Two Swap. Figure from [15]

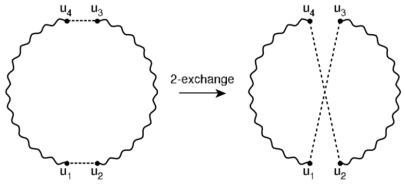
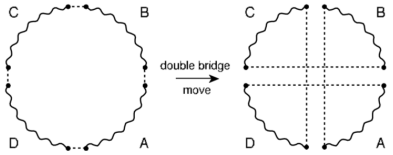


Fig. 2. Double Bridge Move. Figure from [15]



5.1.1 *Local Search.*

Algorithm 9: LC

input : s_0 the random initial solution, *adj* the adjacent matrix of the graph
output : A solution s after Local Search
Initialize *neighbors*
Construct *neighbors* using 2-opt
for *Every neighbor in neighbors* **do**
  Find the neighbor with minimum cost
end
if $neighbor_{best}$ *equals to* s_0 **then**
  return s_0
end
else
  return $LC(neighbor_{best}, adj)$
end

First, I randomly generate a solution as the start solution of Local Search. Then I would use 2-opt method to construct the neighbor set as neighbors. Then I have a score for every neighbor which is the cost of the solution. I will pick the neighbor with minimum cost as best neighbor. If the best neighbor is better than current solution, then recursively call LC on the best neighbor; otherwise, return the current solution since there is no improvement.

In conclusion, for the Local Search, I did not allow the algorithm to choose the worse neighbor at some probability. Since for the Iterative Local Search, the Perturbation will help to escape from local optimal, there is no need to do it in Local Search. I did not involve tabu memory. Since I am only "climbing" the hill, and once

I reach to one local optimal I will return, there is also no need to involve tabu memory in Local Search.

5.1.2 *Perturbation.*

Algorithm 10: Perturbation

input : s the solution that need to perturb, adj the adjacent matrix of the graph, $history$ is the tabu memory
output: A solution s after Perturbation
Initialize $pivots$
do
  while $pivots.size() < 4$ **do**
    Construct valid $pivot$ randomly
    $pivots.add(pivot)$
  end
  $s_{new} \leftarrow$ Execute 4-exchange using $pivots$ on s
  $time++$
while $history.contains(s_{new})$ *and* $time <= k$;
if $(!history.contains(s_{new}))$ **then**
  $history.add(s_{new})$
end
return s_{new}

For Perturbation, I would randomly choose 8 locations in the current solution, and use 4-exchange to change the solution to a new one. The reason I choose 4-exchange is that the goal is to escape from local optimal, and compared to 2-opt and 3-opt, 4-exchange will theorectically have a better effect. I add tabu memory *history* in Perturbation, because different from Local Search, after Perturbation, the solution may already been searched.

5.1.3 *Iteraive Local Search.*

Algorithm 11: Iterative Local Search

input : adj the adjacent matrix of the graph
output: A solution s after ILS
Initialize $history$
$time = now$
$s_0 \leftarrow$ randomly generate the initial solution
$s_{lc} = LC(s_0, adj)$
while $start.plusSeconds(m).isAfter(now)$ **do**
  $s_{perturb} = Perturbation(s_{lc}, adj, history)$
  $s_{perturb\_lc} = LC(s_{perturb}, adj)$
  if $s_{lc}.cost > s_{perturb\_lc}.cost$ **then**
    $s_{lc} = s_{perturb\_lc}$
  end
end

For Iterative Local Search, the termination condition is time. The algorithm will keep calling Perturbation and run Local Search on the solution after Perturbation. If the solution improves, than update; otherwise, keep the solution before Perturbation. The algorithm will keep doing this within the time limit in order to get a better result.

5.2 Simulated Annealing

Simulated annealing was originally proposed separately by [8] and [3] as a fascinating bridge between statistical thermodynamics and combinatorial optimization. The physical process of annealing is commonly used to alter the properties of materials to improve ductility and robustness. From a statistical mechanics perspective, it is the thermal process of obtaining low energy states of a material by placing it in a heat bath [1, 3, 8]. A heat bath is initially set to be at a temperature higher than the melting point of the material of interest. The solid is then placed in the heat bath, and the temperature is reduced slowly. When the material is placed at a certain temperature for a sufficient amount of time, the atoms will settle into some equilibrium state. The system will attempt to stabilize into a state with low energy, but if the temperature is high, there are many possible states that it could settle into. In statistical thermodynamics, it is generally assumed that materials follow the Boltzmann distribution where the higher the temperature, the greater the probability of the system being in any arbitrary state - at high enough temperature, all states are likely. However, as the temperature approaches 0, only the global energy minimization state has non-zero probability. Thus, start the temperature to be high enough that change can be induced, and reduce at a sufficiently slow rate that the system can come into an equilibrium state at each temperature.

From an optimization stand point, simulated annealing is a metaheuristic local search algorithm that encourages transitions to solutions of higher quality (exploitation), while still allowing transitions towards lower quality solutions (exploration) with some probability. As a local search algorithm, simulated annealing also begins with a trial solution and attempts to transition to another solution from a well defined neighborhood. The primary objective of the algorithm is to minimize some cost that is representative of the solution quality. Since the naive greedy strategy of only transitioning to higher quality neighbors is prone to get stuck at sub-optimal solutions, the algorithm allows for transitions into lower quality neighbors with some probability. The distinguishing feature of simulated annealing is that this probability of moving 'uphill' for a minimization problem, is initially set to be sufficiently high, and is slowly reduced as the algorithm progresses. Thus, we encourage the algorithm to initially explore the space instead of getting stuck, and slowly force it to favor exploiting the best solution it has found.

We adapt the general structure of the simulated annealing algorithm from [6] as presented it in Algorithm 12. The following are the main elements of algorithm design that can be varied:

(1) **Initial temperature**: The higher the temperature, the higher the degree of exploration. In our experiments, we test random initialization as well as a dynamic initial temperature dependent on the problem size suggested by [2].
(2) **Initial solution**: If one is able to begin with a reasonable guess for the solution, it may be possible arrive at a better solution as well as reduce the run-time. We experiment with a nearest neighbor heuristic to generate a sensible initial solution.
(3) **Temperature equilibrium**: Theoretically, it is important to allow the system to reach its equilibrium state before reducing

the temperature. While it is possible to devise a rigorous definition of equilibrium, we make the simplifying assumption that equilibrium is achieved in 100 iterations for any temperature. Such a hard threshold on equilibrium is common across prior literature [6].

(4) **Neighbor Selection**: We define our neighborhood as the standard 2-exchange rule as in our iterative local search algorithm. We also experiment with two strategies of pruning the neighborhood of candidate solutions.
(5) **Acceptance Criteria**: Following the literature, and the physical analogy, we always use the Metropolis acceptance criteria. When comparing two solutions, we compute the difference in their costs as the difference in the energies of the two states ΔE. If this difference in energy is less than zero, we have found a better solution and transition into it with probability 1. If not, we accept the solution with probability $\exp(\frac{\Delta E}{T})$, which is the Metropolis criterion. In statistical physics, this criterion is known to simulated the evolution of materials in heat baths to thermal equilibrium [10].
(6) **Temperature Schedule**: We use a simple geometric series where lower the temperature amounts to multiplying it by 0.95.
(7) **Greedy Search after Freezing**: Once the temperature has reached 0, we consider the state to be frozen. At sufficiently low temperatures, it is very rare for the algorithm to move towards a worse solution. At this point, it has been observed [6], that continuing the simulated annealing process does not provide any additional benefit over a simple greedy search over neighbors. Therefore, once the temperature reaches some threshold value (for us 10^{-10}), we consider the system to have frozen, and switch to the naive greedy scheme. This scheme is the same as the simulated annealing process without the possibility of transitioning to worse neighbors
(8) **Search Restarts**: Despite the annealing heuristic, it is still possible that the algorithm will get stuck at sub-optimal solutions, and continue to explore low quality neighborhoods. When this happens, it might be desirable to revert back to the best solution recorded thus far and resume search from there instead. Therefore, we keep track of the number of iterations since the last improvement to the best solution. If there has not been an improvement in $N/2$ iterations, we revert the current solution back to the last best solution. The choice $N/2$ was picked through trial-and-error.

Since the simulated annealing algorithm is a meta-heuristic, it allows for many opportunities to edit the general scheme in Algorith 12. In general, these additions seem to present trade-offs between run-time improvements and solution quality improvements. In our experiments, we decided to prioritize solution quality over run-time whenever necessary as the former is generally the main goal in non-trivial TSP problems. Below, we present our baseline implementation, and some of the additions with which we experimented.

5.2.1 Baseline Simulated Annealing. In our baseline SA implementation, we begin with a randomly sampled initial solution, and an arbitrarily chosen initial temperature of 100. We define our neighborhood as the standard 2-opt neighborhood. Given a solution, we randomly sample the node u_1 and u_4 shown in Fig 1. We then split up the solution into three lists. The first list contains all nodes from the starting node up to u_1, and the third list contains all nodes from u_4 to the final node in the tour (which is connected to the starting node). The second list contains the nodes from u_2 to u_3. We can achieve the desired two-swap by simply reversing the second list, and stitching the three lists back together.

We implement the baseline algorithm on four cities to obtain a sense of solution qualities and run-times as shown in Table 1. We picked these cities as they correspond to cities with some of the smallest (10,20) and largest nodes (109,230). The baseline results are presented in the first row of Table 1.

5.2.2 Initial Solution Selection. We implement a simple greedy nearest neighbor heuristic to obtain an alternative to random initialization as shown in 13. As shown in Table 1, this heuristic resulted in reduced average distances for all but the smallest city, Cincinnati. It does however seem to result in minimal increase in run-time.

Algorithm 12: Basic Simulated Annealing

```
T,S ← Generate an initial temperature and initial solution
S* ← S
best_counter = 0
while time < (start + cut_off_time) do
    while T > 0 do
        for i in 1 : equilibium_iters do
            S' ← find_neighbor(S)
            ΔE ← cost(S') - cost(S)
            if ΔE ≤ 0 then
                S ← S'# Move towards better solution
                if cost(S) < cost(S*) then
                    S* ← S # Update best solution
                    best_counter = 0
                end
            else
                #Potentially move towards worse solution
                if accept_solution(ΔE) == True then
                    S ← S'
                end
            best_counter += 1
            if best_counter > max_best_iterations then
                S = S* #Revert back to best solution
                best_counter = 0 #Reset best_counter
            end
        end
        T ← lower_temperature(T)
    end
    Run greedy local search for remaining time duration
end
return S*
```

	Cincinnati		Atlanta	
Strategy	Avg. Dist	Avg. Time (s)	Avg. Dist	Avg. Time (s)
B	283804	0.0040	2059180	0.0198
B + NN	294318 (+0.037)	0.0026 (-0.35)	2043117 (-0.007)	0.0206 (+0.040)
B + NN + IT	277952 (-0.0021)	0.297 (+73.25)	2039450 (-0.009)	0.4358 (+21.0)
B + NN + IT + TB	**277952 (-0.0021)**	**0.328 (+81)**	**2021219 (-0.018)**	**0.487 (+23.5)**
B + NN + IT + PR	277952 (-0.0021)	0.49 (+121.5)	2007766 (-0.025)	0.448 (+21.6)
	Toronto		Roanoke	
Strategy	Avg. Dist	Avg. Time (s)	Avg. Dist	Avg. Time (s)
B	1345051.8	3.5384	701517.4	27.315
B + NN	1235257.6 (-0.08)	3.5578 (+0.005)	696659.2 (-0.0069)	23.845 (-0.127)
B + NN + IT	1260372 (-0.06)	3.459 (-0.02)	701109 (-0.00058)	27.95 (+0.022)
B + NN + IT + TB	**1222639 (-0.100)**	**4.9172 (+0.39)**	**705502 (+0.00567)**	**25.97 (-0.048)**
B + NN + IT + PR	1245874 (-0.08)	3.92 (+0.097)	699713 (-0.0025)	24.289 (-0.108)

Table 1. Experiments with Improving Simulated Annealing. The strategies correspond iterative improvements to the baseline (B) algorithm with (NN) Nearest Neighbor solution initialization (IT) Bonomi initial temperature (TB) Tabu list (PR) Bonomi Neighborhood Pruning. The B+NN+TB+IT strategy was ultimately chosen since it incorporated both euclidean and geometric distances, and was similar to the alternative with neighborhood pruning.

Algorithm 13: Nearest Neighbor Heuristic

```
adj_mat = get_full_matrix(adj_mat)
tour = [random integer between 0 and len(cities) − 1]
tour_cost = 0
while len(tour) < num_cities do
    # Find neighbors of the last city added to tour
    last_city_added = tour[-1]
    neighbors = adj_mat[last_city_added ]
    # Don't consider cities already in tour
    for city in tour do
        neighbors[city] = ∞
    end
    # Add nearest neighbor to tour
    tour ← tour.append(min_index(neighbors))
    tour_cost += min(neighbors)
end
return tour
```

5.2.3 Initial Temperature. It is difficult to come up with a good estimate for initial temperature for a given problem. Furthermore, it is likely that the choice of internal temperature might be highly specific to the particular problem at hand. In an attempt to use a standardized strategy, we implement the initial temperature suggested by [2].

We begin by computing the length of the smallest square within which the entire city (all nodes) can be fit. When working with Euclidean distances, we compute this length L as the maximum of the highest absolute differences in x and y coordinates. In [2], the authors find that using an initial temperature of $\frac{L}{\sqrt{N}}$, where N is the number of nodes, tends to improve solution quality. In their experiments, this initial temperature tended to be lower than standard. However, for us, this temperature was generally higher than our baseline of 100. Therefore, we anticipated that a higher initial temperature would promote higher exploration and improve solution quality.

As shown in the second row of Table 1, the Bonomi temperature yielded improved solution qualities across the board. However, it does seem to increase the average run time significantly for most cases. This makes sense since a higher starting temperature means that the algorithm accepts more worse neighbors, and therefore spends more time exploring. Since our objective is to maximize solution quality, this improvement is desirable.

5.2.4 Neighborhood Pruning. In an effort to prevent the algorithm from wasting time exploring low quality neighbors, we test out two techniques of pruning the 2-OPT neighborhood.

Firstly, we test out a TABU list to make sure that if a node has been utilized in a recent swap to find neighbors, it is not involved in a swap for a certain number of iterations. Therfore, we maintain a list which of size 5% of the total number of nodes (chosen through trial and error). Whenever a node is selected as part of the 2-swap, we add it to the TABU list. This results in a TABU memory where a node will not be swapped if it participated in a swap in the last $0.05 * N$ iterations. In row 4 of Table 1, we see that the addition of the TABU list further improves solution qualities. As with the initial temperature heuristic. The increase in run-time in row 4 is likely mostly driven by the initial temperature heuristic we employed. While the increase in initial temperature allowed for greater exploration early on, the TABU list should prevent redundant exploration. So it likely complements the earlier improvement, and directs the algorithm further towards better explorations.

We also test the neighborhood pruning scheme suggested by [2], wherein we discourage the addition of very long edges when selecting neighbors. In [2], the authors take the minimum square encompassing all nodes with length L as computed earlier. They then split the box into m^2 evenly spaced cells, and enforce the requirement that when performing a 2-swap the algorithm should

only select swap-nodes that are within 2 cells apart. Bonomi suggest setting $m = L/\sqrt{2}$.

We follow a similar strategy, but instead of placing a hard cut-off, we discourage movements into pruned neighbors. If a neighbor swap resulting in edges larger than those perimitted is selected randomly, we accept the neighbor transition with probability 10%. Furthermore, we do not prune edges that are within 5 cells away, instead of the original two.

As noted in Table 1, the addition of the this Bonomi neighborhood pruning does have similar effects to that of introducing the Tabu list. However, it makes the algorithm slightly complicated and less general. In particular, currently the pruning method is only applicable for euclidean distances. If the improvements in solution quality were significant, it might have been useful to develop an algorithm specifically for euclidean distances, or develop a corresponding version of pruning for geographical distances. However, since the improvements from tabu to Bonomi pruning are minimal, we use the tabu method as our neighborhood pruning strategy.

5.2.5 The Final Model. From our experiments, we find that there seems to be a trade-off between run-time and solution quality when implementing the improvements we tested. As mentioned earlier, we were seeking improvements to simulated annealing that would provide us with the highest solution qualities. This decision was partly motivated by the fact that even with the fastest baseline implementation, the simulated annealing algorithm was unable to compete with the iterated local search in terms of run time. Therefore, we attempted to push the simulated annealing algorithm in the other direction of solution quality.

In all measurements that follow, we use the simulated annealing algorithm with nearest neighbor initial solution heuristic, and the Tabu list neighborhood pruning described in this section.

6 ALGORITHM EVALUATIONS

6.1 Algorithm Comparisons

We report a comprehensive comparison of all algorithms on different TSP datasets [12] in Tables 3 and 4. For each algorithm, we report the time, solution quality, and relative errors with respect to the best known solution. In our experiments, the iterative local search algorithm always provided solutions with the lowest cost. Therefore, we base all solution quality comparisons relative to it. The solution qualities in the table refer to the lowest cost tour obtained from each algorithm for branch and bound and the MST approximation algorithms, and the average cost across 10 runs for the local search algorithms.

All algorithms were designed to terminate only when a predefined amount of time had passed, which was 10 minutes for all algorithms except for simulated annealing. The simulated annealing algorithm was run for 3 minutes, partly due to time constraints and partly due to the fact that none of the instances showed improvements in their solution qualities after that point in prior runs.

The results in Tables 3 and 4 reveal that the iterative local search algorithm yielded the lowest cost distances for all instances. The other local search algorithm, simulated annealing, also demonstrated better solution qualities, on average, compared to the remaining two algorithms. The MST Approximation algorithm yielded the highest cost tours, but had the lowest run-times. When comparing the two local algorithms, there doesn't seem to be a clear winner in terms of run-times. In general, the iterative local search seems to have higher run-times. However, since it also find solutions with much lower cost than simulated annealing, this is not necessarily a con. In fact, this suggests that the perturbations in the iterative local search algorithm allowed it to escape sub-optimal solutions more efficiently than simulated annealing.

We further discuss results for individual algorithms in the following sections.

Algorithm	Iterative Local Search
Neighborhood	2-swap
Perturbation	Double bridge move

Table 2. Main algorithm design aspects of iterative local search algorithm.

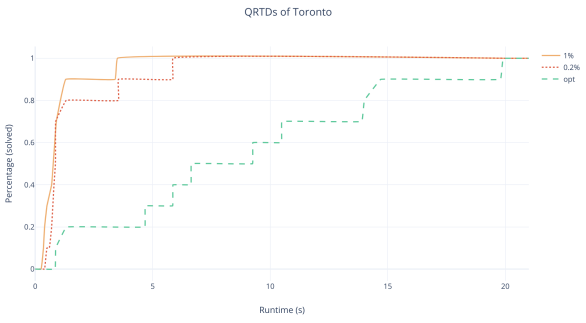
6.2 Local Search Evalutaion

6.2.1 Iterative Local Search.

Platform Detailed Description. The experiments were performed using Java on Intel Core i5 CPU, 2.3GHz with 8GB RAM.

QRTDs Plots. In this part, the Iterative Local Search was implemented on two symmetric TSP datasets from [12]: Toronto and Roanoake. Toronto has 109 locations, while Roanoke has 230 locations. For the set of experiments involving Toronto instance, the termination time is 10min. For the set of experiments involving Roanoke instance, the termination time is 40min. For each instance, I ran the Iterative Local Search 10 times.

Fig. 3. QRTDs Toronto



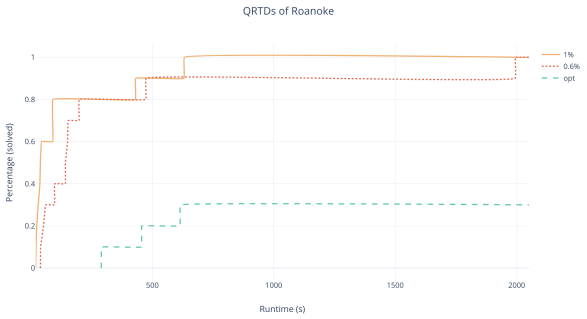
Dataset			Branch and Bound Reduced Matrix			Approximation MST Approximation		
Name	Nodes	Avg. Dist	Time (s.)	Sol. Qual	RelErr	Time (s.)	Sol. Qual	RelErr
Atlanta	20	4584732	0.64	2003763	0	0.012	2270785	0.1333
Boston	40	2864377	1.71	937443	0.0491	0.028	1028494	0.1510
Champaign	55	212792	9.21	56763	0.0783	0.04	61508	0.1684
Denver	83	548993	103.32	113199	0.1271	0.09	124987	0.2445
NYC	68	7316673	525.82	1695384	0.0902	0.065	1825255	0.1738
Philadelphia	30	4138324	126.88	1438010	0.0301	0.019	1626820	0.1654
Roanoke	230	7067078	1.27	790081	0.1779	0.731	789208	0.2054
San Francisco	99	5847251	99.31	919324	0.1347	0.137	1035130	0.2776
Toronto	109	10242268	175.03	1312838	0.1162	0.178	1595529	0.3566
UKansasState	10	116936	0.01	62962	0	0.005	65561	0.0413
UMissouri	106	738908	66.84	161536	0.2172	0.189	153063	0.1534
Berlin	52	29338	329.18	8402	0.1140	0.037	9550	0.2662
Cincinnati	10	334501	0.01	277592	0	0.007	296972	0.0698
ulysses16	16	12214	0.66	6859	0	0.09	7329	0.0685

Table 3. Comprehensive Table for Results for Iterative Local Search and Simulated Annealing. Since the iterative local search algorithm yielded the lowest cost solutions for all cities, we use its results as the baseline when computing relative errors.

Dataset			Local Search Simulated Annealing			Local Search Iterative Local Search	
Name	Nodes	Avg. Dist	Time (s.)	Sol. Qual	RelErr	Time (s.)	Sol. Qual
Atlanta	20	4584732	0.461	2015986	0.006	0.0035	2003763
Boston	40	2864377	0.886	915728	0.025	0.11	893536
Champaign	55	212792	1.552	54513	0.035	1.3405	52643
Denver	83	548993	2.59	106702	0.0601	10.56	100603
NYC	68	7316673	1.486	1637765	0.053	1.97	1555060
Philadelphia	30	4138324	0.6793	1416684	0.0148	0.02	1395981
Roanoke	230	7067078	30.55	691942	0.055	46.44	655454
San Francisco	99	5847251	4.063	877041	0.0825	14.54	810196
Toronto	109	10242268	4.776	1239038	0.053	8.789	1176151
UKansasState	10	116936	0.3955	62962	0.0	0.0	62962
UMissouri	106	738908	5.8153	143192.7	0.078	22.219	132824
Berlin	52	29338	0.5234	7916	0.0495	0.3	7542
Cincinnati	10	334501	0.2909	277952	0.0	0.0015	277952
ulysses16	16	12214	0.0305	7060	0.029	0.0035	6859

Table 4. Comprehensive Table for Results for Iterative Local Search and Simulated Annealing. Since the iterative local search algorithm yielded the lowest cost solutions for all cities, we use its results as the baseline when computing relative errors.

Fig. 4. QRTDs Roanoke



As can be seen in Figure 3 and Figure 4, the Iterative Local Search performed better on Toronto than Roanoke. For Toronto, the algorithm guarantees optimal result within 20 seconds, and guarantees the relative solution quality 0.2% within 10 seconds. For Roanoke, the algorithm ganrantess the relative solution quality 1% within 1000 seconds, and guarantees the relative solution quality 0.6% within 2000 seconds. However, as can be seen in Figure 4, the optimal result is not guaranteed even though I extended the termination time to 40min. This is because the stochastic natural of local search algorithm. The number of locations in Roanoke is 2 times greater than the number of locations in Toronto, therefore the search space

of Roanoke is much larger than Toronto due to the exponential natural. Hence, the probability of Iterative Local Search finding the optimal solution for Roanoke is smaller than finding the solution for Toronto even though Local Search algorithm itself is polynomial.

SQDs Plots. In this part, the Iterative Local Search was implemented on the same two cities as in the QRTDs part with the same parameters configuration.

Fig. 5. SQDs Toronto

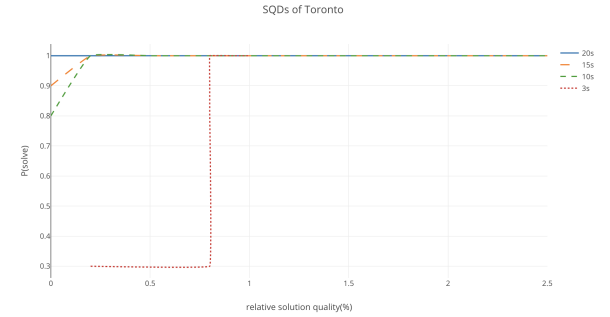
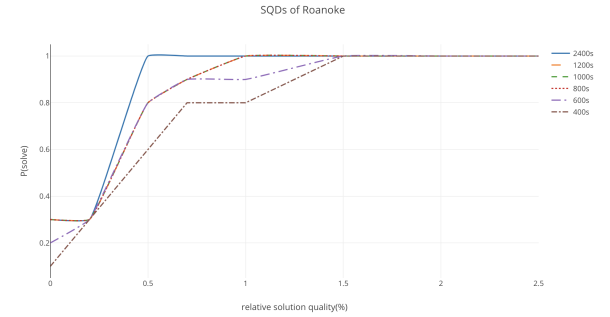


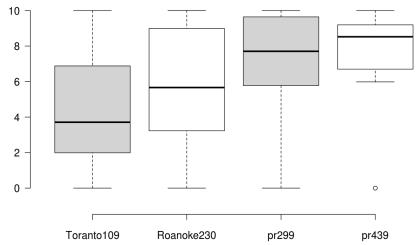
Fig. 6. SQDs Roanoke



As can be seen in Figure 5, running the algorithm 20s on Toronto instance can guarantee the optimal solution, running the algorithm 15s or 10s on Toronto instance can guarantee the relative solution quality smaller than 0.5%, and running 3s can guarantee the relative solution quality smaller than 1%. As can be seen in Figure 6, running the algorithm 2400s on Roanoke instance can only guarantee the relative solution quality 0.5%, running the algorithm 1200s, 1000s, or 800s can only guarantee the relative solution quality 1%, and running 600s or 400s can only guarantee the relative solution quality 1.5%. The SQDs plots indicates the same problem as QRTDs plots, Iterative Local Search performs better on smaller instance.

Box Plots. In this part, since all the data form [12] is comparatively small, I use pr299 with 299 locations and pr439 with 439 locations to make the plot comprehensive. I ran Iterative Local Search algorithm 10 times on each instance. For Toronto, the termination condition is 10min. For Roanoke and pr299, the termination is 20min. For pr439, the termination is 40min. In Figure 7, the y-axis is run time, which is the time when algorithm has last improvement before termination, after normalization.

Fig. 7. Box Plot



As can be seen in Figure 7, the run time of Toronto is comparatively even, the run time of Roanoke is also comparatively even but tall, the run time of pr299 is also even but a long lower whisker, and the run time of pr439 is clearly uneven and also has an outlier. This indicates that for the cities with less locations like Toronto and Roanoke, since the search space is smaller, it's easier to find a improvement. While, for the cities with more locations like pr299 and pr439, since the search space is larger, it's difficult to find a improvement if your random initial solution is already of good quality, which results in the long whisker and outlier.

6.3 Simulated Annealing

Platform Description. The simulated annealing experiments were performed using code written and executed on Python on a MacBook Pro (2013), with a 2.4 GHz Intel Core i5 processor and 16GB DDR3 RAM.

Qualified Runtime Distribution Plots. In Figure 8, we plot the qualified run-time distribution plots for Cincinnati (10 nodes), Atlanta (20 nodes), Toronto (109 nodes), and Roanoke (230 nodes) respectively. These cities serve as a good sample of cities with small and large node sizes in our data sets. This provides us with insight into how our algorithm performs on the two extreme ends with respect to the number of nodes in the graph. For all cities, 30 trials were performed with termination time set to 1 minute. As seen in the figures, the algorithm either reached optimal solutions or converged to a sub-optimal solution well before this cut-off time.

As expected, the total run-time of the algorithm increases with an increase in the number of nodes in the city. For the smallest cities, Cincinnati and Atlanta, with 10-20 nodes, the QRD curves for relative solution quality thresholds of $q = 0.4\%$ down to $q = 0\%$ fully overlap with one-another. In Figure 9, we isolate the QRD curves corresponding to the optimal solutions. We can see that while the algorithm is able to guarantee an optimal solution for Cincinnati within 0.45 seconds, it is only able to guarantee that roughly half

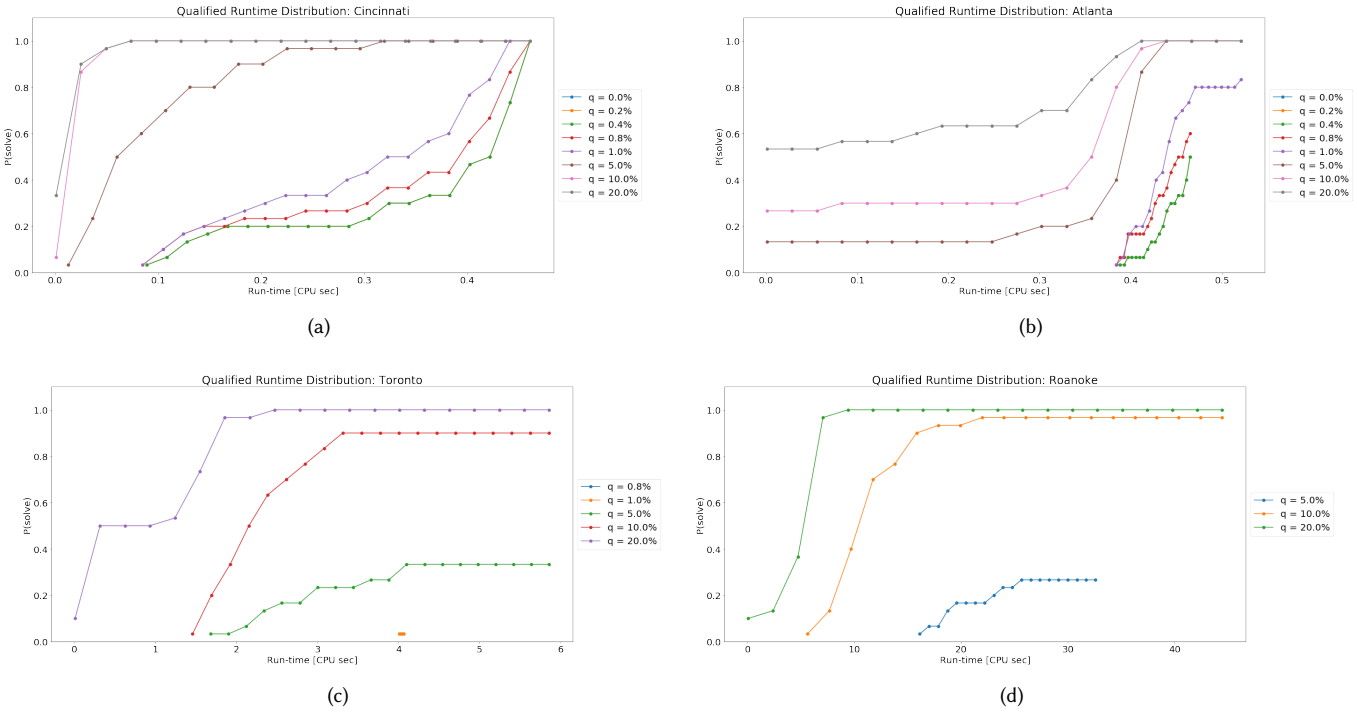


Fig. 8. Qualified Run Time Distributions for the Simulated Annealing Algorithm for Four Selected Cities (a) Cincinnati [10 nodes] (b) Atlanta [20 nodes] (c) Toronto [109 nodes] (d) Roanoke [230 nodes]. Various curves correspond to different values of solution quality thresholds. For instance, $q = 1\%$ indicates solutions that we less than or equal to the $(1.01)\times$ the best solution obtained across all algorithms. All results were obtained through 30 trials performed with termination time set to 1 minute.

the solutions will be optimal for Atlanta even when run for the full run time of 1 minute.

For Cincinnati, the algorithm was able to get a solution with quality within 5% of optimal fairly quickly. When the number of nodes doubles for Atlanta in Figure 8(b), we see that the algorithm demonstrates a similar pattern, but is not able to reach within 1% of the optimal quality.

For higher number of nodes in Figures (c) and (d) in Fig. 8, the algorithm continues to make improvements for much longer. For Toronto (109 nodes), we see improvements up until 6 seconds, while for Roanoke (230 nodes), we see improvements up until the full run time of 1 minute. In both these instances, the algorithm is not able to get to within 10% of the optimal value, and remains stuck at sub-optimal solutions for a large portion of the run-time. However, the slope of the QRD curve drops significantly when moving to thresholds of 1% and lower.

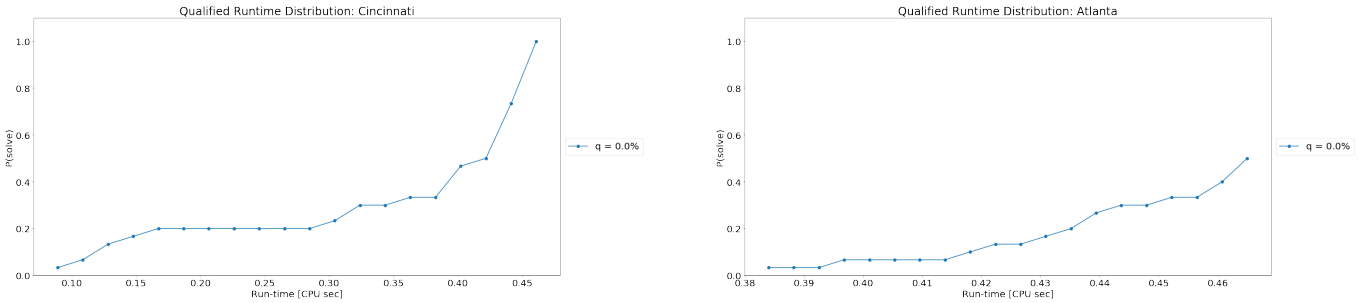
These figures demonstrate that simulated annealing was, in general, unsuccessful in yielding optimal solutions for large number of nodes , especially when compared to iterative local search. This trend exists despite the fact that we made multiple design choices that sacrificed run-time in exchange for improvements in solution quality. Furthermore, even for 20 nodes (Atlanta), the algorithm failed to guarantee optimality across all runs.

Solution Quality Distribution Plots. In Fig. 10, we plot the solution quality distributions (SQD) for the same cities as in the earlier section, with the same parameters. We plot the various curves at time intervals corresponding to the quartiles of the run-times for each city.

For all cities, the SQD plots form a concave curve, with a bend pointing towards the top left corner. The closer the curve is to touching the top left corner, the better the solution quality distribution. This indicates that as we allow for worse solution qualities, the algorithm can ensure a larger portion of the runs will satisfy the quality thresholds. The curves corresponding to lower run-times are further away from the desired top-left corner indicating that the algorithm required more time to reach high quality solutions in most runs.

For the smallest city with 10 nodes (Fig. 10(a)), running the algorithm for 0.46 seconds will ensure optimal results for all runs. However, for all other cities, the algorithm is unable to guarantee optimal solutions within the run time 1 minute. For the second smallest city with 20 nodes (Fig. 10(a)), we are able to ensure that roughly half of all executions run for 0.52 seconds will result in optimal solutions.

As the number of nodes in the city increases, we note that the SQD plots move away from the top left cover towards the bottom right. More importantly, we are no longer able to guarantee that any of runs will result in optimal solutions even when the algorithm is run



(a) (b)

Fig. 9. Qualified Run Time Distribution Plot for Optimal Results for Cincinnati and Atlanta. These are recreated here since the overlap with curves corresponding to higher relative solution qualities in Fig. 8

for up to 1 minute. For the largest city with 230 nodes (Fig. 10(d)), we are able to guarantee that running the algorithm for around 45 seconds will ensure that almost all runs will result in solution qualities only up to 10% of the optimal value. For Toronto (Fig. 10(c)), which had less than half the number of nodes as Roanoke, we can make a similar guarantee within 6 seconds.

Box Plots. In Fig. 11, we show the distribution of normalized run times for the four cities considered in our analysis thus far. The range of run-times varies drastically across these cities from 0.4 up to 50 seconds. Therefore, we normalized the run-times for each city by its largest run time, and multiplying by 10 to obtain a scale from 1-10 for ease of interpretability.

For most runs, we note that the quartile ranges (Q2 and Q4) are evenly spaced around the median. While there are outlier points for all cities, they are minimal. This suggests that our algorithm is, in general fairly robust in terms of run times. This informs us that the distribution of runtimes is fairly uniform i.e. if we run the algorithm a large number of times, it will tend to find the best solution around the same time. The distribution for Atlanta appears to be anamolous in this regard, and is highly skewed towards the lower values. This could indicate that for Atlanta, the greedy nearest neighbor search was more effective in finding a solution very close to optimal during most runs. Since the greedy nearest neighbor also has a random component in picking the first city in the tour, we do not notice this behavior over all runs.

Even though the distribution within Q2 and Q4 tends to be fairly uniform, we do note some long-tails. Particularly, there is a longer tail towards the maximum run-time for the larger cities. This could indicate that our algorithm is prone to get stuck in sub-optimal solutions that are difficult to escape. For these big cities, only in a few instances do we find improvements much later in the overall execution of the algorithm.

7 DISCUSSION

7.1 Branch-and-bound

According to the comprehensive table, branch-and-bound algorithm gave us decent results. On small graphs, specifically 4 graphs with less than 20 cities, branch-and-bound successfully found the optimal solution. While on large graphs, it usually ended up with a non-optimal in limited time, which in our experiments are 10 minutes. Compared to other algorithms, branch-and-bound performed better than 2-approximation but still fell behind those local search algorithms. But the results in comprehensive table clearly shows the advantage of branch-and-bound, that is it guarantees the optimal result. On small graphs, simulated annealing failed to find the optimal results, while branch-and-bound still succeeded. However, branch-and-bound is still not quite useful compared to simulated annealing in practice on large graphs, not to mention the other local search Iterative Local Search algorithm can achieve the optimal solution almost on all cases. Therefore, branch-and-bound still can be a choice for small problems, but with efficient implementations, any other heuristic methods may defeat branch-and-bound.

7.2 Approximation

From the comprehensive table, we can see that for most time, the MST Approximation gives the worst solution. Among all the test cases, the approximation only gave a better solution than Branch and Bound on instances 'UMissouri' and 'Roanake', and the solutions are always worse than the ones got by local search methods. However, the MST method runs really fast since the time complexity is only $O(n^2 \log n)$.

The solution may vary slightly if we select different roots or choose a different neighbor to explore first in depth first search. Since the running time is short, we can run the algorithm for each city being the root and run for two different search directions and pick the best solution. One search direction is to always visit the neighbor with smaller index before visit the ones with larger index, the other direction is the opposite. Solution qualities on most instances are improved after this modification. The relative errors for all cases are between 0.0413 to 0.3566 and the average relative error is 0.1767.

Since the approximation method runs very fast and can only give a solution whose quality is not very good but also not too bad, the method can be used when solution quality is not very important but the running time is limited. For example, it can save a lot of time comparing with Branch and Bound method when dealing with large graphs, while getting a solution with similar quality. Also, the solution got by the approximation method may be used in other methods to generate a lower bound and save some running time.

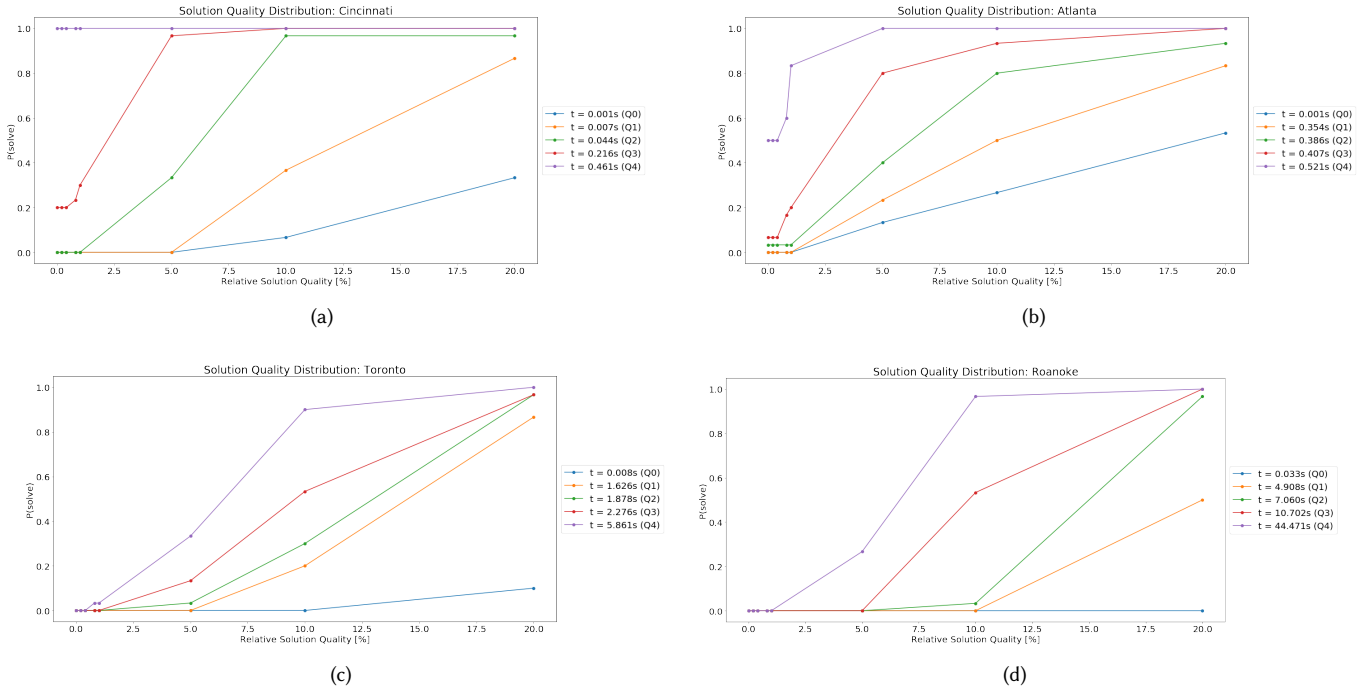
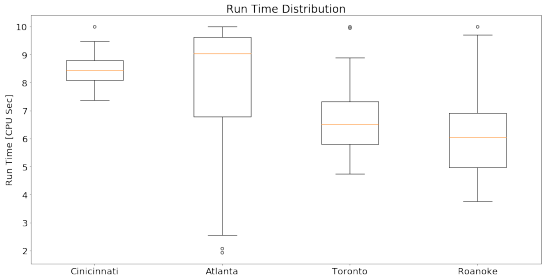


Fig. 10. Solution Quality Distributions for the Simulated Annealing Algorithm Four Selected Cities (a) Cincinnati [10 nodes] (b) Atlanta [20 nodes] (c) Toronto [109 nodes] (d) Roanoke [230 nodes]. Results we obtained through 30 trials performed with termination time set to 1 minute.

Fig. 11. Normalized Box Plot Showing the Distribution of Runtimes across four cities.



7.3 Local Search: Iterative Local Search

From the comprehensive table, we can see the result obtained from Local Search is better than Branch and Bound for most time, and only for small instances like Atlanta and Cincinnati when Branch and Bound can terminate within 10min, they are the same. While, from the table, Local Search is always giving better result than Approximation. As we demonstrated in Local Search Evaluation part, the Local Search will give a better result, even optimal result, within a time limit if the search space of the instance is small. However, the Local Search can perform very bad when search space is very large, and it could stick at one result and has no improvement after a long period of time. Therefore, in the real world scenario, if the search space is very large and cannot get a good result within a certain period of time using Local Search, it's better to choose Approximation. Since Approximation will give a result in $O(n^2 \log n)$ with a certain quality, the result could be better than Local Search within a certain period of time if the search space is very large.

7.4 Local Search: Simulated Annealing

From the results in our comprehensive table in Tables 3 and 4, we see that the simulated annealing algorithm tends to result in better solution qualities than both Branch and Bound and the MST Approximation Algorithm, but worse than Iterative Local Search.

When comparing just the local search approaches, the iterative local search algorithm dominates simulated annealing in both quality and run-time. It should however be noted that when designing the simulated annealing algorithm, we intentionally made choices that sacrificed run-time in favor of solution quality.

When comparing to the MST approximation, simulated annealing almost always has much lower relative errors. When compared to the Branch and Bound approach, simulated annealing yields results that tend to be higher quality much faster. Moreover, it also found the optimal solutions for UKansasState and ulysses16. Thus, simulated annealing appears to be better alternative to both Branch and Bound and MST Approximation in terms of overall solution qualities as well as run-times.

Although, we generally obtain good solution qualities with simulated annealing, our QRD and SQD analyses suggest that it is unable to guarantee optimal results across all runs even when run for a generous amount of time, except for the smallest city with 10 nodes. When the number of nodes increases to 100-200, we are only able to

guarantee that almost all runs of the algorithm will yield solutions with qualities up to 10% of the optimal.

We see a few points of adjustment that might improve the current simulated annealing algorithm. Firstly, the major problem with the algorithm seems to be that it gets stuck in sub-optimal neighborhoods. Incorporating random perturbations such as the double-bridge strategy in our Iterative Local Search might alleviate this problem. Secondly, as simulated annealing is highly flexible meta-heuristic, it provided us with a large number of tunable parameters. In many cases, we picked the parameters based on trial and error. When attempting to solve a particular problem, we could perform a parameter-tuning step to obtain the correct parametrization of the algorithm specific to the problem at hand. This can be done through a sufficiently expansive search over the parameter space, or using more structured methods such as Bayesian hyperparameter optimization.

8 CONCLUSION

In general, this report analyzes some classic algorithm towards Traveling Salesman Problem and presents their performance by comparison. Specifically, the four algorithms discussed are Branch-and-bound, 2-approximation using MST, Iterative Local Search and Simulated Annealing. In total, we test all four algorithms on 13 graphs with size varying from 10 vertices to 230 vertices. The results show that Iterative Local Search generally defeats all three other algorithms, no matter how large the input graph. On small graphs, except for 2-approximation algorithm, all the other algorithms successfully compute the optimal solution. While on large graphs, Iterative Local Search stood out with least running time and the best results (not necessarily optimal). Simulated Annealing may has worse results on large graph than Iterative Local Search, but it is still better than Branch-and-Bound on time performance and solution quality. Branch-and-bound as expected falls behind because of it huge search space. As for the 2-approximation algorithm, the solutions obtained always have higher relative error than other algorithms, since they are just approximate results and the approximation factor 2 is quite loose in TSP. In a nut shell, heuristic algorithm may be the best choice in practice and there are various improvements we can make to get a even better performance. Meanwhile, if the accuracy requirement is not strict, approximation algorithm may be satisfying if implemented in an efficient way.

REFERENCES

[1] E. H.L. Aarts and P. J.M. van Laarhoven. 1989. Simulated annealing: An introduction. *Statistica Neerlandica* (1989). https://doi.org/10.1111/j.1467-9574.1989.tb01245.x

[2] Ernesto Bonomi and Jean-Luc Lutton. 1984. The N -City Travelling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm. *SIAM Rev.* (1984). https://doi.org/10.1137/1026105

[3] V. Černý. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications* (1985). https://doi.org/10.1007/BF00940812

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[5] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. 1996. Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* (1996). https://doi.org/10.1109/3477.484436 arXiv:1706.06208

[6] David S Johnson and Lyle A McGeoch. 1997. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization* (1997), 215–310. http://142.103.6.5/{~}hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf

[7] B. W. Kernighan and S. Lin. 1970. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal* (1970). https://doi.org/10.1002/j.1538-7305.1970.tb01770.x

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by simulated annealing. *Science* (1983). https://doi.org/10.1016/j.ultsonch.2011.06.003 arXiv:arXiv:1011.1669v3

[9] Jon Kleinberg and Eva Tardos. 2005. *Algorithm Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics* (1953). https://doi.org/10.1063/1.1699114 arXiv:5744249209

[11] Chiung Moon, Jongsoo Kim, Gyunghyun Choi, and Yoonho Seo. 2002. An efficient genetic algorithm for the traveling salesman problem with precedence constraints. *European Journal of Operational Research* 140 (08 2002), 606–617. https://doi.org/10.1016/S0377-2217(01)00227-2

[12] Gerhard Reinelt. 1991. TSPLIB–A Traveling Salesman Problem Library. *ORSA Journal on Computing* 3, 4 (1991), 376–384.

[13] G. Shang, Z. Lei, Z. Fengting, and Z. Chunxian. 2007. Solving Traveling Salesman Problem by Ant Colony Optimization Algorithm with Association Rule. In *Third International Conference on Natural Computation (ICNC 2007)*, Vol. 3. 693–698. https://doi.org/10.1109/ICNC.2007.675

[14] Xiaohu Shi, Yanchun Liang, H. P. Lee, C. Lu, and Q. X. Wang. 2007. Particle swarm optimization-based algorithms for TSP and generalized TSP. *Inf. Process. Lett.* 103 (2007), 169–176.

[15] Ümit V. Çatalyürek. 2018. Lecture notes from CSE 6140 (2018).