Algorithms for The Travelling Salesman Problem

Yu Yu
Georgia Institute of Technology
251 Tenth St NW
Atlanta, GA 30318
(+1) 404-247-8221
yyu@gatech.edu

Jiawei Wu
Georgia Institute of Technology
251 Tenth St NW
Atlanta, GA 30318
(+1) 470-232-7358
jwu680@gatech.edu

Liangyu Chen
Georgia Institute of Technology
251 Tenth St NW
Atlanta, GA 30318
(+1) 470-659-3387
liangyu.chen@gatech.edu

Anbang Ye
Georgia Institute of Technology
251 Tenth St NW
Atlanta, GA 30318
(+1) 404-268-9874
aye42@gatech.edu

ABSTRACT
The traveling salesman problem (TSP), also known as the traveling salesperson problem or TSP is one of the most famous NP-complete problem in optimization. The problem statement is as following, "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" [1] TSP has been used intensely as a benchmark for evaluating optimization algorithms since the day it was proposed in the 1930s. In this work, we will demonstrate four algorithms for solving the TSP problem---Branch-and-Bound, Nearest Neighbor Heuristics, Genetic Algorithm and Simulated Annealing. We will cover our approaches, analyze the worst-case running time for those algorithms, and evaluate their performances with empirical analysis. At the end, we will discuss the pro and cons of each algorithm and how we can improve the optimization algorithm in the future.

Keywords
Traveling Salesman Problem, Optimization

1. INTRODUCTION
The traveling salesman problem (TSP) is a well known optimization problem. The problem asks to find the shortest path that visits each city exactly once and returns to the starting point given cities and the distance between every two cities. TSP has been proved to be NP-complete, which means, in the worst case, no algorithm can solve this problem within polynomial time with respect to the number of cities. Because of the difficulty of the problem, TSP has been widely used in examining the efficiency of optimization algorithms since the problem was formally introduced in 1930. It is still an active research problem in computer science and optimization algorithms nowadays. [1]

There are many proposed methods to solve the TSP, or at least get a suboptimal solution. Some algorithms can guarantee to find the exact solution given sufficient time, such as inefficient brute-force algorithm, which runs in $O(n!)$ [2], dynamic programming [3], various backtracking algorithms (branch and bound). Other algorithms do not necessarily generate an exact solution but a solution with similar quality to the optimal ones. including greedy algorithms with proper heuristics, local search algorithms such as simulated annealing and genetic algorithms. Those algorithms can quickly give an approximate solution within a reasonable time. Some of state of art algorithms in this category can solve TSP within 0.05% of the optimal cost. [4]

In this work, we will implement and evaluate four different algorithms on TSP. These are Branch-and-Bound, Nearest Neighbor Heuristics, Genetic Algorithm and Simulated Annealing. Among the four, Branch-and-Bound is exact algorithms, Nearest Neighbor Heuristics, Genetic Algorithm and Simulated Annealing are approximation algorithms.

Table 1. Algorithms

Algorithm Name	Is Exact	Approach
Branch-and-Bound	Yes	Branch-and-Bound
Nearest Neighbor Heuristics	No	heuristic algorithms
Genetic Algorithm	No	Local Search
Simulated Annealing	No	Local Search

2. PROBLEM DEFINITION

Generally speaking, TSP is a name for a kind of problems. In this work, we will deal with Euclidean TSP where the distance is the Euclidean distance between each pair of cities.

We define the TSP problem as, "given the x-y coordinates of N points in the plane (i.e., vertices), and a cost function c(u, v) defined for every pair of points (i.e., edge), find the shortest

simple cycle that visits all N points." [5]

3. Related Works

3.1 Branch-and-Bound

The Branch and Bound algorithm is presented to solve the TSP (traveling salesman problem). The problem is broken up into smaller subproblems and the corresponding lower bound is calculated during the branching process. Eventually, a feasible solution is found that beats the global lower bound. According to R. Radharamanan and L.I. Choi, the algorithm is considered as the general form of solution for STS and MTS (symmetric and asymmetric single and multiple travelling salesman problems), and the transportation routing problems with capacity restrictions. [30] Donald L. Miller and Joseph F. Pekny shows that the algorithm can find exact solutions for some classes of problems and even for problems with large datasets, but for other classes it tends to be useless.[31]

3.2 Nearest Neighbor Heuristic

The nearest neighbor heuristic is a greedy algorithm to solve the TSP, which is considered as a sub-optimal method. It was firstly introduced by Gregory Gutin, Anders Yeo and Alexei Zverovitch in 2002 and then developed in 2007.[25] Basically, it starts at one city and connects with the nearest city with lowest distance cost. Then, it repeats until all cities are visited. After we compare all starting points, we can find a path with the lowest total cost. However, this algorithm does not guarantee the optimal path but can provide a good estimation. In 2004, Jorgen Bang-Jensena, Gregory Gutin, and Anders Yeo published a paper to analyze the situations where the greedy algorithm fails.[27] It can sometimes miss shorter routes which are easily noticed. In the worst case, the algorithm may result in a tour that is much longer than the optimal tour.

3.3 Local Search – Genetic Algorithm

Genetic Algorithm is intensively used to solve Travelling Salesman Problem (TSP). Genetic algorithm is derived from evolutionary algorithms, which are probabilistic search algorithms which simulate natural evolution. Holland (1975) [11] firstly introduced genetic algorithms. In these algorithms, the search space of a problem is represented as a collection of individuals. The quality of an individual is measured by an fitness function. In every iteration, the algorithm select parents to produce next generation. There are many options regarding the implementation of genetic algorithm.

Larranaga(1999) [12]summarized existing implementation details with respect to the key operator in genetic algorithm, including TSP solution representation, the selection method, the crossover method and the mutation method.

So far, there are many literatures proposed improved methods, or studied the effectiveness of existing operator implmentation. Üçoluk [13] adopted an inversion sequence as the representation of apermutation, which outperforms PMX in convergence rate. Deng proposed a strategy that implements initialization based on k-means algorithm, whose performance beats the random initial population and greedy initial population. Also there are literatures claimed that it is necessary to combine other methods, like 2-opt local search or simulated annealing, with genetic algorithm to arrive at high quality solutions[14,15].

3.4 Local Search - Simulated Annealing

Simulated annealing (SA) is an approximation algorithm that explores the solution space to find an optimal solution for the given optimization problem. The "Annealing" in the name of SA comes from the "annealing" in metallurgy. It means to control the temperature of metal during cooling. In the 1970s and the 1980s, several approaches that use methods that resemble SA were developed by scholars independently on different occasions, "including Pincus (1970), Khachaturyan et al (1979, 1981), Kirkpatrick, Gelatt and Vecchi (1983), and Cerny (1985)." [6] The algorithm formally got its well-known name "simulated annealing" in 1983 by Kirkpatrick, Gelatt Jr., Vecchi,[7] who use SA to solve the TSP.

To avoid local minimum, SA adopts the idea of accepting a worse solution with a probability called acceptance probability. Besides, instead of fixing the acceptance probability, SA adjusts the acceptance probability according to temperature (T). The acceptance probability in SA can be expressed as a function of the cost of current solution, the cost of next adjacent solution and a non-negative temperature $P(c_{curr}, c_{next}, T)$. [6] In this equation, a higher T means we are more likely to make a move towards a worse solution ($c_{curr} < c_{next}$), lower T means we are more cautious in taking a movement that can decrease the quality. If T goes to zero, we will only take a move that increases the quality of solution (the same as hill climbing, $c_{curr} > c_{next}$). To make this SA algorithm converge, T is set to some positive initial value and decrease along with time till a lower bound is reached. [6] SA gives some flexibility on how to implement $P(c_{curr}, c_{next}, T)$, one generally applicable way [7] of defining the acceptance probability is described in the following equation.

$$P(c_{curr}, c_{next}, T) = \begin{cases} 1, & c_{curr} > c_{next} \\ e^{-\frac{e_{next} - e_{curr}}{T}}, & c\_\{curr\} < c\_\{next\} \end{cases}$$

As we have mentioned before, SA uses a controlled temperature to determine the probability of accepting worse movement. The temperature is gradually cooled with respect to the running time. One way to perform the cooling is to use a predefined cooling schedule, like linear, quadratic additive, and exponential. However, because the converge time of SA "strongly depends on the 'topography' of the energy function and on the current temperature" and all those variables are usually unknown to the algorithm [6], choosing a suitable cooling schedule is complicated and usually requires experiments to figure out. There are also other approaches that connect cooling speed with the search process, for example, Adaptive Simulated Annealing [8] and Thermodynamic Simulated Annealing [9].

4. Methods

4.1 Branch-and-Bound

4.1.1 Algorithm Overview.

The basic idea is depth first search optimized with branch cut of minimum cost. For depth first search, we expand the path by adding a new stop that doesn't exist in the current path and is closer than all the rest unselected stops (implemented with a distance rank matrix named self.cost_matrix_index). In the whole process, the technique of MST(Minimum Spanning Tree) is used to estimate the cost of a semi-path, which serves as a clue to decide whether to go further or cut the branch. When the path includes all the stops,

which means we find one feasible solution, the total cost will be calculated and compared with that of the best path.

4.1.2 Strongness and Weakness.

In general, Branch and Bound is able to find the exact solution if time permits because it searches through the whole decision tree. The weakness is obvious - too much time to run the algorithm.

4.1.3 Data Structure Used in Implementation.

- A N*N matrix to record the distance cost of all city pairs
- A N*N matrix to record the distance rank of all city pairs
- A map to record distance and the edge coordinates
- A atMostN*1 path list to record the current path in backtracking
- A N*1 path list to record the best path
- A vector to record the cost of the best path
- A map to record visited path and its lower bound cost

4.1.4 Pseudocode.

Algorithm 1 Branch and Bound

```
part I: calculate the cost info
n ←number of cities                                   ▷ get pairwise distance matrix
for city1 from 0 to n do
    for city2 from 0 to n do
        matrix[city1][city2] = distance between city1 and city2
    end for
end for
for start from 0 to n do                              ▷ edge distance mapping
    for end from start + 1 to n do
        mapping[matrix[i][j]] = [i, j]
    end for
end for
startTime ← currentTime()                             ▷ start timer

part II: branch and bound:Kruskal's Minimum Spanning Tree
Algorithm
subtree ←path without current start and end nodes
key ←generate an unique identifier
if key is visited then                                ▷ check if key exists
    return visited[key]
end if

initialize tree map as empty dictionary and totalCost = 0
while there are unused edge do                        ▷ connect edges along the way
    cost ← next smallest edge
    if no cycle formed then
        totalCost ← totalCost + cost
        Connect related edges in tree
    end if
end while
visited[key] ← totalCost

part III: depth first search optimized with branch cut
if len(path) == n then                    ▷ return the results if path is complete
    totalCost ← totalCost + cost
    if totalCost < bestCost then
        update information of best path
    end if
    return
end if

if currentTime() − startTime > cutoff then ▷ stop if time exceeds cutoff
    return
end if
for start from 0 to n do ▷ otherwise, backtracking on the next possible value
    nextPoint ← the next nearest neighbour node
    if nextPoint not in path then
        totalCost ← totalCost + cost
        path.append(nextPoint)
        if partII(path) + newCost < bestCost then
            partIII(path, newCost)
        end if
        path.pop()
    end if
end for
```

4.1.5 Time and Space Complexities.

The algorithm's time complexity is dominated by the part III depth first search O(n!) though part I and part II take O(n^2) for initializing n*n array and iterating through all the edges(n is the number of points). The algorithm's space complexity is dominated by the n*n array we use, so it is O(n^2).

4.1.6 Optimization.

We improve the efficiency of our algorithm in several ways:

1. The use of memoization in MST saves meaningless efforts of solving the same problem many times by checking the unique identifier of the problem at the beginning of part II.
2. In part III, our Expand step always picks the neighbor with the shortest distance from the end of the current path. In this way, we should be able to reach the best solution or a good enough sooner to save time by cutting more branches.

4.2 Nearest Neighbor Heuristic

4.2.1 Algorithm Overview.

Here we choose the nearest neighbor algorithm to construct heuristics. The detailed algorithm is as follows: firstly, we can calculate the cost matrix of pair cities in advance for reference; secondly, choose a city called x and add it to the path; thirdly, find an unvisited city connecting to it with the lowest distance cost and add it to the path; finally, update the current city and repeat the process, until all cities are visited. This is one sub-optimal solution in the greedy algorithm.

4.2.2 Strength and Weakness.

The nearest neighbor algorithm is easy to implement and executes quickly. However, it usually cannot find the globally optimal path. Sometimes, the algorithm may result in a tour that is much longer than the optimal tour when it misses shorter routes which are easily noticed.

4.2.3 Data Structure Used in Implementation.

- A N*N matrix to record the distance cost of all city pairs
- A N*1 path list to record the best path
- A set object to record visited cities

4.2.4 Pseudocode.

As descried, we will calculate the cost matrix first and then find a sub-optimal path using the greedy algorithm. The pseudocode is as follow.

Algorithm: Nearest Neighbor Algorithm

```
# part I: calculate the cost matrix
n = num of cities
matrix = [[0] * n for _ in range(n)]
for i in range(n):
    for j in range(n):
        matrix[i][j] = the distance cost of city i and city j

# part II: find a sub-optimal path using greedy algorithm
cur, path, cost = x, [start], 0
visited = {x}
while |visited| < n:
    cities = [i for i in range(n) if i not in visited]
    costs = [matrix[x][i] for i in cities]
    next = find the corresponding city in cities with min(costs)
    visited.add(next)
    path.append(next)
    cost += min(costs)
    cur = next
cost += matrix[cur][x]
return path, cost
```

4.2.5 Time and Space Complexities.

In order to calculate the cost matrix, the time complexity is $O(n^2)$ and the space complexity is $O(n^2)$. To find a sub-optimal path, it requires to find the nearest neighbor one by one and append it in each step, whose time complexity is $O(n^2)$ and space complexity is $O(n)$. In summary, the algorithm's time and space complexities are both $O(n^2)$.

4.2.6 Approximation Ratio.

The nearest neighbor algorithm does not have a fixed approximation ratio. The approximation ratio depends on the size of the cities in the graph. It usually performs well when the size is small, and cities are clustered. However, it may not perform well when the size is big, and cities are too disperse.

4.2.7 Choose a Starting City x.

Based on our algorithm, we need to choose a starting city first. Here, we adopt a simple method to loop through all cities as starting cities and record their best path and best total cost. Then, we find the best results by choosing the starting city with the lowest best total cost. After we add this step, the time complexity will increase form $O(n^2)$ to $O(n^3)$ while the space complexity is still $O(n^2)$.

4.3 Local Search 1: Genetic Algorithm

4.3.1 Introduction.

Genetic algorithms are a family of computational models inspired by evolution and natural selection process. The algorithm starts from a series of *individuals*, namely solutions, as our first generation or *population*. Individuals with higher fitness function will have a larger probability to be selected as the parents of next generation, so that "good genes" will be kept in the population. Next, the selected parents conduct *crossover* that is a process of exchanging gene fragment. As in the real biology world, the genes will have a probability to mutate and thus our algorithm will have an opportunity to jump out of local optima. Lastly, the individuals with lower fitness score are abandoned to keep the population size remain stable.

4.3.2 Implementation Design and Details.

4.3.2.1 Initialization.

The quality of initial population plays an important role in solving TSP problem [16]. Our path is represented by a "numpy" array, with a length of the cities number. The path along with its fitness score computed as $fitness = \frac{1}{path\_distance}$ make up an individual in population.

The typical way is to construct an initial population randomly, while we choose to initialize it by using a heuristic algorithm nearest neighbor. The choice is basically a tradeoff between diversity and convergence rate. According to our experiment result, as shown below, the nearest neighbor initialization not only have a faster convergence rate but also yield a better solution quality. Therefore, we choose the nearest neighbor as our initialization method.

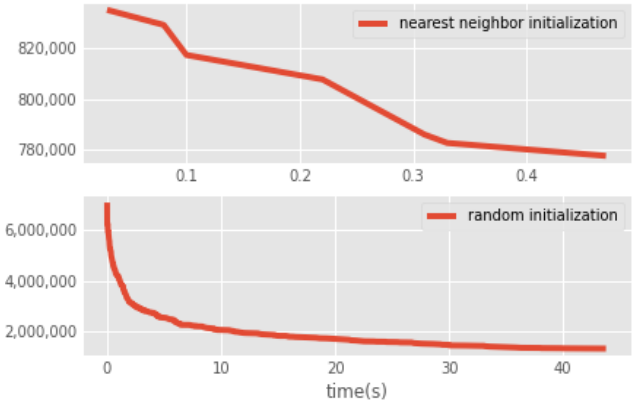

Figure 1. Convergence Comparation on Roanoke

4.3.2.2 Selection

Selection refers to the process of selecting individuals with high fitness value in current population. The most popular choice is proportionate roulette wheel, in which we select parents according to each individual's normalized fitness value.

$$p_s(a_j) = \frac{f(a_j)}{\sum_{i=1}^{n} f(a_i)}$$

However, the drawback of this method is that the some individual with low fitness value only keep theoretical probability of being selected, and "super" individual with high fitness value will dominate the whole population in a few generation, and thus jeopardize the diversity of population [17].

To avoid the premature convergence, we choose to use a rank based selection, where the selection probability is calculated based on its ranking, so that every individual will have a decent chance to be selected.

$$p_s(a_j) = \frac{1/(rank(a_j)+c)}{\sum_{i=1}^{n} 1/(rank(a_i)+c)}$$

Such method can be computationally expensive because it sorts the population every time, but our experiment result shows such sacrifice is acceptable.

4.3.3 Crossover.

Available Crossover method includes *Partially-mapped crossover (PMX)*[18], *Cycle crossover(CX)*[19],*Order based crossover(OX)*[20]. Here we choose PMX as our method.

PMX operator creates offspring in the following way. First, it randomly selects two cut points along the parents' path, between which we call it switching area.

Then the switching area of offspring1 is filled up by copying elements from the second parent path. the non-switching area is filled up by the elements from the first parent path. In case a city is

already present in the switching area, it is replaced according to the mapping defined by the elements in switching area.

$$parent_1(i) \leftrightarrow parent_2(i), i \in \{cut1, cut2\}$$

The same logic applies to offspring2.

4.3.3.1 Mutation

Mutation is a genetic operator used to maintain genetic diversity, so that the offspring could have a probability to jump out of local optima. Popular implementation methods are Displacement mutation (DM) [22], Exchange Mutation (EM) [23] and Insertion mutation (ISM) [24]

In our algorithm, we applied ISM to Mutation process, because compared with other method, ISM could bring more diversification to our searching process. The insertion mutation operator randomly chooses a point in the path, removes it from this path and inserts it in another places.

Another option here is whether to conduct local search like 2-opt search on the mutated individual. According to our test, a local search will significantly increase computational time whereas improve little solution quality, so we didn't add it to our algorithm

4.3.3.2 Pseudocode Code

Algorithm 1 Local Search Genetic Algorithm

```
Initialize Population
population ← ∅
for i = 1 to population size do
    start ← randomly_assigned_city
    individual ← nearest_neighbor_algorithm
    add individual to population
    Evaluate fitness for individuals in population
    Sort population
end for
while new_best_distance > previous_best_distance do
    Randomly Select two parents based on rank
    Crossover and produce two new individuals
    if isMutate then Random choice based on mutation probability
        Mutate new individuals
    end if
    Evaluate fitness for the new individuals
    Add new individuals to population
    Sort population
    Abandon the last two individuals
end while
```

4.3.3.3 Complexities Analysis

4.3.3.4 Time Complexity

The time complexity for each generation can be calculated as follows

- Initialization: $O(n^2 logn) + O(Population * n)$ for sorting all edges, conducting nearest neighbor algorithm, and computing distance
- Selection: $O(Population * logPopulation)$ mainly for sorting all individual
- Crossover: $O(n)$ To form offspring, it needs to iterate over parents
- Mutation: $O(1)$ Natural Selection: $O(Population * logPopulation) + O(n)$ for computing the distance and sort population

4.3.3.5 Space Complexity

$O(n^2)$ **:** The major space consumption is for the adjacency matrix that stores the Euclidean distance between cities.

4.3.3.6 Hyper-Parameter

Table 2. Hyper-parameter (Genetic)

Hyperparameter Name	Value
Population Size	30
Mutation Probability	0.8

All hyper-parameters in our algorithm is a matter of the tradeoff between convergence rate and solution quality. Larger population size and mutation probability leads to a better population diversification, and thus enhance the opportunity of jumping out of the local optima. Since our experiment result shows that most execution of our algorithm converged quickly (less than 10s). Hence, we chose relatively larger hyper-parameters.

4.4 Local Search 2: Simulated Annealing

Our implementation of Simulated Annealing was adapted from the standard SA solution for TSP [7] introduced by Kirkpatrick.

4.4.1 Implementation Design and Details.

4.4.1.1 General Procedure

- Set max temperature, minimum temperature, and set initial temperature to the max temperature. Set the cutoff iteration number.
- Initialize a solution
- Terminate the algorithm if one of the following conditions is met. First, reaches the max iteration number. Second, exceeds the time cutoff. Third, temperature is lower than a predefined minimum threshold. Third, no better cycle found in last 1 second.
- Generate a neighborhood solution through 2 opt.
- Decide whether to accept the neighborhood solution or not.
- Decrease temperature according to the cooling schedule.
- Repeat steps 3 to 7 until the algorithm is terminated.

4.4.1.2 Algorithm Design

We replaced the random initialization with a simple greedy algorithm that expand an existing Hamilton path by adding the closest vertex to any of its ends to the path until all vertices is added. We convert the Hamilton path into a Hamilton cycle by connecting the start and the end. We use a modified 2 Opt method to find neighborhood for given states. In standard 2 Opt, we reverse the order of traversing the path between vertex A and vertex B (inclusively), where A and B are randomly chosen with replacement from the set of all vertices (V). In our implementation, we put some restriction on the separation distance between vertex A and B when the temperature is lower than 10. We select only vertices that are no more than 10 vertices away from vertex A on the current path when T is less than 10. The purpose here is to avoid changing the path too rapidly when temperature is low. We use the acceptance function that is described in Section 3.4 and we use linear cooling schedule below. We do not use random start in our implementation. Table 2 shows some hyperparameters we used in our SA approach.

$$T = T_{min} + (T_{max} - T_{min}) \times \frac{IterMax - step}{IterMax}$$

Table 3. Hyper-parameter (Simulated Annealing)

Hyperparameter Name	Value
Temp_Max	100
Temp_Min	1
IterMax	60000
cooling	"linear"

4.4.2 Pseudocode

Algorithm 1 SimulatedAnnealing

```
Input:
    State, tempMax, tempMin, IterMax, cutOff
//Initialization
State ← initializeStateWithGreedyAlgorithm(State)
bestCost ← costOf(State)
currCost ← bestCost //cost of current path
bestCycle ← States
t ← tempMax
step ← 0 //base case
while step < IterMax and t >= tempMin do
    neigh ← 2Opt(States) //get a new neighbor
    neighCost ← costOf(neigh)
    if neighCost < bestCost then
        bestCost ← neighCost
        bestCycle ← neigh
        add trace checkpoint
        if more than 1 sec from last update then
            converge, terminate program, return trace, bestCost, bestCycle
        end if
    end if
    //check whether or not we should accept current neighbor
    if currCost − neighCost > log(random() + ϵ) ∗ t then
        //incase currCost − neighCost is too large and overflow
        State ← neigh
        currCost ← neighCost
    end if
    if time exceed then
        terminate program, return trace, bestCost, bestCycle
    end if
    //update t
    t ← tempMin + (tempMax − tempMin) × ((iterMax − step)/iterMax)
//linear cooling
    step ← step + 1
end while
return trace, bestCost, bestCycle
```

4.4.3 Data Structure

- A list of vertices that represent a cycle (the start is connected to the end) where each element is in the following form (x_location, y_location, id).
- A cost matrix of size N by N that stores the distance between two cities.

4.4.4 Strongness and Weakness

The advantage of SA algorithm is that it can get approximate solution within a reasonable time. However, SA algorithm doesn't guarantee to find exact solutions, it usually ends up with some sub-optimal solutions.

4.4.5 Complexity

4.4.5.1 Time Complexity

The 2 Opt function generate a neighbor in $O(n)$. Calculating the cost of the new solution takes $O(n)$ time. Deciding whether to accept the new solution or not is $O(1)$. Update the temperature takes $O(1)$. The running time through the loop body once is $O(n)$. The number of loops is bounded only by a constant $MaxIter$, the cutoff iteration number (see 4.4.1 step 2 and Equation 2). The final running time of our algorithm is $O(n \times MaxIter) = O(n)$.

4.4.5.2 Space Complexity

The space requirement of the simulated annealing algorithm is just the space used to store the cost matrix and state (ignore the trace file), which is $O(n^2)$.

5. Results

All these algorithms have been tested on a platform with following configuration

- Mac Monterey 12.0.1
- Apple M1
- 16GB of RAM
- Python 3.9.6
- Numpy 1.21.2

5.1 Branch-and-Bound

5.1.1 Overall Performance

In terms of running time, the observation is that the larger the dataset is, the longer running time it will need to find a solution with Branch and Bound. This is as expected because the time complexity is O(n!) in the worst case. The two shortest running time is 0.01 and 0.02 seconds from the smallest datasets of UKansasState and Cincinnati, which have only 10 nodes per dataset. The Boston dataset has twice as many as the nodes of the Atlanta dataset but the running time of the former is almost 30 times more than the latter one. The running time does increase extraordinarily as the size of the dataset goes up.

There are also exceptions. Roanoke has 230 locations (the most) but only takes 35.85 seconds to run in the test. The relative error of it is the worst, 19%. San Francisco is in a similar situation. Therefore, Branch and Bound doesn't perform stably in the real-world datasets and need to try different approaches.

Table 4. Banch-and-Bound Comprehensive Table

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	14.81	2003763.0	0.00
Berlin	1.07	8091.0	0.07
Boston	432.16	949538.0	0.06
Champaign	379.12	53493.0	0.02
Cincinnati	0.02	277952.0	0.00
Denver	15.79	109238.0	0.09
NYC	41.36	1774195.0	0.14
Philadelphia	192.59	1437553.0	0.03
Roanoke	35.85	779829.0	0.19
SanFrancisco	1.75	896722.0	0.11
Toronto	336.34	1220114.0	0.04
UKansasState	0.01	62962.0	0.00
UMissouri	539.17	156867.0	0.18

5.2 Nearest Neighbor Heuristic

5.2.1 Overall Performance

For most of the datasets, they run very quickly and have low relative error.

Table 5. NN Heuristic Method Comprehensive Table

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.00	2,039,906	0.02
Berlin	0.01	8,182	0.08
Boston	0.01	1,029,009	0.15
Champaign	0.01	61,828	0.17
Cincinnati	0.00	301,259	0.08
Denver	0.07	117,619	0.17
NYC	0.02	1,796,653	0.16
Philadelphia	0.00	1,611,716	0.15
Roanoke	1.09	773,359	0.18
SanFrancisco	0.26	857,731	0.06
Toronto	0.23	1,243,370	0.06
UKansasState	0.00	69,987	0.11
UMissouri	0.17	155,305	0.17

5.2.2 *Running Time Comparison*

Although the time complexity is $O(n^3)$, the running time in the code is less than or equal to 1 second since most of the datasets have relatively low size. Especially, Atlanta, Cincinnati and Philadelphia have the lowest running time while Roanoke has the highest running time.

If we take a closer look at their sizes, Atlanta, Cincinnati and Philadelphia have less than or equal to 30 dimensions while Roanoke has 230 dimensions. The size of Roanoke is less than 10 times of other 3 cities but running time is 1.09 compared to 0.00. This indicates that the running time increases dramatically with the increase of dimensions.

5.2.3 *Relative Error Comparison*

Next, let's take a closer look at the relative error. As we can see, all datasets have their relative errors less than 20%, which is not bad. Especially, Atlanta has the best relative error, 1.90% while Roanoke has the worst relative error. Unlike running time, the relative error does not increase dramatically with the number of dimensions. They tend to stay below a certain bound, for instance, 20%. Overall, compared to other algorithms, the nearest neighbor algorithm is a relatively quick and accurate algorithm.

5.3 Local Search 1: Genetics Algorithm

5.3.1 *Overall Performance*

Table 6. Local Search 1 Comprehensive Table

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.06	2039905.0	0.02
Berlin	1.55	7969.1	0.06
Boston	2.36	954263.5	0.07
Champaign	3.48	55887.2	0.06
Cincinnati	0.08	280857.6	0.01
Denver	0.81	117115.0	0.17
NYC	2.27	1742981.4	0.12
Philadelphia	1.44	1420641.5	0.02
Roanoke	2.10	779359.5	0.19
SanFrancisco	0.76	882289.9	0.09
Toronto	0.88	1243399.4	0.06
UKansasState	0.05	63406.0	0.01
UMissouri	1.06	155818.6	0.17

The table illustrates the average result from 10 execution of our algorithm on all instances. The algorithm performs well with respect to running time, all shorter than 4 seconds. In terms of relative error, the algorithm yielded worse results in larger-scale instances like Roanoke.

5.3.2 *Box Plot*

The box plot displayed the run time distribution of our algorithm on instance Champaign and NYC. Although there are some outliers, the overall run time is controlled to a low level.

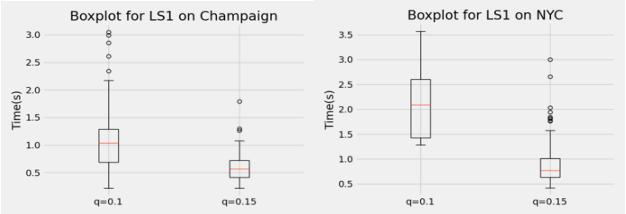

Figure 2. Box Plots for LS1

5.3.3 *QRTD Plot*

The figures displayed the probabilities movement of our algorithm reaching a certain level of quality requirement as the time grow Since we used a nearest neighbor method to initialize population, the algorithm reached a decent level of quality quickly, as shown by the lines q = 0.2 and 0.25. However, although we have set larger hyper-parameters to improve its opportunities of jumping out of local optima, the ultimate relative error is still not satisfying. There is only about 20% of our execution reached a relative error of 0.1 on instance NYC.

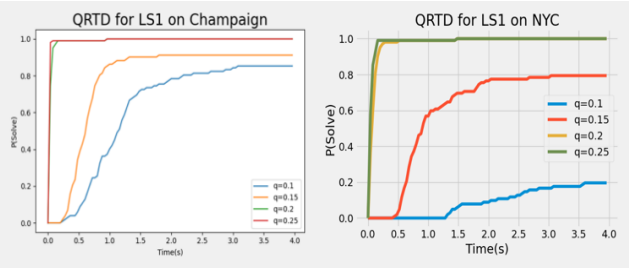

Figure 3. QRTD Plots for LS1

5.3.4 *SQD Plot*

The SQD plot displayed the probability of reaching a quality after a certain time running. For both cases, 100% case can achieve a 0.2 relative error after 0.5 seconds.

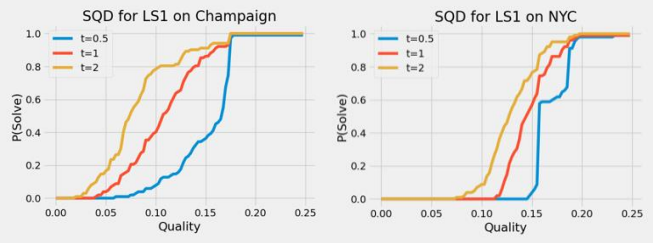

Figure 4. SQD Plots for LS1

5.4 Local Search 2: Simulated Annealing

5.4.1 *Overall Performance*

Our SA algorithm works well on most given test instances within a reasonable time. The relative error of our SA algorithm is bellow or equal to 0.05 for 69% of test cases and only one out of 12 test cases get relative error greater than 15%. The running time is less than 4 secs for 62% of test cases. Only the test instance Roanoke ran significantly longer than the others (15.71 sec). Consider the fact that size of input of test case Roanoke is much larger than others (230 vertices), it is reasonable that our SA algorithm runs much longer. The running time, absolute quality and relative error of SA on each test case is shown in the table below.

Notice that our theoretical running time is $O(n)$, in the worst case, the constant factor is large in our case (60000). So, the SA algorithm is usually slower than we expect.

In the next part, we will display the evaluation results for our SA algorithm with one good test case (NYC) and one worst test case (Roanoke).

Table 7. Local Search 2 Comprehensive Table

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.13	2040606.9	0.02
Berlin	3.76	7879.2	0.04
Boston	1.08	941784.3	0.05
Champaign	2.03	53555.1	0.02
Cincinnati	0.01	279859.5	0.01
Denver	4.26	105217.4	0.05
NYC	2.62	1648835.8	0.06
Philadelphia	0.31	1430746.8	0.02
Roanoke	15.71	758710.0	0.16
SanFrancisco	6.47	879290.9	0.09
Toronto	5.53	1217782.8	0.04
UKansasState	0.02	62962.0	0.00
UMissouri	6.55	143780.0	0.08

5.4.2 *Box Plot*

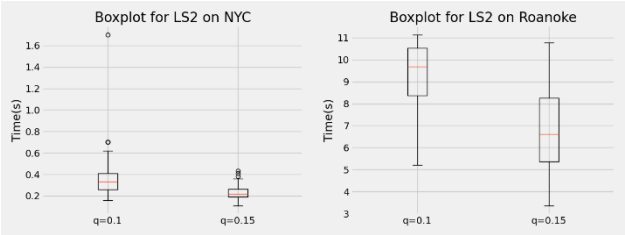

Figure 5. Box Plots for LS2

Here we visualize the box plot of SA algorithm on test case NYC and Roanoke. To achieve 0.1 relative error on NYC, our algorithm needs to run on average about 0.3 sec. To achieve 0.15 relative error, our algorithm needs to run on average about 0.2 sec. Within 0.6 sec, 75% runs achieve relative error less than 0.1. Some outliers need much more time to achieve the same level of quality (1.7 sec). In test case Roanoke, the average running time to achieve 0.1 quality is 9.7sec. The running time range from 5 to about 11 sec in order to get a quality of 0.1.

5.4.3 *QRTD Plot*

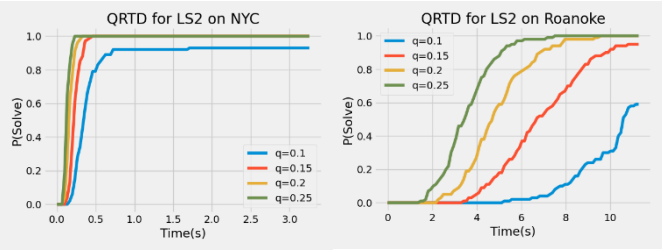

Figure 6. QRTD Plots for LS2

Here we visualize the QRTD plot for SA on test case Roanoke and NYC. On test case Roanoke, the SA algorithm can achieve less than 0.2 relative error in 10 seconds for all test cases and bout 95% chance that we can get quality better than 0.15 relative error after 9 secs. On test case NYC The SA algorithm can achieve less than 0.15 relative error in 0.5 seconds for all test cases and bout 92% chance that we can get quality better than 0.1 relative error after 7 secs.

5.4.4 *SQD Plot*

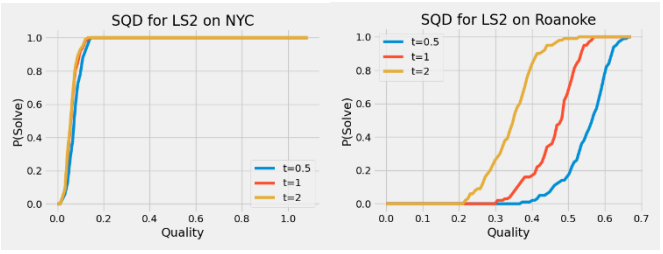

Figure 7. SQD Plots for LS2

Here we visualize the SQD plot for SA on test case NYC and Roanoke. Given an execution time of 0.5 sec, nearly all test cases achieve a relative error less than 0.14. For test case Roanoke, given an execution time of 1 sec, nearly all test cases achieve a relative error less than 0.6. 60% of the test cases achieve a relative error less than 0.5. Given an execution time of 2 sec, nearly all test cases achieve a relative error less than 0.5. 60% of the test cases achieve a relative error less than 0.35.

6. Discussion

6.1 Branch-and-Bound

For the empirical evaluation result of branch and bound, it performs better on smaller dataset due to its exponential time complexity. When it comes to larger dataset, it takes extremely long running time and has more than 10% relative error rate due to time cutoff. Therefore, it is recommended to apply branch and bound to problems with small dataset size and approximation heuristic or local search for those with large dataset size.

6.2 Nearest Neighbor Heuristics

As we mentioned above, the nearest neighbor algorithm performs extremely well and the relative errors are also acceptable. With the increase of dimensions, the empirical running time increases dramatically, which matches the worst-case complexity. For small datasets, nearest neighbor can be an efficient and quick algorithm to solve the TSP problem.

6.3 Local Search 1: Genetic Algorithm

Our genetic algorithm is basically an extension of our heuristic algorithm, as the search is based on a nearest neighbor initialization. As shown in figure below, the left part and right part compared the run time and solution quality respectively on all instances. It takes time to reach a new convergence in GA, whereas the performance is disappointing. Quality improvement only can be found in some instances, like Philadelphia and NYC. Clearly, the weak ability of jumping out of local optima caused such unsatisfying result.

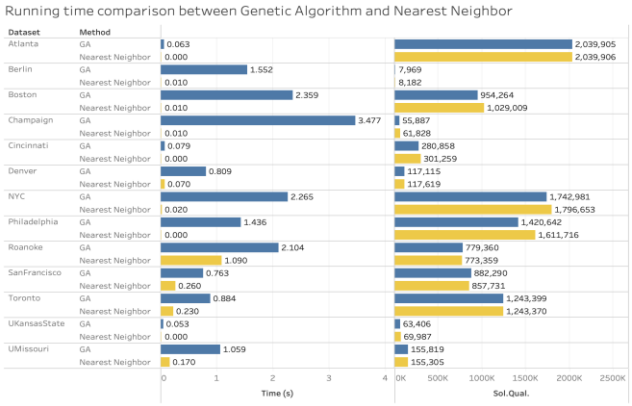

Figure 7. Comparison between Genetic and NN Heuristic

6.4 Local Search 2: Simulated Annealing

Our SA algorithm gives approximate solutions with good quality and within satisfiable time. The theoretical running time of SA is O(n), which is verified in the plot bellow. The O(n) running time suggest our SA algorithm maybe efficient dealing with TSP with a large number of points. Although the algorithm is in O(n), the constant factor is large for our case (60000), which means that our SA algorithm is not as efficient as other algorithms with worse theoretical running time in dealing with TSP with a small number of points.

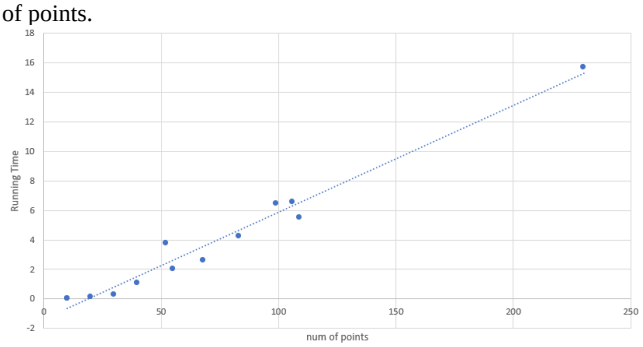

Figure 7. Time Complexity Verification for LS2

6.5 Comparison

The Branch-and-Bound algorithm is supposed to provide exact results. However, due to its exponential running time. The branch-and-bound algorithm performs worse than the other three algorithms on test instances with many points and takes an extremely long time to run. For instances with a small number of points, the algorithm can get exact solutions as expect. The Nearest Neighbor Heuristic method executes fastest among all algorithms. It also gives relatively high accuracy, especially for small datasets. However, for higher dimension datasets, the quality of solutions generated by this algorithm is not as good as our two local search algorithms. The two local search algorithms have similar performance in terms of solution quality. Simulated Annealing is a little bit better than the Genetic Algorithm in most cases. As for running time, the Genetic Algorithm outperforms the Simulated Annealing by a huge margin despite the fact that Genetic Algorithm is $O(n^3)$ and Simulated Annealing is $O(n)$.

7. Conclusion

In this report, we implement four different algorithms to solve the Travling Salesman Problem--- Branch-and-Bound, Nearest Neighbor Heuristics, Genetic Algorithm, and Simulated Annealing. Branch-and-Bound is good at finding exact solutions for instances with a small number of points. The Nearest Neighbor algorithm is easy to implement, it executes quickly and has relatively high accuracy but does not perform well on large-scale instances. Simulated Annealing and Genetic Algorithm both give approximate solutions with good quality within a satisfiable time. Given the nature of NP-complete of the TSP, it seems that finding an approximate solution is much more time efficient and realistic than finding an exact solution, especially when the scale of the instance is large. With suitable heuristics or local search approaches, greedy algorithms and local search algorithms can also get very good sub-optimal solutions. The answer to which is the best algorithm to solve TSP really depends on the application. If quality is important and running time is not a constrain or if the instance is small, Branch-and-Bound can return exact solutions. If we are strict with execution time and do not care too much about quality, a greedy algorithm with the Nearest Neighbor Heuristic may be a good choice. If we weigh both the running time equally important, the two local search algorithms are better.

As for future works. First, in this work, only 13 test instances are used to evaluate the performance of algorithms and the scales of those test instances are under 230. It still needs to show the algorithms' performance on test cases with more points. Second, based on our analysis, it is possible to build an effective ensembled algorithm that chooses the most proper algorithm to run based on the scale of problems.

8. REFERENCES

[1] Website. 2021. Travelling salesman problem - Wikipedia. 2021. Retrieved 3 December 2021, from https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[2] Mark de Berg et. al. 2018. An ETH-Tight Exact Algorithm for Euclidean TSP. eprint arXiv:1807.06933.

[3] Bellman R. 1962, Dynamic Programming Treatment of the Travelling Salesman Problem. J. Assoc. Comput. Mach., 9, 61–63.

[4] Website. 2021. World Traveling Salesman Problem. (n.d.). Retrieved 3 December 2021, from http://www.math.uwaterloo.ca/tsp/world/.

[5] Xiuwei Zhang. 2021. CSE6140/CX4140 Fall 2021 TSP Project.

[6] Website. 2021. Simulated Annealing - Wikipedia. 2021. Retrieved 3 December 2021, from https://en.wikipedia.org/wiki/Simulated_annealing#cite_note-1.

[7] Kirkpatrick S., Gelatt Jr, C. D., Vecchi, M. P. 1983. Optimization by Simulated Annealing. Science. 220 (4598), 671–680.

[8] L Ingber. 1996. Adaptive simulated annealing (ASA): Lessons learned. Control and Cybernetics. Vol. 25 No. 1, 33–54.

[9] De Vicente Juan, Lanchares Juan, Hermida, Román. 2003. Placement by thermodynamic simulated annealing. Physics Letters A, 317 (5–6), 415–423.

[10] Laarhoven, P. J. M. van (Peter J. M.) (1987). Simulated annealing : theory and applications. Aarts, E. H. L. (Emile H. L.). Dordrecht: D. Reidel. ISBN 90-277-2513-6. OCLC 15548651.

[11] Frederick Hayes-Roth. 1975. Review of Adaptation in Natural and Artificial Systems by John H. Holland. The U. of Michigan Press, 1975. ACM SIGART Bulletin 53 (1975), 15–15.

[12] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. 1999. Genetic algorithms for the travelling salesman problem: A review of representations and operators. Artificial intelligence review 13, 2 (1999), 129–170.

[13] Göktürk Üçoluk. 2002. Genetic algorithm solution of the TSP avoiding special crossover and mutation. Intelligent Automation & Soft Computing 8, 3 (2002), 265–272.

[14] Peter Merz and Bernd Freisleben. 1997. Genetic local search for the TSP: New results. In Proceedings of 1997 Ieee International Conference on Evolutionary Computation (Icec'97), IEEE, 159–164.

[15] David E Goldberg, Bradley Korb, and Kalyanmoy Deb. 1989. Messy genetic algorithms: Motivation, analysis, and first results. Complex systems 3, 5 (1989), 493–530.

[16] Vedat Toğan and Ayşe T Daloğlu. 2008. An improved genetic algorithm with initial population strategy and self-adaptive member grouping. Computers & Structures 86, 11–12 (2008), 1204–1218.

[17] Andre, J., Siarry, P., & Dognon, T. (2001). An improvement of the standard genetic algorithm fighting premature convergence in continuous optimization. Advances in Engineering Software, 32(1), 49–60.

[18] David E Goldberg, Robert Lingle, and others. 1985. Alleles, loci, and the traveling salesman problem. In Proceedings of an international conference on genetic algorithms and their applications, Carnegie-Mellon University Pittsburgh, PA, 154–159.

[19] IM Oliver, DJd Smith, and John RC Holland. 1987. Study of permutation crossover operators on the traveling salesman problem. In Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA, Hillsdale, NJ: L. Erlhaum Associates, 1987.

[20] Lawrence Davis and others. 1985. Applying adaptive algorithms to epistatic domains. In IJCAI, 162–164.

[21] Zbigniew Michalewicz, Cezary Z Janikow, and Jacek B Krawczyk. 1992. A modified genetic algorithm for optimal control problems. Computers & Mathematics with Applications 23, 12 (1992), 83–94.

[22] Michalewicz, Z., Janikow, C. Z., & Krawczyk, J. B. (1992). A modified genetic algorithm for optimal control problems. Computers & Mathematics with Applications, 23(12), 83–94.

[23] Banzhaf, W. (1990). The “molecular” traveling salesman. Biological Cybernetics, 64(1), 7–14.

[24] Syswerda, G. (1991). A study of reproduction in generational and steady-state genetic algorithms. In Foundations of genetic algorithms (Vol. 1, pp. 94–101). Elsevier.

[25] G. Gutin, A. Yeo and A. Zverovitch, Exponential Neighborhoods and Domination Analysis for the TSP, in The Traveling Salesman Problem and Its Variations, G. Gutin and A.P. Punnen (eds.), Kluwer (2002) and Springer (2007).

[26] G. Gutin, A. Yeo and A. Zverovich, Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. Discrete Applied Mathematics 117 (2002), 81–86.

[27] J. Bang-Jensen, G. Gutin and A. Yeo, When the greedy algorithm fails. Discrete Optimization 1 (2004), 121–127.

[28] G. Bendall and F. Margot, Greedy Type Resistance of Combinatorial Problems, Discrete Optimization 3 (2006), 288–298.

[29] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. 1963. An Algorithm for the Traveling Salesman Problem. Oper. Res. 11, 6 (December 1963), 972–989

[30] R. Radharamanan, L.I. Choi, A branch and bound algorithm for the travelling salesman and the transportation routing problems, Computers & Industrial Engineering, Volume 11, Issues 1–4, 1986, Pages 236-240, ISSN 0360-8352

[31] Donald L. Miller, Joseph F. Pekny, Exact Solution of Large Asymmetric Traveling Salesman Problems, Science 1991-02-15 251(4995): 754-761