CSE6140 Project
Traveling Salesman Problem

Implementation of Four Well-known Algorithms

Shahrokh Shahi
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia
shahi@gatech.edu

ABSTRACT

The Traveling Salesman Problem (TSP) is one of the most well-known NP-complete problems in computer science and the most prominent member of the rich set of combinatorial optimization problems, in general. In this project, four different algorithms have been implemented to find the optimal or nearly optimal solutions of this problem, including branch-and-bound, minimum spanning tree (MST) approximation, neighborhood 2-opt exchange, and simulated annealing approaches. This paper presents a general definition of this problem and the related works. Then, all the implemented algorithms are discussed. Finally, an empirical evaluation will be presented to compare the capability and performance of the implemented approaches.

KEYWORDS

Traveling salesperson problem, Branch and bound, Approximation, Heuristics, Local search, 2-opt exchange, Simulated annealing

1 INTRODUCTION

For roughly 70 years, the TSP has served as one of the famous challenge problems motivating various general approaches to coping with NP-hard optimization problems, and perhaps no other problem in computer science has been as extensively studied as the TSP. The purpose of this problem is to find a route through a given set of cities with shortest possible length. In other words, given a set of cities and the distance between each of them, which is also referred to as the cost, the problem is to find the best possible way of visiting all the cities and returning to the starting point while minimizing the total travel distance [10].

The study of this problem has attracted many researchers from a wide range of fields, e.g., Mathematics, Physics, Biology, or Artificial Intelligence, etc., and there is a vast amount of literature on it, accordingly. This is due to the fact that, although this problem is easily formulated, it exhibits all aspects of combinatorial optimization and has served and continues to serve as the benchmark problem for new algorithmic ideas like simulated annealing, tabu search, neural networks, and evolutionary methods [16].

In this project, different algorithms are implemented in Python to study various proposed approaches for solving this problem including (i) one exact algorithm using branch-and-bound method, (ii) one construction heuristic approach using MST-approximation algorithms, which is guaranteed to be a 2-approximation algorithm, and (iii) two local search approaches including 2-opt exchange and simulated annealing algorithms. Each method has been briefly discussed and is evaluated by solving multiple instances.

Accordingly, this paper is structured as follows. First, a formal definition of the traveling salesman problem is presented in Section 2. Then, the related works are briefly studied in Section 3. Section 4 describes the implemented algorithms. Finally, the empirical evaluation and discussion are presented in the next sections.

2 PROBLEM DEFINITION

In the general version of TSP, the input is a complete undirected graph $G = (V, E)$, with a nonnegative cost $c_e \geq 0$ for each $e \in E$. The goal is to compute the TSP tour with the minimum total cost, where a TSP tour is a simple cycle that visits each vertex exactly once.

In this project, the metric version of the TSP problem is presented, in which the $x - y$ coordinates of N points (vertices) are given in the plane and the cost function $c_e = d(u, v)$ for every pair of points defined as Euclidean distance between these two nodes u and v,

$$d(u, v) = \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2} \tag{1}$$

In this way, all edge costs are symmetric and satisfy the triangle inequality. Therefore, the following conditions are met,

(1) Non-negativity:

$$\forall u, w \in V, d(u, w) \geq 0 \tag{2}$$

(2) Symmetry:

$$\forall u, w \in V, d(u, w) = d(w, u) \tag{3}$$

(3) Triangle Inequality:

$$\forall u, w, z \in V, d(u, w) \leq d(v, z) + d(z, w) \tag{4}$$

Generally, when $d(\cdot)$ satisfies these three properties, it is considered a metric. Therefore, by this definition, any Euclidean space using Euclidean distance is a metric space.

Definition 1. *A vertex tour of a graph G is a path which visits all vertices and returns to its starting vertex, which is equivalent to a Hamiltonian cycle on G.*

Accordingly, in this project, the metric traveling salesman problem is defined as follows:

Definition 2. Given a graph $G = (V, E)$, where V is a set of points on a plane and $E = V \times V$, and $d(\cdot)$ defined as the Euclidean distance between each two pairs, the traveling salesman problem is to find a tour $\pi = \{v_1, v_2, \ldots\}$ such that the total cost (weight) of the tour

is minimized,

$$d(\pi) = \sum_{i=1}^{|\pi|-1} d(v_i, v_{i+1}) \tag{5}$$

3 RELATED WORKS

In this section, the available proposed algorithm to solve TSP problem is briefly discussed.

3.1 Exact Algorithms

In general, exact algorithms solve an optimization problem to optimality. However, for NP-complete problems, exact algorithms can work reasonably fast for only small size problems. In fact, an exact algorithm for an NP-hard optimization problem cannot run in worst-case polynomial time, unless $P = NP$.

Brute Force algorithms are the most basic and general approach exact algorithms. Such algorithms enumerate all possible solutions for the problem and then check which one is the optimal one. Thus, this approach is too computationally expensive and it is not recommended and feasible to use for real-world and large input size problems.

Branch-and-Bound algorithms is a slightly smarter version of the Brute Force approaches, in which the exact solution obtained faster by pruning the all possible solutions tree, i.e. eliminating unnecessary cases. This elimination cut down the time spent in searching the solutions tree.

One of the very first branch-and-bound algorithms to solve the TSP is proposed by Little *et. al.* [11] in which the set of feasible solutions (all tours) is divided into increasingly small subsets by a procedure known as branching. For each subset, a lower bound in the length (weight) of the tours therein is obtained and the solutions with greater length will be eliminated. Eventually, a subset is found that contains a single tour whose length is not greater than a lower bound of every tour [11].

3.2 Approximation Algorithms

Although the exact algorithms, e.g. branch-and-bound method, provides the exact solutions, they are computationally expensive and will take a long time to obtain a solution in most cases. Sometimes, we need a quick, yet good enough solution. This is where the approximation algorithms come in. Approximation algorithms run in polynomial time and always provide a solution close to the optimal with a guaranteed performance bound. An algorithm is called an α-approximation algorithm if it runs in polynomial time, and always produces a solution within at most α times as the optimal solution (in minimization problems).

In the Traveling Salesman Problem, the given graph G is a complete graph with nonnegative edge costs, and the goal is to find a minimum cost tour. The key to designing approximation algorithm is to obtain a lower bound on the optimal value which is provided by the minimum spanning tree (MST). Therefore, in this project, the MST approximation algorithm is implemented and it will be discussed in the next section.

Latest approximation algorithms can find a solution with 2-3 percent error within reasonable time [13]. Christofides' algorithm [3] is based on the original MST algorithm and presents a 1.5-approximation method which provides a solution that is at most 1.5 times worse than the known optimal. This algorithm improves the lower bound of TSP and applies the concepts of Eulerian tour to achieve such approximation [3]. Another approach is the Nearest Neighbor algorithm which is also known as greedy algorithm and has approximation factor $\Theta(\log(V))$, where V is the total number of cities [5].

3.3 Local Search Algorithms

In addition to the exact and approximation approaches, local search algorithms have been extensively employed to solve the traveling salesman problem. This class of algorithms iteratively improve the current solution by searching for a better result in a predefined neighborhood. The algorithm stops when there is no improvement in the solution in the given neighborhood or if a certain number of iterations has been reached [16].

A well-known local search algorithm is 2-opt which is first proposed by Croes [4] in 1958, in which the main idea is considering a route that crosses over itself and then re-ordering it so that it does not, which can be resulted in a cheaper route. Later, 3-opt algorithm and other variants, e.g. Lin-Kernighan are proposed which are shown can obtain a result within 2-4 percent of the optimal solution [14].

Tabu Search algorithm is another local search approach which is known as the most widely used meta-heuristic procedure that guides a local heuristic search procedure to explore the solution space beyond local optimality, in which an adaptive memory is employed to create a flexible search behavior [1]. This algorithm starts from an initial tour as the current solution and searches for the best solution in a suitably defined neighborhood. Then, it updates the current solution. This procedure will continue until certain conditions are met.

Another family of meta-heuristic algorithms employed to solve TSP problem are Evolutionary algorithms such as genetic algorithm [6, 9] and ant colony algorithm [7]. Such algorithms are inspired by biological evoulution, such as reproduction, mutation, recombination, and selection, and at each iteration, the quality of the solution is determined by a fitness function.

Another good approach to solve TSP problem is stimulated annealing (SA) algorithm [17]. This approach was first independently proposed as a search algorithm for combinatorial optimization problems [2, 8] and then widely employed as a popular iterative meta-heuristic algorithm to solve discrete and continuous optimization problems. The main idea of this approach is escaping from local optima by allowing hill-climbing moves to find global optimum [17].

4 ALGORITHMS

In this project, four different algorithms belonging to the categories presented in the previous section are implemented in Python. These four algorithms are discussed in this section.

4.1 Branch and Bound

As an exact algorithm, branch-and-bound approach is implemented in this project. In this algorithm, considering a solution tree including all possible tours, we start from the top as the current node for which, a bound (in the case of TSP, a lower bound) needs to be

Algorithm 1: `Recursive_Branching`

Input: *nNode*, *tour*, *bound*, *cost*, *step*
Result: *optimumTour*, *minCost*, the optimum tour and the corresponding minimum cost of traveling

```
1  if (step > nNode) then
2      currentCost ← cost + compute_cost(tour)
3      if (currentCost < minCost) then
4          minCost ← currentCost
5          optimumTour ← tour
6      end
7  end
8  for (i = 1 : nNode) do
9      if (node i is not visited) then
10         currentCost ←
             cost + compute_cost(tour[step-1])
11         lowerBound ← compute_bound(tour)
12         if (lowerBound + currentCost < minCost) then
13             tour[step] ← i
14             mark i as visited
15             Recursive_Branching(nNode, tour, lowerBound,
16             currentCost, step + 1))
17         end
18         currentCost ←
             cost − compute_cost(tour[step-1])
19         reset visited list
20         mark nodes i ∈ tour as visited
21     end
22 end

23 return optimumTour, minCost
```

calculated which determines a bound on the best possible solution that can be obtained if we go down this node. If the bound on the best possible solution is worse than the current best which is computed so far, then we can ignore (prune) the corresponding sub-tree which this node is its root. Therefore, the most important feature in this algorithm is finding a way to calculate the bound on the best possible solution. In general, the cost through a node includes two costs; (1) the cost of reaching the node from the root, and (2) cost of reaching an answer from current node to a leaf. When we reach to a node, we have the first cost computed, but for the second cost, we calculate a bound on this cost to decide whether prune it or not. A variety of approaches are proposed in the literature for obtaining a bound in TSP problem. For instance, MST can easily be used as a lower bound. A natural and faster way to calculate a lower bound is presented in the following [12].

For a given tour T, if we consider two edges through every node $u \in V$ ans sum their costs, the overall sum for all vertices will be twice of cost of the tour T, since we have considered every edge twice. Therefore,

$$\texttt{cost\_of\_tour} = \frac{1}{2}\sum_{u \in V}(\texttt{sum\_of\_two\_adjacent\_edges(u)})$$

It is also clear that sum of the two adjacent edges is always greater than or equal to the sum of the two minimum weight adjacent edges. Hence, the cost of any tour must be greater than or equal to the sum of the cost of two minimum weight edged which are adjacent to $u, \forall u \in V$. Now, we can easily implement our branch-and-bound approach. Algorithm (1) and (2) illustrate the pseudo-code of the implemented branch-and-bound algorithm. Algorithm (2) is the main function that calls the recursive branch-and-bound function.

Algorithm 2: `Branch-and-Bound`

Input: *cities coordinates*
Result: *optimumTour*, *minCost*, the optimum tour and the corresponding minimum cost of traveling

```
1  nodes ← extract_nodes(cities)
2  nNode ← number_of_nodes
   /* initialize all variables to call recursive
      branching function                          */
3  initTour ← [−1, . . . , −1]
4  initTour[0] ← 0
5  initBound ← compute_bound(nodes)
6  mark node 0 as visited
7  optimumTour, minCost ←
     Recursive_Branching(nNode, tour, lowerBound,
8    currentCost, step + 1))

9  return optimumTour, minCost
```

For the time complexity, in theory, the worst case complexity of branch and bound algorithm remains same as the brute force approach with enumerating and checking all possible routs (which is checking $O(n2^n)$ sub-problems in linear time that is equal to $O(n^2 2^n)$) because in the worst case, the algorithm may never get a chance to prune any node in the search space. However, in practice, this algorithm performs much better than brute force algorithm. In this way, choosing a proper bounding method is very important in this algorithm.

4.2 MST-Approximation

As mentioned before, the MST algorithm can be employed to find an approximation for the TSP problem with approximation guarantees, by constructing a tree walk based on the MST of the complete graph build upon the given nodes. To find the MST, a greedy algorithm like Kruskal's or Prim's algorithm can be used. Then, we run a depth-first-search (DFS) on the MST which meets every edge exactly twice and has the cost twice as the cost of MST and therefore, is less than or equal to two times optimum cost since the MST gives a lower bound of TSP. In other words,

$$cost = 2 \times MST \leq 2 \times OPT$$

Now, we can write the list of nodes in the approximated tour by writing the first time of appearance of each node in the DFS walk. A simple pseudo-code of the implemented MST-approximation algorithm is presented in Algorithm (3).

Algorithm 3: `MST-Approx`

Input: *cities coordinates*
Result: *optimumTour*, *minCost*, the optimum tour and the corresponding minimum cost of traveling

```
edges ← extract_edges(cities)
/* sort edges with respect to distances        */
edges ← sort(edges)

MST ← Kruskal(edges)
MST_deepcopy ← MST

tour ← ϕ
while (MST_deepcopy ≠ ϕ) do
    tour ← tour ∪ MST[0]
    nextEdge ← find_next_edge()
    if nextEdge ≠ ϕ then
        tour ← tour ∪ MST[0]
    else
        backtrack()
    end
end
minCost ← compute_cost(tour)
optimumTour ← tour

return optimumTour, minCost
```

In general, the complexity of the MST algorithm is $O(n^2)$ and the complexity of our DFS implementation is $O(n+m)$ where $n = |V|$ and $m = |E|$. Therefore, the overall complexity is dominated by $O(n^2)$. Similar to the previous algorithm, the space complexity is $O(n^2)$ which is required to store the adjacency (distance) matrix.

Algorithm 4: `UpdateTour`

Input: *tour*, *i*, *j*
Result: *tourUpdated*, the updated tour

```
tourUpdated ← ϕ
tourUpdated ← tourUpdated ∪ tour[0, i − 1]
tourUpdated ← tourUpdated ∪ reverse(tour[i, j])
tourUpdated ← tourUpdated ∪ tour[j + 1, end]

return tourUpdated
```

4.3 Neighborhood 2-opt Exchange

The 2-opt algorithm is most probably the simplest local search algorithm for solving TSP problem. The main idea, as mentioned earlier, is considering a tour that crosses over itself and then re-order it such that it does not. To implement this algorithm, at first we consider an initial tour which can be obtained from other algorithms, such as greedy algorithm. We call this tour as the current tour. Then, we choose two breaking points i and j inside the tour with which we can create a new tour by combining the following components:

- (1) Path from the beginning, node 1, to node $(i-1)$ with the same order
- (2) Path from i to j in reverse order
- (3) Path from $(j+1)$ to the end

Afterwards, we calculate the cost of this new tour and if it is better (less) than the current tour, we update the current tour. We continue this procedure any new better tour cannot be found. This procedure is demonstrated in Algorithm (4). Accordingly, Algorithm (5) illustrates the main step of the 2-opt local search algorithm. The time complexity of this algorithm is $O(n^2)$ because each node can be exchanged with at most $n-1$ other nodes and there are n nodes in total. The space complexity is also remains $O(n^2)$.

Algorithm 5: `2-Opt-Exchange`

Input: *cities coordinates*
Result: *optimumTour*, *minCost*, the optimum tour and the corresponding minimum cost of traveling

```
while (still can swap) do
    minCost ← compute_cost(tour)
    i ← 0
    j ← i + 1

    while (i < number_of_nodes_can_swap) do
        while (j < number_of_nodes_can_swap) do
            newTour ← UpdateTour(tour, i, j)
             newCost ← compute_cost(newTour) if
            (newCost < minCost) then
                minCost ← newCost
                optimumTour ← tour
            end
        end
    end
end

return optimumTour, minCost
```

4.4 Simulated Annealing

Simulated Annealing (SA) algorithm is the second local search algorithm which is implemented in this project. This algorithm is a stochastic local search approach inspired by the physical process of annealing in metallurgy, which is a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce the defects. In the inspired SA algorithm, the main idea is escaping from local optimums by allowing hill-climbing moves to find the global optimum. In the metal annealing process, when the temperature is high, atoms move around easily and when it is cooling down, atoms move slower and start to find a configuration with lower internal energy. Accordingly, the higher and lower temperature in the metal is associated with more greater and smaller probability of accepting worse neighboring solutions. In this way, we can use this concept to implement a hill-climbing mechanism.

The accepting criterion of a neighboring solution over current solution can be obtained by the Metropolis condition,

$$Pr(s', s) = \begin{cases} 1, & \text{if } f(s') > f(s). \\ \exp\left\{\frac{f(s')-f(s)}{T}\right\}, & \text{otherwise} \end{cases}$$

where s' and s are new and old solutions, respectively, and T is the temperature. This condition implies that the neighboring solution is accepted if it has a greater evaluated value. However, if the new solution is worse, we use a probability which is related to the temperature T for deciding to accept the new solution. This function is exponential and as the temperature is decreasing, the probability of accepting a worse new solution is also decreasing. Therefore, it is very important to determine and tune the effective parameters, such as initial temperature, and decide how to decrease the temperature and determine stopping criteria. The general steps of the SA algorithm which is discussed in class is illustrated in Algorithm (6).

Algorithm 6: `Simulated Annealing`

Input: *problem*, *schedule*
Result: *solution*, the optimum solution of the problem

```
currentSol ← Make_Node(Initial-State[problem])
while (The stopping criteria not met) do
    // updating temperature
    T ← schedule[t]
    if (T = 0) then
        return greedy(currentSol)
    end
    nextSol ← Random_Successor(currentSol)
    ΔE ← f(nextSol) − f(currentSol)
    if (ΔE > 0) then
        currentSol ← nextSol
    else
        currentSol ← nextSol with probability exp(ΔE/T)
    end
end
return currentSol
```

To implement this algorithm, we start from a random initial solution s_0 and set the step counter$i = 0$ and the initial temperature T_0. We also need to calculate the initial evaluation $f(s_0)$ which is the length of the initial solution. Now, we start the iterative procedure by setting $T = T_i$ and forming the neighboring solution s' and computing its length $f(s')$. Then, we can check the Metropolis criterion and update the current solution, if it the criterion is satisfied ($\Delta E > 0$ or the probability condition $\exp(\frac{\Delta E}{T})$ is satisfied). Afterwards, we proceed the iterations ($i \leftarrow i + 1$ and updating T) until meeting the stopping criteria.

The time complexity of the implemented algorithm can be considered as $O(n)$ because the while-loop is executed for specific number of times which is typically less than the size of the problem and the only calculation is evaluating the solution. Therefore, we need $O(n)$ operations. The space complexity of the implemented algorithm is still $O(n^2)$ due to the adjacency (distance) matrix. The required parameters have been tuned in a trial and error procedure.

5 EMPIRICAL EVALUATION

In this section, the outputs produced by the implemented algorithms applied to a set of real world problems are studied. To have a measure of the performance of the developed program, at first, the specification of the platform is presented. Then, the numerical results and evaluation plots are discussed.

5.1 Platform Specification

All programs are written in Python (version 3.7.3) and executed on a MacBook Pro Laptop with the following specifications:

- OS: macOS Catalina (Version 10.15)
- Processor: 1.4 GHz Quad-Core Intel Core i5
- Memory: 8 GB 2133 MHz

5.2 File Structure

The dataset includes 13 different cities in the worlds with various number of nodes. The nodes are specified by their coordinates. The program is written in a modular fashion in which each algorithm implemented as a separated class. The associated python files are as follows,

- `exhustive.py` This file includes the *Exhustive* class to run an exhustive search for solving TSP problem, which was not a requirement in this project. I developed this class only for testing and creating benchmarks for small size instances.
- `branch_and_bound.py` This file includes the *BranchAndBound* class to create and run the explained branch-and-bound algorithms.
- `heuristics.py` This MST-Approximation algorithm is implemented in the *MinSpanningTreeApprox* class in this file.
- `local_search.py` Both discussed local search algorithms (2-opt and simulated annealing) are implemented in this file in two separate classes, *TwoOpt* and *SimulatedAnnealing*, respectively.

Furthermore, two other python files are developed separately which are includes helper functions required by the implemented algorithms:

- `tsp_utils.py` This file includes all the helper functions required by various implemented algorithms including the function to calculate the adjacency (distance) matrix, etc.
- `visual_utils.py` This file includes the helper functions required for the visualization, which was not a requirement in this project, but I added to improve my own understanding of the data files. For instance, Figure (1) illustrates an initial visualization after interpreting an input file.

The third type of developed files are the main executive files:

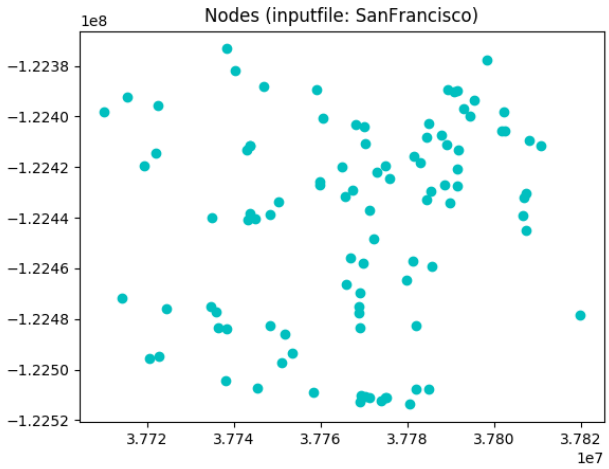

Figure 1: Initial visualization of the nodes after interpreting the coordinates of an input file (San Francisco)

- `tsp_main.py` This file is the main executive file that input parameters as command arguments in terminal and after calling the proper algorithm function and obtaining the solution, generates the two required types of output files (`*.sol` and `*.trace`)

- `driver.py` Since there exist many input files and also there are various (i) output files, (ii) numerical tables, and (iii) evaluation plots which are required to generates for this report, this executive file has been developed to facilitate this procedure. In fact, this function automatically execute `tsp_main.py` with the proper argument and for all the input files. then it gathers required information from multiple runs and generates the required tables (and stores them in `*.tex` files for LaTeX) and the evaluation plots (and saves them in `*.png` files). A complete execution of `driver.py` on a platform with mentioned specification (Section 5.1) takes less than 2 hours with `cut_off_time` equal to 600 seconds.

5.3 Comprehensive Tables

The numerical results of the developed program are presented in this section. For each implemented algorithm, a separated table is generated (automatically by `driver.py`) in which the first column indicates the name of the problem instances (cities), the second column is the elapsed run time that is less than the determined cutoff time, the third column demonstrates the solution quality which is the minimum total weight of the solution (tour) which has been found within the given cutoff time, and the fourth column demonstrates the relative error which calculated with the following formula,

$$RelErr = \frac{ALG - OPT}{OPT}$$

where ALG is the solution quality obtained by the corresponding algorithm and OPT is the optimum solution quality which is the weight of the optimum tour for the given problem instance. The reported results of the local search algorithms are obtained by taking the average of the elapsed running time and solution quality over **15** independent runs (which have been done automatically by `driver.py`).

- *Branch-and-bound*
  The following table illustrates the numerical results of the branch-and-bound algorithms with 10 minutes cutoff time (600 S). It is obvious that most of the time the algorithm is terminated in near cutoff time which shows that most probably the program could not find the exact optimum solution within the given cutoff time. The relative error values prove this hypothesis.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	504.20	2575079	0.2851
Berlin	406.55	19233	1.5501
Boston	329.66	2203741	1.4663
Champaign	567.65	215760	3.0986
Cincinnati	0.27	277952	0.0000
Denver	398.75	540412	4.3809
NYC	217.97	7163135	3.6063
Philadelphia	109.82	3645073	25.1112
Roanoke	212.07	6849948	103.5076
SanFrancisco	230.36	5581318	5.8888
Toronto	333.10	9116969	6.7515
UKansasState	0.20	62962	0.0000
UMissouri	441.62	658567	3.9625

- *MST-Approximation*
  The following table demonstrates the numerical results obtained by the MST-Approximation algorithm. We can see that the execution of this algorithm is much faster than the branch-and-bound algorithm. As mentioned earlier, the performance bound of the approximation algorithm is guaranteed and in the MST-Approximation is a 2-approximation algorithm, where the numerical values of the relative errors demonstrate this fact.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.00	3278557	0.6362
Berlin	0.01	8468	0.1228
Boston	0.00	1701580	0.9043
Champaign	0.01	83154	0.5796
Cincinnati	0.00	307439	0.1061
Denver	0.02	199163	0.9831
NYC	0.01	2416459	0.5539
Philadelphia	0.00	252938	0.8119
Roanoke	0.18	126941	0.9367
SanFrancisco	0.02	1012907	0.2502
Toronto	0.03	2209635	0.8787
UKansasState	0.00	109994	0.7470
UMissouri	0.02	159463	0.2016

- *Local Search: 2-opt*
  The numerical outputs of the 2-opt local search approach is presented in the following table. This algorithm is the fastest

one among implemented approaches; however, as expected, the performance bound is not guaranteed and there are a few cases in which the relative error is a large number.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.00	2147827	0.0719
Berlin	0.00	9206	0.2206
Boston	0.00	1150295	0.2874
Champaign	0.00	78400	0.4893
Cincinnati	0.00	350839	0.2622
Denver	0.01	141909	0.4130
NYC	0.01	2030897	0.3060
Philadelphia	0.00	2211362	14.8409
Roanoke	0.14	902765	12.7732
SanFrancisco	0.02	1143100	0.4109
Toronto	0.02	1706840	0.4512
UKansasState	0.00	83218	0.3217
UMissouri	0.03	191385	0.4421

- *Local Search: Simulated Annealing*
  The following table illustrates the numerical results obtained by running the SA algorithm. Similar to the other implemented local search algorithm, the performance bound is not guaranteed but we can see, at least for the given set of instances, it has a very impressive performance.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.02	2045745	0.0210
Berlin	0.95	8361	0.1086
Boston	0.19	963238	0.0780
Champaign	0.52	54461	0.0345
Cincinnati	0.00	277952	0.0000
Denver	1.32	103761	0.0332
NYC	1.53	1594760	0.0255
Philadelphia	0.07	1400046	9.0291
Roanoke	10.98	733362	10.1887
SanFrancisco	3.80	866110	0.0690
Toronto	2.22	1193277	0.0146
UKansasState	0.01	62962	0.0000
UMissouri	4.64	142932	0.0770

5.4 Evaluation Plots

In this section, a set of evaluation plots are presented to study the implemented local search algorithms. To obtain the meaningful results to generate this plots **100** independent runs (with different seed numbers) has been conducted by `driver.py` and the associated trace results are gathered and stored in some binary files. To generate this results, two instances with more than 50 vertices are chosen; `Berlin.tsp` and `NYC.tsp` where for each of them three types of graphs are presented:

- Qualified Runtime for various solution qualities (QRTDs) in which the x-axis is the run time in seconds, and the y-axis is the fraction of the algorithm runs (in this case, out of 100 independent runs) that solved the problem with respect to various relative solution quality q^*.

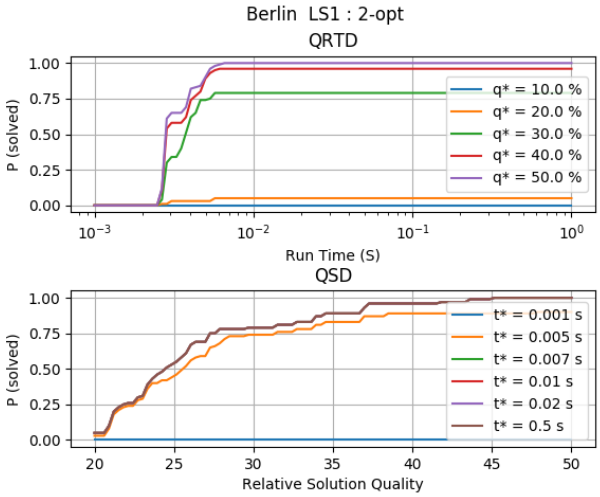

Figure 2: QRTD and SQD plots of 2-opt local search algorithm obtained by 100 independent runs to solve Berlin instance

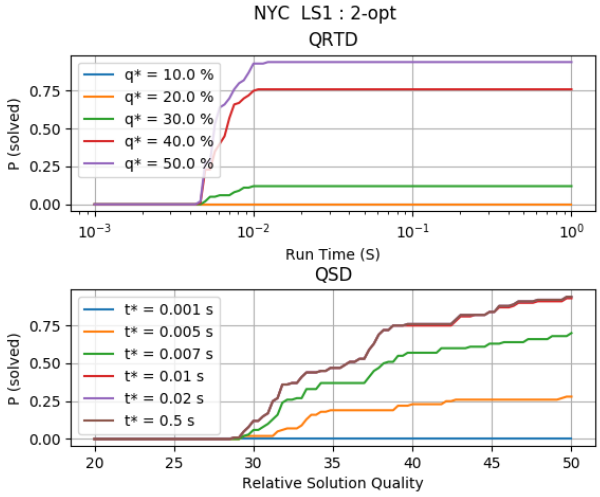

Figure 3: QRTD and SQD plots of 2-opt local search algorithm obtained by 100 independent runs to solve NYC instance

- Solution Quality Distributions for various run-times (SQDs) in which the x-axis the relative solution quality (q), and the y-axis is the fraction of the algorithm runs that solved the problem within various given running time (in seconds)

- Box plots which demonstrate the distribution of the running times of the local search algorithms.

With this explanations, Figure (2) and (3) illustrate the QRTD and SQD plots of 2-opt local search algorithm to solve Berlin and NYC instances, respectively. We can see that both plots have the similar behavior as described in class.

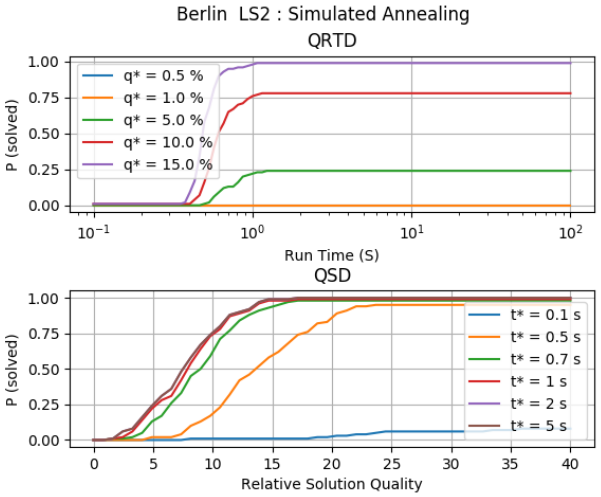

Figure 4: QRTD and SQD plots of SA local search algorithm obtained by 100 independent runs to solve Berlin instance

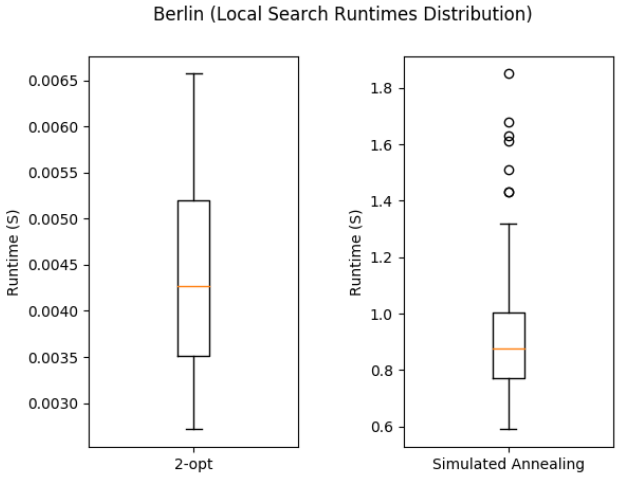

Figure 6: The distribution of the running times of the local search algorithms in 100 independent runs to solve Berlin instance

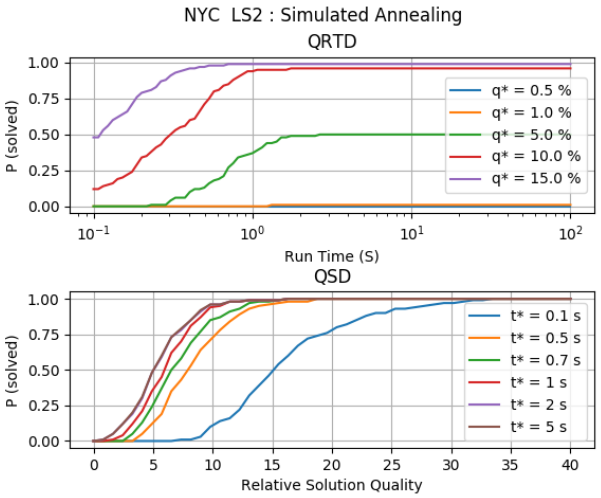

Figure 5: QRTD and SQD plots of SA local search algorithm obtained by 100 independent runs to solve NYC instance

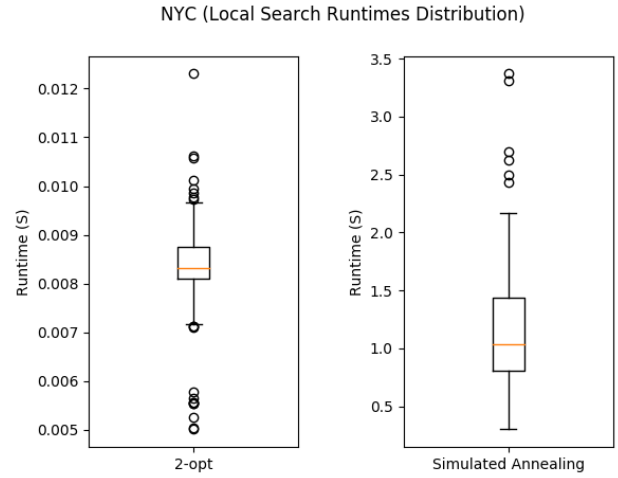

Figure 7: The distribution of the running times of the local search algorithms in 100 independent runs to solve NYC instance

Figure (4) and (5) demonstrates the QRTD and SQD plots of simulated annealing local search algorithm to solve the same instances. For each local search algorithm, the plots shows almost the same behavior for both instances. Comparing the plots for these two local search approaches reveals that even though the running time of the 2-opt algorithm is much less than the SA algorithm, the solution quality obtained by SA algorithm in much better than the solution quality of the 2-opt algorithm. The running times of these algorithms are compared in box plots presented in Figure (6) and (7). In both of the solved instances, the running time of the 2-opt algorithm is much less than that of the SA algorithm.

6 DISCUSSION

Most of the results and findings are discussed in the previous section. The numerical results presented in the tables (Section 5.3) reveals that although the branch-and-bound approach is an exact algorithm, most of the time it cannot obtain the optimum solution within limited running time. This shows that even the branch-and-bound algorithm is very computationally expensive and works well only for the very small size problems. According to the results, it is obvious that the implemented lower bound is not good enough and it is better to implement more rigorous lower bound which in turn leads to better pruning and reducing the computational

time. Essentially, the branch-and-bound algorithm can obtain the exact optimal solution given enough time. However, as mentioned in Section 4, the worst case time complexity remains the same as that of the Brute Force approach which can take an extremely long time to obtain the exact solution.

The approximation algorithms are a better alternative for solving TSP problem; in particular, when a nearly optimum solution is also acceptable. This algorithms provide a bound on the quality of the obtained solutions. In this project the MST-Approximation approach is implemented which is a 2-approximation algorithm in $O(n^2)$ time. That implies that in the worst case, we obtain a solution which its quality is $2 \times OPT$, where OPT is the optimal quality. We can see that the numerical values of the relative errors in the corresponding table of the MST-Approximation are less than 1 which proves this fact,

$$OPT \leq ALG \leq 2OPT \implies 0 \leq \frac{ALG - OPT}{OPT} \leq \frac{2OPT - OPT}{OPT} = 1$$

In comparison to the approximation algorithms, the local search algorithms are faster (as expected due to their theoretical time complexity), but unfortunately there is no guaranteed bound for their solution quality. In other words, we cannot be sure about the solution quality and cannot know if it obtains a local minimum. This can be seen in the numerical results that are presented for local search algorithms. By comparing the tables and also the box plots, it is obvious that the 2-opt algorithm is the fastest algorithm among implemented algorithms, but the solution quality of the simulated annealing algorithm is better than 2-opt approach.

7 CONCLUSION

In this project, four different approaches for solving the TSP problem have been discussed, implemented and their performance have been studied, and the merits and trade-offs of each of them are discussed. To summarize, the exact algorithms are very computationally expensive and can be employed only to solve small size problems. In comparison to the exact algorithms such as branch-and-bound, the approximation algorithms with guaranteed approximation bound are very fast and a good alternative to solve the TSP problem when a nearly optimum solution is enough. An advantage of this type of algorithms is that we can be sure that the performance approximation is within a specific bound. The third type of algorithms studied in this project is local search algorithms which are impressively fast but there is no guaranteed bound on their solution quality and there is no guarantee to obtain the global optimum. It is also observed that using a hill-climbing technique, as it used in simulated annealing algorithm, can effectively improve the solution quality of local search algorithms.

REFERENCES

[1] Basu, Sumanta, and Diptesh Ghosh. 2008. A review of the tabu search literature on traveling salesman problems.
[2] Černý, Vladimír. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm." Journal of optimization theory and applications 45, no. 1, 41-51.
[3] Christofides, Nicos. 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. No. RR-388. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.
[4] Croes, Georges A. 1958. A method for solving traveling-salesman problems. Operations research 6.6, 791-812.
[5] Johnson, David S., Gregory Gutin, Lyle A. McGeoch, Anders Yeo, Weixiong Zhang, and Alexei Zverovitch. 2007. Experimental analysis of heuristics for the ATSP. In The traveling salesman problem and its variations, pp. 445-487. Springer, Boston, MA.
[6] Freisleben, Bernd, and Peter Merz. 1996. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. In Proceedings of IEEE international conference on evolutionary computation, pp. 616-621.
[7] Hlaing, Z. C. S. S., and May Aye Khine. 2011. An ant colony optimization algorithm for solving traveling salesman problem. In International Conference on Information Communication and Management, vol. 16, pp. 54-59.
[8] Kirkpatrick, Scott, C. Daniel Gelatt, and Mario P. Vecchi. 1983. Optimization by simulated annealing. science 220, no. 4598, 671-680.
[9] Larranaga, Pedro, 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, no. 2, 129-170.
[10] Lin, Shen, and Brian W. Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. Operations research 21, no. 2, 498-516.
[11] Little, John DC, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. 1963. An algorithm for the traveling salesman problem. Operations research 11, no. 6, 972-989.
[12] Narahari, Y. 2000. Data structures and algorithms. Retrieved November 15, 2019 from http://lcm.csa.iisc.ernet.in/dsa/dsa.html
[13] Rego, César, Dorabela Gamboa, Fred Glover, and Colin Osterman. 2011. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. European Journal of Operational Research 211, no. 3, 427-441.
[14] Johnson, David S., and Lyle A. McGeoch. 1997. The traveling salesman problem: A case study in local optimization. Local search in combinatorial optimization 1.1, 215-310.
[15] Reinelt, Gerhard. 1991. TSPLIB—A traveling salesman problem library. ORSA journal on computing 3, no. 4, 376-384.
[16] Reinelt, Gerhard. 1994. The traveling salesman: computational solutions for TSP applications. Springer-Verlag.
[17] Skiścim, Christopher C., and Bruce L. Golden. 1983. Optimization by simulated annealing: A preliminary computational study for the tsp. In Proceedings of the 15th conference on Winter Simulation-Volume 2, pp. 523-535. IEEE Press.