Algorithms for Metric Solving Traveling Salesman Problem

Chengrui Li
cnlichengrui@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

Rui Feng
rfeng68@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

Haodan Tan
htan74@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

ABSTRACT

The traveling salesman problem (TSP) is one of the best-known NP-hard problems, meaning there is no guarantee to obtain the exact optimal solution in polynomial time. This paper evaluates the performance of four algorithms solving TSP from different aspects: a Branch and Bound (BnB) method trying to find exact solution, an approximate heuristic algorithm (Approx) based on Minimal Spanning Tree (MST), and two local search algorithms (LS1 and LS2) based on neighborhood 2-opt and 3-opt exchange respectively. The performance of the four algorithms will be evaluated in terms of running time, solution quality, and empirical analysis. From a comprehensive evaluation and analysis of the obtained results, we find that: The local search algorithms provided very promising results for nearly all data sets if enough time is given, while the Branch and Bound algorithm still shows worse accuracy even under the constraint of 600 seconds. The approximate algorithm always gives a fast and stable solution even when feeding very large input instance. Thus, the choice of the best approach is highly depending on the application context.

KEYWORDS

Traveling Salesman Problem (TSP), Approximate Solution, Branch and Bound, Heuristic, Local Search, NP-hard

ACM Reference Format:
Chengrui Li, Rui Feng, and Haodan Tan. 2021. Algorithms for Metric Solving Traveling Salesman Problem. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The traveling salesman problem (TSP) is a widely studied problem due to its extensive usage in numerous fields, including computational sciences, theoretical computer science, and operational research. The problem seeks the shortest route between a set of locations (nodes) that must be visited only once to construct a cycle, thereby forming a TSP solution. In this paper, we analyzed four strategies for solving the optimized version of TSP (finding the smallest cycle), including the Branch and Bound technique trying to find the exact solution, construction Heuristics with approximation guarantees, and two local search algorithms based on neighborhood 2-opt exchange and 3-opt exchange. The performance of each algorithm is tested respectively using real-world data sets, and a comprehensive comparison and analysis is conducted based on the obtained empirical results.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 PROBLEM DEFINITION

Consider an undirected graph $G = (V, E)$, where all vertices V are connected by edges E, and the weight of an edge indicates the distance between the two cities. The TSP is defined to finding the shortest Hamiltonian cycle (a path starting and ending at the same vertex that visiting each vertex exactly once). In particular, the case we study in this paper is only for metric TSP, which means the graph complies the real world situation (all edge costs are symmetric, i.e., $c(u, v) = c(v, j)$; and fulfill the traingle inequality, i.e. $c(u, v) \leqslant c(u, w) + c(w, v),\ \forall u, v, w \in V$). TSP is a well-known NP-complete problem that was included in the list of Karp's 21 NP-complete problems [1, 2].

3 RELATED WORK

Traveling Salesman Problem is one of the first proposed and still actively studied NP-hard problems in theoretical and applied computer science. It has received much attention from researchers for its implication in one of the most important conjectures in theoretical computer science, *i.e.* whether $P = NP$ or not, and its connection to a wide range of applications.

The origin of the problem is unclear. In 1930 [3] Merrill M. Flood considered the mathematical solutions to this problem. It was first proved to be in HP-complete class in 1972 by Karp [4]. The problem can be categorized into symmetric TSP, where distances between points are invariant of directions, and asymmetric TSP, where distances between points can differ by directions. After the problem was formulated, a lot of attempts were made for precise solutions [5] such as branch and bound [6] and approximation methods based on heuristics for locally optimal solutions [5]. The exact methods can find solutions for small data, e.g. less than 15 cities [5]. However, after proven to be in HP-complete class [4] in 1972, the attempts to solve TSP in large-scale datasets rely almost solely on approximation methods based on different heuristics [7–10] that can find a solution with relative error of 1% to 2% of data with potentially minions of points in a reasonable time [11]. It is believed that the Lin-Kernighan heuristic and the improved version [12] has been the most effective approximation heuristics for TSP [12]. Briefly, the Lin-Kernighan-Helsgaun heuristic solves symmetric TSP by swapping pairs of sub-paths to make a new tour with better overall distance, a generalized version of 2-opt and 3-opt experimented in this paper.

4 ALGORITHMS

4.1 Branch and Bound

4.1.1 Description. We experiment with two variants of branch of bound algorithms for TSP. Specifically, we use the same skeleton of the algorithm, i.e. constructing a search tree and expanding vertices from unvisited neighbors of the current vertex and prune the choices where the lower bound exceeds the current best solution. The difference in the two variants lies in the computation of lower bound.

Apparently, a tighter lower bound implies a generally better and more efficient solution. To empirically evaluate the quality of lower bounds, we calculate a "reject rate", as shown in Table 5, which is the percentage of nodes whose lower bound exceeds the best solution found at the time. A higher reject rate implies more nodes are filtered and generally a faster run time.

We used two lower bounds: `smallest-edge` and `reduce-matrix`. `smallest-edge` is established on the intuition that the sum of the weight of two edges is larger than two times the smaller weight. `reduce-matrix` takes the adjacency matrix and reduce each column and row to contain at least one zero, and use the reduced value as the lower bound of expanding a node.

We provide the branch and bound algorithm in Algorithm 1, `smallest-edge` in Algorithm 2 and `reduce-matrix` in Algorithm 3. Code of each module is made available in the package.

Algorithm 1 Branch and Bound Algorithm

BestSolution←∞
BestRoute←[]
function TSP($G(V, E)$)
    l_0 ← Initial estimate of lower bound on G starting from any node
    TSPBranch($G(V, E), v_0$)
end function
function TSPBranch($G(V, E), v_0$)
    Add v_0 to Path
    v_0 is the current node
    for v_i in unvisited nodes **do**
        if Bound($G(V, E), v_0, v_i$) ≥ *BestSolution* **then**
            Continue
        end if
        TSPBranch($G(V, E), v_i$)
    end for
    if All nodes have been visited **then**
        Calculate distance of current path as solution
        if CurrentSolution > BestSolution **then**
            BestSolution ← CurrentSolution
            BestRoute ← CurrentRoute
        end if
    end if
end function

Algorithm 2 Smallest Edge

function InitialEstimate($G(V, E)$)
    Return $\frac{1}{2}\sum_{v \in V}$ sum of two smallest edges connected to$v$
end function
function Bound($G(V, E), u, v$)
    Current path is $[u_1, \ldots, u_k]$
    if Path length is greater than or equal to 2 **then**
        Calculate lower bound by
        Old Lower Bound − (second minimum edge of u + minimum edge of v)/2 + distance of $u \to v$
        Return Lower Bound
    else
        Calculate lower bound by
        Old Lower Bound − (minimum edge of u + minimum edge of v)/2 + distance of $u \to v$
        Return Lower Bound
    end if
end function

Algorithm 3 Reduce Matrix

function InitialEstimate(G)
    Return ReduceMatrix(G)
end function
function ReduceMatrix(G, u, v)
    G is the graph, u, v are source and destination node
    u, v are optional
    Take (previously reduced, if any) adjacency matrix as A
    if u, v are given **then**
        Set $A_{u,:}$ and $A_{:,v}$ as ∞
        Set $A_{v,u}$ as ∞
    end if
    Cost←0
    for each row r in A **do**
        $m \leftarrow \min r$, if infinity, treat as zero
        if m>0 **then**
            Cost ← Cost + m
            $r \leftarrow r - m$
        end if
    end for
    for each column c in A **do**
        Repeat for c what was done for each row r
    end for
    A is the now reduced adjacency matrix
    Return Cost as lower bound at current node
end function

4.2 Approximation

4.2.1 Description. For the approximation algorithm with a upper bound guarantee, we choose to implement the Minimum Spanning Tree (MST) approximation. When given an metric instance (i.e., all edge cost are symmetric and fulfill the triangular inequality), MST algorithm build an MST first. Then, we start from a node as the root of the MST, and follow a depth first search to traverse all nodes, and record the traversing order. Finally, backing to the root node (where we start the traversing) forms a solution with

approximation ratio of 2 as an upper bound. Generally speaking, the MST for a real world instance is unique with probability of 1. So, the only uncertainty of the obtained solution comes from the choice of the root of the MST. Therefore, we can improve the quality of the MST algorithm, by trying all nodes as the MST root, and use the minimum to be the final result. The pseudo-code is shown in Algorithm 4

Algorithm 4 MST Algorithm

```
function MSTApprox(G(V, E))
    mst ← buildMST(G)
    curr_distance ← ∞
    for i = 1, 2, ··· , |V| do
        traverse_list ← depthFirstSearch(mst, i)  ▷ start from node i as the root of the MST to traverse it
        curr_distance ← cycleValue(traverse_list)
        if curr_distance < best_distance then best_distance ← curr_distance best_route ← traverse_list
        end if
    end for
    return best_route, best_distance
end function
```

4.2.2 Time Complexity. Building an MST from a complete graph by prim algorithm is at most in $O(|V|)$, and the depth first search for a tree is at most $O(|V|)$. Taking that trying all nodes as the root to traverse the MST into account, the overall complexity of this algorithm is in $O(|V|^3)$. Since this is a very simple and intuitive algorithm, even though we traverse all of the MST node, the actual running will be very fast compared with other algorithms, because it is very simple and quick to build MST and traverse the MST.

4.3 Local Search - Neighborhood 2-opt Exchange

4.3.1 Description. The first local search(LS) algorithm we applied is 2-opt exchange algorithm. The 2-opt exchange algorithm should start with a solution candidate and explore every of its neighbors. The main idea of obtaining the neighbors is to take a route that crosses over itself and reorder it to eliminate the cross. The 2-opt exchange strategy in the TSP will literately compare every possible valid combination of the swapping route: if the neighboring solution yield a better value than the current solution, the current solution will be updated by the neighboring solution ("the travel distance become shorter than before"). Here we will randomly generate the candidate solution which includes all locations in the city graph, and we define the objective time seed to record the performance in such an exact route. For each step, the 1-neighboring solution are defined to deviated from the current route reversing the middle range of the route but keep the front and back of the route to be the same. Note that a neighboring solution must remain qualified as a valid route. Because reversing the order may case an invalid path, we must remove the invalid route from the search as an eligible candidate for swapping. More details of the 2-opt exchange algorithm are showed in the pseudo-code Algorithm 5.

Algorithm 5 2-opt Algorithm

```
function two_opt(route)
    best_route ← route
    best_distance ← cycleValue(route)
    improved ← TRUE
    while improved do:
        improved ← FALSE
        for i = 1, 2, ··· , length(route) − 2 do
            for j = i + 1, i + 2, ··· , length(route) do
                new_route ← two_opt_swap(route, i, j)
                new_distance ← cycleValue(new_route)
                if new_distance < best_distance then
                    output the updating route
                    best_route ← new_route
                    best_distance ← new_distance
                    improved ← TRUE
                end if
            end for
        end for
        route ← best_route
    end while
    return best_distance
end function

function two_opt_swap(route, i, k)
    1. take route[1] to route[i − 1] and add them in order to the new route
    2. take route[i] to route[k] and add them reversely to the new route
    3. take route[k] to the end and add them in order to the new route
    return route
end function
```

4.3.2 Time Complexity. It's found that the 2-opt exchange algorithm applied here is quite fast and it can show a very promising result. Since 2-opt algorithm consider every possible 2-edge swap, swapping 2 edges when it results in an improved tour, we noticed that it will construct two for loops to trace all valid routes. Therefore, the worst time complexity of 2-opt algorithm will be $O(|V|^3)$ time per iteration where the n is the number of vertices of input graph, but for most cases the time complexity would be much less than $O(|V|^3)$ and more than $O(|V|^2)$. The space complexity would be $O(1)$.

4.4 Local Search - Neighborhood 3-opt Exchange

4.4.1 Description. The second algorithm we chose for local search is 3-opt exchange algorithm.The 3-opt algorithm is a general version of 2-opt algorithm, where three edges are swapped at a time. The starting solution we have for the traveling salesman problem is also generated randomly to includes all locations in the city graph, and we use the seed to record the random value in order to record the performance. The main idea of obtaining the neighbors is to delete three edges in a route to create 3 sub-routes, then there will

be 7 different ways of reconnecting the network to be analyzed to find the more promising solution. For a given route, we will generate all segments combinations, and we will try to improve the route by reversing segments as the instruction above. The reconnecting process repeats for a distinct set of three connections until all possible combinations have been tested. For each interaction, we will compare the current best distance with the new modified distance, then apply the modification and return the relative cost if we find a better cost. Compared to the 2-opt exchange algorithm and 3-opt exchange algorithm, we noticed that the 2-opt algorithm fun faster than the 3-opt algorithm, while 3-opt showed a more optimum solution than the 2-opt algorithm. The reason is that there may exists a sequence of routes will improve the tour but it will increase the distance based on the 2-opt algorithm, therefore, this kind of "bad" move will not be executed when we use the 2-opt optimization only. The pseudo-code of the 3-opt exchange algorithm are showed in Algorithm 6.

Algorithm 6 3-opt Algorithm

```
function THREE_OPT(route)
    while True do
        d ← 0
        best_distance ← ∞
        all_segments ← all segments combinations
        for (a, b, c) in all_segments do
            d ← d + three_opt_swap(route, a, b, c)
            cur_distance ← cycleValue(route)
            if cur_distance<best_distance then
                best_distance ← cur_distance
                record the update route
            end if
        end for
        if d ⩾ 0 then
            break
        end if
    end while
    return route
end function

function THREE_OPT_SWAP(route)
    A, B, C, D, E, F ← cutting points on the route by the given index
    d0 ← distance(A, B) + distance(C, D) + distance(E, F)
    d1 ← distance(A, C) + distance(B, D) + distance(E, F)
    d2 ← distance(A, B) + distance(C, E) + distance(D, F)
    d3 ← distance(A, D) + distance(E, B) + distance(C, F)
    d4 ← distance(F, B) + distance(C, D) + distance(E, A)   ▷ reconnecting the cutting points to rebuild the distinct routes
    compare different reconnecting routes, then reversing the tour if it can make the tour shorter
    return 0
end function
```

4.4.2 Time Complexity. The time complexity of the 3-opt exchange algorithm depends on the number of the locations in a certain city at each step. Since there are 3 sub-routes to be tested in each route, the time complexity of 3-opt algorithm is $O(|V|^3)$ for every iteration where the n is the number of vertices of input graph.

Table 1: Results table for Branch and Bound: reduce matrix lower bound.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	495.35	2044178	0.0202
Berlin	137.46	19237	1.5506
Boston	559.86	1968601	1.2032
Champaign	361.69	177751	2.3765
Cincinnati	0.06	277952	0.0000
Denver	551.17	516227	4.1401
NYC	112.81	6406783	3.1200
Philadelphia	354.96	3285566	1.3536
Roanoke	586.56	6766869	9.3239
SanFrancisco	397.70	5335446	5.5854
Toronto	346.64	8726000	6.4191
UKansasState	0.30	62962	0.0000
UMissouri	588.76	578067	3.3559

5 EMPIRICAL EVALUATION

5.1 Comprehensive Results

In the following, we will use BnB, Approx, LS1, LS2 as abbreviations of Branch and Bound, MST Approximation, Local Search 1 Neighborhood 2-opt Exchange, Local Search 2 Neighborhood 3-opt Exchange, respectively. Our experiments are run on a PC with the following configurations:

- CPU: lntel(R) Core(TM) i7-6700 CPU @3.40GHz, 8 Cores
- RAM: 16.0 GB
- OS: Windows 11, 64 bit

Besides, all of the experiments are implemented and run by Python 3, in the series mode on only one core of the CPU. For BnB, we set the cutoff time to be 600 s; for LS1 and LS2, we run each of the 13 instances with cutoff time 10 s for 10 times using seed 0 to 10 respectively, and average them as the final results. Table 1 to 4 shows the corresponding results of the four algorithms. For each algorithm and each instance, the Time and Solution Quality column represents the *time used in seconds* to achieve the *current best solution* before reaching the cutoff time. The Relative Error is computed as

$$\text{RelErr} = \frac{\text{Sol.Qual.} - \text{OPT}}{\text{OPT}}$$

Thus, the approximation ratio can be deduced as

$$\text{ApproximateRatio} = \text{RelErr} + 1$$

5.2 Analysis

For BnB, we find that the relative error is highly correlated with the instance size, since it follows a fixed procedure to try to find the exact solution. Therefore, an instance with greater size often leads to greater relative error. In particular, for the instance of Cincinnati and UKansasState, both of them only have 10 nodes, so it takes just about 1 s to finish all possible branches and obtain the exact solution.

Table 2: Results table for Approximation: minimum spanning tree.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.00	2407661	0.2016
Berlin	0.03	9777	0.2963
Boston	0.01	1047623	0.1724
Champaign	0.03	61508	0.1684
Cincinnati	0.00	296969	0.0684
Denver	0.05	124990	0.2445
NYC	0.03	1825253	0.1738
Philadelphia	0.00	1708703	0.2240
Roanoke	0.43	789208	0.2041
SanFrancisco	0.05	1073660	0.3252
Toronto	0.07	1596348	0.3573
UKansasState	0.00	66528	0.0566
UMissouri	0.06	153061	0.1534

Table 3: Results table for Local Search 1: Neighborhood 2-opt Exchange.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	0.04	2047460	0.0218
Berlin	1.65	8094	0.0732
Boston	0.55	931470	0.0425
Champaign	2.04	54340	0.0322
Cincinnati	0.99	277953	0.0000
Denver	9.94	107290	0.0683
NYC	4.68	1625272	0.0452
Philadelphia	0.17	1427557	0.0226
Roanoke	8.98	6447342	8.8365
SanFrancisco	9.95	1463736	0.8066
Toronto	9.92	3001857	1.5523
UKansasState	0.62	62962	0.0000
UMissouri	9.93	259892	0.9584

Table 4: Results table for Local Search 2: Neighborhood 3-opt Exchange.

Dataset	Time (s)	Sol.Qual.	RelErr
Atlanta	2.57	2003763	0.0000
Berlin	4.42	7855	0.0415
Boston	5.02	895481	0.0022
Champaign	5.57	53415	0.0147
Cincinnati	0.04	277953	0.0000
Denver	4.20	106004	0.0555
NYC	5.14	1594776	0.0255
Philadelphia	4.35	1395981	0.0000
Roanoke	9.60	705294	0.0760
SanFrancisco	7.10	863644	0.0660
Toronto	5.91	1261738	0.0728
UKansasState	0.04	62962	0.0000
UMissouri	7.93	139927	0.0544

Table 5: Comparison of different lower bound of BnB algorithm for the UMissouri case. RM for reduce matrix and SE for smallest edge.

LB	Time (s)	Sol.Qual.	RelErr	Rej.Rate.	Deepest level
RM	588.76	578067	3.3559	90.14%	92
SE	432.62	657540	3.6992	74.26%	104

But for the instance like Roanoke and Toronto with hundreds of nodes, their relative errors are very high. Furthermore, we also implement the BnB using smallest edge as branch lower bound, to compare the efficiency with the BnB using reduce matrix as branch lower bound. We test, for example, the UMissouri instance under the same setting, and also record the rejection rate (frequency of branch pruning) and the deepest level it reaches. The result shown in Table 5 clearly shows that the reduce matrix lower bound is much more efficient than the smallest edge lower bound.

Compared with BnB trying to find the exact solution, Approx gives a very fast and stable approximation (as shown in Table 2). Although the approximation factor $\rho(|V|) = 2$, the actual approximation rates are far lower than 2 in most of time, and there is indeed no obvious relationship with respect to the size of the input instance. For example, for Roanoke instance with 230 nodes (the largest input instance), it only takes 0.43 s to reach an approximation ratio of $1.2041 < 2$. And the lower bound of the approximation ratio across all 13 instances is 0.05. That means for larger input instance, this approximation algorithm is highly desirable. But the only obvious drawback here is that for small sized input instance, this approximation algorithm still not able to give the exact optimal solution, although it is easily reachable.

When it comes to the two local search algorithms, the relative error now depends on the size of the input instance and the given cutoff time. If someone gives enough time for a large sized input, local search algorithms could still give a quite good solution (refer to the next paragraph for more details). Compared with 2-opt, 3-opt takes longer time but could give a better solution, this is consistent with the theoretical one that 3-opt is slower but give high quality solutions. Besides, when the input size becomes larger, the advantage of 3-opt is much more obvious (e.g., the Roanoke instance).

For local search algorithms, furthermore, we can visualize the performance of the algorithm by plotting the corresponding Qualified RunTime Distribution (QRTD), Solution Quality Distribution (SQD), and run-time boxplot. Specifically, QRTD shows the probability of obtaining a solution within a specific relative error with respect to CPU time (run-time); SQD shows that within a specific cutoff time, the probability of obtaining a solution with respect to the relative error; and run-time boxplot shows distributions of run-time for different Relative Error. For each of the two local search algorithms, we run the Berlin and the Champaign instances for 60 times, to generate the three plots (shown in Figure 1 – 4).

Comparing LS1 (2-opt) with LS2 (3-opt) for both instances, we could see that although 3-opt is slower than 2-opt in theory, 3-opt could produce higher quality solutions. Compared with LS1, the QRTD curve of LS2 grows fast at beginning and quickly converge when approaching to $p = 1$, which means the time used at the

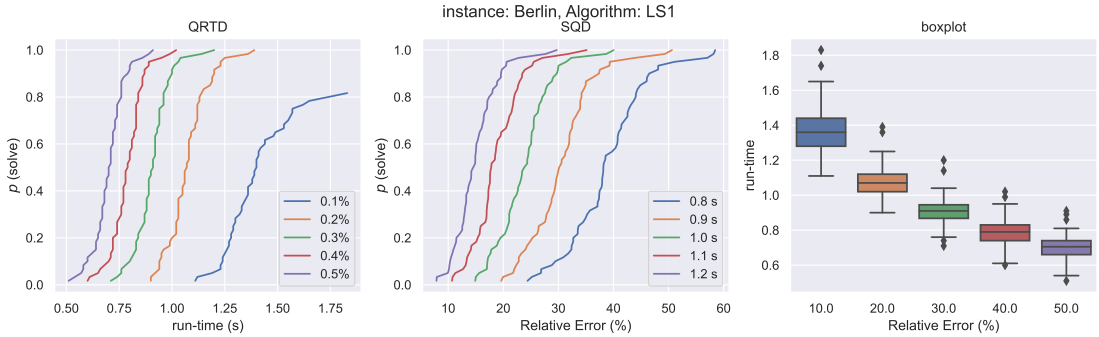

Figure 1: QRTD (left), SQD (middle), and run-time boxplot (right) for LS1 algorithm and Berlin instance.

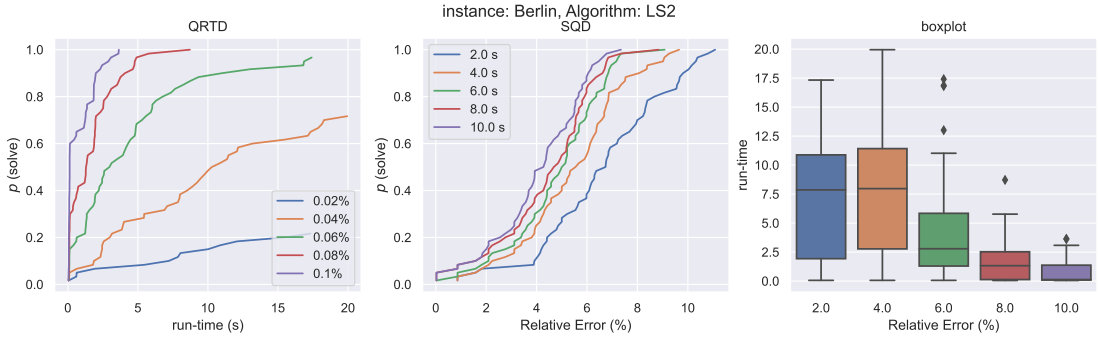

Figure 2: QRTD (left), SQD (middle), and run-time boxplot (right) for LS2 algorithm and Berlin instance.

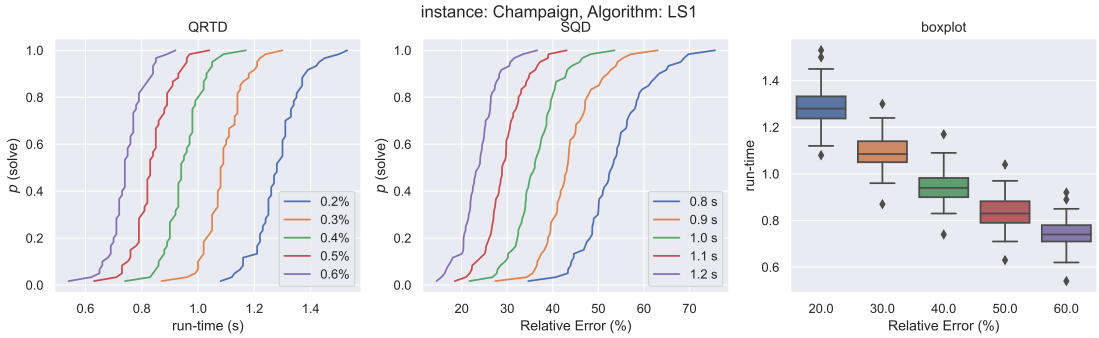

Figure 3: QRTD (left), SQD (middle), and run-time boxplot (right) for LS1 algorithm and Champaign instance.

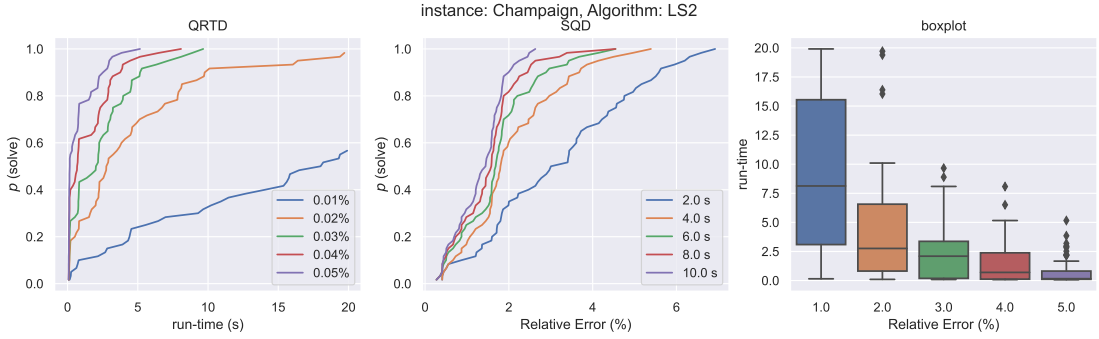

Figure 4: QRTD (left), SQD (middle), and run-time boxplot (right) for LS2 algorithm and Champaign instance.

very beginning of the algorithm is above rubies. But for the QRTD curve of LS1, it keep close to 0 at the beginning for a relatively long time, and then grows up sharply in a specific range of the relative error (which is just the range of relative error the algorithm is able to achieve in practice), and then quickly converges to $p = 1$ after going out of that specific range. This means LS1 needs some time to find a good basin where solutions are better.

In the SQD plots of LS2, a great variation in cutoff time still produces a family of dense curves, with smaller relative error, but varying cutoff of LS1 a little bit gives distancing curves. This means cutoff time is a sensitive factor for LS1, which is consistent with the conclusion drawn in our previous analysis. That is, LS1 takes some time to gradually find better solution, but LS2 quickly find very good solution after passing a specific time point.

In the run-time boxplot, we could see the differences of the run-time distribution under different relative error. For LS1, the average run-time steadily decrease when the accepted relative error increases; and the variance (possible range of the run-time) of the distribution does not change significantly, which means a better solution always takes more time, without exception. But for LS2, both the mean and variance of the run-time distribution decrease when relative error increases. Therefore, if you accept a greater relative error, LS2 always gives you a solution under that relative error in a very short time. But, if you really need a solution that is very close to the exact solution, the efficiency of LS2 will highly depend on the randomized initial condition. Noting that although LS2 producing the worst relative error in about one second, that relative error is still much smaller than that of LS1 under the similar run-time distribution. Therefore, LS2 is a better choice than LS1 in general.

6 DISCUSSION

Up to now, we have shown that the performances of the four TSP algorithms so that a comprehensive comparison is within reach. Based on reduce matrix lower bound, BnB is proficient in finding the exact optimal solution for small instances. However, when the input instance is a little bit large, the time cost grows exponentially since its time complexity is in $O(n^2 2^n)$, and hence lost its feasibility. For a very great sized input instance, we may turn to use Approx, since it always gives a solution in 1 second with a constant approximation factor 2. Although it has the same time complexity $O(|V|^3)$ with the local search algorithm, the run-time in practice is quite smaller than that of local search algorithms, because it always follow an intuitive way to produce a result, and which is very stable (not depending on some random factors). However, the drawback of the Approx is that for smaller instances, it still cannot give a very good solution, since the intuitive approach is not obviously depending on input size. So, if we hope to find the exact solution for a small input instance, Approx is not a good choice. Besides, Approx is the only one that its solution quality not depending on the cutoff time, which means giving more time for getting a better solution is useless.

If the situation is more flexible, local search algorithms may be better candidates. Specifically, if we hope to find a solution that is very close to the exact solution, we can give local search algorithm enough time to improve the solution quality; and if we want to get a feasible solution in a short time, we can just stop at any time we want, and taking the current best solution. Compared with BnB and Approx, local search algorithms play intermediate roles. For small-sized input instances, they could give a very good solution in a very short time; and if the cutoff time is a little bit longer, the solution will be very close to the exact optimal solution. For large-sized input instances, local search algorithms could still give a very good solution in a few seconds, especially the 3-opt algorithm. And when comparing the two local search algorithms, 3-opt is a little bit slower but gives better solution than 2-opt. This benefit of 3-opt is significant under larger-sized input instance. Therefore, although 3-opt takes a little bit longer time, it could obtain a better solution, and which is useful in real world cases. And if the input size is very large, 3-opt could make a better using of the current solution to find better solution than that of 2-opt, and now 3-opt is the better one.

In summary, for NP-hard problems like TSP, there is no perfect algorithm unless P = NP. Therefore, we have to choose the suitable algorithm under the given condition. Besides, the cutoff time and the quality of the solution we desire will also affect the our selection strategy.

7 CONCLUSION

A detailed analysis on 4 distinct algorithms have been presented in this report. In conclusion, the choice of the algorithm to solve the TSP problem should be made based on the resources available, the accuracy required, and the usage scenario. For an small-sized application requiring a high degree of accuracy with no constraint on the running time and computing power, the BnB algorithm would be the best choice. Compared with BnB trying to get the exact solution within a fixed procedure, the Approximation approach can always satisfy the requirement to get the quick and stable approximation but sacrifice the accuracy, especially for small-sized instance. For two local search algorithms, the LS1 (2-opt) run faster than the LS2 (3-opt) while the LS2 algorithm provides more promising solution than the LS1 algorithm, and this benefit is significant under large-sized input.

REFERENCES

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971, pp. 151–158.

[2] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[3] E. L. Lawler, "The traveling salesman problem: a guided tour of combinatorial optimization," *Wiley-Interscience Series in Discrete Mathematics*, 1985.

[4] S. G. Henderson and B. L. Nelson, *Handbooks in operations research and management science: simulation*. Elsevier, 2006.

[5] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245–2269, 1965.

[6] J. D. Little, K. G. Murty, D. W. Sweeney, and C. Karel, "An algorithm for the traveling salesman problem," *Operations research*, vol. 11, no. 6, pp. 972–989, 1963.

[7] B. L. Golden and W. Stewart, "The traveling salesman problem. a guided tour of combinatorial optimization, chapter empirical analysis of heuristics," 1985.

[8] M. J. G. R. G. Rinaldi, G. Jünger, and G. Reinelt, "The traveling salesman problem," 1994.

[9] M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges: Fifth and Sixth DIMACS Implementation Challenges: Papers Related to the DIMACS Challenge on Dictionaries and Priority Queues (1995-1996) and the DIMACS Challenge on Near Neighbor Searches (1998-1999)*. American Mathematical Soc., 2002, no. 59.

[10] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The traveling salesman problem*. Princeton university press, 2011.

[11] K. L. Hoffman, M. Padberg, G. Rinaldi *et al.*, "Traveling salesman problem," *Encyclopedia of operations research and management science*, vol. 1, pp. 1573–1578, 2013.

[12] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *European journal of operational research*, vol. 126, no. 1, pp. 106–130, 2000.