Coping with Travelling Salesman Problem (TSP)

Hanchen Wang (hwang787)[1] Jing Shi (jshi84)[2] Tiantian Fu(tfu46)[3] and Xingjian Wang (xwang971)[4]

Abstract— **In this project, we explore numerous approaches to cope with one famous NPC problem, TSP. Several algorithms were implemented and compared with each other in terms of performance, accuracy, speed and optimum of the results. Further we discuss potential improvement can be made and which algorithm is more suitable in a situation-specific manner.**

I. INTRODUCTION

The traveling salesman problem (TSP) is a classic NP-complete problem with various applications in fields including computational biology, astronomy, VLSI design, robot control and management of resources. In this project, various algorithms for approaching the TSP problem have been performed and their theoretical and experimental complexities have been evaluated. Specifically, four algorithms are implemented and studied, i.e. branch-and-bound algorithm, an approximate algorithm (MST), and two local search algorithms. Evaluation of the performance of these algorithms have been performed and analyzed.

II. PROBLEM DEFINITION

TSP can be modelled as an undirected weighted graph G = (V, E). Cities are the vertices V in the graph and paths are the edges E. The weight of an edge represents the corresponding path's distance. The goal is to find the shortest path which starts and ends at a specific vertex and goes through every other vertex exactly once.

More formally, We define the TSP problem as follows: 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), the goal is to find the shortest simple cycle that visits all N points. This version of the TSP problem is metric, i.e. all edge costs are symmetric and satisfy the triangle inequality.

III. RELATED WORK

TSP was one of the first problems to be proven NP-hard by Karp in 1972. Various algorithmic techniques have been applied to TSP, including but not limited to branch and bound, simulated annealing, Lin-Kernighan type methods, approximation algorithms.

Exact algorithms work only for small problem sizes. Various branch and bound algorithms have been developed, like 2 shortest edges, MST bounding function.

Another category is devising heuristic and approximation algorithms that deliver relatively good solutions in a reasonable time. The nearest neighbor (NN) algorithm lets the salesman choose the nearest unvisited city as next move. Rosenkrantz et al.[1] showed that the NN algorithm has the approximation factor $\Theta(log|V|)$. The MST-approximation algorithm has the approximation factor 2. Christofides' Algorithm[2] combines the minimum spanning tree with minimum-weight perfect matching, which can give a TSP tour which is at most 1.5 times the optimal.

Local search algorithms such as simulated annealing, iterated local search, have also produced good results for TSP problem. They iteratively improve the current solution by searching for a better one in its predefined neighborhood. Historically, 2-opt[3] was one of the first successful algorithms to solve larger TSP instances. The 2-opt technique involves iteratively removing two edges and replacing these with two different edges that reconnect the fragments to form a new and shorter tour. Similarly, the 3-opt technique removes 3 edges and reconnects them to form a shorter tour. Lin and Kernighan[4] extended this idea to more complex neighborhoods.

IV. ALGORITHM

Our project include four main algorithm which are Branch and Bound, Iterated local search, Simulated annealing and MST approximation, which are described in more detail in the following sections.

A. Branch and Bound

The main idea of branch and bound is to use lower bound to prune the tree. If the lower bound on the node is worse than best possible solution, then we ignore the subtree rooted with the node.

We choose 2 shortest edges as the lower bound function, that is, we calculate new bound by removing the edge with smallest value of a node and adding the distance from this node to last node to previous low bound.

To calculate bound, first we need to set the rule of what the bound stands for. In such case, since we judge the result with distance, the bound here obviously represent the minimum total distance we need to cover.

We can find, in such problem, every node will be connected, meaning each one will have exact two edges connecting with. If two edges of each node are its two edges with two smallest value, then the total distance will be the smallest, and such case thus can be set as the most optimal case.

If we use min_i to represent the edge with minimum value of $node_i$ and $secmin_i$ to represent the edge with second minimum value of that node, then, the total distance in such case is

$$\frac{1}{2}\sum_{i=1}^{n}(min_i + secmin_i)$$

The reason why the sum should be half is one edge can only be counted once, however, we count every edge when we find a node connected with it. Because one edge is connected with 2 nodes, the edge is thus considered twice, needing half.

With the initial bound, we need to use two edges function to increase it until we get the solution. For $node_i$ we are considering, if the previous node is $node_p$ and distance of edge from $node_i$ to $node_j$ is $dis(p,i)$, we can then get the new bound according to the formula below:

$$bound_{new} = bound_{old} + dis(p,i) - min_p - min_i(p = 1)$$

$$bound_{new} = bound_{old} + dis(p,i) - secmin_p - min_i(p > 1)$$

The change in the formula is that when we work on the first node, it has no edge connected with, however, when we handle other nodes, we need included second minimum edge for previous node, because the minimum edge cost has already been subtracted in previous level.

If we find the total distance is larger than the current best total value when we move to the $node_i$ at this step, we remove it from the list and thus we do not need to consider nodes in its subtree as its next node.

In order to increase the accuracy within the given time, we use the MST's result as the lowerbound at first, and then during the traversing, when we find the bound smaller than it, we update it and add certain city into the list.

Algorithm 1 Branch and Bound

```
Initialize tempbound using MST
Initialize lowerbound using two shortest ways from every pair of cities.
dis←MST.get(dis)
path←[]
for i in random(city numbers) do
    if distance between last city in path and i city !=0 and i has not been visited then
        add distance between last city and i city to current distance.
        lowerbound=lowerbound-(minimum distance from i city+ last city's second shortest path)/2
        lowerbound+=distance between last city in path and i city
        if lowerbound<tempbound then
            tempbound=lowerbound
        end if
        if tempbound<tempdis then
            add i to dis[]
            check next random city
            set i as visited
            BNB()
        end if
    end if
end for
if no update then
    return tempvalue,dis
else
    return tempvalue, path[]
end if
```

B. Simulated Annealing

The main idea of simulated annealing is as follows. First, we randomly choose a path as the initial solution. Then, we randomly choose two nodes which are not start nodes to decide exchanging or not. The rule is if the distance is decreased after exchanging, then we do the swap; otherwise do swap according to the probability. We update the path and distance every time we exchange. When the path and distance are not updated after iteration times we set, the algorithm finishes.

Simulated annealing can accept worse solutions which allows for a more extensive search for the global optimum solution. The probability of accepting worse solutions gradually decreases as the solution space is explored. And the probability we choose to use is 1.1^{-n}, in which n means exchange times. The pseudo-code is as below:

C. Iterated Local Search

Usually, the simple local search can get stuck in a local optimum, where no improving neighbors can be found. While by applying permutations to the current minimum and applying local search on modified solution, the Iterated Local Search go beyond the local optimum and find the global optimum.

The main idea of Iterated Local Search is as follows: First, we construct an initial solution and do local search on it to find a local optimum s^*. Then we apply a perturbation that leads to an intermediate state s'. Then we apply local search to s' and we reach a solution s'^*. If s'^* is a better solution, we use it for the next iteration of the walk; otherwise, we return to s^*. To be more specific, we use 2-opt as the local search algorithm and double bridge as the perturbation method. The double bridge method involves changing 4 vertices at the same time, ensuring it can't be easily achieved by 2-opt.

The main strength of Iterated Local Search is that it won't be stuck in a local optimum.

D. MST approximation

In our problem, the distance satisfies the triangle inequality, ensuring we can get at most 2 times of the optimal distance using MST approximation.

The main ideas of MST approximation is as follows. First, compute the minimum spanning tree (MST) of the TSP graph. Then we do a depth-first search (DFS) of MST, hitting every edge exactly twice. Last, we write each vertex only the first time it appears in the DFS tour. The pseudo-code is as below.

The MST approximation algorithm is a deterministic algorithm. The time complexity for MST approximation is $O(n^2 logn)$.

Algorithm 2 Simulated Annealing

```
$final_{path} \leftarrow$ choose nodes from input randomly
$final_{value} \leftarrow$ calculate total distance based on $final_{path}$

while it< times we set do
  pick two nodes randomly
  if distance increases after exchanging two nodes positions then
    exchange their position and update path
    $it \leftarrow 0$
  else
    exchange based on the probability
    if exchange then
      $it \leftarrow 0$
      update distance and path
    end if
  end if
end while
return $final_{path}, final_{value}$
```

Algorithm 3 Iterated local search (ILS)

```
Construct an initial solution $s_0$.
$s^*$ = 2-opt-localSearch($s_0$)
while Runtime is less than cutoff and duplicate count is less than max duplicate count do
  $s^{'} \leftarrow$ do double bridge perturbation on $s^*$.
  $s^{'*}$ = 2-opt-localSearch($s^{'}$)
  if cost($s^{'*}$) < cost($s^*$) then
    $s^* \leftarrow s^{'*}$
  end if
end while
```

V. Empirical evaluation

A. Experimental platform

The program is written in Python. All the algorithms are run on a Lenova Y40-70 laptop with 8GB RAM and 2.60 GHz Intel core i7.

B. Experimental procedure and evaluation criteria

First, for all algorithms and all instances, the time, solution quality (the best solutions found by the algorithms), and the relative error are considered. The relative error is computed as $(Alg - OPT)/OPT$. To get this data, run MST approximation, branch-and-bound, SA (LS1), iterated local search (LS2) for all instances. The cutoff time for branch-and-bound is set to 600 seconds. And for local search algorithms, the cut-off time is 300 seconds. Since two local search algorithms rely on initial path construction, run each instance for 10 times to get the average value.

Second, for two local search algorithms, we choose two instances, Atlanta.tsp (20 vertices) and Champaign.tsp (55 vertices). We run each instance for 20 times with the cut-off time as 100 seconds and obtained the corresponding traces. Then we plot qualified runtime for various solution qualities

Algorithm 4 MST approximation

```
Compute MST of TSP graph using Prim's algorithm.
tour = DFS(MST)
$final_{path}$ = write each vertex the first time in the tour.
return $final_{path}$
```

(QRTDs), solution quality distributions for various run-times (SQDs), and box plots for running times.

C. Results

Table I shows the the time, the solution quality, and relative error for all instances for all algorithms.

From the tables above, we could see that the relative errors for MST approximation algorithm is within range 0% to 38%, which is within the theoretical approximation ratio 2. At the same time, the runtime for MST is much better than other algorithms since it's a deterministic algorithm. The relative errors for branch-and-bound algorithm is also within range 0% to 38%. We can notice that most relative errors are consistent with those gotten from MST approximation. This is because MST here assign initialization for upper bound and current best solution, (details in pseudo-code) By doing so, we have tighter upper bound and lower bound to eliminate unnecessary sub-problem. And branch-and-bound doesn't find better solutions within the cut-off time.

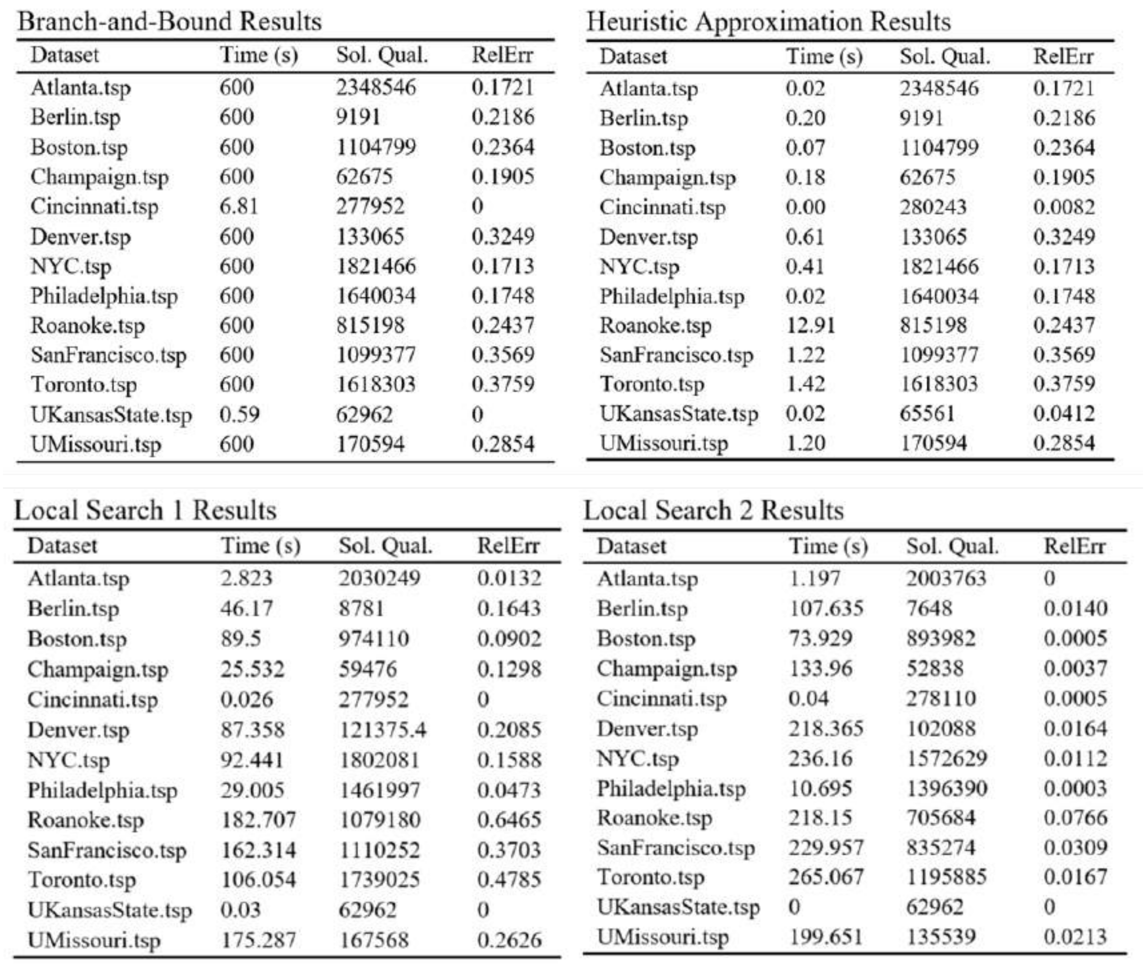
The relative errors for simulated annealing is within range 0 to 64.65%. The largest relative error occurs when the dataset has the most vertices, i.e., Roanoke. The relative errors for iterated local search is within range 0 to 7.66%. In terms of solution quality, iterated local search performs better.

The QRTD and SQD are shown in Figure 1-2. The box plots are shown in Figure 3.

VI. Discussion

A. Qualified Runtime Distribution (QRTD) for Various Solution Qualities

We obtained the plot of QRTD of two Local Search algorithms (LS1 and LS2) by varying the relative solution quality q*. q* ranges from 1% to 25% for LS1 while ranges from 0.5% to 20% for LS2. Analyzing the results, we found that LS2 algorithm can give more accurate results than LS1 algorithm. Therefore, we used higher values of q* for the QRTD plot of LS2 algorithm. The four plots for the two algorithms and two cities are shown in Figure 2, which indicates that for both the two cities (Atlanta and Champaign), LS2 algorithm is able to generate more accurate results in less time. Specifically, for the smaller dataset of Atlanta, solutions from LS2 are more accurate than those from LS1. LS2 can compute a solution within a relative error of 1% within 1 second while LS1 cannot provide a solution with that quality within 5 seconds. For the larger dataset of Champaign, the solution quality decreased for both of these two algorithms. Therefore, higher values of q* are adopted. The performance of LS1 algorithm is still worse



Branch-and-Bound Results

Dataset	Time (s)	Sol. Qual.	RelErr
Atlanta.tsp	600	2348546	0.1721
Berlin.tsp	600	9191	0.2186
Boston.tsp	600	1104799	0.2364
Champaign.tsp	600	62675	0.1905
Cincinnati.tsp	6.81	277952	0
Denver.tsp	600	133065	0.3249
NYC.tsp	600	1821466	0.1713
Philadelphia.tsp	600	1640034	0.1748
Roanoke.tsp	600	815198	0.2437
SanFrancisco.tsp	600	1099377	0.3569
Toronto.tsp	600	1618303	0.3759
UKansasState.tsp	0.59	62962	0
UMissouri.tsp	600	170594	0.2854

Heuristic Approximation Results

Dataset	Time (s)	Sol. Qual.	RelErr
Atlanta.tsp	0.02	2348546	0.1721
Berlin.tsp	0.20	9191	0.2186
Boston.tsp	0.07	1104799	0.2364
Champaign.tsp	0.18	62675	0.1905
Cincinnati.tsp	0.00	280243	0.0082
Denver.tsp	0.61	133065	0.3249
NYC.tsp	0.41	1821466	0.1713
Philadelphia.tsp	0.02	1640034	0.1748
Roanoke.tsp	12.91	815198	0.2437
SanFrancisco.tsp	1.22	1099377	0.3569
Toronto.tsp	1.42	1618303	0.3759
UKansasState.tsp	0.02	65561	0.0412
UMissouri.tsp	1.20	170594	0.2854

Local Search 1 Results

Dataset	Time (s)	Sol. Qual.	RelErr
Atlanta.tsp	2.823	2030249	0.0132
Berlin.tsp	46.17	8781	0.1643
Boston.tsp	89.5	974110	0.0902
Champaign.tsp	25.532	59476	0.1298
Cincinnati.tsp	0.026	277952	0
Denver.tsp	87.358	121375.4	0.2085
NYC.tsp	92.441	1802081	0.1588
Philadelphia.tsp	29.005	1461997	0.0473
Roanoke.tsp	182.707	1079180	0.6465
SanFrancisco.tsp	162.314	1110252	0.3703
Toronto.tsp	106.054	1739025	0.4785
UKansasState.tsp	0.03	62962	0
UMissouri.tsp	175.287	167568	0.2626

Local Search 2 Results

Dataset	Time (s)	Sol. Qual.	RelErr
Atlanta.tsp	1.197	2003763	0
Berlin.tsp	107.635	7648	0.0140
Boston.tsp	73.929	893982	0.0005
Champaign.tsp	133.96	52838	0.0037
Cincinnati.tsp	0.04	278110	0.0005
Denver.tsp	218.365	102088	0.0164
NYC.tsp	236.16	1572629	0.0112
Philadelphia.tsp	10.695	1396390	0.0003
Roanoke.tsp	218.15	705684	0.0766
SanFrancisco.tsp	229.957	835274	0.0309
Toronto.tsp	265.067	1195885	0.0167
UKansasState.tsp	0	62962	0
UMissouri.tsp	199.651	135539	0.0213

TABLE I: Runtime, solution quality, relative error for all instances for all algorithms

than LS2, especially with its much longer running time. The plots for both these two algorithms shows the trend that when the value of q* decreases, the successful probability of the algorithms are lowered.

B. Solution Quality Distributions (SQD) for Various Runtimes

We obtained the plots of SQD of two Local Search algorithms (LS1 and LS2) by varying the runtime t*, as shown in Figure 3. Figure 3 shows intuitively that more accurate solutions can be obtained with longer running times. Besides, the SQD plots also indicate that LS2 performs better than LS1. By comparing the data of Atlanta/LS1 and Atlanta/LS2, we can see that LS2 can achieve better solution quality (lower relative error) than LS1 within a similar time scale. By comparing the data of champaign / LS1 and champaign / LS2, we can find that LS2 can obtain a solution within a relative error of 10% in 5 seconds. In contrast, LS1 cannot do that even with 100 seconds.

C. Box Plots For Runtime Comparison

Local Search algorithms rely on random steps during their process of finding the optimum solution. Therefore, the obtained results are uncertain. For relieving the uncertainty, people tend to run the algorithm multiple times and then use the average value as the result. Here we use a similar approach to analyze the runtime complexity of these two Local Search (LS1 and LS2) algorithms with two cities Atlanta and Champaign. Box plots (shown in Figure 3) are used for each combination of different algorithms and cities so that the distribution of the results can be shown intuitively.

Note that to make two local search algorithms comparable, we fixed the solution quality. For Atlanta dataset, 12.5% solution quality is used; for Champaign dataset, 22.5% solution quality is used. Both SA and iterated local search can reach 12.5% quality within 0.1s. The upper quartile, median, and the lower quartile of iterated local search is slightly less than SA. For the Champaign dataset, we can observe a few outliers in SA, which takes more than 20 seconds to reach the 22.5% solution quality, while the median is less than 10 seconds. No outliers are examined in iterated local search. It can reach 22.5% solution quality within 1.5s.

VII. Conclusions

We have implemented several algorithms to compute the traveling salesman problem. Our experiments show that the MST-approximation algorithm can calculate the

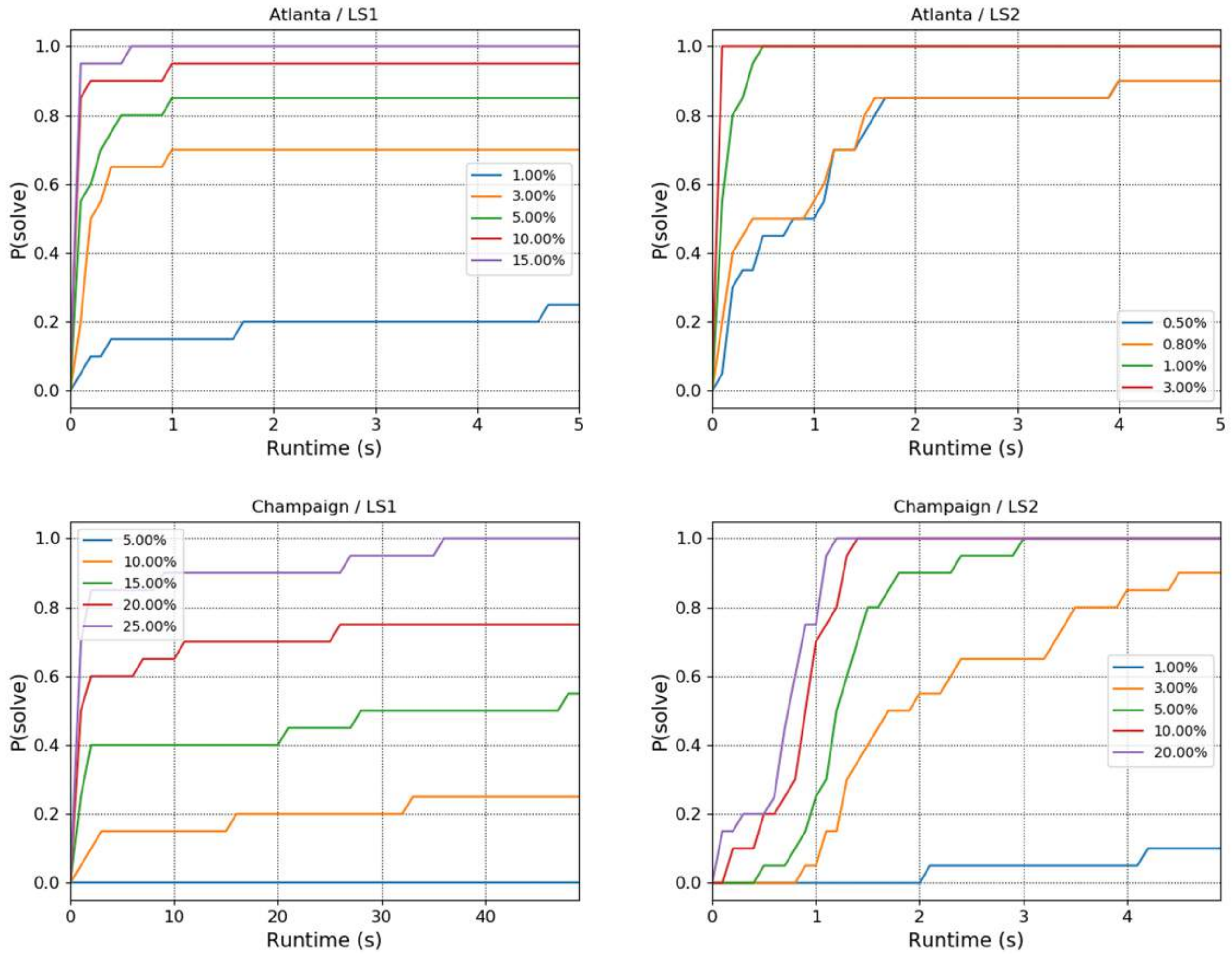


Fig. 1: Qualified Runtime for Various Solution Qualities graphs (QRTD) of two cities, Atlanta and Champaign, with two local search algorithms (LS1 and LS2).

distance with the fastest speed and yet a pretty good result compared to other algorithms. The branch-and-bound can obtain optimal results but the time it took exceed the cutoff time. Therefore, branch-and-bound algorithm cannot return the optimal results within the cutoff time. The iterated local search algorithm can solve the problem with a faster speed and higher accurracy than the local search algorithm with simulated annealing. Specifically, BnB with MST approximation as initialization turns out to be one effective method for improving the accuracy and speed. As MST will assign Bnb a tighter lower bound and upper bound, which can improve the performance.

References

[1] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "Approximate algorithms for the traveling salesperson problem," in *Proceedings of the 15th Annual Symposium on Switching and Automata Theory (Swat 1974)*, SWAT '74, (Washington, DC, USA), pp. 33–42, IEEE Computer Society, 1974.

[2] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[3] G. CROES, "A METHOD FOR SOLVING TRAVELING-SALESMAN PROBLEMS," *OPERATIONS RESEARCH*, vol. 6, no. 6, pp. 791–812, 1958.

[4] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, no. 2, pp. 498–516, 1973.

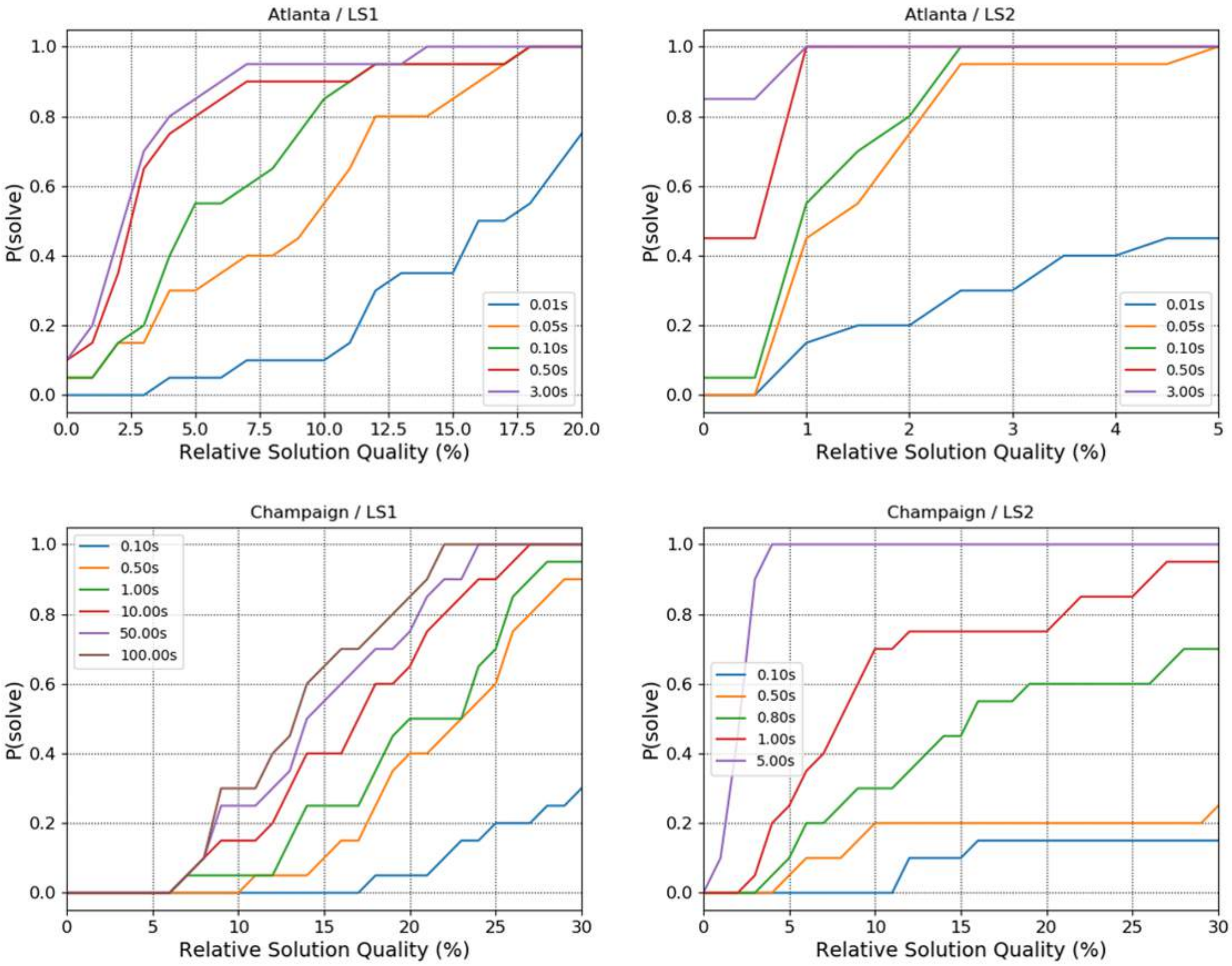


Fig. 2: Solution Quality Distributions for the runtimes of two cities, Atlanta and Champaign, with two local search algorithms (LS1 and LS2).

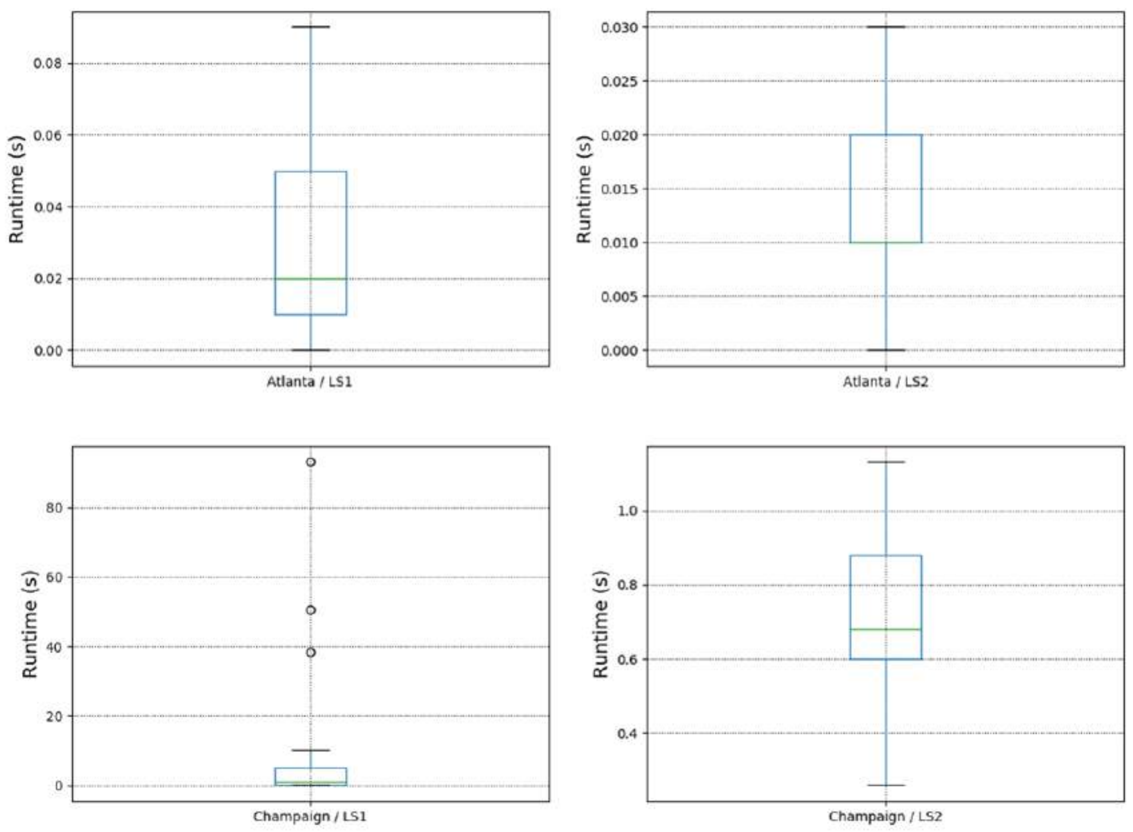


Fig. 3: Boxplots for the runtimes of two cities, Atlanta (relative error set to 12.5%) and Champaign (relative error set to 22.5%), with two local search algorithms (LS1 and LS2).