A Study and Evaluation of Algorithms for Solving Traveling Salesman Problem

HENGHAO LI, Georgia Institute of Technology, USA
WENTING TAN, Georgia Institute of Technology, USA
XIA WU, Georgia Institute of Technology, USA
YUNHAN ZOU, Georgia Institute of Technology, USA

The traveling salesman problem is a classic NP-hard problem in computer science and operations research, focusing on providing optimal solutions[15]. We proposed and implemented four algorithms in this study: branch and bound, MST approximation, iterative local search and simulated annealing. All four algorithms were tested and evaluated on the data set provided on TSPLIB[11].

Key Words and Phrases: Traveling salesman problem, exact algorithm, MST approximation, approximation algorithm, NP-complete

1 INTRODUCTION

The traveling salesman problem (TSP) has been served as one of the most intensively studied problem[12], which was first formulated in 1930s[16]. It is an NP-hard computational problem and used as a benchmark for several optimization problems over the years[16]. There already exist a large scale of exact solutions and approximating methods. Besides, it can be widely applied in many different fields other than computer science, such as DNA sequencing and astronomy.

In this study, we solve this problem with four different algorithms. Branch-and-bound is a classic method and can help us find the exact solution by sacrificing the running time. MST heuristic is an approximation method that can reduce the time complexity to "nearly" linear, but may introduce offsets to the result. Iterative local search and simulated annealing are two efficient local search algorithms that can find a solution, if not at, near global minimum to this optimization problem.

2 PROBLEM DEFINITION

2.1 Travelling salesman problem

The traveling salesman problem, which is a classic NP-hard problem[14] in combinatorial optimization, aims to solve the following problem: given a list of cities and the distances between these cities, find the shortest possible route that visits all cities exactly once before returning to the original city [16]. This problem can be formulated as a undirected weighted graph problem: given a list of N nodes and weighted edges $d(u, v)$ for $u \in N$ and $v \in N$, find the shortest path that visits all the nodes only once before returning to the starting node. Note that in this project, the adjacency matrix is symmetric and dense[13].

More specifically, we will parse input data and save the distances between any two given locations in an n-by-n symmetric matrix $D = (d(u, v))$, where u, v are locations and $d(u, v)$ is the distance between them. Our goal is to arrange these locations in a cyclic order such that the sum of $d(u, v)$ between any two consecutive points are minimal[5]. Note that all the edge costs here are symmetric and satisfy the triangle inequality.

2.2 Data

The input data are all stored as *.tsp* file, whose format is as used in *TSPLIB*[11]. In this project, we have information of 14 different cities, each of which contains some $x - y$ coordinates and corresponding IDs, with data type of either Euclidean or geographical. After preprocessing inputs,

we will store the distances in a 2-dimensional matrix, where the indices represent for the vertex ID and the values represent for the corresponding distance between them.

2.2.1 Euclidean 2D Format. Let $u = (x_u, y_u)$ and $v = (x_v, y_v)$ denote any two locations, and let $d(u, v)$ denote the distance between them, then the Euclidean distance is calculated by $distance_x = x_u - x_v$, $distance_y = y_u - y_v$ and $distance(u, v) = \lfloor \sqrt{distance_x{}^2 + distance_y{}^2} \rceil$.

2.2.2 Geographical Format. Let $u = (x_u, y_u)$ and $v = (x_v, y_v)$ denote any two locations, where x and y represent for latitude and longitude of `DDD.MM` format, respectively. Then $d(u, v)$ is their distance on the idealized sphere with radius $RRR = 6378.388$ kilometers. Thus we need to convert these data points into radians and the Geographical distance is calculated as

$$distance(u, v) = \lfloor (RRR * \arccos(0.5 * ((1.0 + \cos(rad(y_u) - rad(y_v))) * \cos(rad(x_u) - rad(x_v)) \\ -(1.0 - \cos(rad(y_u) - rad(y_v))) * \cos(rad(x_u) + rad(x_v)))) + 1.0) \rfloor$$

2.3 Output

Two typical types of output data are all stored in folder named "output". First type is Solution file ending with ".sol", which saves the best quality and its corresponding tour. Second type is Trace file ending with ".trace", which contains the trace of when the better solution was found.

3 RELATED WORKS

Traveling salesman problem is a classic NP-hard problem in computer science field. Many works and solutions have been made and raised in the past years. Kruskal et al. provided a greedy method to find the minimum spanning tree in a graph, which can be applied to solve the traveling salesman problem[9]. S. Lin et al. proposed an effective heuristic algorithm based on general approach to heuristics, and solved the problem by joining the two end points with a small distance[10]. V Černý presented approximation solutions using Monte Carlo algorithm, obtaining results that were very close to the optimal ones. It generates the permutations of trip stations randomly, where the probabilities count on the length of the route[3]. George Dantzig et al. found a solution of large-scale TSP, showing that by a certain arrangement of 49 cities in the US, one can find the shortest road distance for one of each of the 48 states and Washington D.C.[5]. David E Goldberg et al. showed a genetic algorithm that led to a partially-mapped crossover operator. They formulated TSP as a n-permutation problem without allele values and obtained a "nearly" optimal solution[8]. Although in this study we mainly focus on sequential methods, there exist some distributed algorithms, for example, the ant colony system introduced by Marco Dorigo et al., which outperforms other natural algorithms significantly for both symmetric and asymmetric TSPs[6].

4 ALGORITHMS

We implemented four different algorithms that falls in three categories, exact algorithm, approximation algorithm, and local search algorithm. The pseudo-codes are listed in Appendix A.

4.1 Branch-and-Bound

4.1.1 Overview. The branch and bound is an exact algorithm for computing TSP. It works by enumerating a search tree and breaking up the possible solution set into smaller subsets (child node), calculating lower bounds for each subset and discarding (pruning) those having a lower bound larger than a upper bound. The algorithm consists of three components:

(1) Branching: We have to decide what constraints we impose on a child node.
(2) Bounding: We have to find a way to compute lower bound for a node.

(3) Selection of next node to explore: We have to determine in what order we traverse the tree. There are two common techniques: two-node expansion and multi-node expansion.

4.1.2 Implementation. In our implementation, the lower bound is calculated by performing matrix reduction: summation of minima of each row and each column. Though this value is not a valid solution, any valid solution cannot be smaller than this number, thus the lower bound. The multi-node expansion was chosen because a solution with good quality can be found faster than using two-node expansion. Furthermore, A priority queue was implemented so the subsets with the smallest lower bound can be explored and expanded first, leading to promising solution faster.

4.1.3 Optimization. Since branch and bound is an exact algorithm with exponential run time, for large data sets, it's not able to yield a good quality solution within a time frame of 10 minutes. In our implementation, we designed the priority queue to work with multi-node expansion, which has the advantage of finding the best solution faster, but has the disadvantage of taking up too much memory when the search tree goes deep enough. To tackle this problem, we designed the following optimization techniques,

(1) We designed a greedy algorithm for initializing upper bound. This tight upper bound helps prune more nodes down the search tree.
(2) We observed that most of the nodes would not generate promising results and most of them would end up getting pruned. For this reason, instead of having one giant priority queue to hold all nodes, we designed an array of priority queues to hold nodes at each level. Thus, the length of array is equal to the number of cities. In addition, we set a threshold k as the maximum number of nodes at any level the priority queue can contain. Once a priority queue exceeds the size limit, only the first k nodes will be kept and the rest will be pruned.
(3) If a node popped from priority queue has a lower bound larger than the upper bound, the priority queue will be cleared at that level to save memory, since the rest of the nodes are not going to yield promising results due to the property of min heap.

The pseudocode is outlined in Appendix A. Due to space limitation, the pseudocode for helper functions GreedyBound() and Reduce() are omitted but are described in 4.1.2.

4.1.4 Time and space complexities. In this exhaustive search algorithm, all candidate solutions are tested, so the run time has an exponential growth at worst case. The space complexity using Priority Queue is at worst case also exponential even with pruning and optimization techniques applied. In summary, the time complexity is $O(n!)$ and space complexity is $O(n!)$.

4.1.5 Strength and weakness. One major strength of this algorithm is that it is guaranteed to return optimal solution shall it complete. However, the obvious weakness is that it doesn't have friendly time and space complexities, almost guaranteed to result in out of memory and timeout errors for even medium data set where $|\text{Cities}| > 70$.

4.2 MST-Approximation

4.2.1 Overview. From lecture we learned that the output produced by MST-approx is never more than twice the optimal solution. First observation is that the total cost of a minimal spanning tree is no greater than the optimal cost, $cost(MST) \leq cost(OPT)$. Second observation is an Eulerian tour can be obtained through traversing through the MST and including each edge twice as they are visited from both directions. Thus we have the relation $cost(T) = 2 * cost(MST)$, where T is an Eulerian tour. Third observation is that traversing the MST in preorder and including each edge exactly once when they are first visited gives us a valid solution of TSP. Furthermore, we have

the relation $cost(TSP) \leq cost(T)$, by applying the triangle inequality,assuming the TSP problem is metric. Combining three results we have the relation $cost(OPT) \leq cost(TSP) \leq 2cost(OPT)$.

4.2.2 Implementation. The main function lies in *compute.py*. It takes in two inputs *input* and *d*, which are the cost matrix and its dimension. Its output is *tsp* and *c*, which are the the 2-approximate tour and its total cost. First step is to compute a MST of the graph. Second step is to perform DFS on the MST to obtain a pre-order traversal. The details of these steps are shown in the *MST* and *DFS* functions in Algorithm 4, respectively.

Our MST implementation is based on Prim's algorithm. The reason we chose it instead of Kruskal's is that there is an edge between each pair of nodes thus $m = n^2$, making it a dense graph. For the same reason, we chose normal list representation rather than binary heap or Fibonacci heap, making the time complexity of $O(n^2)$ at worst case. Also referencing from the lecture slide Graph-2, we know Prim's is a greedy algorithm, in that at each step it chooses the next smallest-weighted edge connected to the partially constructed spanning tree. The correctness lies in the Cut Property, which essentially states that the smallest-weigted edge of any cut of the graph must be in the minimal spanning tree.

Our DFS implementation is stack-based. Since the *tsp* obtained from MST is a list where the index represents one node and the value represents its parent. The root of the tree is also the parent of itself. For each node we choose, we have to search through the list to find its children to expand. Thus a potentially better implementation is to construct a tree first, reducing time complexity from $O(n^2)$ to $O(n \log n)$. The pseudocode is listed in Appendix A.

4.2.3 Time and space complexities. Time complexity is $O(n^2)$, and space complexity is $O(n^2)$.

4.2.4 Strength and weakness. One obvious strength is that it guarantees the solution quality, that is, the minimal cost is no larger than two times the optimal solution. Another strength is that the time and space complexity is efficient both in theory and in practice. Essentially, it is a polynomial solution to NP-hard problem. On the other hand, one weakness of approximation algorithm is that it cannot be improved in solution quality even with more time and/or space. Another weakness is that the algorithm itself is hard to improve upon. For example in the MST-approximation, no performance tuning of any parameter is possible.

4.3 LS1: Iterated Local Search

4.3.1 Overview. Two-opt is a simple local search algorithm for solving TSP. It iteratively chooses every pair of two nodes on the current route, examines the new cost after performing 2-exchange on such pair of nodes. If new cost is smaller, the exchange operation is performed. However, this method is prone to getting stuck in local minimum. We implemented iterated local search to tackle this problem.

Iterated local search is an approximation algorithm that perturbs the local minimum reached by two-opt algorithm and restarts the local search. The objective of perturbation is to move the solution out of the local minimum, to search in a more promising state, leading to a possible global minimum.

4.3.2 Implementation. We first randomly generate a valid solution, then perform the two-opt algorithm as follow:

(1) Initialize with a valid solution.
(2) At every iteration, pick two cities and examine the total cost of the route after performing an 2-exchange on them.

(3) If the new total cost is smaller than the former one, perform the 2-exchange and keep the new solution. This is equivalent to reversing the segment of route between these two cities.
(4) Keep iterating until no improvement can be made by an 2-exchange move or time runs out.

After two-opts algorithm reaches an local minimum, we perturb the solution using a double-bridge move (4-exchange move). This is a relatively strong perturbation that cannot be easily reverted by a single 2-exchange move.

(1) Initialize a probability p = 1 and decay rate 0 < d < 1.
(2) With probability $1-p$, terminate the iteration and return the current solution. With probability p, randomly select 3 cities on the route. Along with the starting point, these 4 cities break the entire route into 4 segments. Let these 4 segments annotated by A B C D. Reconnect these 4 segments in the order of A D C B. Re-run the two-opt algorithm using this new route as initial solution.
(3) Examine the cost of new local minimum returned by two-opt. If the new total cost is smaller than the former one, accept current route, set p = 1 and repeat from step 1. Otherwise, revert back to previous route, set p = p * d and repeat from step 1.

Note that iterated local search does not need a terminating condition. It can keep iterating and looking for better solution until time limit is reached. However, in this implementation, in order to obtain relatively good solutions in short amount of time, a decaying probability is applied to decide whether to make next perturbation if the previous one fails to provide a better solution for local search. The pseudocode is outlined in Appendix A.

4.3.3 Time and Space Complexities. The two-opt algorithm iterates through $\frac{n*(n-1)}{2}$ pair of nodes. For each pair of nodes, updating cost takes $O(1)$ time, and performing the 2-exchange takes $O(n)$ time. In the worst case (although extremely unlikely), two-opt algorithm performs 2-exchange for every pair of nodes, and its time complexity is $O(n^3)$.

Iterated local search can run two-opt algorithm indefinitely number of times until time runs out, so its time complexity is hard to analyze.

The space complexity of two-opt and iterated local search is $O(n)$, as all the swap operations are implemented in-place (If the input distance matrix of size $\Theta(n^2)$ is not counted).

4.3.4 Strength and weakness. Iterated local search perturbs local optimal solution using 4-exchange. This method keeps the route within each of 4 segments unchanged, thus it retains more information from last iteration compared to a random restart.

This specific implementation of iterated local search makes a trade-off between solution quality and run-time using a probability to abort. This is analogous to the "patience" of algorithm. If many consecutive attempts cannot provide a better solution, the algorithm loses its patience as the chance to abort without re-running two-opt algorithm again increases. The probability to abort is reset to zero once a better solution is found. In practice, this methods returns surprisingly accurate result within much shorter time (usually less than one minute) compared to branch and bound and simulated annealing we implemented.

However, this algorithm does not make use of the full period of time limit, and has higher chance than the simulated annealing to omit the global optimal solution within each run.

4.4 LS2: Simulated Annealing

4.4.1 Overview. Simulated annealing algorithm is an optimization technique that finds an approximation of the global minimum of a function. It was originally inspired from the process of annealing in metallurgy which involves two processes, heating and cooling, to alter metal's internal properties. The process starts at a high temperature, causing its internal structure to undergo constant changes.

As the metal cools down, its new structure becomes permanent, consequently causing the metal to retain its newly obtained properties.

4.4.2 Implementation. In our implementation, we keep a temperature variable to simulate the heating and cooling processes. We set it very high initially and allow it to gradually cool down over time. While the temperature is high, a worse solution is accepted at a higher frequency. This allows the algorithm to jump out of any local minimum it finds itself in. As the temperature drops, so is the probability of accepting worse solutions, thus allowing the algorithm to focus on exploring promising search space that could yield the global minimum. Applying the simulated annealing technique to TSP can be summarized as follow:

(1) Create an initial path randomly.
(2) At every iteration, randomly pick two cities and reverse the path between them, including them[4]. This forms a candidate solution.
(3) If the new distance is shorter than the former, it is kept as new solution.
(4) Otherwise, it is kept with a certain probability.
(5) We update the temperature in every iteration by slowly cooling down.

The initial temperature, cooling factor, cut off temperature threshold were chosen by running experiments on given data set and the best parameter set was chosen. They turned out to be: $Start\_T = 10^{30}, c = 0.999, End\_T = 1$.

4.4.3 Optimization. The following optimization techniques were implemented to yield the best quality solution:

(1) The new distance was calculated in $O(1)$ time instead of $O(n)$ by disconnecting and reconnecting the current solution.
(2) The current solution is reversed only when we are certain the new path is going to replace the current solution. This happens when either the new distance is shorter or the acceptance probability is larger than the normal distribution.
(3) Restart[7] was implemented to prevent from staying too long at non-promising states and local minimum. Given a time frame of 10 minutes, when the temperature drops below a certain threshold, the procedure will restart at the best solution found so far, and this solution will be updated once a better one is found. The process repeats until time runs out.
(4) The initial temperature will drop at the begining of every restart to increase randomness.

The pseudocode is listed in Appendix A.

4.4.4 Time and Space Complexities. Each cooling procedure has a time and space complexity of both O(n). Since the problem can run indefinitely number of times within time limit, the exact time complexity in practice is hard to come up with.

4.4.5 Strength and weakness. The strength of this algorithm is that it is apparent simulated annealing requires far less computation than an exact algorithm, for example, branch and bound. Often times, it is able to find the global optimum for smaller problem with far shorter running time. On the other hand, its effectiveness is extremely reliant on the choosing of parameters, such as initial temperature, cooling factor and cut off temperature. Finding a good set of parameters can be time consuming. A faster, adaptable way of finding those parameters from problem to problem can significantly increase the effectiveness of my algorithm.

5 EMPIRICAL EVALUATION

5.1 Platform

We ran our algorithms on a computer equipped with Intel Core i5 and 8GB RAM, using Python 2.7.

5.2 Experiments

5.2.1 Evaluation criteria. Four algorithms were tested on all the provided data sets. The best solution found within 10 minutes was returned. The evaluation criteria we used is the relative error which is calculated using RelErr = $\frac{\text{Sol.Qual}-\text{OPT}}{\text{OPT}}$, in which the Sol.Qual is the best solution returned, and OPT is the optimal solution found using online solver[2]. Two local search algorithms were tested 10 times with different seeds and the results were averged. Note that for comparing purpose, these 10 seeds are set to same for both algorithms.

5.2.2 Results. The comprehensive table 1 shows the results of all four algorithms.

Table 1. Comprehensive Table

	Branch-and-Bound			**MST-Approx.**			**Iterated Local Search**			**Simulated Annealing**		
Dataset	Time(s)	Sol.Qual.	RelErr	Time(s)	Sol.Qual.	RelErr	Time(s)	Sol.Qual.	RelErr	Time(s)	Sol.Qual.	RelErr
Atlanta	7.84	2003763	0	0.0008	2415132	0.2053	0.062	2007020.5	0.0016	3.11	2003763	0
Berlin	318.6	8276	0.0973	0.002	10303	0.3661	1.94	7627.8	0.0114	317.31	7579.8	0.005
Boston	113.15	898522	0.0056	0.002	1094649	0.2251	1.16	895200.4	0.0019	196.12	894963.5	0.0016
Champaign	29.10	58347	0.1084	0.003	61508	0.1684	4.01	52686.7	0.0008	317.29	52839.2	0.0037
Cincinnati	0.07	277952	0	0.0005	315452	0.1349	0.001	277952	0	0.61	277952	0
Denver	350.81	119479	0.1897	0.005	126189	0.2565	14.86	101734.2	0.0130	388.62	102973.4	0.0253
NYC	464.48	1624703	0.0448	0.004	1884293	0.2117	4.63	1566153	0.0071	301.35	1565061.3	0.0064
Philadelphia	11.93	1395981	0	0.002	1722655	0.2340	0.11	1395981	0	6.44	1395981	0
Roanoke	0.009	840996	0.2831	0.04	797872	0.2173	379.68	661448.7	0.0091	451.4	689384	0.0518
SanFrancisco	0.002	897275	0.1075	0.007	1099837	0.3575	9.92	814639.8	0.0055	402.71	825676.1	0.0191
Toronto	0.002	1386622	0.1789	0.008	1682030	0.4301	13.72	1176504	0.0003	388.77	1201621.7	0.0217
UKansasState	0.15	62962	0	0.0006	70143	0.1141	0.0008	62962	0	0.44	62962	0
ulysses16	21.43	6859	0	0.0007	7796	0.1366	0.015	6859	0	1.92	6859	0
UMissouri	0.002	164590	0.2402	0.01	153757	0.1586	19.70	134038.2	0.01	413.66	139389.2	0.0503

Figure 1 and 2 show the qualified runtime distributions of both local search algorithms of cities *UkandasState*, *ulysses16*, *Philadelphia* and *Roanoke*, with number of locations 10, 16, 30 and 230, respectively. For our iterated local search algorithm, we can see from figure 1 that even *UkandasState* and *ulysses16* contain similar number of cities, the former can find the optimal solution within a much shorter period time. Besides, as the size of the input graph increases and the value of p^* decreases, the time we take to find the best solution increases and the fraction of $P(solve)$ decreases.

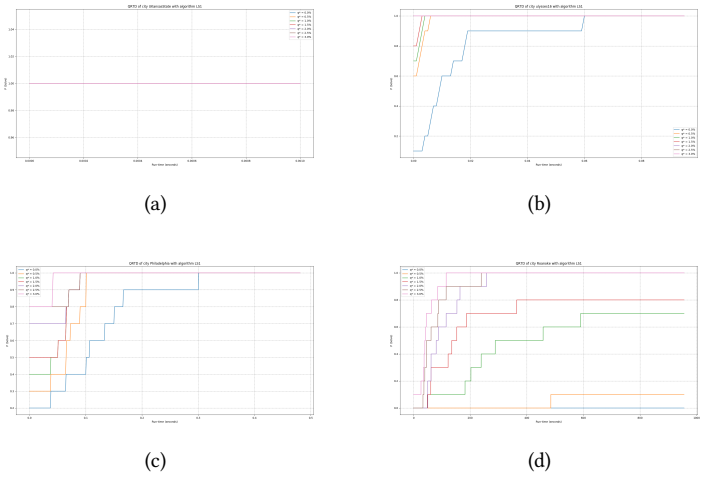
Figure 2 shows a similar trend of different values of p^* with the same four cities in figure 1. However, we can observe that, our simulated annealing algorithm does not solve the largest graph *Roanoke* within 3% relative solution quality and thus gave us a complete zero values of all p^*.

Solution quality distributions of both local search algorithms of the corresponding QRTDs are shown in figure 3 and 4. We can explicitly see that as the number of nodes and the relative solution quality increase, the fraction of solved problems also increases and the computing time decreases.

The box plots for running times of both local search algorithms are shown in figure 5, 6 and 7, which are generated by [1]. We chose to plot the running time for 10 different seeds to reach a value of 3% of q^* here, and the size of input graph increases from left to right. Since the time it takes for city *Roanoke* to find the best solution is much longer than all the others, it was plotted in figure 6 separately.

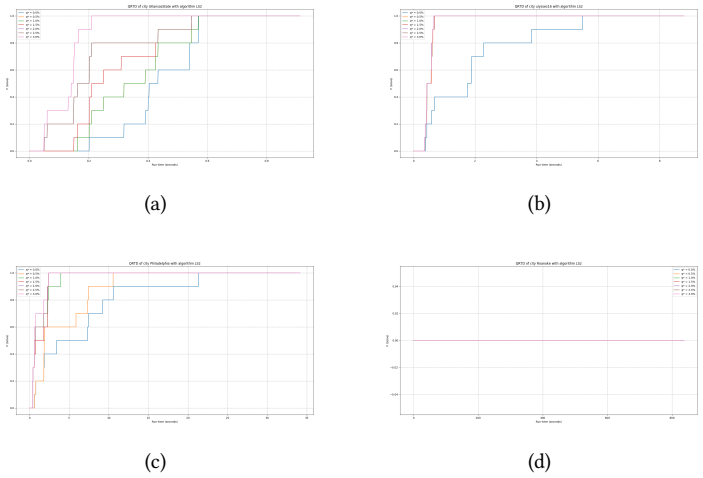
6 DISCUSSION

Branch and bound is an exact algorithm that is guaranteed to find the optimal solution for problems with size < 20. However, the memory cost explodes as the size of problems increases. The search tree can go deep enough without a depth limit. In theory, at level k, there are $(\text{size} - k)!$ nodes to be added to the queue. Each node has its associated reduced matrix and partial path. Thus, it's not a surprise that this algorithm can cause out of memory error easily on problem with size > 40.



(a) (b) (c) (d)

Fig. 1. Qualified runtime distributions of Iterated Local Search algorithm: (a) UKansasState; (b) ulysses16; (c) Philadelphia; and, (d) Roanoke.

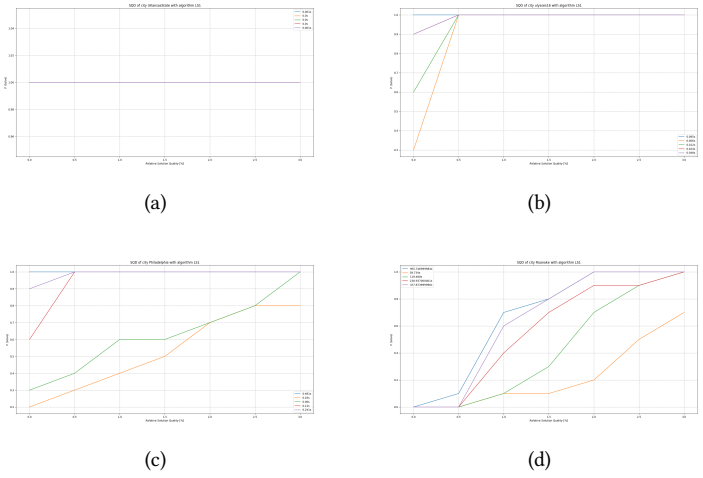


(a) (b) (c) (d)

Fig. 2. Qualified runtime distributions of Simulated Annealing algorithm: (a) UKansasState; (b) ulysses16; (c) Philadelphia; and, (d) Roanoke.

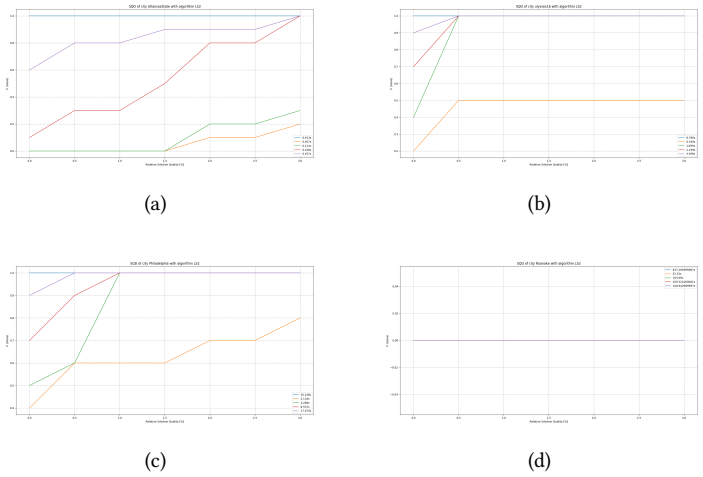
For MST heuristic method, we finished computation in a very short period of time other than the Branch-and-Bound method, and obtained results that give guarantee of the solution quality.

For two local search algorithms, iterated local search and simulated annealing can generate solutions that are extremely close to the optimal ones. However, the iterated local search can find the best solution faster, since it aborts and returns best result found so far with increasing probability after consecutive failed restart attempts. Simulated annealing takes longer to yield the



(a) (b) (c) (d)

Fig. 3. Solution quality distributions of Iterated Local Search algorithm: (a) UKansasState; (b) ulysses16; (c) Philadelphia; and, (d) Roanoke.



(a) (b) (c) (d)

Fig. 4. Solution quality distributions of Simulated Annealing algorithm: (a) UKansasState; (b) ulysses16; (c) Philadelphia; and, (d) Roanoke.

best solution because the parameters, i.e, initial temperature, cooling factor, etc, are critical in its performance, and the best parameter set usually varies from problem to problem.

7 CONCLUSIONS

In this study, four algorithms were implemented in an attempt to solve TSP. Even though TSP falls under the category of NP-complete problems, this study shows that there exist efficient approximation algorithms to generate a solution close enough to the optimal one. The real question

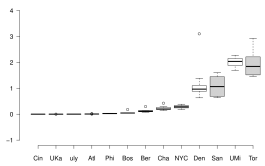


Fig. 5. Box plots for running times of Iterated local search

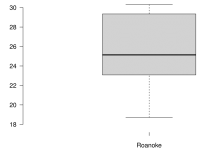


Fig. 6. Box plots for running times of Iterated local search of city Roanoke

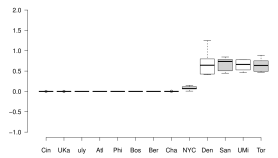


Fig. 7. Box plots for running times of Simulated Annealing

is: is it really worth the effort to come up with an optimal solution, or is it sufficient to come up with an efficient approximation solution that is close to the optimal one? This time-accuracy trade-off exists in many theoretical computer science problems and gives rise to the field of optimization. From a real-world perspective, the exact algorithm should be used for problems with small sizes, while the approximation algorithm is recommended for other problems. The future work may build on the findings of this study for the more efficient algorithms.

REFERENCES

[1] [n. d.]. BoxPlotR: a web-tool for generation of box plots. ([n. d.]). http://shiny.chemgrid.org/boxplotr/
[2] [n. d.]. concorde. ([n. d.]). https://neos-server.org/neos/solvers/co:concorde/TSP.html
[3] V. Černý. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications* 45, 1 (01 Jan 1985), 41–51. https://doi.org/10.1007/BF00940812
[4] J. Chang. 2006. More on Markov chains, Examples and Applications. In *Stochastic Processes*. Chapter 2, 57–65.
[5] George Dantzig, Ray Fulkerson, and Selmer Johnson. 1954. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America* 2, 4 (1954), 393–410.
[6] Marco Dorigo and Luca Maria Gambardella. 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation* 1, 1 (1997), 53–66.
[7] R. Shonkwiler F. Mendivil and M. C. Spruill. 2000. RESTARTING SEARCH ALGORITHMS WITH APPLICATIONS TO SIMULATED ANNEALING. *Applied Probability Trust* 1, 1 (2000), 10–22.
[8] David E Goldberg, Robert Lingle, et al. 1985. Alleles, loci, and the traveling salesman problem. In *Proceedings of an international conference on genetic algorithms and their applications*, Vol. 154. Lawrence Erlbaum, Hillsdale, NJ, 154–159.
[9] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.
[10] Shen Lin and Brian W Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations research* 21, 2 (1973), 498–516.
[11] Gerhard Reinelt. 1991. TSPLIB - A traveling salesman problem library. *ORSA journal on computing* 3, 4 (1991), 376–384.
[12] Tim Roughgarden. [n. d.]. Lecture 16: The Traveling Salesman Problem. ([n. d.]). https://theory.stanford.edu/~tim/w16/l/l16.pdf
[13] Wikipedia. [n. d.]. Wikipedia, Complete graph. ([n. d.]). https://en.wikipedia.org/wiki/Complete_graph
[14] Wikipedia. [n. d.]. Wikipedia, NP-hardness. ([n. d.]). https://en.wikipedia.org/wiki/NP-hardness
[15] Wikipedia. [n. d.]. Wikipedia, Travelling salesman problem. ([n. d.]). https://simple.wikipedia.org/wiki/Travelling_salesman_problem
[16] Wikipedia. [n. d.]. Wikipedia, Travelling salesman problem. ([n. d.]). https://en.wikipedia.org/wiki/Travelling_salesman_problem

A APPENDIX PSEUDO-CODE

A.1 Branch-and-Bound

ALGORITHM 1: Branch-and-Bound

```
Input: distance matrix (CostMatrix), time limit
Output: The best solution found within time limit
UpperBound ← GreedyBound(), Solution ← [];
Preprocess CostMatrix to make diagonal elements ∞
ReducedMatrix, Cost ← Reduce(CostMatrix)
Path ← [0]; Visited ← set(0)
Frontiers ← array of Priority Queues, with a length equal to number of cities
RootNode ← Node(ReducedMatrix, cost, Path, Visited)
Frontiers[0].append(RootNode);
CurrentLevel ← 0;
Duration ← 0;
repeat
    if Frontiers[CurrentLevel].size > k then
        Keep the first k nodes in Frontiers[CurrentLevel];
    end
    TopNode ← Frontiers[CurrentLevel].pop()
    CostSoFar ← TopNode.getCost()
    if CostSoFar > UpperBound then
        Frontiers[CurrentLevel].clear();
    end
    if CostSoFar < UpperBound then
        Get Path, CurrentIndex, ReducedMatrix, VisitedSet from TopNode;
        Neighbors ← Indices currently not in Visited;
        if ReducedMatrix contains only ∞ then
            if CostSoFar < UpperBond then
                Solution ← Path;
                UpperBound ← CostSoFar;
            end
        end
        CurrentLevel++;
        for NextIndex in Neighbors do
            Set ReducedMatrix[CurrentIndex, :] and ReducedMatrix[:, NextIndex] to ∞;
            Set ReducedMatrix[NextIndex, 0] to ∞;
            Cost, NewReducedMatrix ← Reduce(ReducedMatrix);
            NewLowerBound ← CostSoFar + Cost + ReducedMatrix[CurrentIndex, NextIndex];
            if NewLowerBound < UpperBound then
                Path.append(NextIndex);
                Visited.append(NextIndex);
                ChildNode ← Node(NewReduce, NewCost, Path, Visited);
                Frontiers[CurrentLevel].append(ChildNode);
            end
        end
    end
    Update Duration;
until Frontiers.isEmpty() or Duration > TimeLimit;
```

A.2 Approximation

ALGORITHM 2: MST-Approximation

```
Function compute(d, input, root=0): (c, tsp)
    mst = MST(d, input, root)
    tsp = DFS(d, mst, root)
    c = 0
    for i = 0 to d-1 do
        c += input[tsp[i]][tsp[i + 1]]
    end
end
Function MST(d, input, root) : (mst)
    for i = 0 to d-1 do
        mst[i] = i ; a[i] = Inf ; visited[i] = 0
    end
    visited[root] = 1
    while True do
        s = sum(visited)
        if s == d then break
        u = argmin(a)
        if s == 0 then  u = root
        visited[u] = 1
        for v = 0 to d-1 do
            if not visited[v] and input[u][v] < a[v] then
                a[v] = input[u][v]
                mst[v] = u
            end
        end
    end
end
Function DFS(d, mst, root): (tsp)
    tsp = []
    for i = 0 to d-1 do  visited[i] = 0
    stack.push(root)
    visited[root] = 1
    while not stack.empty() do
        cur = stack.pop()
        tsp.append(cur)
        for i = 0 to d-1 do
            if not visited[i] and mst[i] == cur then
                stack.push(i)
                visited[i] = 1
            end
        end
    end
    tsp.append(root)
end
```

A.3 LS1

ALGORITHM 3: Iterated Local Search

Input: distance matrix (CostMatrix), probability decay rate (decay), time limit, seed
Output: The best solution found within time limit or algorithm terminates with probability

```
Set random seed;
Solution ← initial random tour;
Duration ← 0;
p ← 1;
bestpath, bestcost, duration ← TwoOpt(solution);
InitialRoute ← RandomRoute();
Route, Cost ← TwoOpt();
BestRoute, BestCost ← Route, Cost;
Duration ← 0;
repeat
    if Random() > prob then
        return BestRoute, BestCost;
    end
    Route ← perturbate(Route);
    Route, Cost ← TwoOpt(Route);
    if Cost < BestCost then
        BestRoute, BestCost = Route, Cost;
        prob ← 1;
    end
    if Cost > BestCost then
        Route, Cost = BestRoute, BestCost;
        prob ← prob * decay;
    end
    Update Duration;
until Duration > TimeLimit;
```

A.4 LS2

ALGORITHM 4: Simulated Annealing

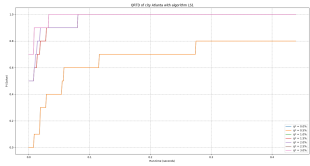
Input: distance matrix (CostMatrix), initial temperature (Start_T), cooling factor (c), cut off temperature (End_T), time limit, seed
Output: The best solution found within time limit

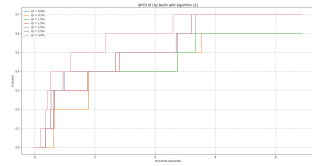
```
Set random seed;
Solution ← initial random tour;
Restart ← Solution;
T_decrease_factor ← 1;
Duration ← 0;
repeat
    Start_T ← Start_T * T_decrease_factor;
    T ← Start_T;
    repeat
        a, b ← randomly pick two indices;
        NewDistance ← UpdateDistance(Solution, a, b);
        if NewDistance < Solution then
            Solution ← ReversePath(Solution, a, b, CostMatrix);
            if Solution < Restart then
                Restart ← Solution;
            end
        end
        if NewDistance > Solution then
            Diff ← CurrentDistance - Solution;
            AcceptProb ← e^(Diff/T);
            if AcceptProb > UniformDistribtion(0,1) then
                Solution ← ReversePath(Solution, a, b, CostMatrix);
            end
        end
    until T < End_T;
    Solution ← Restart;
    T_decrease_factor *= 0.9;
    Update Duration;
until Duration > TimeLimit;
```

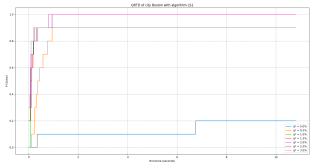
B APPENDIX PLOTS

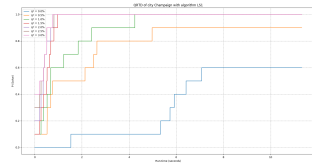
B.1 QRTDs

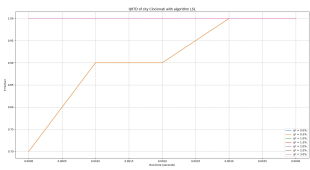
B.1.1 LS1.

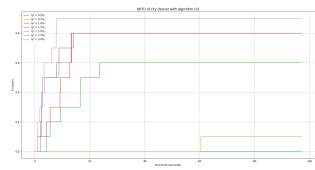


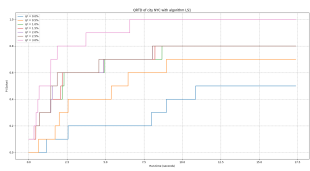


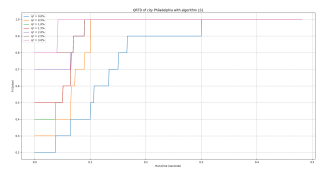


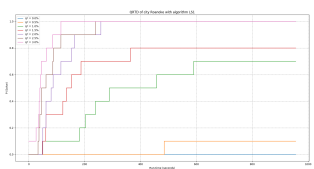


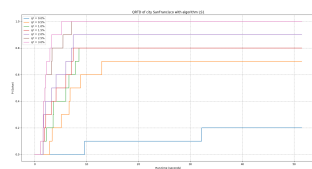


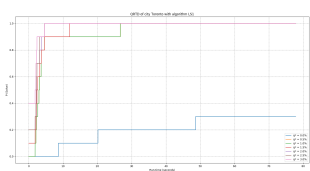


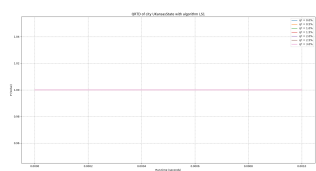


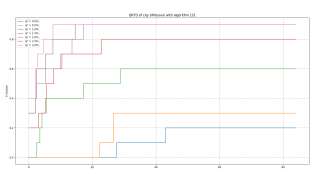


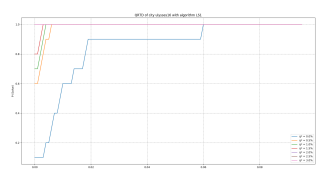




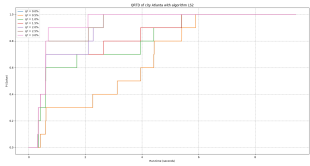


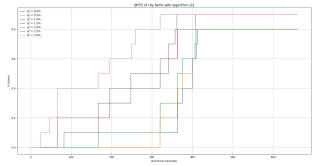


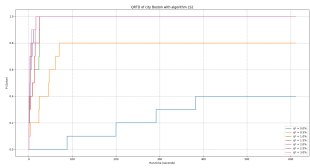


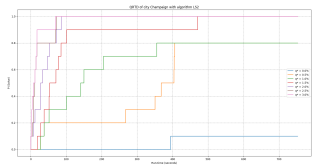


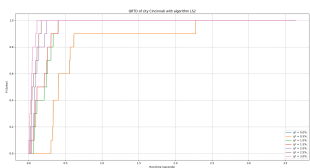
B.1.2 LS2.

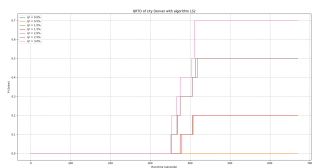

QRTO of city Atlanta with algorithm LS2
Fvalue
Runtime (seconds)

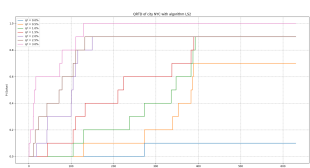

QRTO of city Berlin with algorithm LS2
Fvalue
Runtime (seconds)

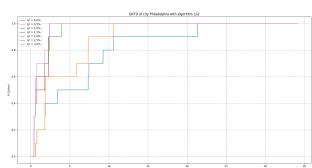

QRTO of city Boston with algorithm LS2
Fvalue
Runtime (seconds)


QRTO of city Champaign with algorithm LS2
Fvalue
Runtime (seconds)

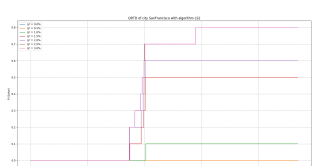

QRTO of city Cincinnati with algorithm LS2
Fvalue
Runtime (seconds)

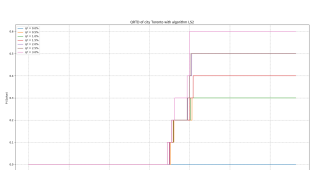

QRTO of city Denver with algorithm LS2
Fvalue
Runtime (seconds)

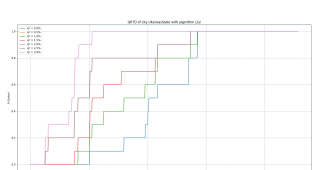

QRTO of city NYC with algorithm LS2
Fvalue
Runtime (seconds)


QRTO of city Philadelphia with algorithm LS2
Fvalue
Runtime (seconds)

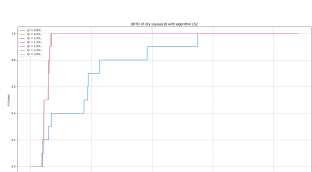

QRTO of city Roanoke with algorithm LS2
Fvalue
Runtime (seconds)


QRTO of city SanFrancisco with algorithm LS2
Fvalue
Runtime (seconds)


QRTO of city Toronto with algorithm LS2
Fvalue
Runtime (seconds)

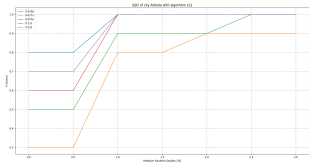

QRTO of city Urbanadelphia with algorithm LS2
Fvalue
Runtime (seconds)

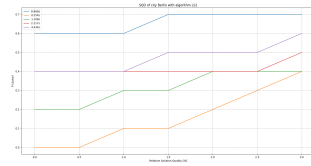

QRTO of city Urbanaut with algorithm LS2
Fvalue
Runtime (seconds)

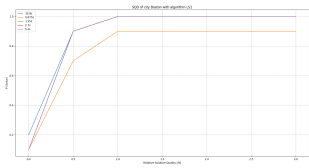

QRTO of city Urbana1K with algorithm LS2
Fvalue
Runtime (seconds)

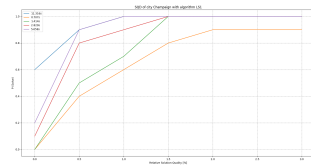
B.2 SQD

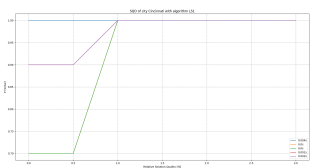
B.2.1 LS1.

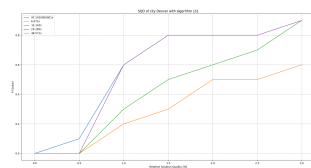


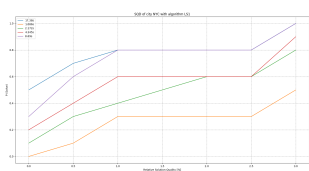


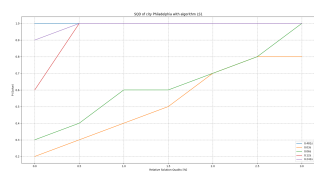


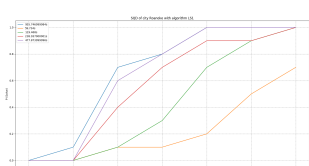


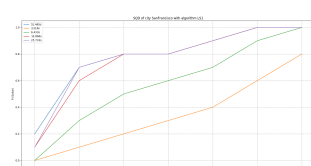


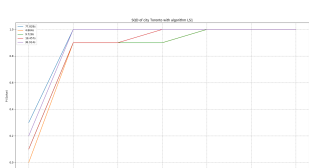


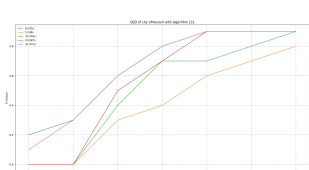


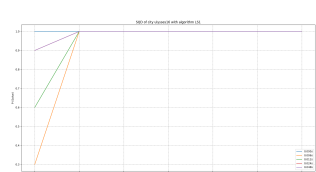




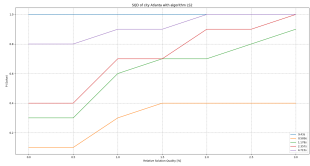


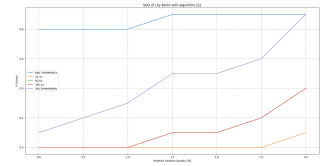


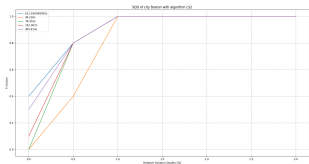


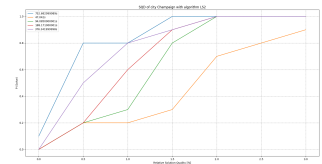


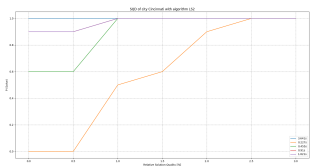
B.2.2 LS2.

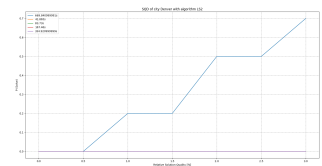

SGD of city Atlanta with algorithm LS2
Fvalue
Instance Solution Quality (%)

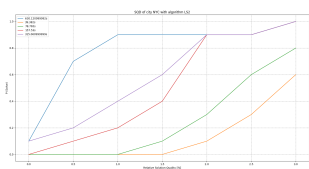

SGD of city Berlin with algorithm LS2
Fvalue
Instance Solution Quality (%)

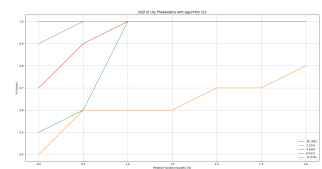

SGD of city Boston with algorithm LS2
Fvalue
Instance Solution Quality (%)

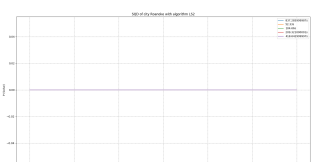

SGD of city Champaign with algorithm LS2
Fvalue
Instance Solution Quality (%)

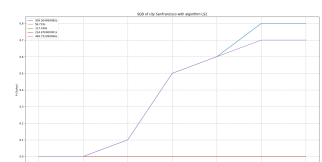

SGD of city Cincinnati with algorithm LS2
Fvalue
Instance Solution Quality (%)

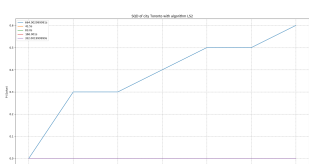

SGD of city Denver with algorithm LS2
Fvalue
Instance Solution Quality (%)

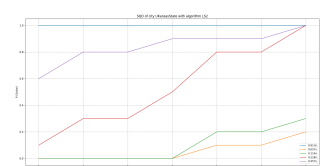

SGD of city NYC with algorithm LS2
Fvalue
Instance Solution Quality (%)

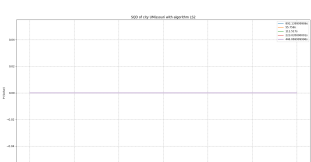

SGD of city Philadelphia with algorithm LS2
Fvalue
Instance Solution Quality (%)

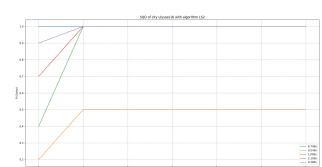

SGD of city Roanoke with algorithm LS2
Fvalue
Instance Solution Quality (%)


SGD of city SanFrancisco with algorithm LS2
Fvalue
Instance Solution Quality (%)


SGD of city Toronto with algorithm LS2
Fvalue
Instance Solution Quality (%)


SGD of city UrbanaChabm with algorithm LS2
Fvalue
Instance Solution Quality (%)


SGD of city UMAssunit with algorithm LS2
Fvalue
Instance Solution Quality (%)


SGD of city UpperLitt with algorithm LS2
Fvalue
Instance Solution Quality (%)