Travelling Salesman Problem: Various Algorithm And Their Evaluation

Goyal, Naman
Georgia Institute of technology
North Avenue NW
Atlanta, Georgia
ngoyal32@gatech.edu

Kumar, Mrinal
Georgia Institute of technology
North Avenue NW
Atlanta, Georgia
mkumar73@gatech.edu

Song, Yang
Georgia Institute of technology
North Avenue NW
Atlanta, Georgia
songyang@gatech.edu

ABSTRACT

In this paper, we describe various techniques to solve Travelling Salesman Problem. We also evaluate and compare those techniques in terms of various metrics such as running time, complexity, ability to reach global optimum and worse case analysis.

Keywords

TSP, MST, Branch and Bound, Local search, Greedy, Approximation.

1. Introduction

The goal of this project is to implement various algorithms for traveling salesman problem (TSP) and compare and evaluate their respective computational efficiency. The algorithms presented in this paper can be classified into three categories: exact solution, approximation algorithms, and local search algorithms. We have implemented Branch and bound, greedy heuristic with furthest insertion, MST 2-approximation, stimulated annealing and genetic algorithm for solving TSP. The dataset consists of cities varying from 14 to 202 taken from tsplib dataset.

2. Problem definition

The Traveling Salesman Problem (TSP) is a well-known and important combinatorial optimization problem. The goal is to find the shortest tour that visits each city in a given list exactly once and then returns to the starting city. Although the objective seems quite simple but finding the optimal solution is a difficult task since it belongs to the class of NP-complete problems. TSP has quite wide range of application in real world that include route planning, job scheduling, circuit design, Laser cutting etc.

3. Related Work

As we said before, the traveling salesman problem is NP-hard so there is no known algorithm that will solve it in polynomial time. We will probably have to sacrifice optimality in order to get a good answer in a shorter time. Here in this section we discuss few algorithm that have been tried for the traveling salesman problem
Greedy Algorithms are a method of finding a feasible solution to the traveling salesman problem. The algorithm creates a list of all edges in the graph and then orders them from smallest cost to largest cost. It then chooses the edges with smallest cost first, providing they do not create a cycle. The greedy algorithm gives feasible solutions however they are not always good.

The Nearest Neighbor algorithm is similar to the greedy algorithm in its simple approach. We arbitrarily choose a starting city and then travel to the city closest to it that does not create cycle. We continue to do this until all cities are in the tour. This algorithm also does not always give good solutions because often the last edge added to the tour (that is, the edge en1 where n is the number of cities) can be quite large.

A minimum spanning tree is a set of $n - 1$ edges (where again n is the number of cities) that connect all cities so that the sum of all the edges used is minimized. Once we have found a minimum spanning tree for our graph we can create a tour by treating the edges in our spanning tree as bidirectional edges.

We then start from a city that is only connected to one other city (this is known as a 'leaf' city) and continue following untraversed edges to new cities. If there is no untraversed edge we go back along the previous edge. We continue to do this until we return to the starting city. This will give us an upper bound for the optimal traveling salesman tour. Note, however, that we will visit some cities more than once. We are able to fix this if whenever we need to traverse back to a city we have already been to, we instead go to the next unvisited city. When all cities have been visited we go directly back to the starting city.

4. Algorithms

4.1 Farthest Insertion Greedy Heuristic:

Greedy algorithm strategy is the simplest strategy where the idea is to make a greedy choice at every step in such a manner that it tries to attain global optimum. There are several greedy strategies for Travelling Salesman Problem which aims to find a path that covers all nodes (cities) and has minimum travel cost. Farthest insertion greedy approach selects a node at maximum distance from set of visited node and then the new node is inserted into the traversed path in such a manner so that the cost of tour is minimal.

Pseudo-Code
Start with any random node in the list.

Then for each node that is not in the visited list we perform following step:

Selection step: In this step algorithm finds a node n_r such that is at a maximum distance from any of the visited node i.e. cost C_{ir} is maximum where n_i is a node is visited set of node.

Insertion step: After we select the farthest node n_r we insert the new node in the currently explored path where inserting a node n_r between n_i and n_j will lead to an insertion cost which is equal to:

$$cost\_new = old\_optimal + [cost(n_i \rightarrow n_r) + cost(n_i \rightarrow n_r) - cost(n_i \rightarrow n_j)]$$

Note: This is the greedy step where the aim of algorithm is to insert the selected node n_r in such a manner that minimizes "*cost_new*".
cost_new is the total cost of the tour calculated by the farthest insertion approach.

Time and Space Complexity:

In selection step, we use the binary max-heap. The total time for selection step is O(|E| log|V|), which is equivalent to O(n^2logn), n denotes the number of the cities, because it is a complete graph. The space for selection step is O(n) cause we use heap to store every node in. Other than that we need to maintain list of traversed path which is the total number of node covered in the tour which takes O(n) space. So, total space complexity of Farthest insertion becomes O(|n| + |n|) i.e. O(n).

Worse case behavior:

$$\frac{length\_of\_farthest\_insertion\_cost}{length\_of\_optimal\_tour} \leq 2\ln(n) + 0.16$$

4.2 MST-APPROX

We are given the x-y coordinates of N points in the plane (i.e. vertices), and a cost function c(u,v) defined for every pair of points (i.e. edge), find the shortest simple cycle that visits all N points. The cost function c(u,v) is defined as the Euclidean or Geo distance between points u and v. Thus the cost function c naturally satisfies the triangle inequality if, for all vertices u,v,w ∈ V ,

$$c(u,w) \leq c(u,v) + c(v,w)$$

In MST-APPROX, we first compute a minimum spanning tree, whose weight gives a lover bound on the length of an optimal traveling-salesman tour. Then we use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight because the problem satisfies the triangle inequality. The way we create a tour is that we recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children. It is a pre-order tree walk. Naturally, we use depth first search to finish this approach. The pseudocode of this process is as followings:

Pseudo-Code:

```
MST-APPROX(G):
        T = MST-Prim(G)
        Choose a vertex as root r
        return preorderTreeWalk(T, r)
```

In prim algorithm, we use the binary min-heap to accelerate it. The total time for Prim's algorithm is O(|E|log|V |), which is equivalent to O(n^2logn), n denotes the number of the cities, because it is a complete graph. The space for Prim's algorithm need is O(n) cause we use heap to store every nodes in it at first. As for pre-order tree walk, we use depth first search to finish this process. Because we use the adjacency list like data structure to store the graph, the time for dfs is O(|E| + |V|), which is equivalent to O(n^2). The space for dfs algorithm need is O(n^2). So the overall time complexity is O(n^2logn); The overall space complexity is O(n^2).

Worse case behavior:

We can prove that MST-APPROX is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality. Let H^* denote an optimal tour for the given set of vertices. We obtain a spanning tree by deleting any edge from a tour, and each edge cost is nonnegative. Therefore, the weight of the MST T provides a lower bound on the cost of an optimal tour:

$$c(T) \leq c(H*)$$

A full walk of T, we call it W, lists the vertices when they are first visited and also whenever they are returned to after a visit to a sub-tree. Since the full walk traverse every edge of T exactly twice, we have

$$c(W) = 2c(T)$$

Thus, we can get that:

$$c(W) \leq 2c(H^*)$$

So the cost of W is within a factor of 2 of the cost of an optimal tour. By the triangle inequality, we can delete a visit to any vertex from W and the cost does not increase. By repeatedly applying this operation,

we can remove from W all but the first visit to each vertex, which is the pre-order walk. Since H is obtained by deleting vertices from the full walk W, we are here to have

$$c(H) \leq 2c(H^*)$$

, which means MST-APPROX is a 2-approximation algorithm.

4.3 Branch and bound

Branch and Bound strategy is an exact solution algorithm for traveling salesman problem. It tries to reduce the exponential search space by finding the lower bound of the partial solution and pruning the partial solutions which have lower bound lesser than already found best solution.

Pseudo-Code:

```
def BranchAndBound(Graph):
1)  Add start node to the priority queue with lower bound.
2)  While(priority queue not empty)
    a.  current ← Queue pop item
    b.  For all neighbors of current that are not covered in partial
        solution.
        i.   Add neighbor in covered nodes
        ii.  check if this is a valid solution
             1.  if cost of this solution < best solution
                 a. best solution=cost of this solution
        iii. if not dead end
             1.  if lower bound of this solution < best Solution
                 a. queue.addItem(partialSolution,
                    lowerBound)
3)  return bestSolution
```

```
def lowerBound(Graph,tour):
1) lowerBound=0
2) For node not in partialTour
    a. min1,min2 ← For adjacent edge of node find minimum two edges
    b. lowerBound+=(min1+min2)/2
3) minOutgoing, minIncoming ← Find minimum edge of a & b in partialTour start and end nodes.
4) return lowerBound+ minOutgoing + minIncoming + cost(partailTour)
```

Time and Space Complexity:

The time complexity of Branch and Bound is exponential since it has to search all the solution space of the problem. The solution space of traveling salesman problem is n!. Although branch and bound tries to reduce the solution space by pruning the nodes which can never give better than already found solution but still in the worst case branch and bound has to traverse whole solution space. So the time complexity is O(n!).

We are using Priority Queue for our best first solution. The queue holds all the partial solution which are yet to be extended. Similar to above argument in the worst case it can require O(n!) space.

4.4 Simulated Annealing

Simulated annealing (SA) is an optimization technique that finds an approximation of the global minimum of a function.

Applying the simulated annealing to the TSP can be described as follow:

First, we create the initial list of cities by shuffling the input list (i.e.: make the order of visit random).

Second, we iteratively move from current position to one of neighboring positions. We randomly choose two edges. We use 2-opt algorithm to search neighborhood. The main idea behind it is to take a route that crosses over itself and reorder it. The 2-opt algorithm basically removes two edges from the tour, and reconnects the two paths created, which I referred to as a 2-opt move, which is shown in Figure 1. The cost value is the distance traveled by the salesman for the whole tour.

Then, if the new distance, computed after the change, is shorter than the current distance, it is kept. However, if the new distance is longer than the current one, it is kept with a certain probability. The neighborhood size of 2-opt runs in O(n2), which involves selecting an edge (u1, u2) and another one edge (u3, u4), making a 2-opt move and completing the cost.

Finally, we can update the temperature at every iteration by slowly cooling down. The probability of the current one maintains when the new distance is longer than the current one depends on the cost of doing so, and the current "temperature" T. Specifically, the method mimics the Boltzmann distribution: the probability of accepting a perturbation is exp(cost/T). In other words, more costly exploration can occur at high temperatures. Specifically, the cost can be defined as:

Cost = (dist(u1, u2) + dist(u3,u4)) – (dist(u1, u3) + dist(u2, u4))

The annealing schedule specifies how T changes over time -- this is generally decreasing. The T cools down using the following:

T = T * r

The "temperature" T gradually cools down to a certain level of "temperature" named T_min. At one iteration, if the "temperature" becomes T_min, we can find that we get the local minimum solution. At this time, in order to find the global minimum and reduce the error rate further, we increase the temperature again:

T = T_initial * 0.1

The iteration terminates when code runs more than the given cut off time. To run the program properly, we need to initialize T, define the cooling rate r and set the T_min. The T must be related to the size of the graph and the weights of the graph, so it can scale well from small instances of graphs to larger ones. So heuristically, the initialization of T is defined as following:

T = dist(ui, uj) * |U|

Edge (ui, uj) is randomly choose from the graph.

We also try to vary the r. When r reduces, the temperature T cools down more quickly and the program runs more quickly. However, the error rate is increasing. So this is a trade-off of optimality of solution vs the time taken to reach ths optimal solution. We need to choice a proper r to satisfy different needs.

We set T = dist(ui, uj) * |U|, T_min = 0.0001, r = 0.9999 (initial). And our algorithm attains a high level of accuracy.

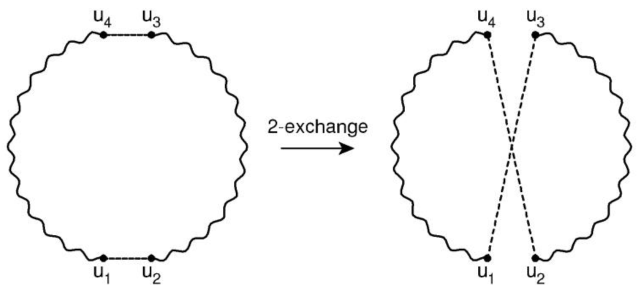


Figure 1, 2-opt move

4.5 Hill Climbing

The Hill Climbing algorithm is great for finding local optima and works by changing a small part of the current state to get a better (in this case, shorter) path. The hill climbing algorithm works as following:

First, we create the initial list of cities by shuffling the input list (i.e.: make the order of visit random) and pick a place to start.

Then like simulated annealing, we iteratively move from current position to one of neighboring positions. We first choose two edges, using 2-opt algorithm to search neighborhood. Then, if the new distance, computed after the change, is shorter than the current distance, it is kept. However, if the new distance is longer than the current one, it is discarded.

In order to comprehensively find all the local solutions without increasing the running time, if we don't find any better 2-opt solution we move to the 3 opt neighborhood search. 3-opt analysis involves deleting 3 connections (or edges) in a network (or tour), reconnecting the network in all other possible ways, and then evaluating each reconnection method to find the optimum one. This process is then repeated for a different set of 3 connections. The 3-opt is shown in Figure 2.

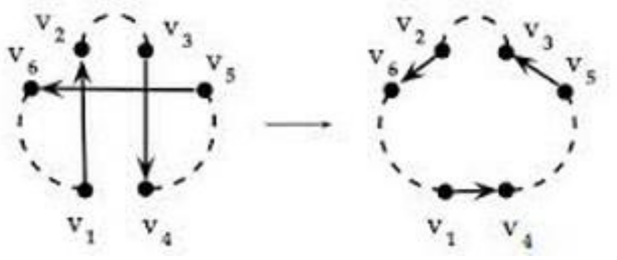


Figure 2, 3-opt move

And when we find out one of the better 3 opt neighbor, we switch back to 2 opt neighborhood search. This saves us from exhaustingly searching over all O(n^3) 3 opt neighborhood and still provides better solutions than 2 opt.

We check if solutions do not change for more than 3 times even after 3 opt search then the algorithm may have reached the partial sub-hill, which is nearly a local optimal or may have come to the global minima. At this time, we randomly restart hill climbing using double bridge move for random perturbation.

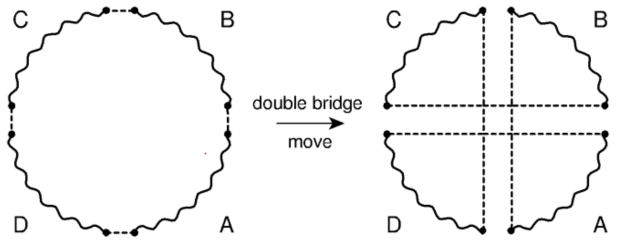


Figure 3, double-bridge move

At each iterations, we takes any step that decreases tour distance, otherwise carry on finding neighbors. The algorithm terminates when code runs more than the given cut off time limit. Our program attains a lower level of error rate which is described in further section.

5. Evaluation

5.1 Comprehension Table

	Branch and Bound			Greedy Heuristic			MST Approximation			Simulated Annealing			Hill Climbing		
Dataset	Time (Sec)	Path Length	Rel Err	Time (Sec)	Path Length	Rel Err	Time (Sec)	Path Length	Rel Err	Time** (Sec)	Path Length	Rel Err	Time** (Sec)	Path Length	Rel Err
burma14.tsp	6.301	3323	0.00	0.0019	3323	0.0000	0.0040	4162	0.2525	300	3328	0.0015	300	3323	0.0000
ulysses16.tsp	1015.37	6859	0.00	0.0019	7023	0.0239	0.0040	7583	0.1056	300	6860	0.0000	300	6859	0.0000
berlin52.tsp	-	-	-	0.0069	8307	0.1014	0.0120	10205	0.3532	300	7542	0.0000	300	8015	0.0627
kroA100.tsp	-	-	-	0.0012	23186	0.0895	0.0200	29996	0.4095	600	21581	0.0140	600	22388	0.0520
ch150.tsp	-	-	-	0.0026	7081	0.0847	0.0359	8629	0.3218	600	6862	0.0512	600	7028	0.0765
gr202.tsp	-	-	-	0.0046	45211	0.1258	0.0640	53428	0.3304	600	42550	0.0595	600	42875	0.0676

The table shows the relative performance of various algorithms. It is clearly visible that greedy heuristic error margin keeps on increasing with the size of graph but it outperforms MST 2 for lower instances.

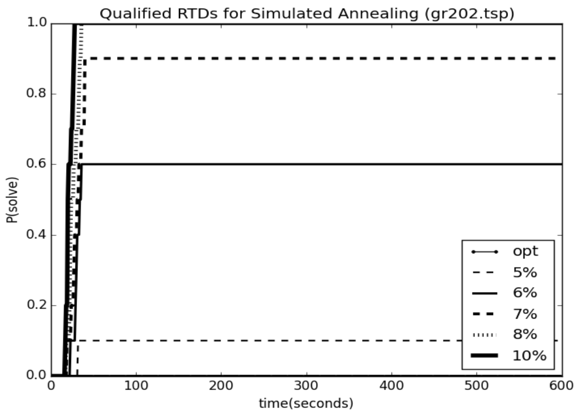
Branch and Bound could find solution only for the two smallest instances because of the exhaustive search space of solution.

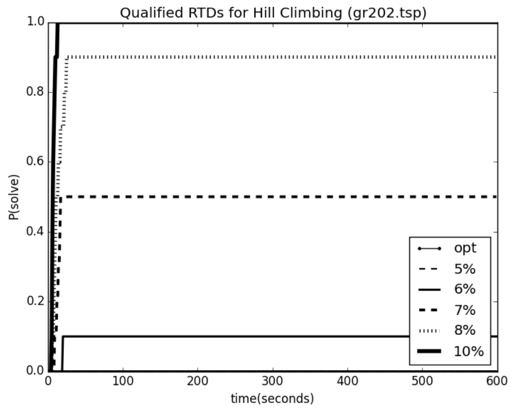
5.2 Qualified Runtime for various solution qualities (QRTD)

Below is the QRTD comparison of both the local search approaches for the largest instance (gr202.tsp):

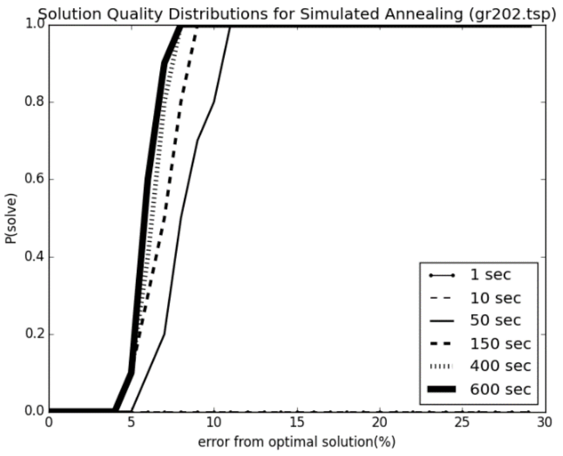
The QRTD comparison of Simulated Annealing vs Hill Climbing shows the tradeoff between these two local search approaches.

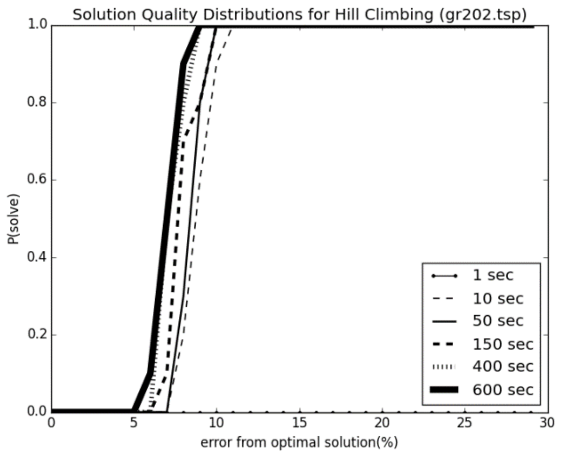
Simulated Annealing finds solution which are quite close to optimal with a very high probability as compared to Hill Climbing but the time taken by Simulated Annealing to reach to a solution is higher than hill climbing approach. The hill climbing approach reaches to the local minima solution faster but it is not able to move towards global minima.The QRTD plots for other graph instances are added in the appendix.





5.3 Solution Quality Distributions for various run-times (SQDs):

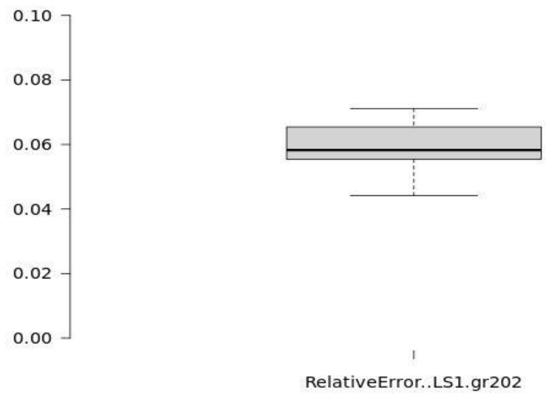


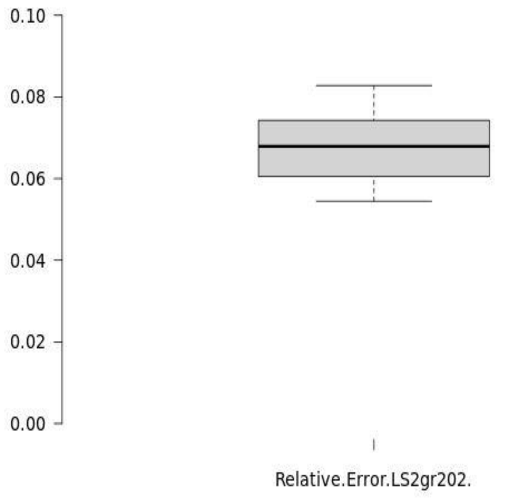


The SQD comparison of simulated annealing and hill climbing shows again that simulated annealing finds the solution closer to the optimal solution. The hill climbing local search compromises on the quality of solution but it is faster to achieve such solution as compared to simulated annealing.

5.4 Box Plots

For randomize local searches following box plots show the comparison between simulated annealing and hill climbing for gr202 instance. The median error is lesser for simulated annealing.





Platform Specification:

Above results are obtained by a python 2.7 interpreter on a i7 quad core machine having 8 GB RAM in idol state without any other load on the machine. For calculating the probability and averages 10 iterations are used for randomized local search algorithms.

6. Discussion

Based of the above evaluation we can compare various algorithms to solve tsp. We can evaluate in which algorithms performs better under what circumstances.

Furthest Insertion Greedy Heuristic vs MST -2 approximation:

The approximation algorithms are good choice when the size of graph is big and there is permissible margin of error. We implemented and evaluated two such approximation algorithms. According to our finding, greedy heuristic with furthest insertion gives pretty good (10%) approximation results but the error keeps on increasing with the number of cities in the tsp instance. On the other hand MST 2-Approx has quite good constant approximation guarantees and it doesn't depend on the size of the tsp instance. But for smaller instances furthest insertion outperforms the MST 2-Approx.

Thus for the smaller instances one might prefer furthest greedy heuristic and for the larger instances one will prefer MST 2- Approx with constant approximation guarantee.

Simulated Annealing vs Hill Climbing:

Randomized local search approaches are used to find the solution close to the optimal solution with randomization and heuristics. According to our evaluation, simulated annealing worked performed quite well. For instances within 50 cities, it was able to reach optimal solution with high probability. Even for the larger instances the error rate was mostly within 5% with 600 sec cutoff.

On the hand Hill climbing found solution comparatively with higher error rates (within 10% as shown by QRTD and SQD). But the biggest difference between the two algorithms is that hill climbing was able to approach towards the solutions faster as compared to simulated annealing.

Thus depending upon the requirements and instance size, simulated annealing will be preferred for higher precision solutions and hill climbing will be preferred for better running time.

Branch and bound

This is the only algorithm in our approach that has theoretical guarantees of finding the optimal solution. Out observation matched our intuition that it was not able to scale for more than 16 cities instances because the neighborhood search is exhaustive even after pruning.

Thus Branch and Bound will be preferred when only optimal solution is acceptable and the graph instance is not very large.

7. Conclusion

Thus In this class project, we implemented algorithms in various categories to obtain the solution of classical NP Hard tsp problem. We also evaluated the performance of such algorithms and found that these all algorithms are useful depending on the objective and the exact instance of the problem. We also could explore some innovative ways in which these existing algorithms can be improved. There is a further lot of scope in the improvement for helping our puzzled travelling salesman.

References

Gerard Reinelt. The Traveling Salesman: Computational Solutions for TSP Applications.
Springer-Verlag, 1994.

Larrañaga, Pedro, et al. "Genetic algorithms for the travelling salesman problem: A review of representations and operators." *Artificial Intelligence Review* 13.2 (1999): 129-170.

P. Larrañaga, C.M.H. Kuijpers, R.H. Murga, I. Inza, S. Dizdarevic. "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators", Artificial Intelligence Review

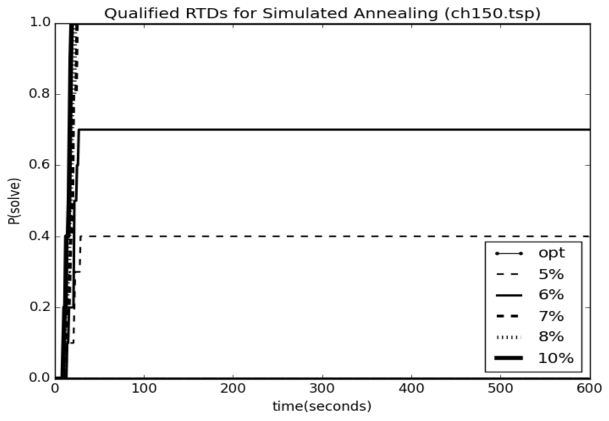
Malek, Miroslaw, et al. "Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem." *Annals of Operations Research*21.1 (1989): 59-84.

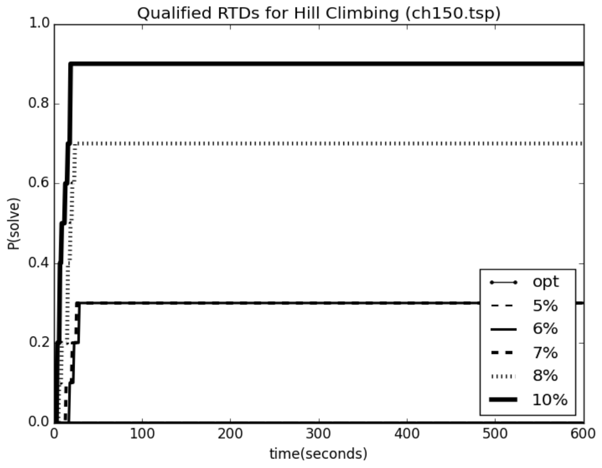
E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. The Traveling
Salesman. JohnWiley and Sons, 1986.

Appendix

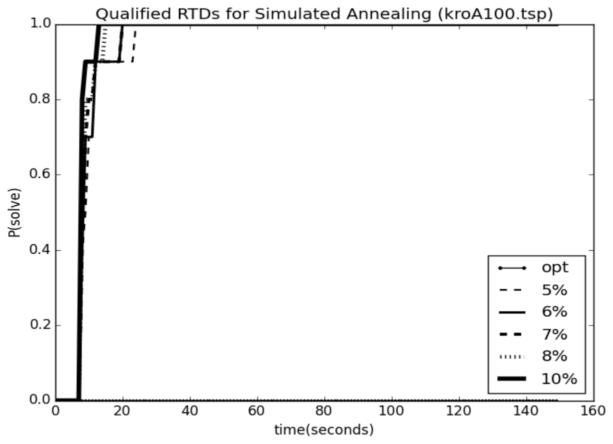
QRTD Plots (for remaining graph instances)

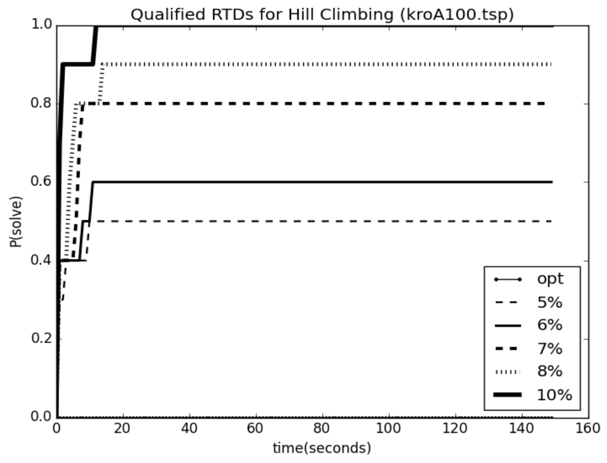
Ch150.tsp





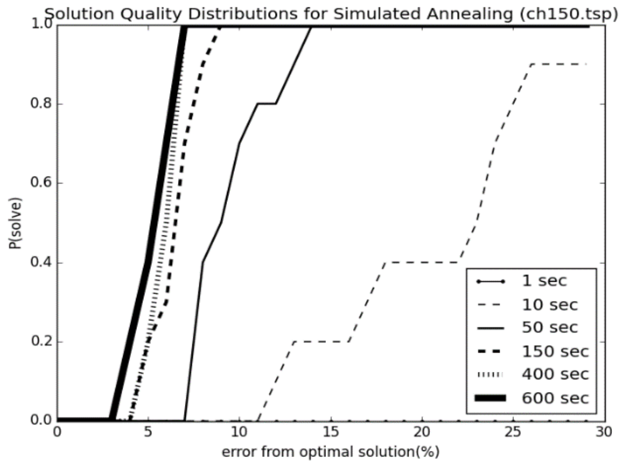
kroA100.tsp

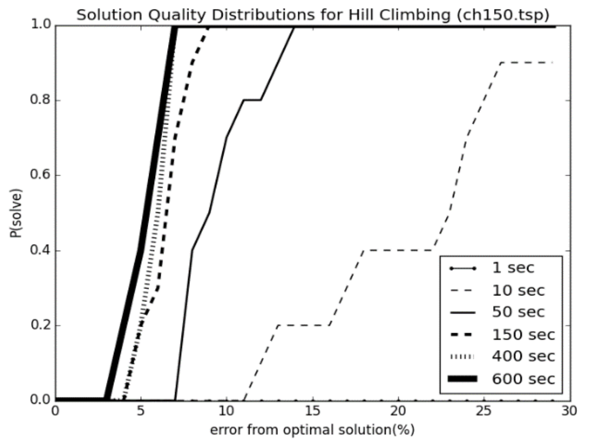




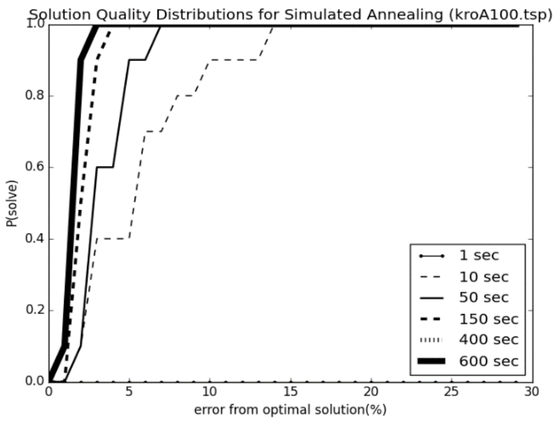
SQD Plots (for remaining instances)

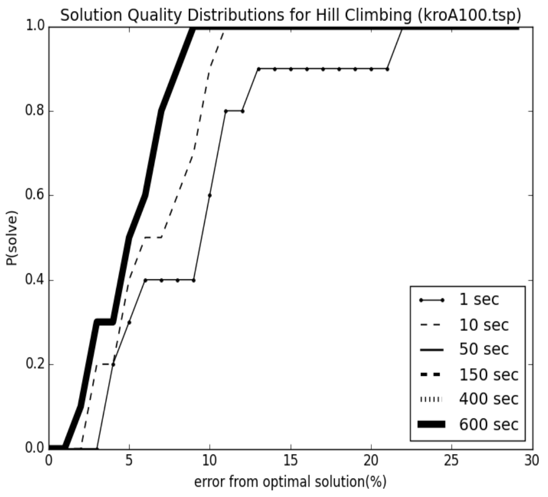
Ch150.tsp





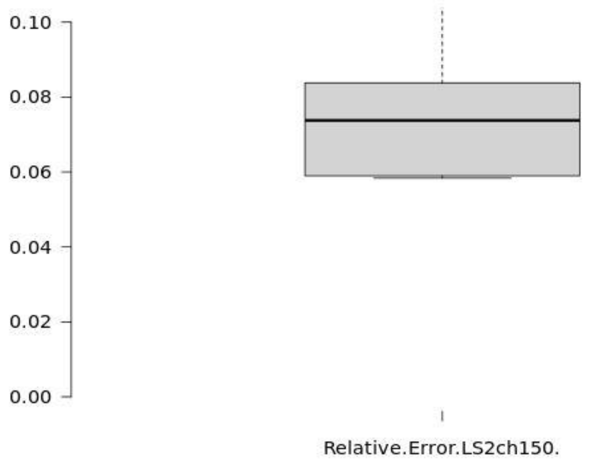
kroA100.tsp

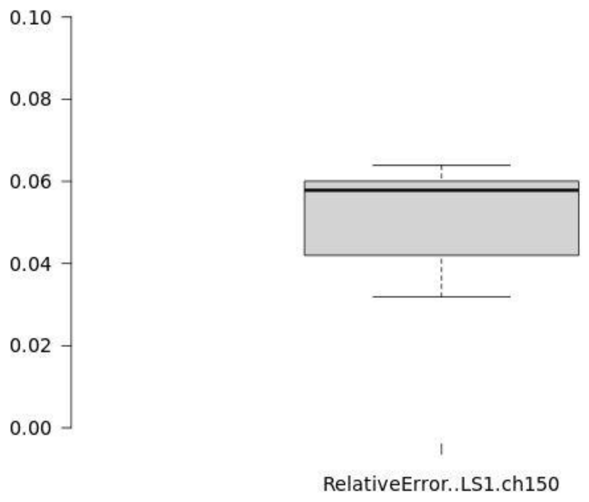




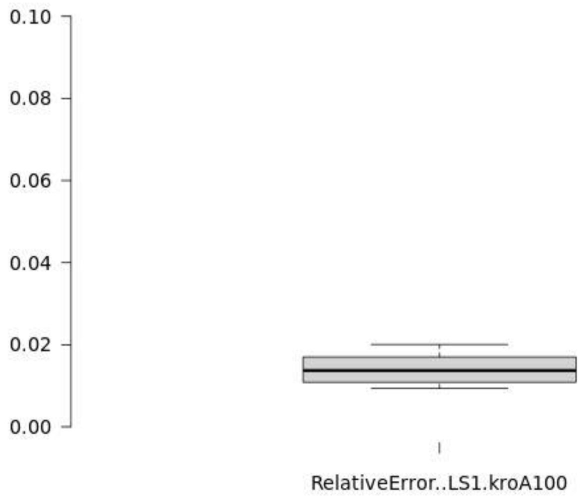
Box plots

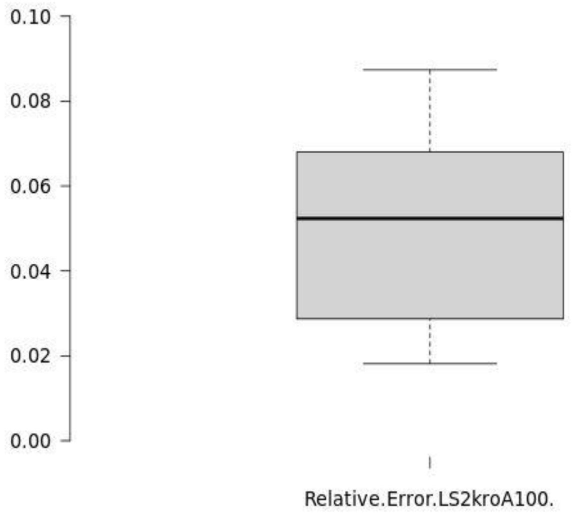
Ch150



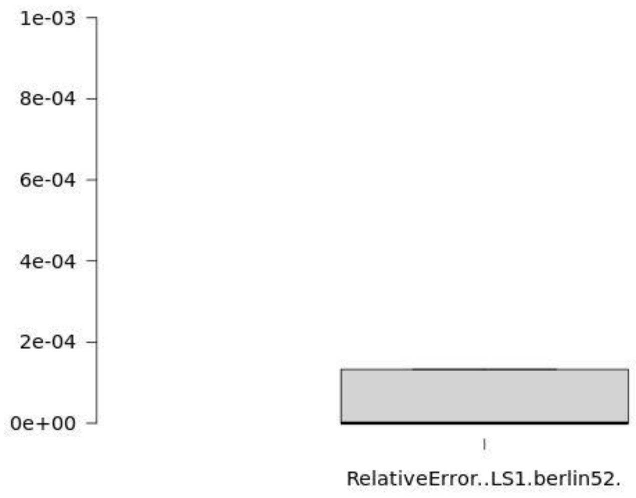


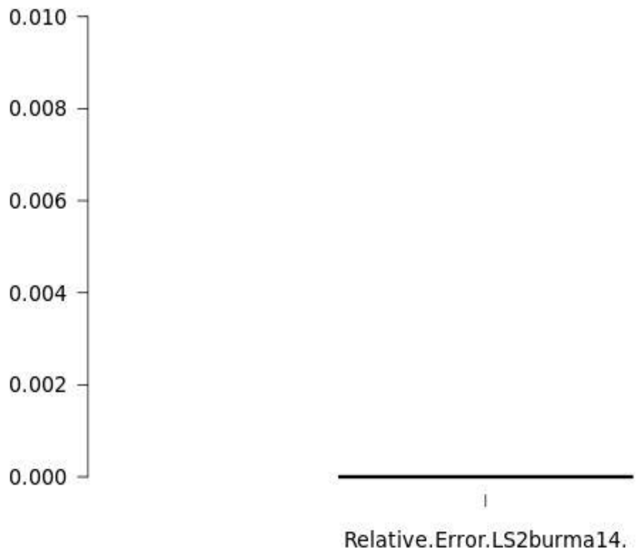
kroA100

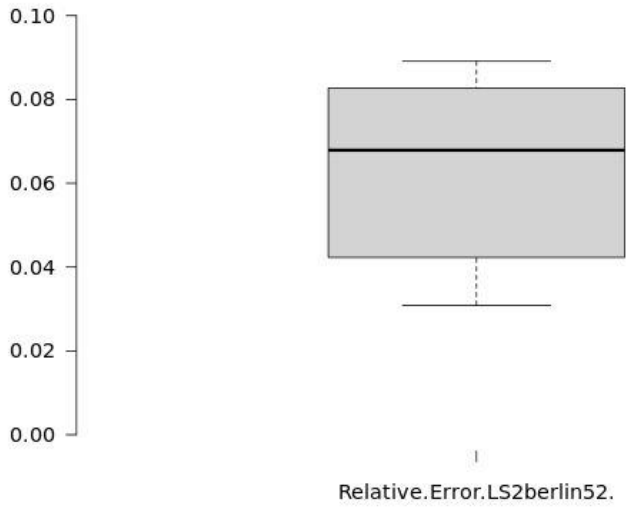




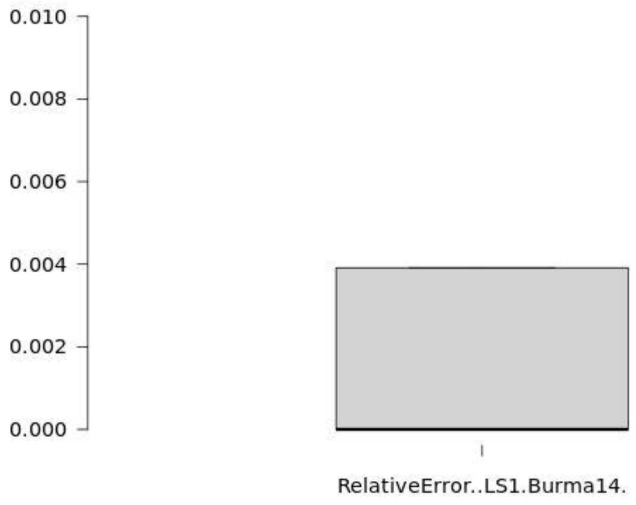
Berlin52







Burma14


RelativeError..LS1.Burma14.