Analysis of Select Approaches for Solving the Travelling Salesman Problem

Parul Awasthy
Georgia Institute of Technology
Atlanta, GA 30332
pawasthy3@gatech.edu

Young Jin Kim
Georgia Institute of Technology
Atlanta, GA 30332
ykim362@gatech.edu

Ankit Srivastava
Georgia Institute of Technology
Atlanta, GA 30332
asrivastava64@gatech.edu

1. INTRODUCTION

Travelling Salesman Problem (TSP) is one of the most intensively studied problems in the world of Theoretical Computer Science. It asks the question that given a list of cities and distance between each pair of cities, what is the shortest possible tour that visits each city exactly once and returns to the origin city. It is an NP- hard problem and so no polynomial time solution to solve the problem has been found till date.

TSP has been the basis of many optimization studies. Many approaches for finding the exact and approximate match are known. In this work we discuss five such strategies. Branch and Bound finds the exact solution. Iterated Local Search and Genetic Algorithm are Local Search approaches that find approximate solution. And Minimum Spanning Tree Approximation and Farther Neighbor Insertion are Construction Heuristic to find approximate solution. For each of the strategy the approach is outlined and analyzed. Each algorithm has been run on six instances of data starting covering up to two hundred cities. Detailed results and analysis is also included. In brief, Construction Heuristics gave quick solutions in the range of 15 to 25% for the bigger datasets, the Local Search Algorithms returning error of less than 10% for bigger datasets and Branch and Bound giving optimal results for small datasets and near optimal results within the 10 minute cut-off time.

We begin by defining the problem in detail in Section 2 and then explore related work in Section 3. We then explain all the algorithms in Section 4. We present our results in Section 5. We discuss the results in Section 6 and finally conclude in section 7.

2. PROBLEM DEFINITION

In any given TSP problem, the main objective is minimizing the total distance of a complete tour. The complete tour means that all cities in the problem should be visited and the number of visiting should be one by a solution. It also should not make two or more cycles in the graph. Only one cycle should cover all cities. The inputs of the problem have to contain following two kinds of information.

- Dimension: The total number of cities in a given problem
- Distance: Any distance from city A to city B when city A and city B are selected arbitrarily

The distance between two cities can be symmetric or not. If the distance from city A to city B is the same with the distance from city B to city A, it is called symmertic distance. Otherwise, the problem is an asymmetric TSP problem.

In this project, the distance is not directly given but the coordinates of all cities are given. In the process of solving a problem, the distance is extracted by using the coordinates of the cities. There are two different kinds of coordinates. The first one is GEO type which uses x coordinate as a latitude and y coordinate as a longitude. The other one is EUC_2D type which uses 2 dimensional coordinates in a 2 dimensional plane. Distances between two cities are calculated based on the Euclidean distance. The TSP problem can be expressed mathematically as below.

$$\min \sum_{i=0}^{n} \sum_{j \neq i, j=0}^{n} d_{ij} x_{ij}$$

$$x_{ij} = \begin{cases} 1 & \text{if the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

$$d_{ij} = \text{distance from city } i \text{ to city } j, \quad \forall i, j$$

$$\sum_{i=0, i \neq j}^{n} x_{ij} = 1, \quad \forall j$$

$$\sum_{j=0, j \neq i}^{n} x_{ij} = 1, \quad \forall i$$

$$u_i \in Z, \quad \forall i$$

$$u_i - u_j + n x_{ij} \leq n - 1, \quad 1 \leq i \neq j \leq n$$

Two equality constraints enfoce that all cities visited only once. The last inequality constraint enforces that there is only a single tour covering all cities, and not two or more disjointed tours that only collectively cover all cities.

3. RELATED WORK

In order to solve the TSP problem, various approaches have been suggested. They can be categorized according to the correctness of acquired solutions in exact and non-exact solutions. One of the most famous algorithm which returns an exact solution is Branch-and-Bound algorithm. There have been a lot of research about this algorithm. Richard Wiener suggested several different Branch-and-Bound implementations including sequential and parallel algorithms. The first algorithm he suggested is a single threaded algorithm with the general TSP problems.[3] He used a depth first search algorithm and the 2 shortest edges as the lower bound of the algorithm. He also suggested a multi threaded algorithm and a multi processing algorithms.[4] In this task, only sequential algorithms were studies and implemented.

The non-exact solution algorithms can be divided into two different strategies. The first one is the construction heuristics and the other one is the local search algorithm. For the construction heuristics, there exist lots of algorithms including the MST approximation, farthest insertion, random insertion and etc. For this task, the MST approximation algorithm and the farthest insertion algorithm were implemented. For the local search algorithms, there are many variants of these. The iterated local search, the simulated annealing and the genetic algorithm are some part of this local search algorithm. There exist a lot of previous research to solve the TSP problem using the local search algorithm. For navigating the local neighborhood Stattenberger et. have outlined a 2.5-Opt move also called Node Insertion Move or Jump in [2]. This move randomly selects a node and an edge. It removes the randomly chosen node from its original position and places it between the end points of the randomly selected edge, which is cut for this purpose. The neighborhood size generated by this move is $N(N-2)$ and thus also of order $O(N^2)$.

In case of the genetic algorithm, there can be many different choices of implementing each individual operation in the algorithm. For the crossover operation, Zakir H. Ahmed suggested a new way of algorithm which improved the quality of the solution dramatically. Generally, random crossover strategies including generalized N-point crossover are widely used. But he developed a new crossover operation based on a greedy heuristic algorithm and much better quality could be acquired.[1] In this project, this crossover operation is used to improve the quality of a solution.

4. ALGORITHMS

4.1 MST Approximation

Metric TSP instances are the ones in which all the edges follow triangle inequality i.e. cost of going from node A to node B directly is always lesser than cost of going from A to B via any other node. There exists an algorithm for getting near optimal tour of metric TSP instances in polynomial time with guarantees on quality of the solution. Since all the given TSP instances are metric, this algorithm can be used for getting the near optimal cost.

4.1.1 Algorithm

The algorithm first obtains a minimum spanning tree (MST) for the instance. Although the time complexity for getting the MST depends on the algorithm chosen, it can be done in polynomial time. The algorithm then does a pre-order walk of the MST and adds the nodes to the tour in the order they are visited. This can also be done in polynomial time. The algorithm therefore runs in polynomial time. It can be shown that the cost of the tour thus obtained is no more than twice the optimal. Therefore, the algorithm is a polynomial-time 2-approximation algorithm for metric TSP.

4.1.2 Implementation

There were two algorithm choices for obtaining MST: Kruskal's algorithm, which runs in $O(|E| \log |V|)$ time with union-find, and Prim's algorithm, which runs in $O(|E| + |V| \log |V|)$ with Fibbonacci heaps. The corresponding runtime of Kruskal and Prim on cliques, with $|E| = O(|V|^2)$, is $O(|V|^2 \log |V|)$ and $O(|V|^2 + |V| \log |V|)$ $(= O(|V|^2)$ respectively. Since all the TSP instances under consideration are cliques, Prim's algorithm was chosen for getting the MST as it promised a better runtime. Preorder walk of the MST was done by doing a recursive depth first traversal of the MST and adding nodes to the tour the first time they were visited. This is done in $O(|E|)$ $(= O(|V|^2)$ for cliques) time. The tour thus obtained was returned as TSP tour of the given instance.

Algorithm 1 MST Approximation

```
 1: procedure PrimMST(root)
 2:     mst ← φ
 3:     heap ← φ
 4:     for node ∈ tspInstance do
 5:         node.key ← ∞
 6:         insert node in heap
 7:     root.key ← 0
 8:     while heap ≠ φ do
 9:         u ← extract minimum element from heap
10:         add u to mst
11:         for v ∈ heap do
12:             if distance between u & v < v.key then
13:                 v.key ← distance between u & v
14:     return mst
15: procedure PreorderWalk(mst, root)
16:     walk ← φ
17:     children ← mst[root]
18:     for child ∈ children do
19:         childWalk ← PreorderWalk(mst, child)
20:         append childWalk to walk
21:     return walk
22: procedure MSTApproximation
23:     root ← 1
24:     mst ← PrimMST(root)
25:     approxTour ← PreorderWalk(mst, root)
26:     return approxTour
```

4.2 Farthest Neighbor Insertion

4.2.1 Algorithm

Farthest Neighbor Insertion is a Greedy Heuristic to find an approximate solution for the TSP. As this is a greedy construction heuristic it chooses cities that fit the heuristic best at that point of time with no consideration to global optimality. It is considered a quick approach to solve the TSP with results that are only 5 - 10% longer than the optimal.

The algorithm builds the tour from the start by inserting any city first and then choosing the city farthest to it and inserting it to the tour. From there on the next city is selected by choosing a city, not belonging to the tour, that is farthest to any city in the tour, and inserting it a position that does increases the tour length by the least. The approach essentially creates a frame out of the farthest apart cities and then fill it in with cities that would then be closer.

4.2.2 Implementation

For implementing Farthest Neighbor Insertion a distance matrix is created using vector of vector list. The distance between any two points can be found by simply indexing the list by those two points as row and column. Then the farthest neighbor is searched and added to the tour in the appropriate place. The complexity of this approach is $O(n^3)$ as it uses vectors for searching the farthest node $O(n^2)$ and repeats this for all nodes $O(n^3)$.

Algorithm 2 Farthest Neighbor Insertion

```
1: procedure FarthestNeighborInsertion
2:     tour ← two farthest nodes from tspInstance
3:     notInTour ← tspInstance − tour
4:     while notInTour ≠ φ do
5:         find node ∈ notInTourthat is farthest to tour
6:         insert node in tour s.t. min increase to tour len
7:     return tour
```

4.3 Branch-and-Bound

4.3.1 Algorithm

Branch-and-Bound is an algorithm to find an optimal solution by exploring a tree-structured solution space. To prevent huge iterations, this algorithm make use of a bounding logic which can reduce the number of nodes generated. At the time of expanding tree, a lower bound is calculated and a node which has greater lower bound than the current best solution is pruned. Basically, this algorithm looks through all possible candiate solutions so the time complexity of this algorithm can be $O((n-1)!)$ in the worst case. So, the way how to set appropriate bound and how to select efficient expanding direction can determine the performance of the algorithm.

4.3.2 Implementation

To implement a branch-and-bound algorithm for the travelling salesman problem, there can be several decisions including the selections of a bounding function and a search direction. In this impelementation, a 2 shortest edges bound was selected. At every time a new node is expanded at the tree, the lower bound is calculated and if the node's lower bound is greater than the cost of the best tour obtained so far, the node is pruned. For the expansion of the tree, a depth-first search was implemented using recursive function calls. There are two other possible implementations which are best-first search and breadth-first search. However, the depth-first search is better in memory usage in a large dataset thus it is used in this project. The constraints to maintain the solution path as feasible are also inspected. Any edges which can make premature cycles, is excluded at the time of expanding tree. By this approach, many unnecessary exploring of the tree are pruned. The last approach, which is taken to improve the performance of the algorithm, is the strategy which uses initial upper bound. Approximation algorithms find feasible solutions which have certain level of quality very quickly even for the large dataset. Therefore, by using these solutions as a initial upper bound, a lot of nodes can be pruned without deep exploration to the tree. By using this strategy, it is assured that the implemented branch-and-bound algorithm can return at least as good as the approximated solutions.

Algorithm 3 Branch-and-Bound

```
1: procedure Solve
2:     bestTour ← ∞
3:     BranchAndBound(root, 0)
4:     return bestTour
5: procedure BranchAndBound(treeNode, nextEdge)
6:     if treeNode is a complete path then
7:         if treeNode.lb < bestTour.lb then
8:             bestTour ← tour length of treeNode
9:         return
10:    leftNode ← Expand(leftEdge)
11:    rightNode ← Expand(rightEdge)
12:    if leftNode.lb > bestTour.lb then
13:        prune leftNode
14:    if rightNode.lb > bestTour.lb then
15:        prune rightNode
16:    if leftNode.lb ≤ rightNode.lb then
17:        BranchAndBound(leftNode.lb, nextEdge)
18:        if rightNode.lb > bestTour.lb then
19:            prune rightNode
20:        else
21:            BranchAndBound(rightNode, nextEdge)
22:    else
23:        BranchAndBoun(rightNode, nextEdge)
24:        if leftNode.lb > bestTour.lb then
25:            prune leftNode
26:        else
27:            BranchAndBound(leftNode, nextEdge)
28: procedure Expand(edge)
29:    Check feasibility
30:    lb ← Compute lower bound
31:    return newNode
```

4.4 Iterated Local Search

4.4.1 Algorithm

Iterated Local Search is a strategy that uses many local searches in different neighborhoods and returns the best solution found. In ILS a solutions searched in the local neighborhood till it no better solution can be found. The current best solution is then stored and the neighborhood is perturbed reach a new neighborhood and break out of local maxima. The best solution in this neighborhood is then searched and compared with the current best. The better quality solution is saved. These steps are repeated till the acceptance criteria is met. Various configuration can be tried to improve the performance of ILS e.g. using 2-Opt or 3-Opt to navigate the neighborhood, letting the algorithm make some bad moves, etc. It is relatively easy to change these configurations and see the results. The configurations that work best can be stored.

4.4.2 *Implementation*

Iterated local search has been implemented using 2.5-Opt move for traversing the neighborhood. The 2.5-Opt move randomly selects a node and an edge and places the node between the end points of the randomly selected edge, which is cut for this purpose. The neighborhood size generated by this move is $N(N-2)$ and thus also of order $O(N^2)$. The double bridge move for perturbing the tour to break the local maxima. It involves partitioning a permutation into 4 pieces (a,b,c,d) and putting it back together in a specific and jumbled ordering (a,d,c,b). Other than the above, the algorithm was allowed five bad moves in the local neighborhood before perturbing the neighborhood. The initial solution was chosen as the greedy solution.

To get to the above configurations, that resulted in best performance, many other configurations were tried. Experiments were done with 2-opt vs. 2.5 opt, 5 bad moves vs. 3 bad moves, random initial solution vs greedy initial solution, etc.

As the criteria was set that the algorithm should run for the given time and keep searching for better solutions, time complexity was not considered as a parameter. The double bridge move is a 4-Opt move that can result in a neighborhood of size $O(n^4)$. This was largely the time complexity of the algorithm. The 2.5 opt did not increase this complexity as because of the 5 bad moves criteria, the 2.5-opt exchange was called very few times.

Algorithm 4 Iterated Local Search

```
1:  procedure ITERATEDLOCALSEARCH
2:      bestTour ← GreedySolution
3:      while acceptanceCriteria() ≠ True do
4:          badMoves ← 0
5:          while badMoves ≤ 5 do
6:              currentTour ← twoPointFiveOpt(bestTour)
7:              if cost(currentTour) < cost(bestTour) then
8:                  bestTour ← currentTour
9:                  badMoves ← 0
10:             else
11:                 badMoves ← badMoves + 1
12:         currentTour ← Perturb(bestTour)
13:     return bestTour
14: procedure TWOPOINTFIVEOPT(tour)
15:     remove node w ← randomfromtour
16:     edge u, v ← randomfromtour
17:     insert w between u and v
18:     return tour
19: procedure PERTURB(tour)
20:     divide tourin segments a, b, c, d
21:     create tour as b, a, d, c
22:     return tour
```

4.5 Genetic Algorithm

4.5.1 *Algorithm*

Genetic algorithm is a search heuristic that mimics the process of natural selection. It uses operations like selection, crossover and mutation which are imitating natural evolution. The selection operation is a way to find some good parents without losing genetic variation. In order for this purpose, several different conceptual approaches were suggested including a roulette selection, a tournament selection and etc. The crossover operaion is the procedure generating offsprings using parentś genes. Basically, its purpose is mixing some good characteristics of parents to generate offsprings. The mutation operation is the step to add some randomness to the gene by perturbing any gene. The randomness in these operations makes the solution space not to be stuck in a local optimal solution. It is important to determine how to represent the solution as a genetic form and to select the appropriate fitness function to compare two solutions and to prioritize solutions. Figure 1 shows the process of the evolution of generations.

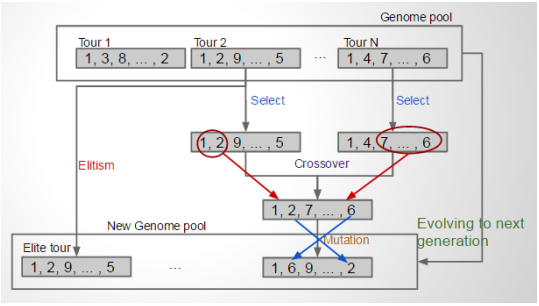

Figure 1: Genetic algorithm cycle

4.5.2 *Implementation*

In order to formulate the TSP as a problem to be solved by a genetic algorithm, it is more important how to determine the genetic form of a solution and how to define the genetic operations. Regarding the fitness fuction, it is obvious to use the inverse of tour cost as the fitness function. It means the shorter tour is more fitted in this gene pool. On the other hand, the genetic operations should be carefully selected because there is Hamiltonian cycle constraint in this problem. In some representations, the solutions acquired from operations can be infeasible.

For the selection operation, a tournament selection logic has been implemented in this project. It randomly extracts genes from the previous generation and selects the best gene among them. The size of a tournament has been set proportional to the size of one generation. A roulette selection logic tends to lose the variation of the gene pool easily so the tournament selection is better for finding global optimum. With this selection operation, an elitism operation is also used. It assures that the gene pool do not lose the best gene with evolving generations.

Regarding the crossover operation, two different kinds of crossover logic were implemented. The first one is generalized N-point crossover. This picks randomly two cities in one parent and the sequence between two cities in that parent is inherited to an offspring. Then, the other parent's sequence is modified not to destroy the complete tour and is inherited to the offspring. This operation performed well for the small TSP instances but it was too random for the larger instances to find a better offspring. Therefore, another crossover logic was studied and implemented. This is called sequential constructive crossover and uses a heuristic merge of two parents. It is consist of following four steps.

- Step 1: - Start from 'node 1' (i.e., current node p =1).

- Step 2: - Sequentially search both of the parent chromosomes and consider the first 'legitimate node'(the node that is not yet visited) appeared after 'node p' in each parent. If no 'legitimate node' after 'node p' is present in any of the parent, search sequentially the nodes 2, 3, ..., n and consider the first 'legitimate' node, and go to Step 3.
- Step 3: Suppose the 'node α' and the 'node β' are found in 1st and 2nd parent respectively, then for selecting the next node go to Step 4.
- Step 4: If $d_{p\alpha} < d_{p\beta}$, then select 'node α', otherwise, 'node β' as the next node and concatenate it to the partially constructed offspring chromosome. If the offspring is a complete chromosome, then stop, otherwise, rename the present node as 'node p' and go to Step 2.

This crossover logic performs much better than generalized N-point crossover. Because it keeps good characteristics of both parents, it has the higher change to go to the optimal solution.

In case of the mutation operation, a simple mutation logic was implemented. By any given rate, two cities are picked randomly and two cities are swapped in a sequence of a tour.

Algorithm 5 Genetic Algorithm

```
 1: procedure SOLVE
 2:     genePool ← INITIALPOPULATE
 3:     while numEvolution < setNumEvolution do
 4:         bestTour ← the best tour in genePool
 5:         while sizeOffspring < sizeGeneration do
 6:             parent1, parent2 ← SELECT(genePool)
 7:             offspring ← CROSSOVER(parent1, parent2)
 8:             offspring ← MUTATE(offspring)
 9:             append offspring to offspringPool
10:         genePool ← offspringPool
11:     return bestTour
12: procedure INITIALPOPULATE
13:     while sizeGenePool < sizeGeneration do
14:         make a new gene
15:         append gene to genePool
16: procedure SELECT(GenePool)
17:     randomly pick n gene from GenePool
18:     append gene to tournament
19:     select the bestGene among tournament
20:     return bestGene
21: procedure CROSSOVER(P1, P2)
22:     gene ← φ
23:     while sizeGene < sizeTour do
24:         append the closer city among P1 and P2 to gene
25:     return gene
26: procedure MUTATE(Gene)
27:     randomly pick 2 cities in Gene
28:     swap selected 2 cities in Gene
29:     return Gene
```

5. EMPIRICAL EVALUATION

We used *C++11* for implementing the algorithms. *g++* (v4.8.2), with option *-std=c++0x* specified for enabling C++11 features and optimization level of *-O3*, was used for compiling the code base. The experiments were run on Linux platform, runnning on 8 core Intel® Core i7-4770, 3.40GHz CPU. The machine had 16 GB RAM. For performing the experiments, a cutoff time of 600 seconds (10 minutes) was used for all the runs and the best solution found until the cutoff time was reported. If an algorithm returned before the cutoff time then the time when it returned was noted. The two Local Search algorithms were run 10 times, each run with a different seed in range [1, 10]. Results of the experiments are tabulated in Table 1.

MST Approximation.

As discussed in Section 4.1, MST Approximation is a 2-approximation algorithm for metric TSP and it can be seen from Table 1 that quality of the solutions output by our implementation of the algorithm is always less than twice that of optimal tour cost. In fact, the quality of all the solutions was found to be between $15\% - 34\%$ of the optimal solution.

Farthest Neighbor Insertion.

It can be seen from Table 1 that the quality of solutions output by our implementation of this Greedy Heuristic is less than 5% of optimal, while it is less than 17% of the optimal solution for bigger instances.

Branch-and-Bound.

Our implementation of Branch-and-Bound finds optimal solution for *burma14* and *ulysses16* within given cutoff time. However, for *berlin52*, and other instances with more nodes, it is unable to improve the upper bound, returned by the greedy heuristic, within given time.

Iterated Local Search.

Our implementation of Iterated Local Search terminates before allotted time for *burma14*, *ulysses16*, *berlin52*, and *kroA100* with a solution within 10% of the optimal solution. For larger instances, it runs for the whole allotted time and gets to within 10% of the optimal.

For solutions obtained by running Iterated Local Search on the two biggest instances, *ch150* and *gr202*, Figure 2 and Figure 5 show Qualified Runtime Distribution (QRTD) plots, Figure 3 and Figure 6 show Solution Quality Distribution (SQD) plots, and Figure 4 and Figure 7 show the boxplots of run time versus relative solution quality for the instances.

Genetic Algorithm.

Our implementation of Genetic Algorithm terminates before allotted time for *burma14*, *ulysses16*, and *berlin52* and finds a solution within 5% of the optimal solution. Similar to Iterated Local Search, for larger instances, it runs for the whole allotted time and gets to within 10% of the optimal.

For solutions obtained by running Genetic Algorithm on the two biggest instances, *ch150* and *gr202*, Figure 8 and Figure 11 show QRTD plots, Figure 9 and Figure 12 show SQD plots, and Figure 10 and Figure 13 show the boxplots of run time versus relative solution quality for the instances.

6. DISCUSSION

Solution Quality.

As discussed earlier, MST Approximation is a 2-approximation

Table 1: Experimental results for all five algorithms on all six datasets. Time is in seconds, and Relative error (RelErr) is computed as $(AlgPathLength - OPTPathLength)/OPTPathLength$). Bold values for RelErr highlight the closest algorithm to the optimum for a given dataset.

Dataset	Branch and Bound			Farthest Neighbor Insertion			MST Approximation			Iterated Local Search			Genetic Algorithm		
	Time	Length	RelErr	Time	Length	RelErr	Time	Length	RelErr	Time	Length	RelErr	Time	Length	RelErr
burma14	1.43	3323	**0.00**	0.00022	3487	0.049	0.00023	3814	0.15	0.036	3323	**0.00**	0.22	3339	0.0048
ulysses16	75.31	6859	**0.00**	0.00022	7054	0.028	0.00028	7903	0.15	0.058	6868	0.0013	0.42	6868	0.0013
berlin52	600.00	7783	0.032	0.00038	7783	0.032	0.00033	10114	0.34	8.41	7573	0.0041	77.73	7558	**0.0021**
kroA100	600.01	24534	0.15	0.0017	24534	0.15	0.00086	27210	0.28	126.08	21488	**0.0097**	600.01	21883	0.028
ch150	600.01	7651	0.17	0.0023	7651	0.17	0.0017	8413	0.29	600.00	6964	**0.067**	600.01	7070	0.083
gr202	600.01	44776	0.11	0.017	44776	0.11	0.013	51990	0.29	600.00	43478	**0.083**	600.02	43785	0.090

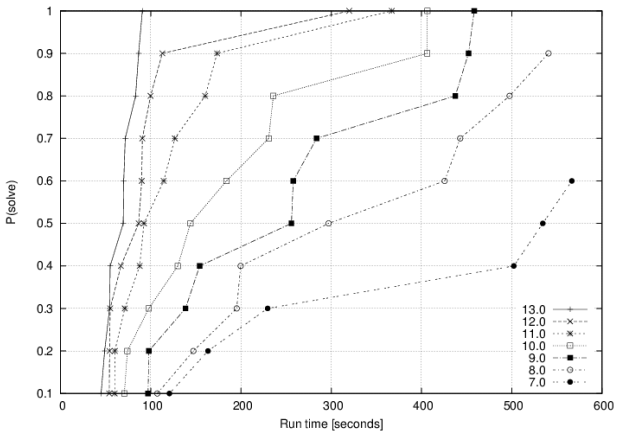

Figure 2: QRTD for *ch150* with Iterated Local Search

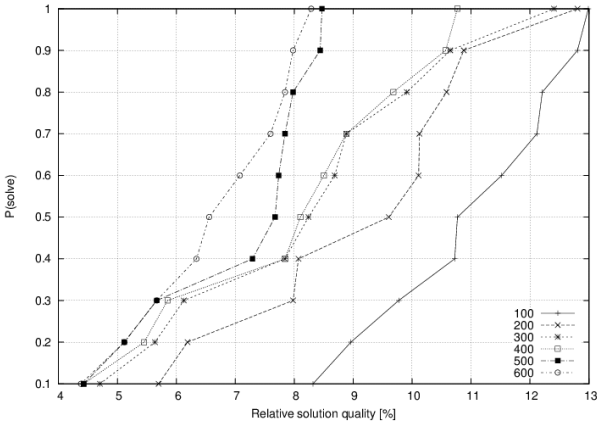

Figure 3: SQD for *ch150* with Iterated Local Search

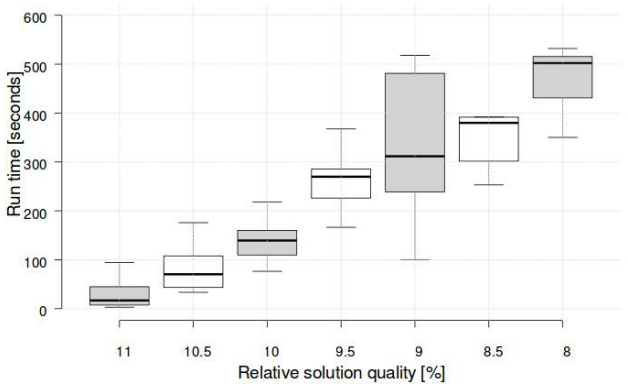

Figure 4: Boxplot for *ch150* with Iterated Local Search

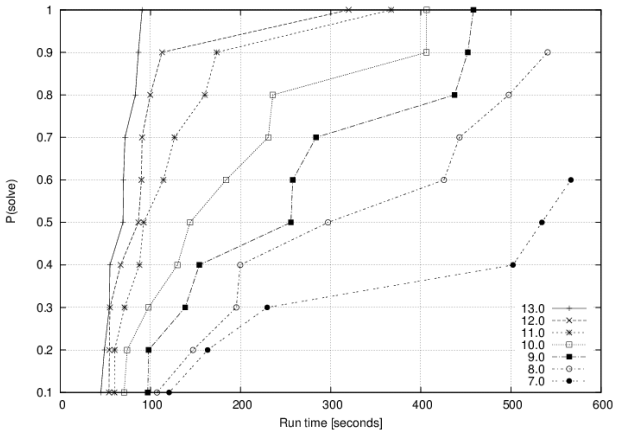

Figure 5: QRTD for *gr202* with Iterated Local Search

algorithm for metric TSPs. We found that the results produced by our implementation of MST Approximation were indeed less than twice the optimal cost. For the test instances, we found the solution returned by the algorithm to be within 35% of the optimal, in the worst case. The quality of the solution generally got worse with increase in the size of the instance. Greedy Heuristic performed generally better than MST Approximation, giving better quality ap-

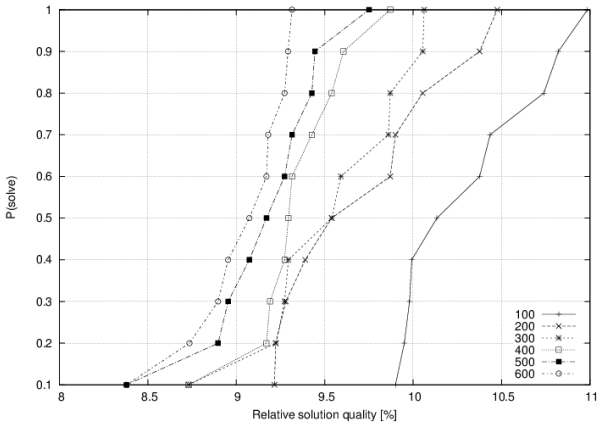

Figure 6: SQD for *gr202* with Iterated Local Search

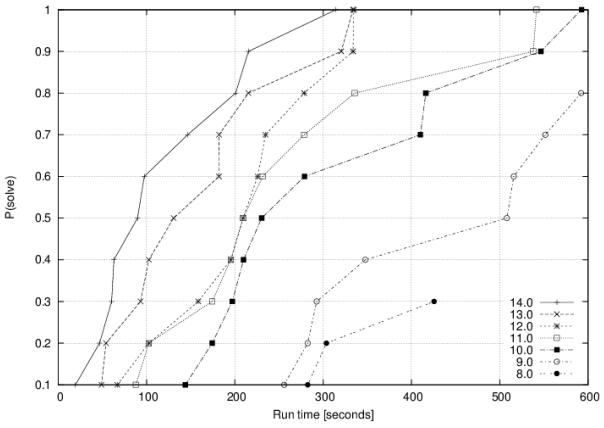

Figure 8: QRTD for *ch150* with Genetic Algorithm

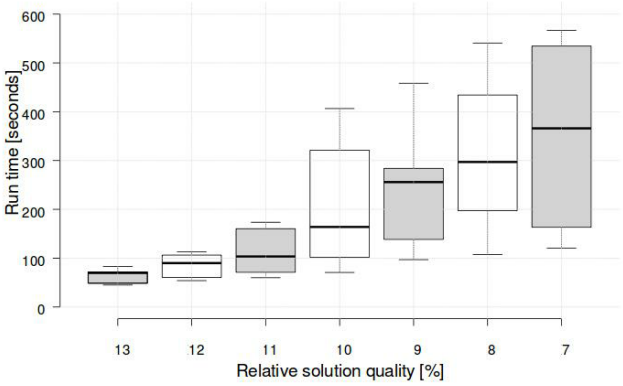

Figure 7: Boxplot for *gr202* with Iterated Local Search

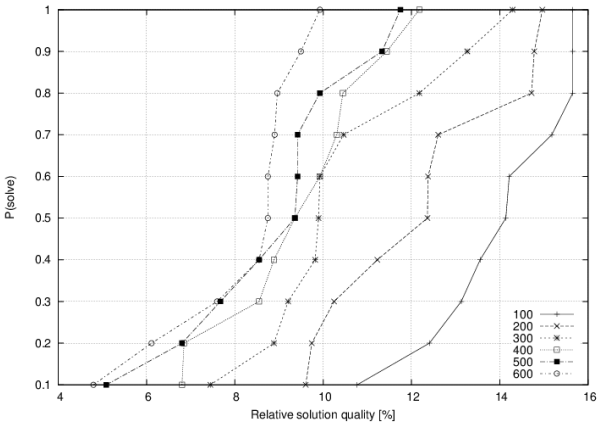

Figure 9: SQD for *ch150* with Genetic Algorithm

proximation in almost the same time. As in case of MST Approximation, the solution quality got worse with increase in size of the instance. However, even in worst case, the quality of the solution was within 17% of the optimal.

Branch-and-Bound used the approximate greedy cost as the initial upper bound and was able to find optimal solution for smaller instances: *burma14* and *ulysses16*. However, it was unable to improve the greedy approximation for bigger instances, within given time limit. Our branch-and-bound implementation didn't suffer from memory requirement issues, therefore, given enough time it is expected to return optimal solution in all the cases.

Of the two Local Search approaches that we implemented, Iterated Local Search generally gave better results than Genetic Algorithm, terminating after reaching within 1% of the optimal value for smaller instances and reaching a relative solution quality of less than 9% for bigger instances. While Genetic Algorithm also gave similar results, the time taken by Genetic Algorithm for reaching similar quality was observed to be more than that taken by Iterated Local Search.

Run Times.

We calculated time complexity of $O(n^2)$ for MST Approximation and $O(n^3)$ for Farthest Neighbor Insertion in sections above, where n is the size of the TSP instance. Although it is difficult to predict if the implementations are following the theoretical complexity with just 5 data points, we were able to fit a quadratic curve and cubic curve to the data points for the two algorithms.

All the other algorithms don't provide any specific guarantees on the run time and therefore it is difficult to analyze their run time. However, from the observed data, Iterated Local Search seems to be a better Local Search method in terms of time taken to reach a certain quality.

7. CONCLUSION

The research done to develop the five algorithms - Branch-and-Bound, Greedy Farthest Neighbour, MST Approximation, Iterated Local Search and Genetic Algorithm - has helped us discover new approaches to solve the Traveling

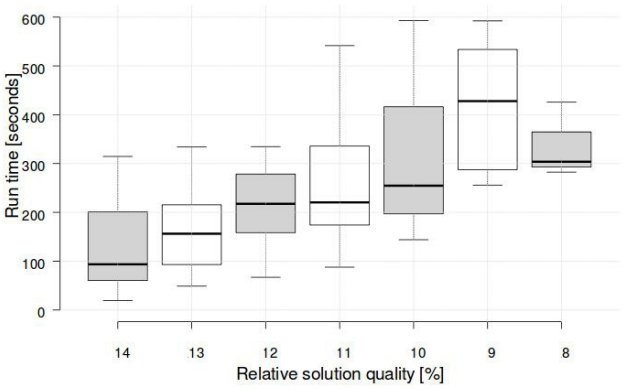

Figure 10: Boxplot for *ch150* with Genetic Algorithm

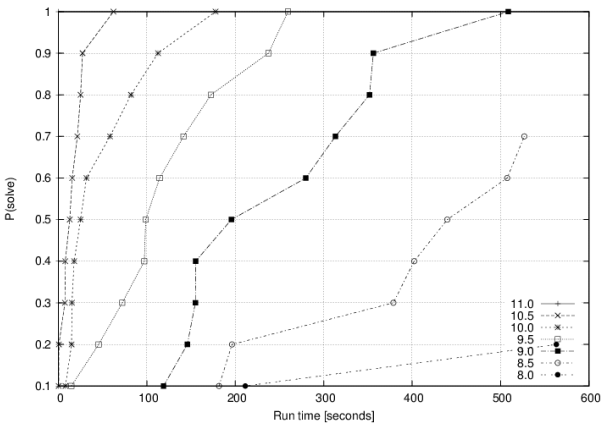

Figure 11: QRTD for *gr202* with Genetic Algorithm

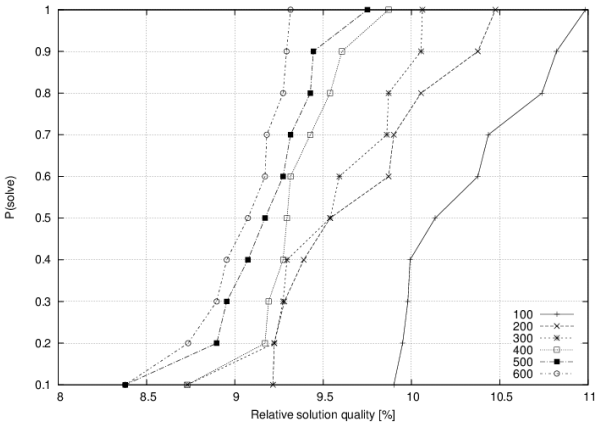

Figure 12: SQD for *gr202* with Genetic Algorithm

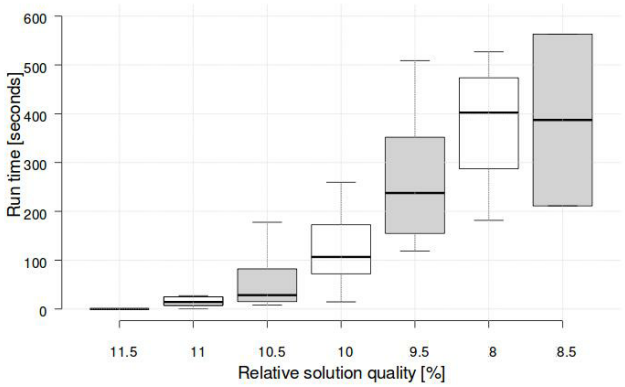

Figure 13: Boxplot for *gr202* with Genetic Algorithm

Salesman Problem. We have been able to experiment with different approaches and contrast their behavior or augment the performance of one approach with other.

The project helped us discover many interesting things that theoretical study did not equip us with. E.g. the performance of a greedy contraction heuristic like farthest neighbor insertion was much better empirically than in theory. Or that the performance of Local Search Algorithms improves drastically by changing a few parameters. Or even that increasing the size of the instance marginally might exponentially degrade the performance of exact solutions like Branch and Bound.

Experimental results for current version of the implemented algorithms are tabulated in Table 1. Though we can see from the table that the performance for the Local Search algorithm was better than the construction heuristic, the learning from the project is that the combination of parameters that were fed to the local search is what is important in improving its performance.

We would like to acknowledge the guidance and encouragement we recieved from the Professor and the Teaching Assistants in the course of this project, In all, this has been a very fruitfull learning experience, and has succeeded in allowing us to implement what we learnt in the course to a practical problem.

8. REFERENCES

[1] Z. H. Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics & Bioinformatics (IJBB)*, 3(6):96, 2010.

[2] G. Stattenberger, M. Dankesreiter, F. Baumgartner, and J. J. Schneider. On the neighborhood structure of the traveling salesman problem generated by local search moves. *Journal of Statistical Physics*, 129(4):623–648, 2007.

[3] R. Wiener. Branch and bound implementations for the traveling salesperson problem - part 1: A solution with nodes containing partial tours with constraints. *Journal of Object Technology*, 2(2):65–86, Mar. 2003. (column).

[4] R. Wiener. Branch and bound implementations for the traveling salesperson problem - part 3: Multi-threaded

solution with many inexpensive nodes. *Journal of Object Technology*, 2(4):89–100, July 2003. (column).