Different approaches to solve Traveling Salesman Problem

Revant Kumar, Xin Wei, Aashu Singh

Different approaches to solve Traveling Salesman Problem*

Group E

Xin Wei†
GT ID: 902939677
xwei36@gatech.edu

Revant Kumar‡
GT ID: 903024989
rkumar74@gatech.edu

Aashu Singh§
GT ID: 903081538
aashu@gatech.edu

ABSTRACT

In this paper, we have provided the details and implementation of different approaches to solve Traveling Salesman Problem (TSP). We have implemented five algorithms Greedy Heuristic (Farthest Insertion), MST-2 Approximation, Branch and Bound, Local Search - Hill Climbing and Local Search - Simulated Annealing. We have reported exhaustive evaluations for all the five algorithms. We have obtained our best results for the Local Search – Hill Climbing Algorithm with the solution gap within a value of 3%. It runs within 60s and achieved optimal solution for burma14.tsp and ulysses16.tsp (solution gap of 0%). The performance of our greedy algorithm (Farthest Insertion) is good with solution gap within 11%. The greedy solution is quite fast and runs within 0.1s. For branch and bound, we have achieved optimal solution for burma14.tsp and ulysses16.tsp (solution gap of 0%).

Keywords

Traveling Salesman Problem, Local Search, Hill Climbing, Simulated Annealing, MST-2 Approximation, Branch and Bound, Farthest Insertion

1. INTRODUCTION

The traveling salesman problem is a problem in graph theory requiring the most efficient (i.e., least total distance) Hamiltonian cycle a salesman can take through each of n cities. No general method of solution is known, and the problem is NP-hard.

The following image from http://xkcd.com/399/ explains the complexity of the traveling salesman problem as shown in Figure 1.

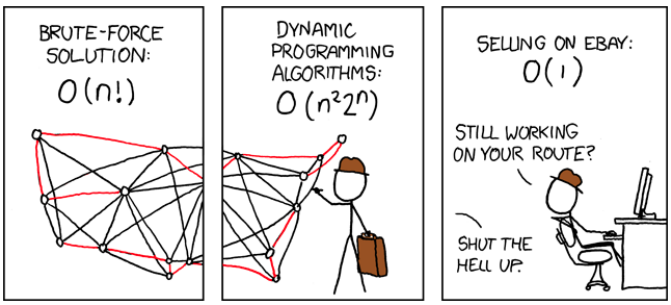

Figure 1: xkcd: Traveling Salesman Problem

Unfortunately for TSP fans, no good algorithm is known for the problem. The best result thus far is a solution method, discovered in 1962, that runs in time proportional to $n^2 2^n$. Although not good, this growth rate is much smaller than the total number of tours through n points, which we know is $(n-1)!/2$.

We have used the following approaches to solve the Traveling Salesman Problem:

- Greedy Heuristic (Farthest Insertion)
- MST-2 Approximation
- Branch and Bound
- Local Search - Hill Climbing
- Local Search - Simulated Annealing

The results obtained are:

- The solution gap for the greedy algorithm (Farthest Insertion) is within 11%. Also, greedy heuristic is quite fast and runs for gr202.tsp in 0.1 seconds.
- The solution gap for Local Search – Hill Climbing is within 3%. The results for burma14.tsp and ulysses16.tsp are optimal with solution gap of 0%. Time taken by hill climbing to run is within 1 minute (60s).

*This is the final report of the project of CSE6140.
†PhD. in Operations Research, Georgia Tech
‡MS in Computer Science, Georgia Tech
§MS in Computer Science, Georgia Tech

- The solution gaps for Branch and Bound for the burma14.tsp and ulysses16.tsp are optimal with a value of 0%. Since the algorithm ran out of time for the last four cases, we used nearest neighbor to greedily construct a solution in the end.
- The solution gap for MST-2 Approximation is within 40%.
- The solution gap for Local Search – Simulated Annealing is within 10%. It takes about 300 seconds to achieve this solution gap.

Thus, we will now discuss about all the five algorithms and their evaluations in the following sections.

1.1 Problem Definition

The decision version of the traveling salesman problem says the following:

> Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

So, the classical Traveling Salesman Problem is formulated as follows:

Input: A finite set of "cities" $C = \{c_1, \ldots, c_n\}$ and a matrix of "distances" $d(c_i, c_j)$ between c_i and c_j for each pair i, j.

Output: A permutation $\pi(1), \ldots, \pi(n)$ of indices $1, \ldots, n$ such that the sum

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$$

is as small as possible. This sum is the length of a tour starting in the city $c_{\pi(1)}$, visiting all the cities in a certain order and returning back to $c_{\pi(1)}$.

1.2 Related Work

TSP can be modeled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.

In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether the graph has any tour shorter than L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases super-polynomially (or perhaps exponentially) with the number of cities.

In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions.

The Euclidean TSP, or planar TSP, is the TSP with the distance being the ordinary Euclidean distance. The Euclidean TSP is a particular case of the metric TSP, since distances in a plane obey the triangle inequality. Like the general TSP, the Euclidean TSP is NP-hard. With discretized metric (distances rounded up to an integer), the problem is NP-complete.

TSP can be formulated as an Integer Linear program(ILP) as well. Let V denote the set cities with labels $0, \ldots, n$ and define:

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

For $i = 1, \ldots, n$, take c_{ij} to be the distance from city i to city j. Then TSP can be written as the following integer linear programming problem:

$$\begin{aligned}
&\min \sum_{i=0}^{n} \sum_{j \neq i, j=0}^{n} c_{ij} x_{ij} \\
&s.t. \sum_{i=0, i \neq j}^{n} x_{ij} = 1 && j = 0, \cdots, n \\
&\sum_{j=0, j \neq i}^{n} x_{ij} = 1 && i = 0, \cdots, n \\
&\sum_{i,j \in U} x_{ij} \leq |U| - 1 && \forall U \subset V, 2 \leq |U| \leq |V| - 2 \\
&x_{ij} \in \{0, 1\} && i, j = 0, \cdots, n
\end{aligned}$$

The first set of equalities requires that each city be arrived at from exactly one other city, and the second set of equalities requires that from each city there is a departure to exactly one other city. The last constraints(also called subtour elimination constraints) enforce that there is only a single tour covering all cities, and not two or more disjointed tours that only collectively cover all cities.

Exact Algorithms: The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute force search). The running time for this approach lies within a polynomial factor of $O(n!)$, the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. One of the earliest applications of dynamic programming is the Held-Karp algorithm that solves the problem in time $O(n^2 2^n)$. Other approaches include various branch-and-bound algorithms, which can be used to process TSPs containing 40-60 cities.

Heuristic and approximation algorithms: Various heuristics and approximation algorithms, which quickly yield good solutions have been devised. Modern methods can find solutions for extremely large problems (millions of cities) within a reasonable time which are with a high probability just 2-3% away from the optimal solution.

The nearest neighbor (NN) algorithm (or so-called greedy algorithm) lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. For N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path. However, there exist many specially arranged city distributions which make the NN al-

gorithm give the worst route.

Optimized Markov chain algorithms which use local searching heuristic sub-algorithms can find a route extremely close to the optimal route for 700 to 800 cities.

TSP is a touchstone for many general heuristics devised for combinatorial optimization such as genetic algorithms, simulated annealing, Tabu search, ant colony optimization, river formation dynamics (see swarm intelligence) and the cross entropy method.

ACS (Ant Colony System) is a heuristic method described by Artificial intelligence researcher Marco Dorigo in 1997 which generates "good solutions" to the TSP using a simulation of an ant colony. It models behavior observed in real ants to find short paths between food sources and their nest, an emergent behavior resulting from each ant's preference to follow trail pheromones deposited by other ants.

2. ALGORITHMS

In this section, we will explain the implementation of each of the five algorithms that we have used in order to solve the Traveling Salesman Problem.

2.1 Branch and Bound

Branch-and-Bound(BnB) algorithm is an exact algorithm that can give optimal solution but takes exponential time in worst case. In our BnB algorithm, we start with an initial vertex as the starting vertex denoted as **a**. Then each node in the BnB tree is a path from **a** to **b** going through a subset of vertices T. So for each node, we store the path (**a**-T-**b**) and the remaining vertices R. Also, we store the lower bound of that node which are used to compare different nodes when branch and prune nodes having high lower bound.

We tried all 3 types of bound function introduced in class and found that Minimum Spanning Tree (MST) bound is the tightest among these 3 types of bound. And MST is used in our code to calculate the lower-bound. The algorithm used to calculate MST is Kruskal's algorithm. Kruskal's algorithm is easier to implement than Prim's algorithm for the data structure we used. Also since all datasets are small graphs, there is no significant difference in running time between these two algorithms. The branching strategy is to always choose the node with minimum lower-bound to expand since it is the most promising node that can reach the optimal solution. The drawback of this strategy is that we can't even get a solution within 30 minutes for large datasets. The pseudo-code is provided:

Branch-and-Bound(D) // Input: distance matrix D
{
01 $F = \{(\emptyset, P)\}$ // Frontier set of nodes
02 $B = (+\infty, (\emptyset, P))$ // Best cost and solution
03 **while** F not empty **do**
04 Choose(X,Y) in F - the node with minimum lower-bound
05 Expand(X,Y) in F
06 Let $(X_1, Y_1), (X_2, Y_2), \ldots, (X_k, Y_k)$ be new nodes
07 **for** each nodes (X_i, Y_i) **do**
08 Check (X_i, Y_i)
09 **if** (solution found) **then**
10 **if** (cost(X_i)<B cost **then** //update lower-bound
11 B = (cost(X_i),(X_i, Y_i))
12 **if** (not dead end) **then**
13 **if** (lb(X_i)<B cost **then** //update F
14 $F = F \cup \{(X_i, Y_i)\}$ // else prune by lb
15**return B**
}

We didn't try DFS algorithm to do branching. DFS will quickly give a solution but often with bad quality. However, we did construct a solution for large datasets. What we did is to find the current best node with minimum lower-bound. Then we choose vertices in R greedily to construct a valid solution. At each time, we choose the nearest neighbor of **b** and terminate until R is empty. From the result we can find that this procedure will usually give a solution within 20% of optimal.

The time complexity of BnB algorithm is exponential since it may search all $|V|!$ number of cycles in worst case. Also the space complexity is exponential since we need to store a node which corresponds to a partial solution in memory unless we expand it or prune it by lower-bound.

2.2 Local Search

Local search algorithms are widely used in solving NP-hard problem. Though they are inexact algorithms, the result of using these algorithms to solve TSP problems is very good considering time and space complexity. Greedy or approximation algorithm can easily find us a tour. However, they usually have bad quality. The idea behind those local search algorithms is to improve an existing tour by locally select a better neighboring tour and change to it without considering further steps.

Popular local search algorithms in solving TSP are *Hill Climbing*,*Simulated Annealing* and *Genetic Algorithms*,etc. Good algorithms often combine those ideas together. So we implemented 2 different local search algorithm to solve TSP problems.

2.2.1 Iterated Hill Climbing

To use ***Hill Climbing***, we first need to define the neighbor of a tour. We claim that a tour C' is a neighbor of C *iff* C' can be derived from C by doing k-opt exchange once. The k-opt exchange technique is to delete k edges of tour C and reconnect the remaining k pieces together to form a tour C'. The easiest one is 2-opt exchange and there is only one way to reconnect. For 3-opt exchange, there are 4 different ways to reconnect the tour. When $k = 5$, there are totally 148 different ways to reconnect the tour. For simplicity, we use 3-opt exchange technique in our *Hill Climbing* algorithm. And the 4 different cases of reconnecting the tour is shown in Figure 2.

Since the greedy solution has good quality, we use it as the initial tour and perform 3-opt exchange. We find the tour which has best improvement of our current tour at each iteration until we can't find a better tour. Then *Hill Climbing* stops at a local optimal solution which is often the case unless we are very lucky to obtain the global optimal. To go further, we need to perturb the current local optimal

solution to a different tour and redo *Hill Climbing*. The perturbed(or so-called kicked) tour may have worse quality than the current local optimal solution, but are promising to reach a better local optimal solution by doing *Hill Climbing*. This is the main idea of perturbation.

To do this, we used a particular 4-opt exchange technique (Figure 3) which is proved to be effective in practice. This perturbation will often perturb the current tour to a new tour that we will not come back to the same tour by *Hill Climbing*. In our implementation, we randomly select 4 edges in the tour and perform the corresponding perturbation. However, from the result of our experiment, this perturbation often results in bad quality solutions.

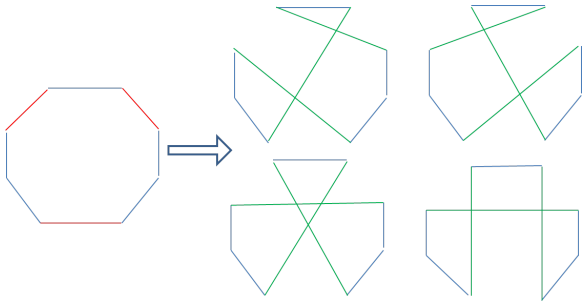

Figure 2: 4 different cases of 3-opt exchange

To overcome this problem, we simply store some of these bad quality solutions for further reference rather than throw them away. Since our result from *Hill Climbing* is very good(within 6% of optimal). There is a very high probability that the perturbed solutions are worse. However, we can only improve our solution by starting from those bad quality solutions. The problem is that we don't know which "bad" solution can drive us to better solution. So we just store some of promising "bad" solutions in a stack and reuse them repeatedly. The best solution will be put into the bottom of the stack and worse solutions on top. The so-called "promising" solutions are those solutions with value within m times of our current best solutions, which means their values are not far from optimal. We use $m = 2$ in our code. It can definitely be tuned to a better choice. From the result, we find that it's better to use large m for small graphs and small m for large graphs. On the other hand, the number of "bad" solutions we stored is 10. Again, this is another parameter that can be tuned to improve the performance of the algorithm. The pseudo-code is provided:

Hill-Climbing(D) // Input: distance matrix D
{
Step1: Start with a greedy solution;
Step2: Do{
Search for the best 3-opt exchange scenario;
Perform the 3-opt exchange;
} While(reach local optimal C)
stack.pushback(C);
Step3: Randomly perturb C to C'; Perform step2 to C';
Step4: temp=stack.top();
If(value(C')<value(*temp*))
stack.pop(); Goto step4;
Else if(value(C')<2*value(min)&& stack.size()<10)
Stack.pushback(C');
Goto step3;
Else
Goto step3;
End when exceeds cutoff time
}

The result of this algorithm is pretty good in the sense that we can get solutions within 4% of optimal for all datasets in just 30 seconds. For small datasets(e.g. burma14,ulysses16), the algorithm can get optimal solution within 1 second. The good performance of this algorithm shows that when we want to imporve a near optimal solution to optimal, we need to compromise the current solution to a worse solution and reach the optimal from it.

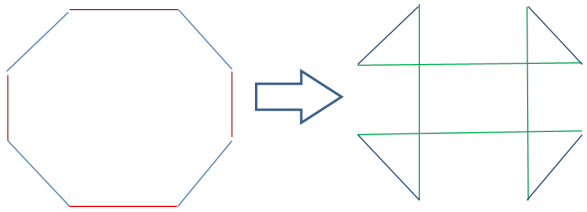

Figure 3: 4-opt exchange technique to do perturbation

For the complexity of the algorithm. Theoretically, perform a 3-opt exchange once or 4-opt exchange once takes constant time since we list all different cases in the code. To find the best improving neighbor of current tour, it checks all possible combinations of 3 edges and there are $O(V^3)$ number of them. For each combination, we will check 4 different cases. So the total number of checks is also $O(V^3)$. For one 3-opt exchange, the number of changes is $O(V)$. So the time complexity of one iteration of *Hill Climbing* is $O(V^3)$. The perturbation also takes constant time since we randomly delete 4 edges and reconnect the tour. In practice, the algorithm will run until it exceeds the cutoff time. So the total running time is exactly the cutoff time. However, for each iteration, it only takes polynomial time in the size of input. The space complexity is just $O(V)$ since we only keep the best tour in memory for hill climbing. In the perturbation process, we only store a constant number of solutions which is also $O(V)$.

2.2.2 *Simulated Annealing*

Simulated annealing (SA) is a generic probabilistic metaheuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities).

SA is based on neighborhood search. SA is a strategy which occasionally allows uphill moves. Uphill moves in SA are applied in a controlled manner. It has a strong analogy to the simulation of cooling of material. Uphill moves are allowed with a temperature dependent probability. Generic and problem-specific decisions have to be taken at implementation. Experimental tuning is very important in case of SA.

The SA implementation requires:

- State space with fitness measure:
  For each case of the TSP we are given a set of cities. So the search space is most naturally divided into states consisting of unique orderings of those cities. The "fitness" (or "cost") C of each state is the length of a path through the cities in that order. The goal is to find a state with the smallest possible fitness.
- Temperature schedule:
  The key to SA's ability to find global optimums, and what differentiates it from simple hill-climbing algorithms, is that at each iteration it has a (decreasing) probability of moving to less fit states. It always moves to more fit states but, presented with the prospect of moving to a less fit state, it will do so with probability equal to $e^{-C/T}$, where T is temperature. Theoretical analysis of SA indicates an optimal (maximum) cooling rate that ensures ergodicity and thus a positive probability of escaping from local optimums and eventually finding the global optimum.

Initial Temperature: TI must be high enough in order for the final solution to be independent of the starting one.

The rate at which temperature is reduced is governed by:

- Temperature length (TL): number of iterations at a given temperature
- Cooling ratio (f): rate at which temperature is reduced

The pseudo-code is provided:

Simulated-Annealing(D) // Input: distance matrix D
{
01 construct initial solution x_0; $x^{now} = x_0$
02 Set initial temperature T = TI
03 **repeat** F not empty **do**
04 **for** i = 1 to TL do
05 Generate random neighbor solution $x^{'} \in N(x^{now})$
06 compute change of cost $\Delta C = C(x^{'}) - C(x^{now})$
07 **if** $\Delta C \leq 0$ then
08 $x^{now} = x^{'}$ (accept new state)
09 **else**
10 Generate q = random(0,1)
11 **if** $q < e^{-\Delta C/T}$ then
12 $x^{now} = x^{'}$
13 set new temperature $T = f$
14 until stopping criterion (when exceeds cutoff time)
15 return solution corresponding to min cost function
}

The solution gap for Local Search – Simulated Annealing is within 10%. It takes about 300 seconds to achieve this solution gap.

2.3 Approximation Algorithm

MST-2 Approximation is a heuristic and approximation algorithm. In our case, constructions based on a minimum spanning tree have an approximation ratio of 2.

We have used Prim's Algorithm to find a Minimum Spanning Tree MST of the dense graph. The matrix representation explained above gives an efficient time complexity of $O(V^2 \log V)$, which is equal to that given by Fibonacci heap in this scenario.

Further, after we have obtained the MST, we choose one of the nodes as the source nodes. The 1st city is chosen as the starting node, and then pre-order traversal is performed on the minimum spanning tree. For an MST this takes - $O(V)$ time. As the new nodes are encountered in the traversal they are stored onto the stack, and finally we have the order in which, a complete tour can be done visiting a city exactly once.

Summing the intercity distance between the adjacent nodes, we get final path length.

Time Complexity: $O(V^2 \log V)$ & Space Complexity: $O(V)$

The pseudo-code is provided:

Approx-TSPTour(D) // Input: distance matrix D
{
Step1: Find a MST T; We used Prim's Algorithm.
Step2: Choose a vertex as root r; We have chosen the first city as root
Step3: return preorderTreeWalk(D, r);
}

preorderTreeWalk(D,r) // Input: distance matrix D, r is the root vertex
{
Step1: Depth First Search (DFS) in the tree, and output each node the first time that you enter it
Step2: Exactly the same order as the Eulerian tour and the shortcutting
}

The solution gap for the MST-2 approximation is within 40%.

2.4 Greedy Algorithm

In the greedy algorithm, we have used the approach of Farthest Insertion. Thus, we start from a single vertex (in our case we have started from city 0), and then insert new points into this partial tour one at a time until the tour is complete. The version of this heuristic that seems to work best is farthest point insertion.

In the farthest point insertion, of all remaining points, we insert the point v into a partial tour T such that

$$max_{v \in V/T} \; min_{i=1}^{|T|}(d(v, v_i) + d(v, v_{i+1}) - d(v_i, v_{i+1}))$$

The "min" ensures that we insert the vertex in the position that adds the smallest amount of distance to the tour, while the "max" ensures that we pick the worst such vertex first.

Time Complexity: $O(V^2)$ & Space Complexity: $O(V)$

The pseudo-code is provided:

Farthest-Insertion(D) // Input: distance matrix D
{
Step1: Start with a sub-graph consisting of node i only

(Here it will be the 1st city); Now inserts new points into this partial tour one at a time until the tour is complete.

Step2: In the farthest point insertion, of all remaining points, we insert the point v into a partial tour T such that $max_{v \in V/T}\ min_{i=1}^{|T|}(d(v,v_i) + d(v,v_{i+1}) - d(v_i,v_{i+1}))$

Step3: If all the nodes are added to the tour, stop. Else go to step 2

}

The solution gap for the greedy algorithm (Farthest Insertion) is within 11%. Also, greedy heuristic is quite fast and runs for gr202.tsp in 0.1 seconds.

Greedy heuristics always move from the current solution to the best neighboring solution.

3. EMPIRICAL EVALUATION

3.1 System Details

The detailed description of our platforms can be seen in Table 1.

3.2 Results

Table 2 gives the results for the all the algorithms that we have implemented.

3.3 Plots

- Box Plots for LS – Hill Climbing:
  - Box Plot for ch150.tsp: (Refer Figure 4)

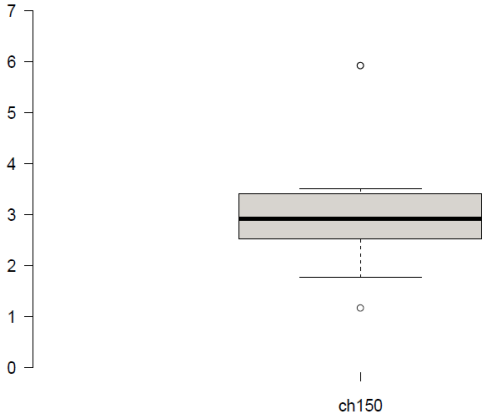

Figure 4: Box Plot for ch150 (60s) – Hill Climbing

  - Box Plot for gr202.tsp: (Refer Figure 5)
- QRTD Plots for LS – Hill Climbing:
  - QRTD Plot for ch150.tsp: (Refer Figure 6)
  - QRTD Plot for gr202.tsp: (Refer Figure 7)
- SQD Plots for LS – Hill Climbing:
  - SQD Plot for ch150.tsp: (Refer Figure 8)
  - SQD Plot for gr202.tsp: (Refer Figure 9)
- Mean Error Plots for LS – Hill Climbing:

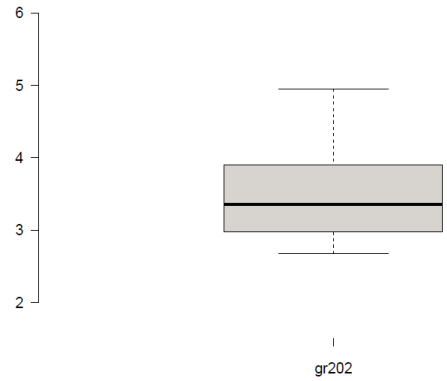

Figure 5: Box Plot for gr202 (60s) – Hill Climbing

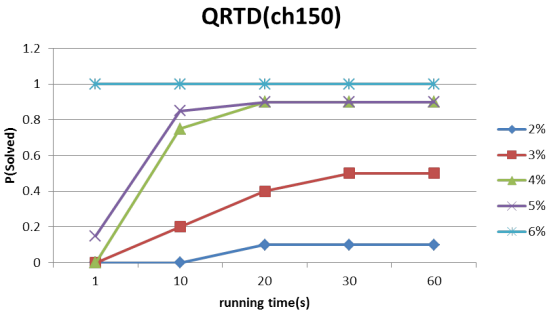

Figure 6: QRTD Plot for ch150 – Hill Climbing

  - Mean Error Plot for ch150.tsp: (Refer Figure 10)
  - Mean Error Plot for gr202.tsp: (Refer Figure 11)
- Box Plots for LS – Simulated Annealing:
  - Box Plot for ch150.tsp: (Refer Figure 12)
  - Box Plot for gr202.tsp: (Refer Figure 13)
- QRTD Plots for LS – Simulated Annealing:
  - QRTD Plot for ch150.tsp: (Refer Figure 14)
  - QRTD Plot for gr202.tsp: (Refer Figure 15)
- SQD Plots for LS – Simulated Annealing:
  - SQD Plot for ch150.tsp: (Refer Figure 16)
  - SQD Plot for gr202.tsp: (Refer Figure 17)
- Mean Error Plots for LS – Simulated Annealing:
  - Mean Error Plot for ch150.tsp: (Refer Figure 18)
  - Mean Error Plot for gr202.tsp: (Refer Figure 19)

4. DISCUSSION

For *Hill Climbing* algorithms, they are very effective in solving TSP problems. Though the results may not be optimal, they can produce very good solutions (within 3% of optimal) in a reasonable time. The key issue for local search algorithms is to define a neighbor of the tour and how to

Table 1: Description of our platforms

Person	CPU	RAM	Language	Compiler
Xin Wei	Intel Core i7 2.9 GHz	8 GB	C++	g++
Revant Kumar	Intel Core i5 2.5 GHz	4 GB	C++	g++
Aashu Singh	Intel Core i7 2.9GHz	8 GB	C++	g++

Table 2: 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 Insertion			MST-2 Approx			Hill Climbing			Simulated Annealing		
	Time	Length	RelErr	Time	Length	RelErr	Time	Length	RelErr	Time	Length	RelErr	Time	Length	RelErr
burma14	1.14	3323	**0.0000**	0.01	3442	0.0358	0.02	4003	0.2046	0.1	3323	**0.0000**	300	3442	0.0358
ulysses16	398.71	6859	**0.0000**	0.02	6950	0.0132	0.02	7788	0.1354	0.1	6859	**0.0000**	300	6950	0.0132
berlin52	600	9327*	0.2366	0.02	8170	0.0832	0.07	10402	0.3792	60	7571	**0.0038**	300	8170	0.0832
kroA100	600	27755*	0.3041	0.05	22352	0.0502	0.10	30516	0.4338	60	21594	**0.0146**	300	22352	0.0502
ch150	600	8206*	0.2570	0.07	7248	0.1102	0.20	9315	0.4269	60	6605	**0.0117**	300	7248	0.1102
gr202	600.02	48646*	0.2113	0.10	44364	0.1046	0.26	52617	0.3101	60	41235	**0.0267**	300	44364	0.1046

9327* These results are constructed by nearest neighbor algorithm at the end of time

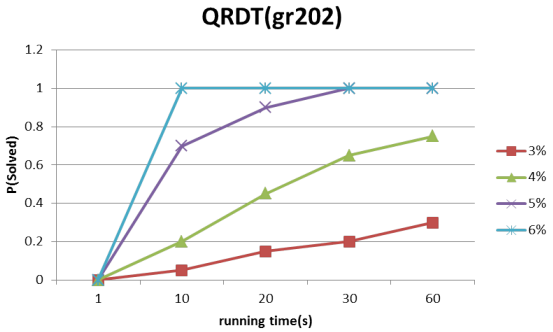

Figure 7: QRTD Plot for gr202 – Hill Climbing

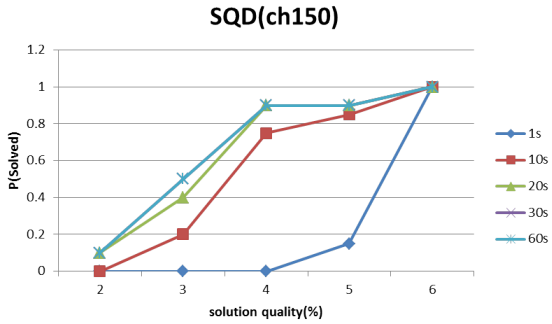

Figure 8: SQD Plot for ch150 – Hill Climbing

choose a promising one. Different choices can result in totally different solutions and qualities. On the other hand, *Hill Climbing* often gives a local optimal solution. To make further improvement, we need to change to a new tour and redo the process until we reach the global optimal. Perturbation technique is a good way to produce a new tour that is hard to be revisited. However, in practice, the perturbed solution often has worse quality than the previous local optimal. And the better the local optimal solution, the harder to perturb to a even better solution. We can tackle this problem by memorizing a certain amount of "worse but promising" perturbed solution and repeatedly do hill climbing to find a better solution. This is the procedure we do and it turns out to be effective. Other methods like *Simulated Annealing* are good approaches to tackle this problem, too. However, the key idea behind all these methods is how to effectively choose the "worse but promising" solution which can finally obtain global optimal solution.

For *Branch-and-Bound*(BnB) algorithm, it is an exact algorithm but often runs in exponential time. For large datasets, it takes very long time and quickly exhaust all memories. However, this is an effective approach in practice when we have tight bound function and branching strategy. A good branching stategy can quickly generate a solution whose value can be used to prune nodes. A tight bound function is also effective to guide us to the optimal solution and prune unpromising nodes. For example, we can model the TSP problem as an integer program(IP) and solve the LP relaxation of it. The optimal value of the LP relaxation is a good bound and branching on a fractional variable is a good branching strategy. Also by adding cutting planes to the LP relaxation, we can quickly get integer solutions and in turn the optimal solutions.

For *Greedy Algorithm*, we have used the Farthest Insertion approach. It is basically a Heuristic and approximation algorithm. The algorithm is quite fast and even the largest path in gr202.tsp run in 0.1 seconds. The solution gap achieved is within 11%. The minimum solution gap was achieved by the greedy algorithm was 1.32% for ulysses16.tsp. Greedy heuristics always move from the current solution to the best neighboring solution.

For *Simulated Annealing*, the solution gap is within 10%.

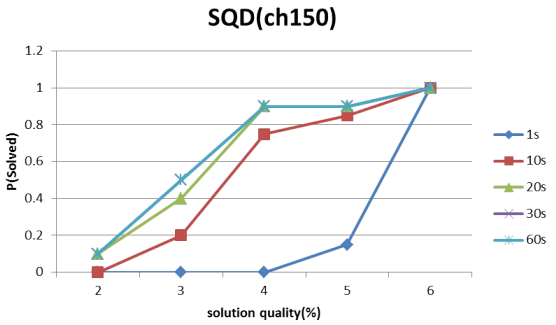

Figure 9: SQD Plot for gr202 – Hill Climbing

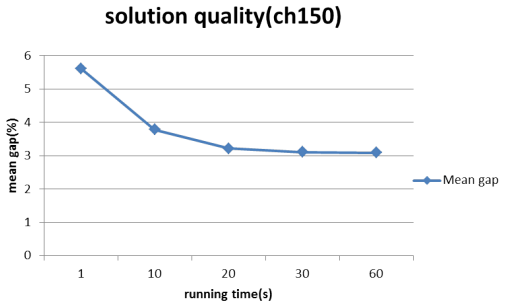

Figure 10: Mean Error Plot for ch150 – Hill Climbing

The initial solution path is randomly chosen. The initial temperature is set quite high enough, and it varies with the length of the graph. The cooling factor is taken to be constant. We can analyze from the solution obtained, that the solution gets stuck in the local minima. We use 2-opt exchange to search through the neighborhood space. This simple heuristic ensures that we iterate over all the possible solutions solutions. It takes about 300 seconds to achieve this solution gap.Simulated Annealing is an optimized Markov chain algorithms which uses local searching heuristic sub-algorithms to find a route extremely close to the optimal route for 700 to 800 cities, and thus probably running for more time in our case would return optimal solution.

For *MST-2 Approximation* Algorithm, we have achieved the solution gaps within 40%. MST-2 Approximation is a heuristic and approximation algorithm. In our case, constructions are based on a minimum spanning tree, and the pre-order walk through the tree gives a solution within twice the optimal.

5. CONCLUSIONS

Thus, we have successfully implemented and evaluated five algorithms – Greedy Heuristic (Farthest Insertion), MST-2 Approximation, Branch and Bound, Local Search - Hill Climbing, & Local Search - Simulated Annealing for solving the Traveling Salesman Problem. We have reported exhaustive evaluations for all the five algorithms. We have obtained our best results for the Local Search – Hill Climbing Algorithm with the solution gap within a value of 3%. It runs within 60s and achieved optimal solution for burma14.tsp and ulysses16.tsp (solution gap of 0%). The performance of our greedy algorithm (Farthest Insertion) is good with solution gap within 11%. The greedy solution is quite fast and runs within 0.1s. For branch and bound, we have achieved optimal solution for burma14.tsp and ulysses16.tsp (solution gap of 0%).

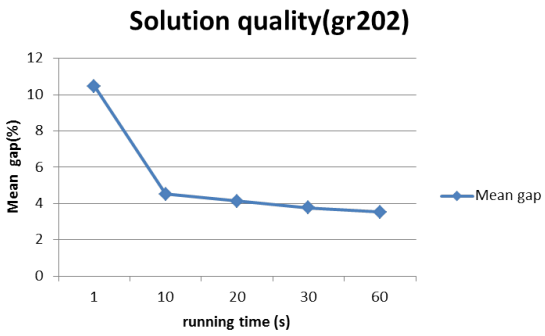

Figure 11: Mean Error Plot for gr202 – Hill Climbing

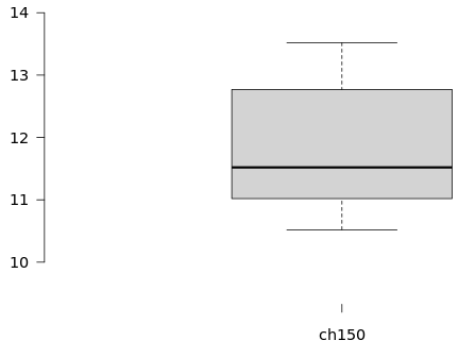

Figure 12: Box Plot for ch150 (60s) – Simulated Annealing

6. ACKNOWLEDGMENTS

We would like to thank Professor Bistra Dilkina and TA Elias Khalil for their valuable guidance during the entire project.

7. REFERENCES

[1] Chapter 8. Stochastic Local Search: Foundations and Applications.

[2] Chapters 1 & 2. Stochastic Local Search: Foundations and Applications.

[3] Cook,W. 2012. In pursuit of the Traveling Salesman: Mathematics at the limits of Computation. Princeton University Press.

Figure 13: Box Plot for gr202 (60s) –Simulated Annealing

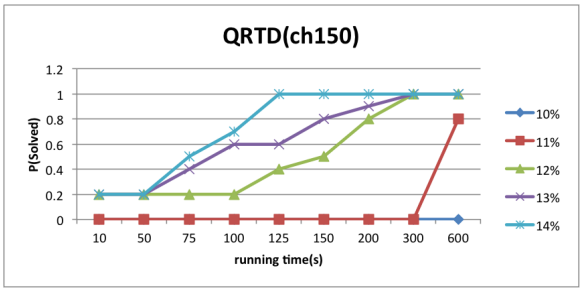

Figure 14: QRTD Plot for ch150 – Simulated Annealing

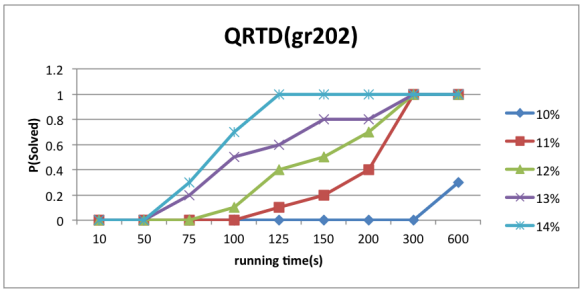

Figure 15: QRTD Plot for gr202 – Simulated Annealing

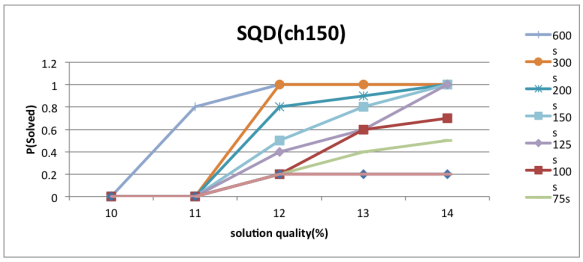

Figure 16: SQD Plot for ch150 – Simulated Annealing

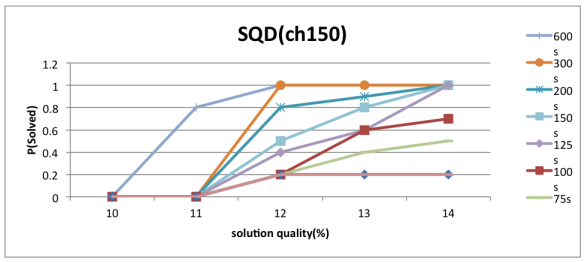

Figure 17: SQD Plot for gr202 – Simulated Annealing

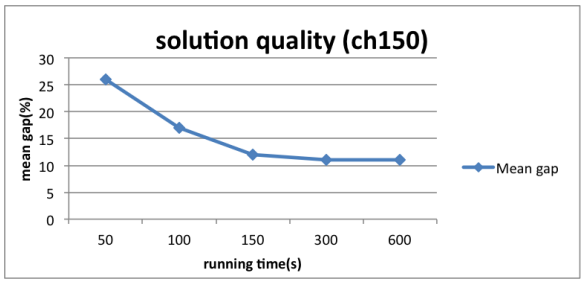

Figure 18: Mean Error Plot for ch150 – Simulated Annealing

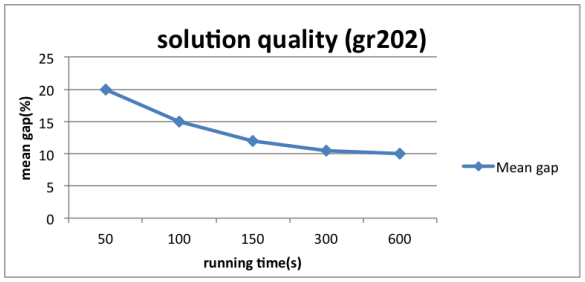

Figure 19: Mean Error Plot for gr202 – Simulated Annealing