Traveling Salesman Problem (TSP):

MD OHIDUL BARIK (SL1816013)

The Traveling Salesman Problem (TSP) is one of the most famous problems in computer science. Back in the days when salesmen traveled door-to-door hawking vacuums and encyclopedias, they had to plan their routes, from house to house or city to city. The shorter the route, the better. Finding the shortest route that visits a set of locations is an exponentially difficult problem: finding the shortest path for 20 locations is much more than twice as hard as 10 locations. An exhaustive search of all possible paths would be guaranteed to find the shortest, but is computationally intractable for all but small sets of locations. For larger problems, optimization techniques are needed to intelligently search the solution space and find near-optimal solutions.
Mathematically, traveling salesman problems can be represented as a graph, where the locations are the nodes and edges (or arcs) represent direct travel between the locations. The weight of each edge is the distance between the nodes. The goal is to find the path with the shortest sum of weights.

SECTION I

Greedy Algorithm for TSP:

A greedy algorithm, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision. **I applied the algorithm below to do my greedy algorithm application.**

Algorithm procedure:

1: **procedure** GREEDY($start\_pos, C$)
2: $current\_pos \leftarrow start\_pos$
3: $V \leftarrow \{\}$
4: **while** $|V| < |C|$ **do**
5: $next\_pos \leftarrow \min_c dist(current\_pos, c), \forall c \in C \setminus V$
6: $V \leftarrow V \cup next\_pos$
7: $current\_pos \leftarrow next\_pos$
8: **end while**
9: **end procedure**

Experiment and Result:

I write the greedy algorithm based on the above procedure and run for the provided datasets, time taken for these datasets, the shortest path distance and the optimized path is below:

Table 1: Result of greedy algorithm for two datasets.

	n=10	**n=100**
shortest path distance	10726.544518592038	28345.28036602417
total time taken (sec)	0.004934072494506836	0.01576542854309082

Optimized path order when n = 10:
[9 0 8 5 3 2 1 7 4 6]

Optimized path order when n = 100:
[138 90 82 58 22 15 75 1826 40 31 36 81 11 85 10 98 94 62 78 66 49 51 39 549 30 57 35 29 77 27 59 701 46 33 88 86 53 60 83 20 48 14 42 63 44 50 64 38 74 25 87 52 16 68 61 32 17 45 920 21 72 23 99 43 793 47 41 37 24 557 34 69 19 91 84 96 26 12 56 65 71 76 8945 67 95 73 28 93 97 80]

Dynamic Algorithm for TSP:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)? To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending . **I applied following procedure to apply algorithm for TSP problem**.

Algorithm procedure:

```
Initialize D_TSP with values ∞ ;
Initialize a table P to retain predecessor locations ;
Initialize v as an arbitrary location in V ;
foreach w ∈ V do
    D_TSP({w}, w) ← c(v, w) ;
    P({w}, w) ← v ;
for i = 2, ..., |V| do
    for S ⊆ V where |S| = i do
        foreach w ∈ S do
            foreach u ∈ S do
                z ← D_TSP(S \ {w}, u) + c(u, w) ;
                if z < D_TSP(S, w) then
                    D_TSP(S, w) ← z ;
                    P(S, w) ← u ;
return path obtained by backtracking over locations in P starting at P(V, v) ;
```

Experiment and Result:

I write the dynamic algorithm for the tsp based on the above procedure and run for the provided datasets, time taken for these data sets, the shortest path distance and the optimized path is below:

Tabel 2: Result of dynamic algorithm for two datasets:

	n=10	**n=100**
shortest path distance	10127.552143541276	NC
total time taken (sec)	0.019023999999999985	NC

Optimized path order when n = 10:
[4, 6, 7, 1, 3, 2, 5, 8, 9, 1]

Branch and Bound Algorithm for TSP:

The term Branch and Bound refers to all state space search methods in which all the children of E-node are generated before any other live node can become the E-node. E-node is the node, which is being expended. State space tree can be expended in any method i.e. BFS or DFS. Both start with the root node and generate other nodes. A node which has been generated and all of whose children are not yet been expanded is called livenode. A node is called dead node, which has been generated, but it cannot be expanded further. In this method we expand the node, which is the most promising, means the node which promises that expanding or choosing it will give us the optimal solution. **I do following procedure to apply algorithm for TSP problem**.

Algorithm procedure:

1 Set $L = \{X\}$ and initialize $\hat{x}$
2 **while** $L \neq \emptyset$**:**
3 Select a subproblem S from L to explore
4 **if** a solution $\hat{x}' \in \{x \in S \mid f(x) < f(\hat{x})\}$ can be found**:** Set $\hat{x} = \hat{x}'$
5 **if** S cannot be pruned**:**
6 Partition S into $S_1, S_2, \ldots, S_r$
7 Insert $S_1, S_2, \ldots, S_r$ into L
8 Remove S from L
9 Return $\hat{x}$

Experiment and Result:

I write the branch and bound algorithm for the TSP based on the above procedure and run for the provided datasets, time taken for these datasets, the shortest path distance and the optimized path is below:

Tabel 3: Result of dynamic algorithm for two datasets:

	n=10	**n=100**
shortest path distance	10127.552143541276	NC
total time taken (sec)	0.46016931533813477	NC

SECTION II

For n = 100, two of the above algorithms, namely, dynamic and greedy don't converges, so we applied heuristic: genetic algorithms and learning: simulated annealing algorithm for the TSP and compare their performances on two different datasets.

Genetic Algorithm for TSP:

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. Few important definitions for genetic algorithms are:

Gene: a city (represented as (x, y) coordinates)
chromosome: a single route satisfying the conditions above
Population: a collection of possible routes (i.e., collection of individuals)
Parents: two routes that are combined to create a new route
Mating pool: a collection of parents that are used to create our next population (thus creating the next generation of routes)
Fitness: a function that tells us how good each route is (in our case, how short the distance is)
Mutation: a way to introduce variation in our population by randomly swapping two cities in a route.
Elitism: a way to carry the best individuals into the next generation

My GA will proceed in the following steps:

1. Create the population

2. Determine fitness

3. Select the mating pool

4. Breed

5. Mutate

6. *Repeat*

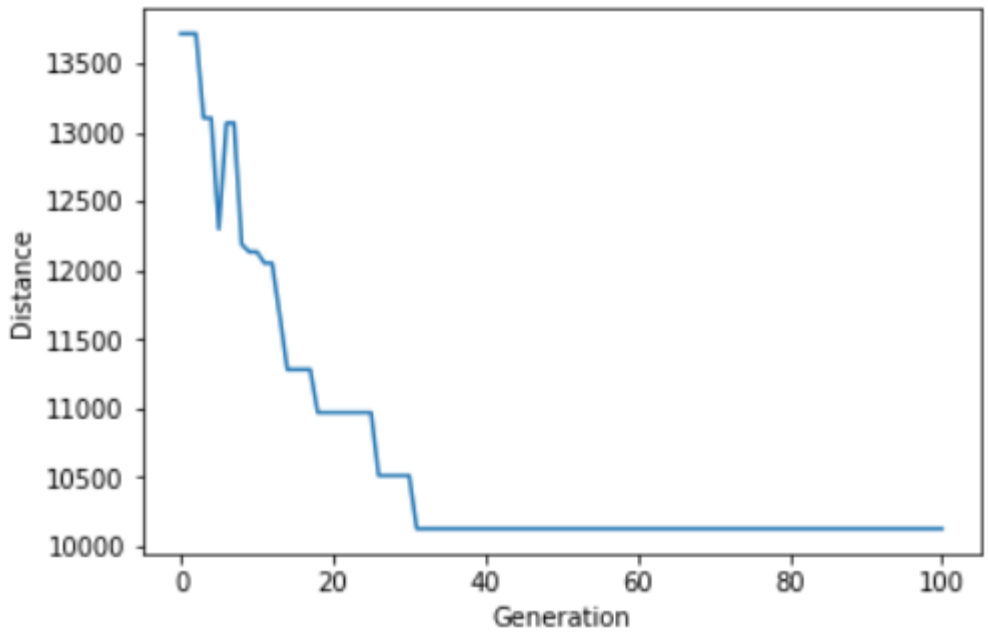
Experiment and Result:

I write the genetic algorithm for the TSP based on the above procedure and run for the provided datasets, time taken for these data sets, the shortest path distance and the optimized path is below:

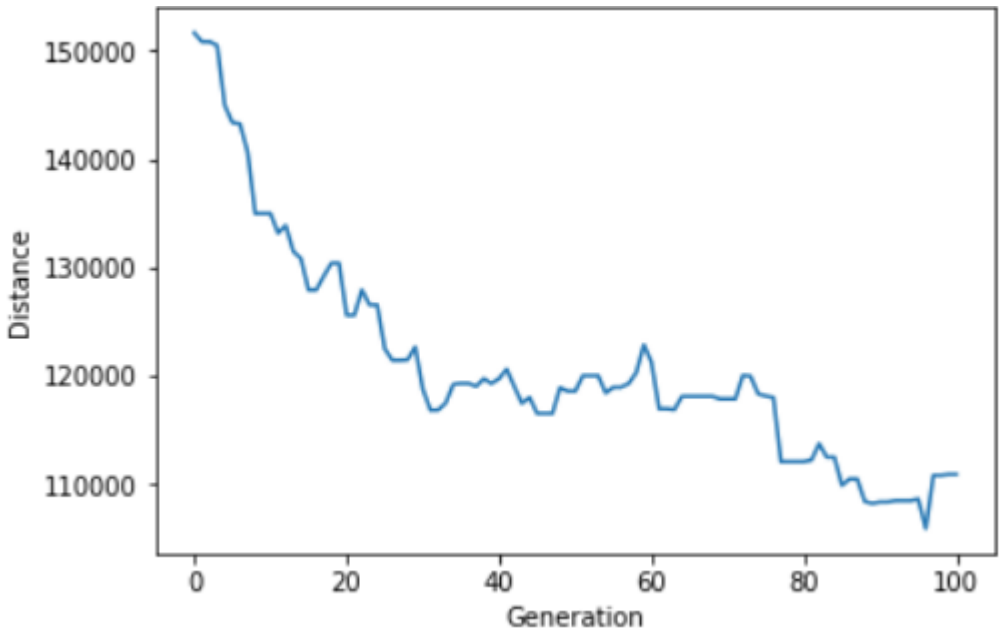
Table 4: Result of Genetic Algorithm for two datasets.

	n=10	**n=100**
shortest path distance	10127.552143541274	nearly equals to 94000 (optimal solution)
total time taken (sec)	9.16734790802002	27.159043073654175

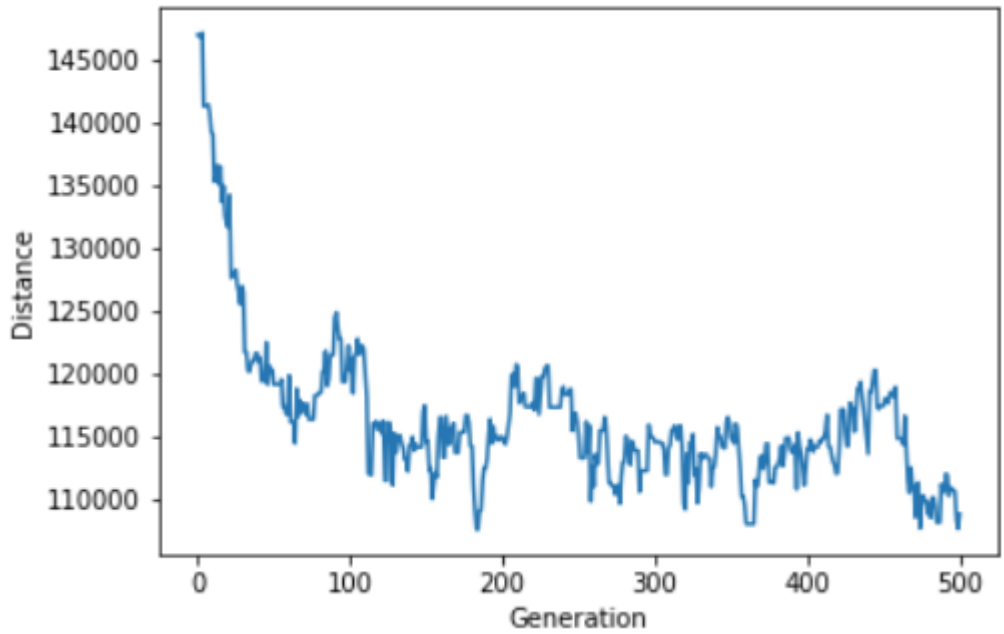
Convergence curve for n = 10:



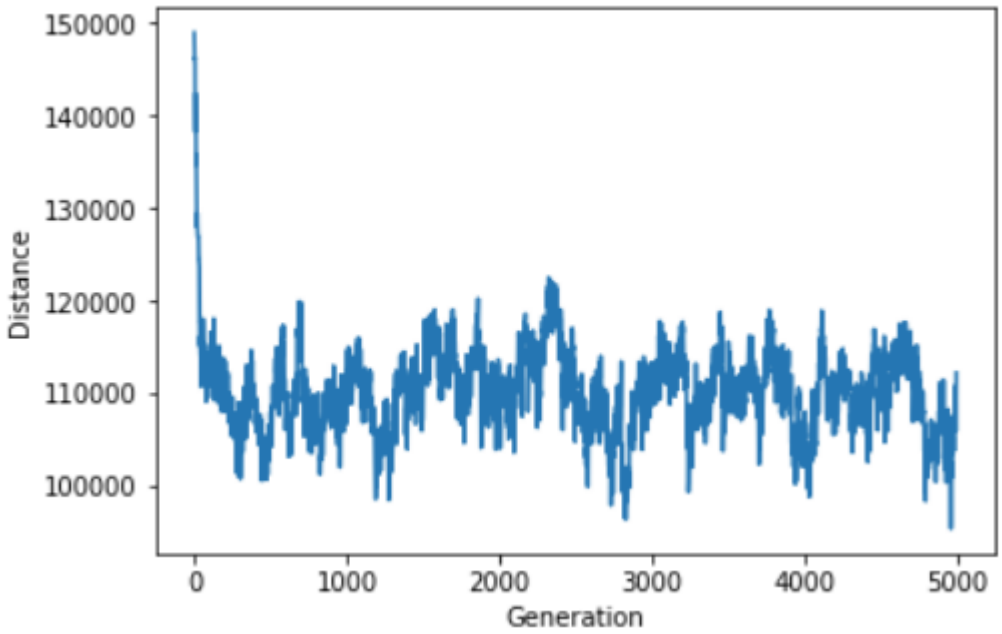
Convergence curve for n = 100:
(When generation = 100)



When generation = 500



When generation = 5000



Result Analysis:

For the datasets having only 10 cities the genetic algorithm curve gives optimal value after 33 generation but for datasets having 100 cities genetic algorithm convergence curve actually didn't converge so well but it gave minima at 188th generation. So this metaheuristic (genetic) algorithm is not so suitable for this type of problem having large number of datasets.

Simulated Annealing Algorithm for TSP:

Simulated annealing is an optimization technique that finds an approximation of the global minimum of a function. When working on an optimization problem, a model and a cost function are designed specifically for this problem. By applying the simulated annealing technique to this cost function, an optimal solution can be found.

I do following procedure to apply simulated annealing algorithm for TSP.

Pseudocode:

1) Choose a random state s and define T_0 and β
2) Create a new state s' by randomly swapping two cities in s
3) Compute $\delta = \frac{C(s')-C(s)}{C(s)}$
    a. If $\delta \leq 0$, then $s = s'$
    b. If $\delta > 0$, then assign $s = s'$ with probability $P(\delta, T_k)$
        i. Compute $T_{k+1} = \beta T_k$ and increment k
4) Repeat steps 2 and 3 keeping track of the best solution until stopping conditions are met

Experiment and Result:

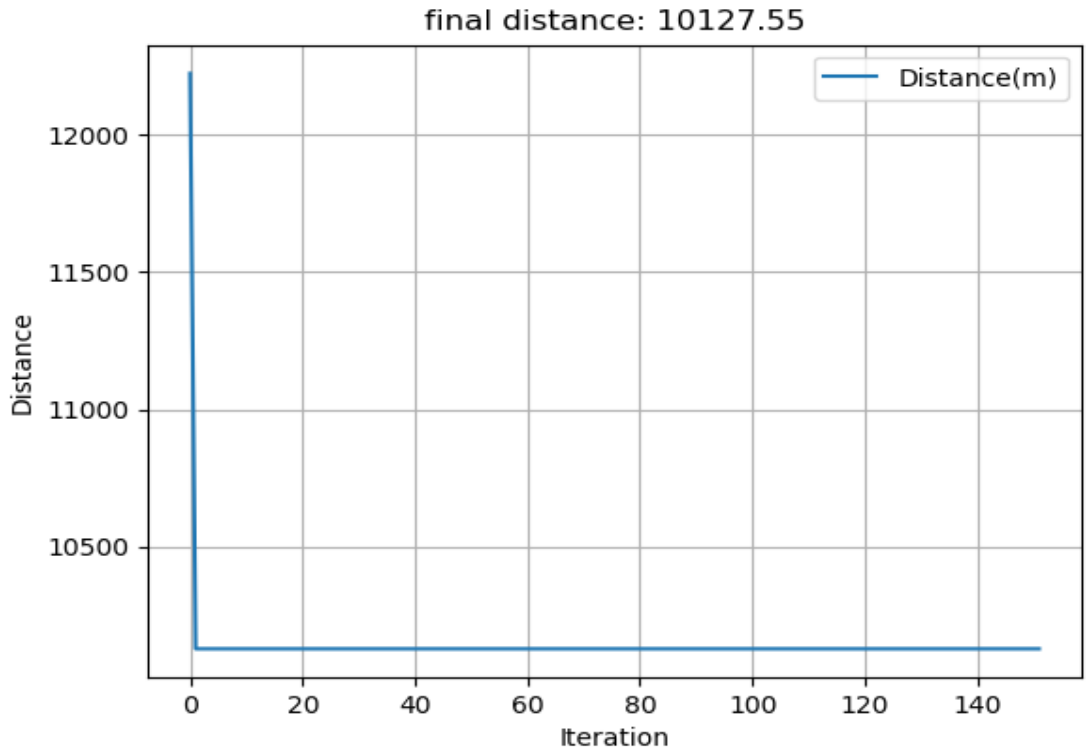
I write the simulated annealing algorithm for the TSP based on the above procedure and run for the provided datasets, time taken for these data sets, the shortest path distance and convergence curve is below:

	n=10	**n=100**
shortest path distance	10127.55	24861.64
total time taken (sec)	0.04402279853820801	9.3510422706604

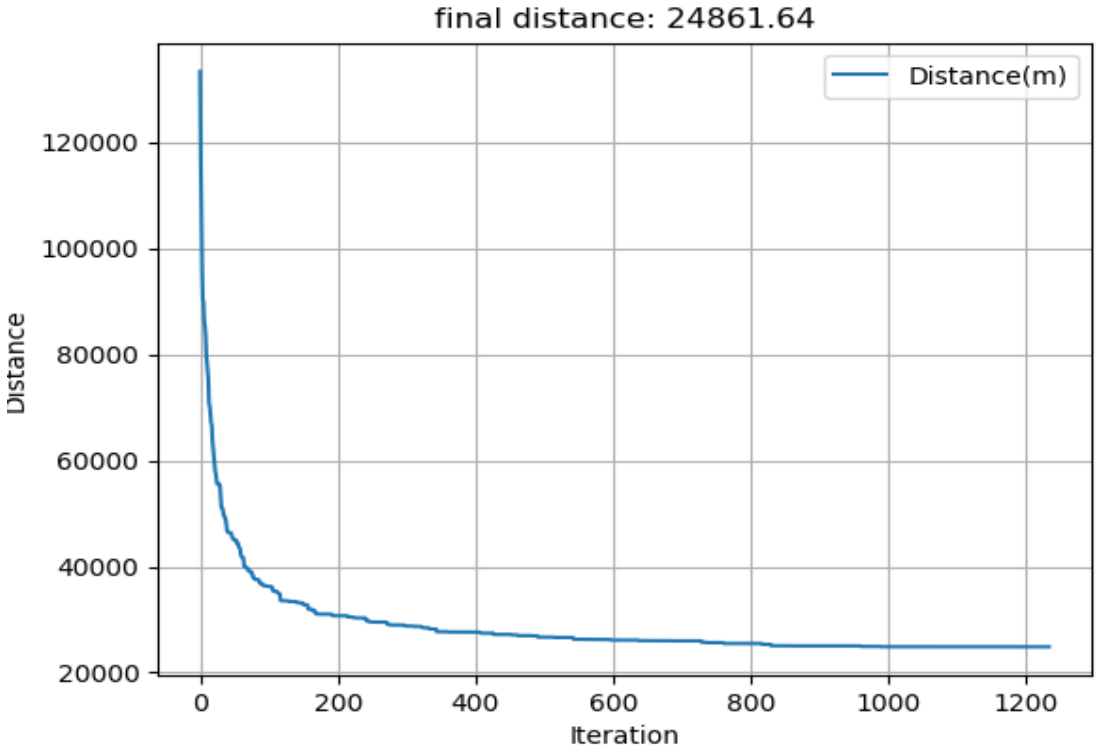
Optimized path for n = 10:
[1 7 6 4 0 9 8 5 2 3]

Optimized path for n = 100:
[72 92 45 17 68 16 61 32 80 97 93 67 95 73 28 22 58 82 90 15 13 8 75 18
10 85 11 2 6 40 31 36 81 5 4 89 51 49 39 54 9 30 57 35 29 77 27 59
46 1 70 76 71 66 78 62 94 98 38 74 25 52 87 42 63 44 50 64 12 56 65 60
83 14 48 20 26 96 53 86 33 88 84 91 19 69 34 3 47 41 37 7 55 24 79 43
99 23 0 21]

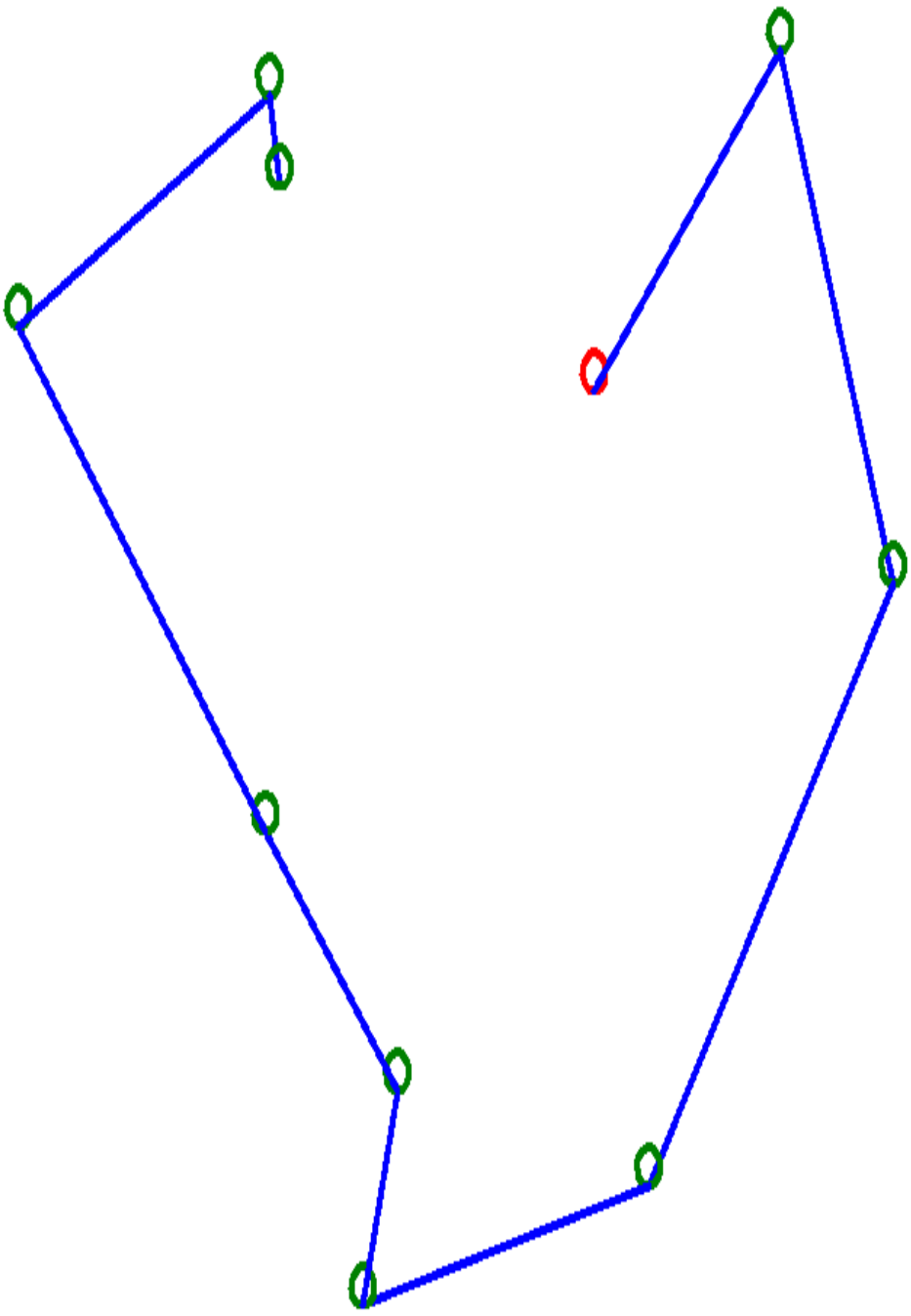
Convergence curve when n = 10:



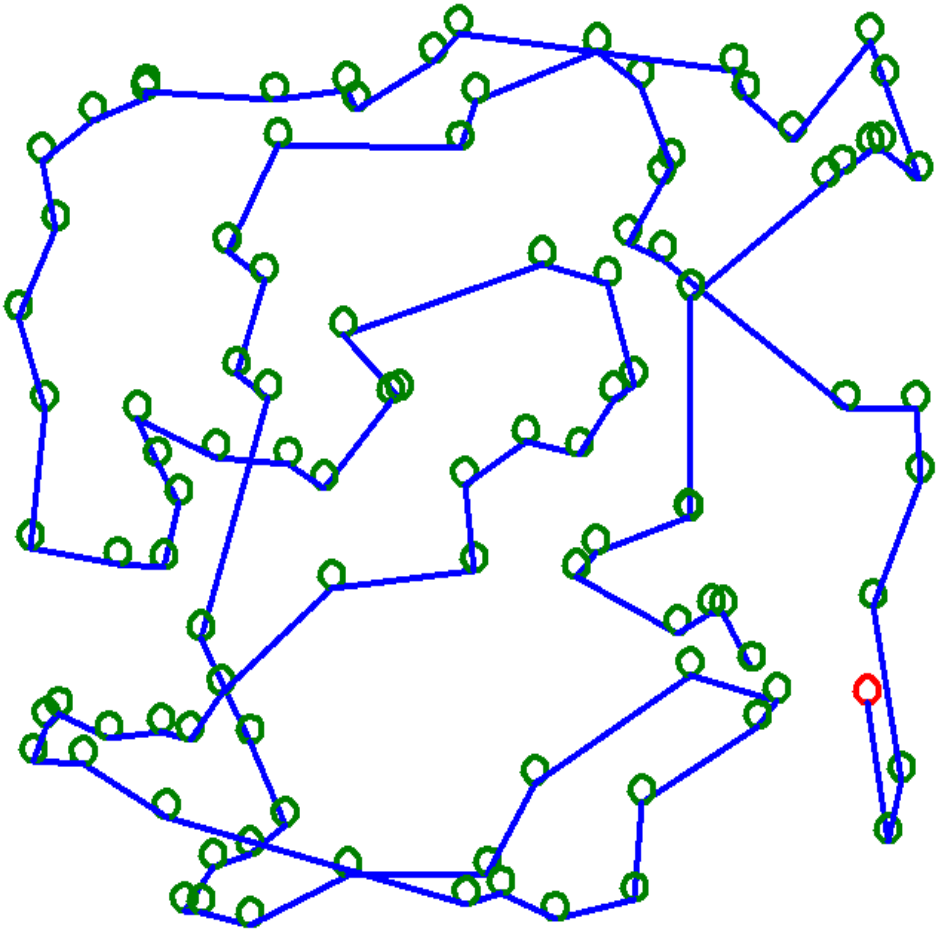
Convergence curve when n = 100:



TSP map for n = 10 by using simulated annealing algorithm:



TSP map for n = 100 by using simulated annealing algorithm:



Conclusion:

I have applied for a total of 5 algorithm for the TSP problem in which simulated annealing works best. The total time cost of simulated annealing algorithm when n = 10; is 0.04402279853820801 second and start to converge from 3rd iteration, when n = 100 the total time cost is 9.3510422706604 and start to give optimal result at 823th iteration. Dynamic and branch and boundaries algorithm didn't converge for the large datasets (n=100), where as genetic algorithm just gives the optimal value but which is not the best optimal value compared to simulated annealing. In my experiment greedy algorithm works fine after simulated annealing giving almost same result with simulated annealing in short time.