CSI - 5137 - AI-Enabled Software Verification and Testing
Assignment - 1

Submitted by - Deepank Kartikey (300247255) & Osama Alghamyan (300218382)

In brief, the traveling salesman problem (also called the traveling salesperson problem or TSP) is about finding the shortest possible route that visits each city exactly once and returns to the origin city when given a list of cities and the distances between each pair of cities.

This assignment aims to solve the traveling salesman problem using a Metaheuristic search algorithm. We implemented the Simulated Annealing algorithm with some modifications using python and tested it with varying sizes of datasets of **TYPE**: TSP and **EDGE WEIGHT TYPE**: EUC 2D. We will discuss the datasets and results after discussing the algorithm.

Algorithm

As discussed in one of the lectures, Simulates Annealing is inspired by the annealing process in metallurgy where metal is heated to a high temperature quickly and then cooled down slowly which increases its strength and makes it easier to work with.

It is a stochastic global search optimization algorithm that makes use of randomness as part of the search process. The usage of randomness makes it perform better on nonlinear optimization problems like TSP where local search algorithms fail to perform well. It also helps in overcoming the problem of local optima that are posed by the famous Hill Climbing algorithm. Hence, we adopted Simulated (named SA further) as one of the famous Heuristic methods to solve the TSP. In figure 1, you can see how random in SA algorithm helps in avoiding local optima situations.

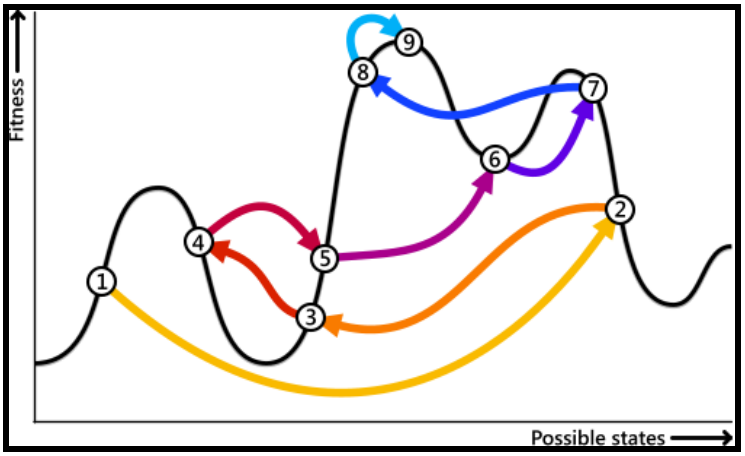

Figure 1

The SA algorithm starts with an arbitrary solution and high value of temperature then iteratively attempts to find a better solution by reducing the temperature and choosing a random solution state, compares the achieved total distance (which is our Fitness Function here in this problem), and does that till reaches a next solution state.

It is worth mentioning that distance between cities is **Fitness Function**, changing arguments that affect the achieved best total distance like cooling rate, mutation rate, and maximum iterations are our **operators**, and all possible paths or state space is our **representation**.

Considering the **cost-benefit analysis**, even though the SA algorithm provides a Metaheuristic view to solve the TSP by randomly selecting a further state(s) it proves to be costly at times. We applied our solution that takes only a mutation of all possible states on a dataset with 1748 cities (vm1748.tsp) and it took almost 20 times more than a dataset with 76 cities (pr76.tsp) to execute and it confirms how costly it could be to apply SA on all paths both timely and computationally wise.

Thus, we optimized our algorithm to extract the most benefit out of the vanilla Simulated Annealing.

Optimization

Instead of selecting a random solution from the complete population, we employed a restrictive random restart to make the process faster. We will show a comparison of results on the basis of a different number of random states selected towards the end. We also applied two different mutation rates, start city, and max iterations in the algorithm to compare the results that we will observe in the result section.

The method *get_random_solution_from_population()* in python implementation, selects a specific number of states randomly and then returns the best out of them based on the fitness (i.e. distance).

Random Restart function gets calculated matrix of cities' distances, home as start city, state as initial state, the maximum number of iterations, and mutation rate which is the number (actually percentage) of cities to be replaced with each other for creating a new state. Eventually, it returns a path with the shortest total distance among those states that the initial point of which was from the state.

Experiment

To test the algorithm, we ran it on two different datasets: **pr76.tsp** and **a280.tsp** where the first one consists of 76 cities and the latter consists of 280 cities to explore the best solution.

As discussed, we applied the SA algorithm with random restart and chose 2 and 20 for start cities (*Home*) with maximum iterations of 500 and 1000 and mutation rates of 0.2 and 0.4.

Home is the first city, *max_iterations* is the number of interactions of the algorithm and *mutation_rate* indicates the percentage of cities that will be replaced with each other. We chose a decent *mutation_rate* as SA is a **global search algorithm** so we ought to find states, **not** in the near neighborhood.

Dataset: **pr76.tsp**

Home= 2

Max Iteration	Mutation Rate	Distance	Time(in seconds)
500	0.2	335890	31.49
500	0.6	448630	35.17
1000	0.2	301039	64
1000	0.6	429484	76.1

Home= 20

Max Iteration	Mutation Rate	Distance	Time(in seconds)
500	0.2	338467	32
500	0.6	426267	37.46
1000	0.2	308501	63.73
1000	0.6	433319	77.95

Dataset: **a280.tsp**

Home= 2

Max Iteration	Mutation Rate	Distance	Time(in seconds)
500	0.2	26642.86	119.56
500	0.6	29349.94	147.94
1000	0.2	26612.38	250.36
1000	0.6	28584.36	285.48

Home= 20

Max Iteration	Mutation Rate	Distance	Time(in seconds)
500	0.2	26442.86	119.95
500	0.6	29219.39	140.74
1000	0.2	26595.24	260.86
1000	0.6	28729.64	286.74

Results

As shown in tables, increasing max iterations of the algorithm from 500 to 1000 led to a better or equal solutions with an exception of Home = 20 in a280.tsp dataset. When mutation rates are decreased to reduce the size of random search space it also has a positive effect in all cases of both the datasets with Home = 2 and 20.

Hence our algorithm tends to perform better when iterations are high and search space is limited to an extent (still searching randomly within the restricted search space).