Travelling Salesman Problem using Genetic Algorithm

Apurv Priyam
apriyam3@gatech.edu
Georgia Institute of Technology

1 TRAVELLING SALESMAN PROBLEM

The Traveling Salesman Problem (TSP) is a problem arising in operations research and computer science. The TSP is widely studied due to its many applications to practical problems in STEM fields. Despite being a fundamental problem in research since its first formulation in the 1930's, the TSP remains to be solved completely - there currently does not exist a method for producing an exact solution in polynomial time.

Intuitively, the TSP is the following: given a list of cities and their locations, find the shortest tour which visits all cities exactly once and returns to the starting city.

Formally, this can be stated as, given a graph consisting of edges and vertices, $G = (V, E)$, if each edge $e = (v_i, v_j)$ has corresponding weight w_{ij}, find the cycle that minimizes the sum of weights such that each node is visited exactly once. In graph theory, a cycle that visits each node exactly once is referred to as a Hamiltonian Cycle. Thus, the TSP is equivalently to find the minimum Hamiltonian Cycle. the TSP is an integer linear programming problem:

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

$$\text{s.t.} \quad \sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad \forall j \in \{1, ..., n\}$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad \forall i \in \{1, ..., n\}$$

$$\sum_{i \in S} \sum_{j \in S}^{n} x_{ij} \leq |S| - 1 \qquad \forall S \subsetneq \{1, ..., n\}, |S| \geq 2$$

$$x_{ij} \in \{0, 1\}$$

$$\text{where } \mathrm{x}_{ij} = \begin{cases} 1 & \text{if edge } (v_i, v_j) \text{ is included} \\ 0 & \text{else} \end{cases}$$

The objective function is to minimize the sum of weights for chosen graph edges. Constraints 1 and 2 ensure that each city is visited only once. Constraint 3 ensures that the constructed tour returns to the starting point last.

1.1 Classification

Presentation of the optimization form of the TSP lends itself to a mention of the difficulty of TSP. The TSP falls into the NP-complete class of problems. Without providing a formal proof, this can be easily seen due to the relation that the TSP has to Hamiltonian cycles. The TSP is simply a special case of the Hamiltonian cycle problem. It is well known that finding a Hamiltonian cycle in a graph is an NP-complete problem, and thus it can be shown via a reduction that the TSP also falls into this class.

Due to its NP-completeness, there does not exist an algorithm to efficiently and exactly solve the TSP. A naive approach to solve the problem would be with brute force, however, the number of computations expands exponentially as the number of cities increases - even with a few hundred cities, total enumeration could take a computer years to solve. For this reason, and because of the TSP's wide applicability, there have been many different approaches studied for finding near-optimal solutions efficiently.

1.2 Solution Approache - Local Search (Genetic Algorithm)

Although there exist many approaches for solving the TSP, this section will detail one class of approache used.

- Local Search: Local search methods in many cases fall in between exact methods and construction heuristics in regards to balancing increased computation and increased solution quality.

  The idea of local search methods is to iteratively improve the solution by searching within a given neighborhood. This is done until the algorithm exceeds its allotted computation time/iterations or there is no improving solution in the current neighborhood.

  Generally speaking, a problem's neighborhood is the set of solutions differing from the current solution in some specified way. For example, with the local search algorithm 2-OPT, the neighborhood is the set of all possible tours that could be constructed after removing two particular edges in the current tour. Many local search algorithms exist, each with their own way of defining a neighborhood.

2 RELATED WORK

2.1 TSP Local Search - Genetic algorithm

Many research has been done in the field of genetic algorithm due to its flexibility to use in different types of problems. Therefore, there are noteworthy publications that review genetic algorithm and give a comprehensive overview of the field. Getting influenced by the Darwin's theory of evolution, John Holland introduced genetic algorithm in 1960. Afterwards, it was extended by his student in 1989.

One of the earliest significant contributions was made by Fogel [**?**] in 1988. He points out the importance of crossover operator in the genetic algorithm and emphasizes that without crossover, the algorithm wouldn't be more than a random search. Fogel also warns of the over-ambitious mutation operators that could destory the link between offspring and parent but could lead to more exploration.

Many contributions have also been made in solving TSP through genetic algorithm by introduction of various types of crossover operator. Goldberg and Lingle [**?**] proposed partially mapped crossover in the year 1985. The order crossover (OX) was proposed by Davis [**?**] in 1985. Both of these methods have been tried in this paper and detailed information is provided in Algorithm section.

Various attempts have also been made to create hybrid genetic K-means algorithm [**?**] which aims to divide the large scale TSP problems into smaller sub-problems using clustering and then use genetic algorithm to find the shortest path for each sub-problems. Finally, the sub-groups are combined into one using an effective connection method to give the result for the main problem.

3 ALGORITHMS

3.1 Local Search: Genetic Algorithm

A genetic algorithm (GA) is a metaheuristic method for solving optimization problem inspired by the process of natural selection and 'survival of the fittest'. In a genetic algorithm, a group (called population) of candidate solutions (called individuals or chromosomes) to an optimization problem is evolved in each generation. Weak individuals are killed and stronger ones are given more chance to breed. A typical genetic algorithm has a genetic representation of the solution and a fitness function to score the solution. A standard representation of solution is an array of bits. Traditionally, solutions are represented in binary as strings of 0s and 1s, but encoding of solution can change depending on the problem, just like Travelling Salesman Problem.

A travelling salesman problem can be formulated as a genetic algorithm with following definitions:

- Gene: City (represented as their id)
- Individual (chromosome): A string of genes representing one route satisfying the constraints of TSP
- Population: A group of individuals
- Fitness function: A function which tells how good a solution is (the total distance for TSP)
- Parents: Two routes selected for mating based on their fitness
- Mating pool: A collection of parents selected for creating next generation
- Mutation: A way to bring variation in the solution (by swapping two cities in the route)
- Crossover: Substrings of two parents are interchanged to create two new routes.

The algorithm starts with a initially generated population of sequences. These individuals can be generated randomly or a small number of individuals can be generated using some heuristics to speed up the convergence. During each generation (or iteration), three operations are generally performed in sequence with some probabilities - selection, crossover, mutation. While the selection does the exploitation of good chromosomes, mutation and crossover do the exploration. Figure 1, shows a typical flowchart of genetic algorithm.

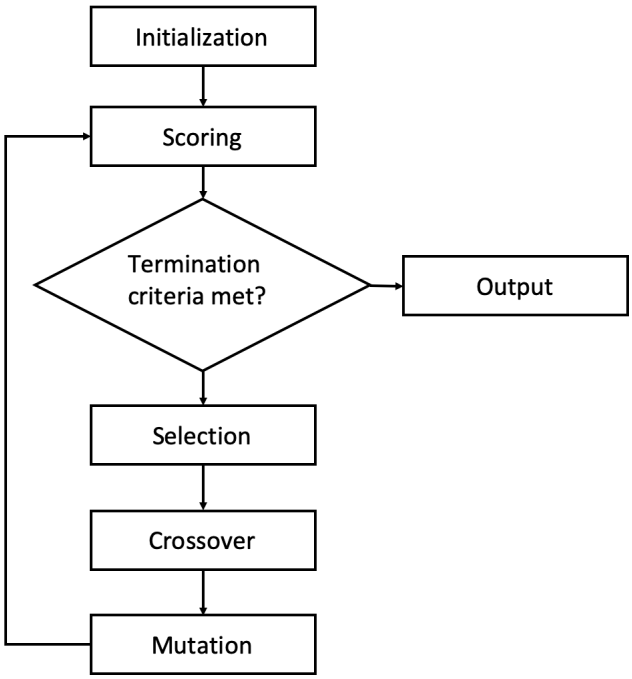


Figure 1: Steps of genetic algorithm

3.1.1 Initial population generation. While the most simplest method to generate the initial population is to generate the individuals randomly, several better methods have been proposed. Such methods uses algorithms like K-Means clustering to create the initial population. For this problem a simple Nearest Neighbour has been implemented to create the initial few individuals. To maintain the diversity in population only few individuals are being created using nearest neighbour.

3.1.2 Genetic Operators.

(1) Selection: Selection operation is done on the population to select the fittest sub-population which will generate the offspring. Fitness value can be used as a criteria to judge which individuals are more fit. There are many methods developed on how to select the best parents, but in this project two of the most common methods have been implemented -
   (a) Roulette wheel selection: Probability of a individual to get selected is directly proportional to its fitness value. If f_i is the fitness of individual i in a population with N individuals, then its probability of selection will be

$$\frac{f_i}{\sum_{n=1}^{N} f_n} \tag{1}$$

   Table 1 shows the probability of 5 hypothetical chromosomes and figure 2 shows the probability with roulette wheel selection method.
   (b) Rank based selection: Roulette's wheel selection criteria will have challenges when the difference in fitness values is

Table 1: Table shows the fitness value and probability of selection for 5 routes

Route	Fitness value	Probability
Route 1	5	5%
Route 2	5	5%
Route 3	10	10%
Route 4	15	15%
Route 5	65	65%

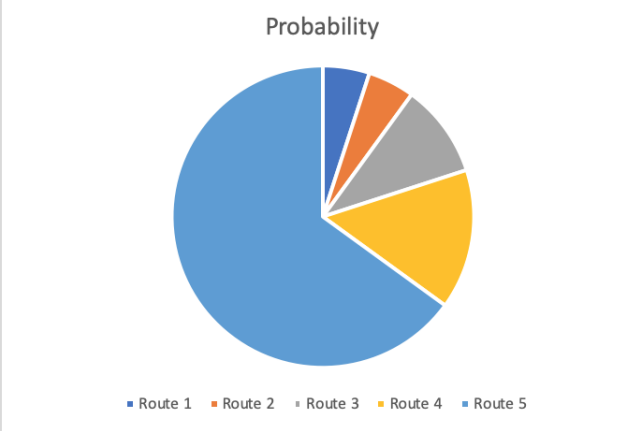


Figure 2: Probability distribution with roulette wheel selection

Table 2: Table shows the updated fitness value and probability of selection for 5 routes

Route	Fitness value	Rank	New fitness	Probability
Route 1	5	5	1	6.7%
Route 2	5	4	2	13.3%
Route 3	10	3	3	20.0%
Route 4	15	2	4	26.7%
Route 5	65	1	5	33.3%

very high among the individuals. For example, if the fittest individual has its probability of 90% then the remaining individuals will have very small chance of selection. The rank based approach sorts all the individual by their ranks and assign probability based on the new fitness value, which is their rank. The worst with rank N will have fitness of 1, and strongest individual with rank 1 will have fitness value of N. After this all chromosomes will have some chance to get selected.

Since, the initial random populations generated in this project had distance of almost 10 times the optimal value, this approach was also tried, and found to perform better with faster convergence.

(2) Crossover: The new generation is genetic algorithm is generated from the existing population by a set of operators -

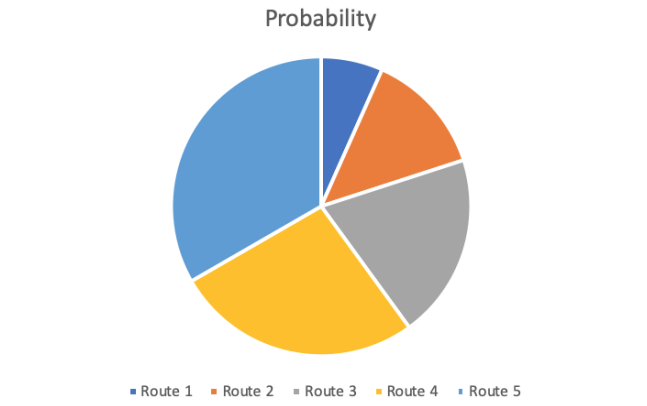


Figure 3: Probability distribution with rank based selection

crossover and mutation. The crossover operator produces two new individuals from two parents by copying substring from each parents. In a typical crossover, the gene at position i in each offspring is copied from the gene at position i from either of the parents. After the mating pool is selected, two parent chromosomes is selected from the pool and based on a probability, called crossover probability, crossover operation is performed on them. Generally this crossover probability is kept high (0.6-0.8) to ensure that there is enough chance for a new offspring to be created which holds good genes of both the parents. In this project four types of crossover were implemented and tried -

- Single point with PMX crossover
- Single point with OX crossover
- Two points with PMX crossover
- Two points with OX crossover

Each of the four types of crossover is explained below:

(a) Single point crossover: A point on both parents' chromosomes is picked randomly, and designated a 'crossover point'. Genes to the right of that point are swapped between the two parent chromosomes. This results in two offspring, each carrying some genetic information from both parents.

(b) Two points crossover: In two-point crossover, two crossover points are picked randomly from the parent chromosomes. The bits in between the two points are swapped between the parent organisms.

Since not all the individual presents a feasible solution in a TSP problem due to presence of constraints, 2 specialized crossover techniques were tried along with single/two points crossover:

(c) partially mapped crossover (PMX) crossover: The crossover points are decided and the genes between these points from each parents are copied to the two new offspring (From parent P1 to offspring O2 and P2 to O1). The remaining genes in O1 are taken from P1 such that no city is repeated. If a city from P1 is already present in O1, the city

is taken from the interchanged genes which were transferred to O2 from P1. Similar process is repeated for O2. Figure 4- 7 shows the steps of PMX.

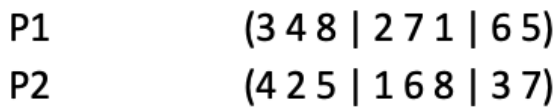


Figure 4: PMX step 1 - Two parents

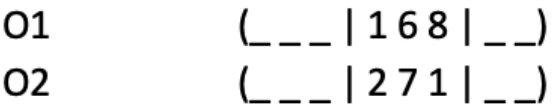


Figure 5: PMX step 2 - Genes from the crossover points are copied to two offsprings (P1 to O2 and P2 to O1

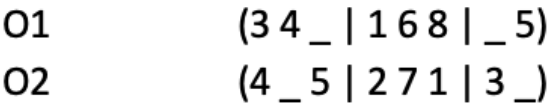


Figure 6: PMX step 3 - Genes are copied from parents such that they are not copied (P1 to O1 and P2 to O2

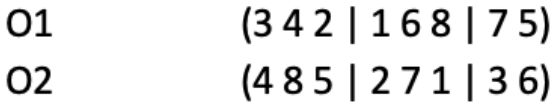


Figure 7: PMX step 4 - Remaining genes are filled (O1 are filled with whatever cities were transferred to O2 from P1

(d) Ordered (OX) crossover: After the crossover points are decided, the genes between these two points are copied to new offspring (from parent P1 to offspring O2 and P2 to O1). Since O1 has the genes from P2, the rest are filled from P1 in the order they appear in P1, as long as the repetitions are avoided. Then same process is followed for O2. Figure 8- 10 shows the steps of OX.

(3) Mutation: Typically with 0s and 1s encoded individual, the mutation operator randomly changes 0 to 1 and vice-versa. But since for TSP, the constraints needs to be satisfied that one city should only be visited one, the mutation operation swaps two genes (city) in the individual (route) with some small probability, called the mutation probability. Figure 11 shows the mutation process.

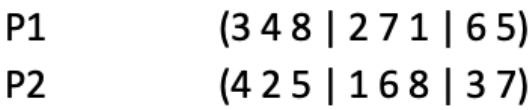


Figure 8: OX step 1 - Two parents

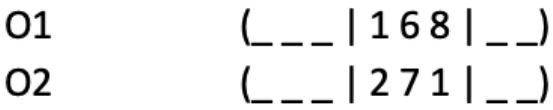


Figure 9: OX step 2 - Genes from the crossover points are copied to two offsprings (P1 to O2 and P2 to O1

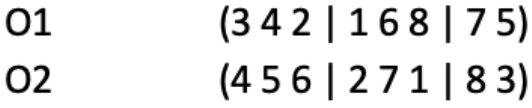


Figure 10: OX step 3 - Genes are filled in order they appear in parent without repetition. If repetition is found next gene is used (P1 to O1 and P2 to O2

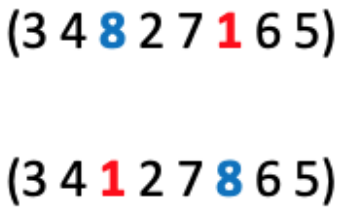


Figure 11: Step showing mutation between reg and green colored genes

3.1.3 Fitness function. Since the probability of selected increases as the fitness value increases, the minimization problem needs to be converted in to a maximization problem. The fitness function was taken as the inverse of the total distance of the route. Since no route will have 0 distance, this approach was suitable. One other method was also tried where a diversity factor was included in the fitness score. The weight of diversity in the fitness score was reduced after each iteration like simulated annealing. During the initial generations, With high diversity weight, the algorithm forced the individuals to spread across the solution space. As the generations progressed more weights on distance were put to converge on minimum distance solution. This approach only gave marginal better

Table 3: Result comparison after including randomly generated individuals in each iteration

TSP	Error %	# Generations with (without) random sequences
Atlanta	0.6%	109 (126)
UKansasState	0.0%	8 (19)

result but increased both the complexity and run-time, therefore diversity part was removed.

3.1.4 Elitism. After crossover and mutation, there is a big chance that we may loose our current best route (individual). Elitism ensures that the top chromosomes from each generation are copied to the next generation's population without any change. This increases the performance of GA, because it prevents losing the best found solution in each generation. But since having high elite individuals in genetic algorithm can reduce the exploration and cause the algorithm to stuck in a local optimal solution, only a small part of new population is made through elitism.

3.1.5 Next Generation. New population for the next generation were created which had three types of individual:

- Elite chromosomes: Some of chromosomes with highest fitness values were transferred to the next generation
- Newly generated chromosomes were selection based on their fitness value
- Some newly randomly generated chromosomes were added in each iteration to help in exploration. The count of such chromosomes was decided based on the concept of simulated annealing:

$$N_3(i) = round\left(\frac{0.2N}{i^k}\right) \tag{2}$$

Where N is the population size and $N_3(i)$ is the number of chromosomes randomly generated in the generation i and i is the generation number. The value of k was decided on trial and error the final value was selected as 0.2. This improved the result by a small amount. Table 3 shows the comparison for two cities:

3.1.6 Pseudocode and Implementation. The python implementation was done by creating three classes - GeneticAlgorithm and Chromosomes and Node. The GeneticAlgorithm class has following parameters and objects:

- mut_prob: Mutation probability (Numeric)
- cross_prob: Crossover probability (Numeric)
- pop_size: Population size (Integer)
- Populaiton: List of Chromosomes class objects (List of Chromosomes of length pop_size)
- fitness: List of fitness value of all chromosomes (List of length pop_size)
- nodes: List of node id (List of length equal to number of nodes)
- length: Length of chromosomes (Integer)
- distanceMatrix: Dictionary with distance between each nodes
- mutation_type: Only accepts 'swap' (Dictionary of size square of number of nodes)
- crossover_type: Type of crossover (PMX or OX) (String)
- crossover_points: Crossover points (1 or 2) (Integer)
- new_population: New population generated after crossover and mutation. List of Chromosomes class objects (List of Chromosomes of length pop_size)
- new_population_fitness: List of fitness values of new_population (List of length pop_size)
- elite_num: Number of elite chromosomes (elitism) (Integer)
- num_generation: Number of generations to be performed (Integer)
- verbose: Print results at each iteration (0: Don't print, 1: print iteration number and best fitness, 2: print iteration number, best and average score) (Integer)
- seed: Seed value (taken from input) (Integer)
- seed_vals: New sets of seed generated using seed (List of integers)
- smart_pop: Number of initial chromosomes generated using heuristic (Integer)
- cutoff_time: Allowed running time (taken from input) (Numeric)
- tsp_name: TSP instance name (taken from input) (String)
- trace: Dictionary which stores trace of algorithm (Dictionary)
- prev_best: Stores previous best result (Numeric)

The Chromosome class following object and parameters:

- length: Length of sequence (Integer)
- nodes: List of nodes id (List of length equal to number of nodes)
- sequence: Route (List of length equal to number of nodes)
- fitness: Fitness value of the route (Numeric)

The Node class has following objects and parameters:

- id: ID of the node (Integer)
- x: X co-ordinate (Numeric)
- y: Y co-ordinate (Numeric)

Algorithm 1: Genetic Algorithm

```
GA:
    nodes = [1, 2, ..., n]
    distanceMatrix
    Step1: Initial Population Generation
    Step2: Calculate fitness of each individual
    for i ≤ number of generation do
        if Stopping criteria is met then
            STOP
        else
            Step3: Selection (Select mating pool)
            Step4: Select 2 parents from mating pool. Based
              on crossover probability do crossover. Repeat
              for all individuals in the pool.
            Step5: Mutate all crossovered individuals based
              on Mutation Probability.
            Step6: Calculate fitness of newly generated
              sequence.
            Step7: Create population for next generation.
        end
    end
    return Best sequence with minimum distance
```

3.1.7 *Complexity Analysis.*

(1) Time complexity: The process is divided into many parts which runs multiple chunks of code in sequential format. Different parts of algorithm implementation along with thier time complexity:
Notations: P is the population size, N is the sequence length (number of nodes), G is the number of generations.
   (a) Step1 - Initial population generation: One initial random chromosome generation require $O(N)$ time. Therefore for the complete population the time complexity is $O(PN)$
   (b) Step2 - Calculating fitness of initial population: We have to sum all the consecutive pairs of cities in a city, therefore one chromosome will take $O(N)$ time. Total time complexity $O(P * N)$
   (c) Step3 - Selection: Selection is further divided into following parts:
      - Store fitness of all chromosome in a list in loop - Time complexity $O(P)$
      - Calculate probability of each chromosomes using fitness value (sorting of all the fitness values is required which is most time consuming in this step) - Time complexity $O(PlogP)$
      - Create mating pool (To select one individual for the pool, one random number needs to be generated and two search needs to be performed in the sorted fitness scores list) - Time complexity for whole population is $O(P * logP)$
   (d) Step4 - Crossover: For each pair of parents, for loops are required which traverse over all the genes in the sequence. The number of loops is constant and doesn't change with problem size - Total time complexity for whole population is $O(P * N)$

Table 4: Run time analysis

P	N	G	Time(s)
100	20	100	6.7
100	20	200	12.8
100	20	1000	70.1
100	30	100	9.4
100	170	100	65.2
200	20	100	14.6
1000	20	100	83.5

   (e) Step5 - Mutation: For one chromosomes, maximum N genes can be mutates - Total time complexity for whole population is $O(P * N)$
   (f) Step6 - Calculating fitness of modified population: Same as step2 time complexity - Time complexity $O(P * N)$
   Since all the above mentioned steps are in sequence, the total time complexity of 1 generation is $O(PN * logP)$ only. For G generations the time complexity will be $O(GPN * logP)$. Table 4 shows time analysis on different values of P, G, and N.
   The run times in table 4 shows almost linear increase in time with increase in all the parameters. Although increase in time when P is increased in slightly more than when other are increased.
(2) Space complexity: We can calculate the total space complexity by looking into individual classes space complexity:
   - Node class: Holds three numeric values ($O(1)$). Total space complexity - $O(N)$
   - Chromosomes class: Holds 3 numeric values and two lists of length N. Space complexity for 1 chromosome- $O(N)$
   - GeneticAlgorithm class: Holds 12 numeric, 3 strings, 5 lists of numeric values of length N or P, 2 lists having P Chromosomes class objects ($O(P * N)$) and 1 dictionary to store the trace values ($O(G)$ because this will have maximum of G values). Total space complexity - $O(G + PN)$

   Therefore, total space complexity will be $O(G + PN)$

3.1.8 *Strengths and Weaknesses:*

(1) Strengths: Genetic algorithm is suitable when the problem is very complex with very tough mathematics formulations. It covers large solution space simultaneously if formulated correctly. Although genetic algorithm doesn't guarantee global optimal solution, if the problem has many local optimal solutions, genetic algorithm can check most of them simultaneously to find the best among them. It can also support multiple-objective functions.
   Since the operators of genetic algorithm are modular, it can solve various types of problems by changing its operators, encoding and parameters individually.
(2) Weakness: One of the main weakness that genetic algorithm faces is that it is sensitive to the initial population. Similarly generated initial populations can easily converge the genetic algorithm to a local optimal solution. Therefore, genetic algorithm can't guarantee optimal solution and it is

very common to get sub-optimal solution. It is also difficult to prove the optimality of genetic algorithm solution unless we have a very good lower bound which matches the result of genetic algorithm. Unlike simpler local searches, genetic algorithm has more parameters including different probabilities and operators' types. Which makes it difficult to tune and select the best performing parameters and operators. Modeling a genetic algorithm also becomes difficult when the problem has constraints. In such cases, either the operators and chromosome's encoding can be changed (like TSP) or some penalty can be added to the objective function, which again makes it difficult.

4 EMPIRICAL EVALUATIONS

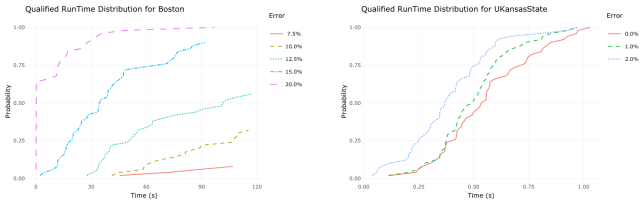
4.1 Genetic Algorithm

The genetic algorithm was implemented in Python3. The experiments were performed on a personal laptop with 8 GB of RAM and 2.7 GHz Intel Dual Core i5 processor. Since the running time varied even for the same dataset, 10 runs were made for each dataset with different random seed and average error and time were recorded. Comprehensive table 5 shows the average time, solution quality, relative error and selected cutoff-time for genetic algorithm performed on difference dataset. Cutoff-time was selected based on the size of the dataset, cutoff was set at 15 seconds for dataset with less than or equal to 20 nodes. The mutation and crossover probabilities were kept at 0.1 and 0.8 respectively with population size of 100. Maximum generations allowed were 1000, but for majority datasets iterations stopped because of cutoff-time. The probabilities were selected which gave overall minimum error over multiple runs for different datasets.

Table 5: Results of GA iteration on different datasets. Cutoff-time was selected based on dataset size.

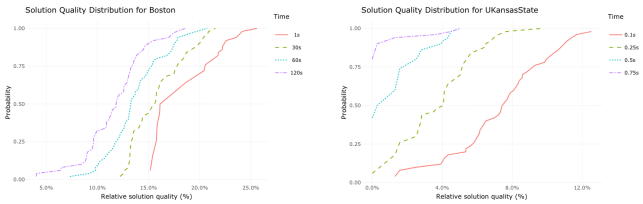
Dataset	Time(s)	Sol.Qual.	RelError	Cutoff (s)
Atlanta	14.67	2031666	0.014	15
Berlin	30.00	8541	0.132	30
Boston	30.00	1020882	0.143	30
Champaign	30.00	61978	0.177	30
Cincinnati	3.66	278461	0.002	15
Denver	30.00	118401	0.179	30
NYC	30.00	1850087	0.190	20
Philadelphia	15	1567328	0.123	30
Roanoke	30.00	786297	0.200	30
SanFrancisco	30.00	927482	0.145	30
Toronto	30.00	1263996	0.075	30
UKansasState	0.57	62962	0.000	15
UMissouri	30.00	157746	0.189	30

Since analyzing accuracy with respect to time for local search is difficult and depends on initialization, Qualified RunTime Distribution (QRTD), Solution Quality Distribution (SQD) and Box plots for running times were added for 50 runs for the dataset UKansasState and Boston. Figure 12 shows the Qualified RunTime Distribution (QRTD) for the two dataset for different values of error. Figure 13 shows the Solution Quality Distribution (SQD) for different values of runtime. Figure 14 shows the runtime box plot for different level of error. As we increased the error threshold, the box plots went down indicating lower runtime requirement to get lower error.



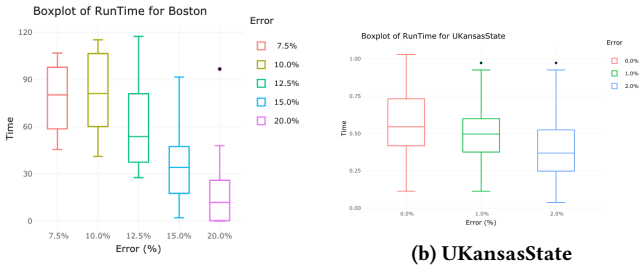
(a) Boston **(b) UKansasState**

Figure 12: QRTD plots for the two dataset



(a) Boston **(b) UKansasState**

Figure 13: RSQ plots for the two dataset



(a) Boston **(b) UKansasState**

Figure 14: Box plots for the two dataset

It was observed that for smaller datasets like Atlanta, UKansasState and Cincinnati almost 0 error rate can be achieved in few seconds. For medium size dataset like Philadelphia, Toronto, Berlin the algorithm gave minimum of of 5-8% error with cut-off time of seconds. And for Boston and SanFrancisco alrogrithm was able to give minimum error of just above 10% in 30 seconds. For some runs on the larger datasets, the algorithm failed to performed better than Nearest Neighbour in under 30 seconds as the result failed to improve after the initial population generation using Nearest Neighbour.