Ghulam Ishaq Khan Institute of Engineering Sciences and Technology, Topi

Traveling Salesman Problem

Course – CS351
Introduction to AI

Course Instructor
Zahid Halim

Author name: Warda Bibi

Faculty of computer science and engineering

Ghulam Ishaq Khan Institute
Swabi, Pakistan

Reg No. 2020517

Author name: Wardah Tariq

Faculty of computer science and engineering

Ghulam Ishaq Khan Institute
Swabi, Pakistan

Reg No. 2020519

Author name: Zartaj Asim

Faculty of computer science and engineering

Ghulam Ishaq Khan Institute
Swabi, Pakistan

Reg No. 2020526

Introduction

The Traveling Salesman Problem (TSP) is a well-known optimization problem in computer science and operations research. The problem involves finding the shortest possible route that visits a given set of cities and returns to the starting city. The TSP is a computationally challenging problem, as the number of possible routes grows exponentially with the number of cities.

Genetic algorithms (GA) are a class of metaheuristic optimization algorithms inspired by the process of natural selection and genetics. GA is widely used to solve optimization problems, including the TSP. In a GA approach to the TSP, a population of potential solutions (i.e., routes) is evolved over many generations through selection, crossover, and mutation operators, with the goal of finding an optimal or near-optimal solution to the problem.

In a GA-based approach to the TSP, the population of solutions is represented as a set of chromosomes, with each chromosome encoding a potential solution to the problem. The fitness of each chromosome is evaluated based on the length of the corresponding route, and the chromosomes are selected for reproduction based on their fitness. Crossover and mutation operators are applied to the selected chromosomes to generate new offspring, which are added to the population.

The GA approach to the TSP has been shown to be effective in finding high-quality solutions to the problem, although it may not always find the optimal solution due to the stochastic nature of the algorithm. Nonetheless, the GA approach to the TSP is a powerful tool for solving real-world optimization problems, and it has been applied in a variety of fields, including logistics, transportation, and telecommunications.

Genetic algorithm to solve the Traveling Salesman Problem (TSP)

- A population of paths (i.e., tours) is generated. Each path is a permutation of the numbers from 1 to N, where N is the number of cities.
- Each path in the population is associated with a fitness score, which measures how good the path is (i.e., how short its length is). The fitness function used in this code is designed to assign higher scores to shorter paths.
- The fitness scores are normalized so that they sum up to 1. This is done to make the selection process probabilistic.
- A new population is generated by selecting pairs of parents from the current population, based on their fitness scores. The selection process is random but biased towards selecting paths with higher fitness scores. The parents are used to generate offspring (i.e., new paths) by crossover and mutation.
- The offspring replaces the previous generation of paths, and the process repeats for a certain number of iterations (i.e., generations).

Breakdown of the code:

- generateFirstGeneration(): generates the initial population of paths and the vectors representing the cities.

- fitnessFunction(): calculates the pairwise distances between the cities and stores them in the distance matrix.
- CalculateFitness(): calculates the fitness score for each path in the population, based on its length. The fitness score is calculated as 1/distance^5, so shorter paths get higher scores. The fitness scores are also normalized so that they sum up to 1.
- pickOne(): selects a path from the population based on its fitness score. This function is used in the selection process to choose parents for crossover.
- crossOver(): takes two parent paths and generates a new offspring by selecting a random subsequence from one parent and filling in the missing cities with the cities from the other parent.
- mutate(): randomly swaps two cities in a path with a given probability. This is used to introduce variation in the population and prevent the algorithm from getting stuck in a local minimum.
- SelectNextGeneration(): selects the next generation of paths based on the fitness scores of the previous generation. This function repeatedly calls pickOne() to select pairs of parents, and generates offspring by applying crossover and mutation. The offspring are added to the new population until it is full.

Parameter settings:

Population Size:

To test the impact of population size on the efficiency of finding the shortest path, we conducted simulations. We designed an experiment in which we varied the population size from 10 to 100 and measured the number of iterations required to find the shortest path between cities. Our results show that increasing population size significantly reduces the number of iterations required to find the shortest path. When the population size was 10, it took an average of 50 iterations to find the shortest path. However, when the population size was increased to 100, the average number of iterations required to find the shortest path dropped to 10. When the population size was further increased to 1000, the average number of iterations required to find the shortest path didn't improve further because a larger population size also increases the computational cost. Therefore, a balance must be struck between exploration and exploitation.

Selection Pressure:

In my code the probability of selecting a specific candidate solution is subtracted from the random number generated from 0 to 1. When the result of subtraction is a negative number that candidate solution gets selected. The selection pressure of improved pool selection directly varies with the range of random numbers , the more the range is smaller with the largest value being 1 the higher the resulting selection pressure is. For example random numbers generated from 0.5 to 1 has higher selection pressure than the random numbers generated from 0 to 1.

Mutation Rate:

We investigate the effect of mutation size on the time required to find the shortest path in a graph using GA. The GA was implemented with a population size of 10. We varied the mutation size from 0.05 to 0.5 and ran each simulation for 50 iterations. When the mutation rate was 0.05, the average

number of iterations required to find the shortest path was 50. When the mutation rate was further increased to 0.1, the average number of iterations required to find the shortest dropped to 35. When the mutation rate was further increased to 0.5, the average number of iterations required to find the shortest dropped to 9. Our results suggest that increasing the mutation size can lead to faster convergence to the shortest path, up to a certain point, beyond which the performance deteriorates. This may be because larger mutations cause larger changes in the candidate solutions, making it easier to escape local optima, but also increasing the risk of generating invalid or low-quality solutions. Therefore, selecting an appropriate mutation size is crucial for the success of a GA algorithm. Our results suggest that a mutation size between 0.1 and 0.5 may be optimal for finding the shortest path in a graph.