Ministry of Science and Higher Education of the Russian

Federation National Research University ITMO

Evolutionary computing

Spring

2024

Laboratory work No. 4

GENETIC ALGORITHM FOR SOLVING THE TRAVELING SALESMAN PROBLEM

1. Implementation of classes:

A) TspSolution: a class that consists of a tour that visits each city exactly once. It encapsulates methods for accessing and manipulating the tour, meeting the requirement of representing a solution to the TSP and providing essential functionality for the genetic algorithm.
B) TspFactory: serves as a factory for generating initial candidate solutions to the TSP. It creates instances of `TspSolution` with randomized tours, adhering to the requirement of generating initial candidate solutions for the genetic algorithm.
C) TspCrossover: here we implement crossover functionality for the TSP genetic algorithm. It performs a crossover operation between pairs of parent solutions, producing two child solutions with characteristics inherited from their parents. The implementation meets the requirement of performing crossover between candidate solutions as part of the evolutionary process.
D) TspMutation: this class implements mutation operators for the TSP genetic algorithm. It offers various mutation strategies, including swap, insert, scramble, and inversion mutations, which apply the following:
- swapMutation: Swaps two cities in a tour randomly.
- insertMutation: Inserts a city from one position to another within the tour.
- scrambleMutation: Shuffles a subset of cities between two random indices within the tour.
- inversionMutation: Reverses the order of cities between two random indices within the tour.
TspFitnessFunction: The `TspFitnessFunction` class evaluates the fitness of candidate solutions based on their tour lengths in the TSP. It calculates the total distance traveled in a tour by considering that the path between cities is straight ($distance = \sqrt{x^2 + y^2}$).

2. Points to explain about the implementation of the problem:

1- Solution Representation:
The solution representation in our implementation follows the typical approach of representing a route as a permutation of city indices. For example, in the TspSolution class, each solution is represented as a list of integers where each integer represents a city index. This representation ensures that each city appears exactly once in the route.
2- Mutation techniques:
A) Swap Mutation:
First, two random indices within the tour are chosen. Then The cities at these indices are swapped to introduce variation in the solution. Which will help exploring different permutations of cities in the solution space.

```
int city1 = solution.getCity(index1);
int city2 = solution.getCity(index2);
solution.setCity(index1, city2);
solution.setCity(index2, city1);
```

B) Insert Mutation:

First, it selects two random indices within the tour, then it removes the city at the first index and inserts it at the second index, shifting other cities accordingly. So technically, it applies relocation of a city within the tour.

```
// Remove the city at index1 and insert it at index2
int city = tour.remove(index1);
if (index1 < index2) {
    index2--; // Adjust index2 after removing a city
}
tour.add(index2, city);
```

3. Scramble Mutation:

First, two random indices are chosen to define a subset of cities in the tour. Then the cities within this subset are shuffled randomly. After shuffling, the subset of cities is placed back into the tour, maintaining the order of the rest of the cities.

```
// Shuffle the subset of cities between index1 and index2 (inclusive)
List<Integer> subset = tour.subList(index1, index2 + 1);
Collections.shuffle(subset, random);

// Update the tour with the shuffled subset
for (int i = 0; i < subset.size(); i++) {
    solution.setCity( index: index1 + i, subset.get(i));
}
```

4. Inversion Mutation:

First, we selects two random indices to define a segment of cities in the tour, then we reverses the order of cities within this segment. So in short, this technique flips a portion of the route.

```
List<Integer> subset = tour.subList(index1, index2 + 1);
Collections.reverse(subset);
```

Each of the previous mutation functions aim to modify the current solution slightly to explore the solution space more effectively with respect to our problem constraints. By applying these mutation operators, the algorithm can traverse different routes and find shorter paths in the search for an optimal solution.

3. Experimentation:

Problem	Size	Pop, gen,	Route length	Nubber of iterations until convergence	Optimal route
xqf131	131	10, 100	3134	91	564
xqg237	237	10, 1000	7734	991	1019
pma343	343	10, 1000	21154	966	1368
bcl380	380	20, 2000	13806	1926	1621
pbl395	395	10, 2000	13902	1958	1281
pbk411	411	10, 2000	15113	1934	1343
pbn423	423	10,2000	15056	1969	1365
pbn423	423	10,3000	14140	2953	1365
pbn423	423	30,3000	16106	2985	1365
pbm436	436	10, 4000	14678	3928	1443
xql662	662	10, 5000	32687	4790	2513

By analyzing the results, we can see that the algorithm is improving only with the increase of the number of generations. Increasing the population size is leading to worse fitness, and despite setting

high selection pressure in the tournament strategy, still results are not satisfactory. After searching more about the possible reasons, we reached to the idea that maybe operators types that are used (mutation and crossover) are not efficient to explore the search space enough (despite setting different values for mutation probabilities). and therefore, maybe other types of mutation and crossover could have given better results.

4. Questions:

1. Is it possible to determine that the resulting solution is a global optimum?

It's difficult to determine definitively whether the resulting solution is a global optimum. The genetic algorithm explores a finite number of solutions within the search space, and it's possible that a better solution exists but was not discovered due to limitations in the algorithm's exploration or exploitation capabilities.

2. Is it possible to allow invalid decisions (with repetition of cities). If yes, then how to handle such a decision and how it will affect algorithm performance?

Allowing invalid decisions, such as repetitions of cities in a route, can have various implications for algorithm performance. If repetitions are allowed, it increases the size of the search space and may lead to a larger variety of solutions. But it also introduces the risk of generating inefficient or invalid routes. To handle such decisions, we can implement additional constraints or penalties to discourage or prevent invalid solutions (as we did for the queen placement when having the choice of not placing a queen). This could involve modifying the crossover and mutation operators to ensure that generated solutions adhere to the problem constraints.

3. How will the problem change if we remove the condition of the need to return to the starting point of the route?

Removing the condition of the need to return to the starting point of the route changes the nature of the problem from a closed tour (where the route must form a loop) to an open tour. In an open tour, the route starts and ends at different points, and there's no requirement to return to the starting point. This change may simplify the problem and potentially lead to different optimal solutions. The algorithm would need to be adapted accordingly to accommodate the new problem formulation and optimize routes without the constraint of returning to the starting point.