Documentation for the Genetic Algorithm Solving the Traveling Salesman Problem

Aliaksei Zimnitski
FIIT STU

October 9, 2024

Introduction

This documentation details the implementation of a genetic algorithm designed to solve the Traveling Salesman Problem (TSP). The primary goal is to determine the shortest possible route that visits each city exactly once and returns to the starting city.

Key Program Variables

- **number_of_cities** — the number of cities to visit.
- **population_size** — the size of the population (number of individuals). For this test, the value was set to 80 based on extensive testing.
- **children_size** — the number of offspring generated in each iteration, equal to 4/5 of **population_size**, with elitism preserving the top 10% of the population.
- **population[fitness, distance, path]** — an array representing the population, where each individual has three elements:
  - **fitness** — the fitness value (inverse of the total distance),
  - **distance** — the total distance of the route,
  - **path** — the actual route (a list of cities).
- **number_of_identical_best_to_stop** — the number of repeated best individuals in the population required to terminate the algorithm. Testing showed that this value strikes a good balance between giving the algorithm enough time to explore solutions and preventing excessive runtime when no better solutions are being found.
- **number_of_identical_best** — the current number of repeated best individuals in the population.

Key Functions of the Program

Parent Selection Functions

- **build_cumulative_probabilities** — calculates the selection probability for each parent.
- **select_one** — selects individuals randomly to become parents.

- **roulette_selection** — performs roulette wheel selection for a parent.
- **tournament_selection** — selects a parent using tournament selection.
- **hybrid_parent_selection** — selects parents using a combination of roulette and tournament selection.

Offspring Generation Functions

- **crossover** — randomly combines parents to create offspring for the next generation. Before adding, it checks if the child already exists to avoid duplication.
- **make_child** — creates a child by copying a random segment from one parent and filling in the rest from the other. Mutation may occur afterward.
- **mutation** — modifies the child with a certain probability. The mutation chance varies based on the number of consecutive generations with the same best solution. If mutation occurs, one of the following four types is chosen:
  - **invert_mutation_chance** — inverts a random section.
  - **scramble_mutation_chance** — shuffles a random section.
  - **shift_mutation** — moves a random segment to a different part of the list.
  - **swap_mutation** — swaps two random elements.
- **adaptive_mutation_chance** — adjusts the mutation probability based on the number of consecutive identical best results.

Auxiliary Functions

- **path** — calculates the total distance between cities.
- **euclidean_distance** — computes the distance between coordinates.
- **fitness** — evaluates the fitness of an individual.

Visualization Function

- **plot_path(cities_coordinates, best_path)** — visualizes the best-found path by plotting the coordinates of the cities and the connecting lines between them. **This function was generated by ChatGPT and not written by me.**

How the Program Works

Initialization

The program starts by creating a two-dimensional array representing the distances between cities. An initial population of random routes is then generated.

Running the Genetic Algorithm

The algorithm proceeds as follows:

1. A two-dimensional array is created to store the distances between cities.
2. An initial population of random routes is generated.
3. The genetic algorithm starts, and the *crossover* function generates offspring for the next generation.
4. For each child, its fitness and route are calculated. If the child is not already present in the population, it is added.
5. The population is sorted based on the fitness value.
6. It is checked whether the best result has changed:
   - If it has, the counter **number_of_identical_best** is reset to zero.
   - If not, the counter is incremented by one.
7. If **number_of_identical_best** reaches the threshold value **number_of_identical_best_to_stop**, the algorithm terminates.

Improvements

- **Adaptive Mutation Probability** — helps avoid stagnation by gradually increasing the mutation chance if the best solution remains unchanged.

```
def adaptive_mutation_chance(number_of_identical_best):
    plus_chance = 100 * number_of_identical_best /
        number_of_identical_best_to_stop
    base_chance = 0.1
    return base_chance + 0.01 * plus_chance
```

- **Uniqueness Checks** — prevent duplication by checking whether the new offspring already exists in the population.

```
        if child not in children:
            children.append(child)


        if child not in population:
            population[i] = child
```

Testing

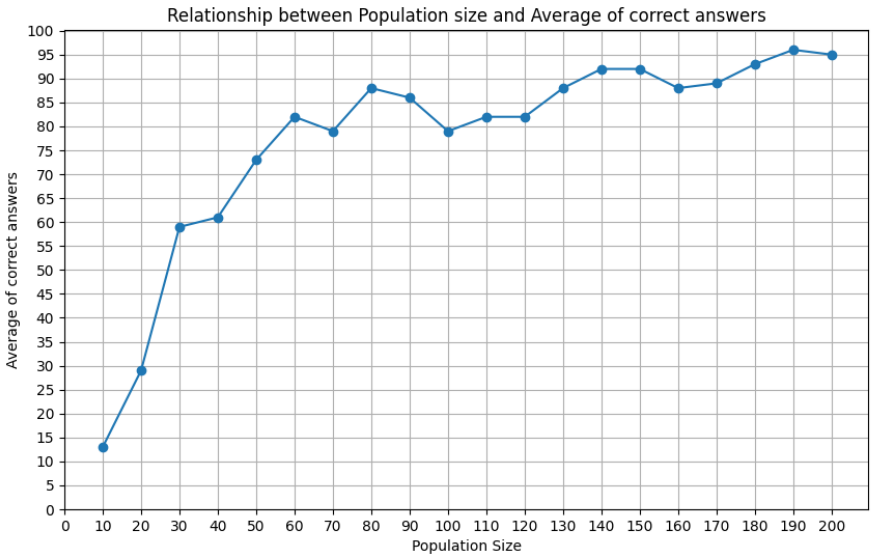
The graphs below illustrate the relationship between the average number of correct solutions and population size for different parent selection methods. The first graph shows hybrid selection (roulette + tournament), while the second graph displays roulette-only selection.

The analysis showed that using only roulette required a larger population size (around 120), whereas the hybrid approach provided good results with a population size of 80.

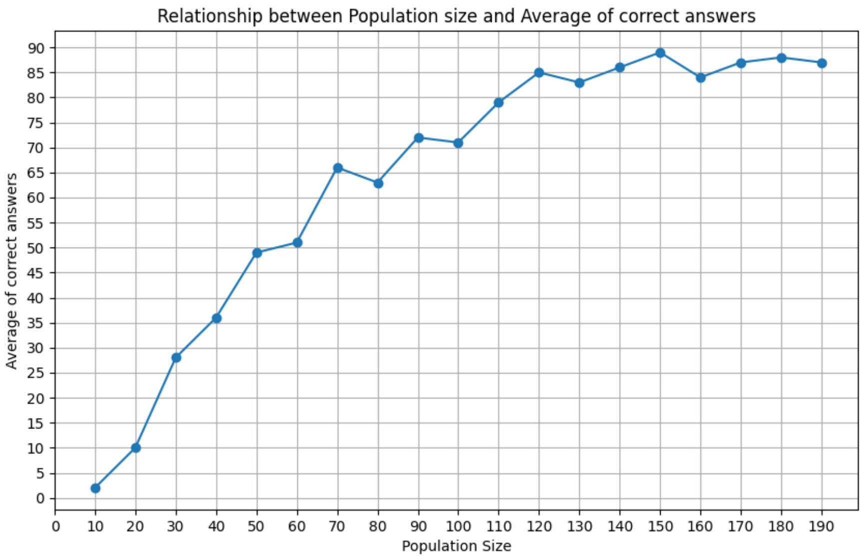
All tests were conducted using the following test case:

```
20
60 200
180 200
100 180
140 180
20 160
80 160
200 160
140 140
40 120
120 120
180 100
```

```
60 80
100 80
180 60
20 40
100 40
200 40
20 20
60 20
160 20
```



Hybrid Selection (Roulette + Tournament)



Roulette Selection Only