Slovak Universiry of Technology in Bratislava

Faculty of Informatics and Information Technologies

Artificial Intelligence

Assignment 2

Travelling Salesman Problem

Implementation of Genetic algorithm and Tabu search to solve Travelling Salesman Problem

Dávid Truhlář ID: 120897 xtruhlar@stuba.sk

Table of Content

The assignment

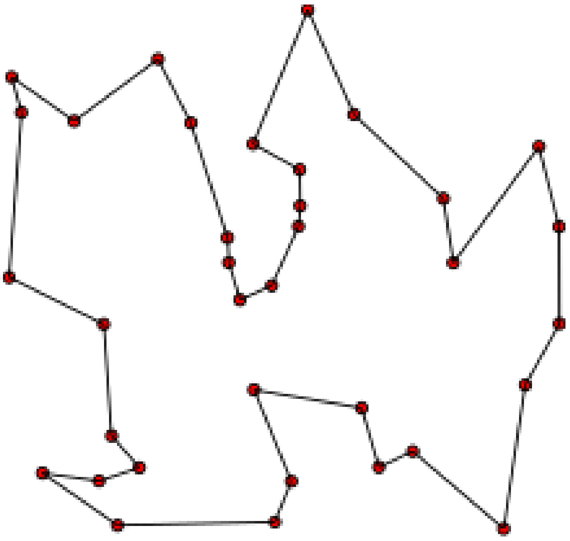
The task involved solving the traveling salesman problem. I had to visit multiple cities while minimizing travel costs, with the route forming a closed loop, meaning I had to return to the city I started from.

I had at least 20 cities at my disposal, each with randomly generated coordinates. The cost of travel between two cities was determined by the Euclidean distance. The total length of the route was determined by a permutation of cities, and my task was to find a permutation with the smallest total distance.

In the case of the genetic algorithm, I represented individuals using a vector with the order of cities. The fitness value of an individual was the inverse of the length of its route. I initialized the first generation and implemented at least two methods for parent selection. I addressed crossover and mutations of individuals. I evaluated the results and compared different methods for generating the next generation or selection.

In the case of tabu search, I also used a representation with a vector of the order of cities. My algorithm generated successors and moved to better states while maintaining a list of tabu states to avoid cycling in local extremes. The length of this list was an important parameter.

Example:



Route between randomly chosen cities.

Implementation

PART I. Genetic Algorithm

Genes representation

In my program, gene representation is achieved using classes. The first class, "*Mesto*," contains information about each city on the map, including its *ID, as well as its x and y coordinates*.

```
class Mesto:
    def __init__(self, num, x, y):
        self.num = num
        self.x = x
        self.y = y
```

The second class, "*Chromosome*," stores the permutation of cities in two formats. It maintains an array of objects (cities) in *self.chromosome* and an array of city IDs in *self.chromosome_representation*. This class also calculates the distance of the permutation, stored in *self.cost*, and its fitness value in *self.fitness_value*.

```
class Chromosome:
    def __init__(self, zoznam):
        self.chromosome = zoznam

        chromosome_representation = []
        for i in range(0, len(zoznam)):
            chromosome_representation.append(self.chromosome[i].num)
        self.chromosome_representation = chromosome_representation

        cost = 0
        for j in range(1, len(self.chromosome_representation) - 1):
          cost += matrix[self.chromosome_representation[j]-1][self.chromosome_representation[j+1]-1]

        self.cost = cost
        self.fitness_value = 1 / self.cost
```

The distance matrix is employed to compute the cost of each Chromosome.

```
def create_distance_matrix(mesta):
    num_cities = config.NUMBER_OF_CITIES
    mat = [[0] * num_cities for _ in range(num_cities)]

    for i in range(0, len(mat)):
        for j in range(0, len(mat)):
            mat[i][j] = math.sqrt((mesta[i].x-mesta[j].x)**2+(mesta[i].y-mesta[j].y)**2)
    return mat
```

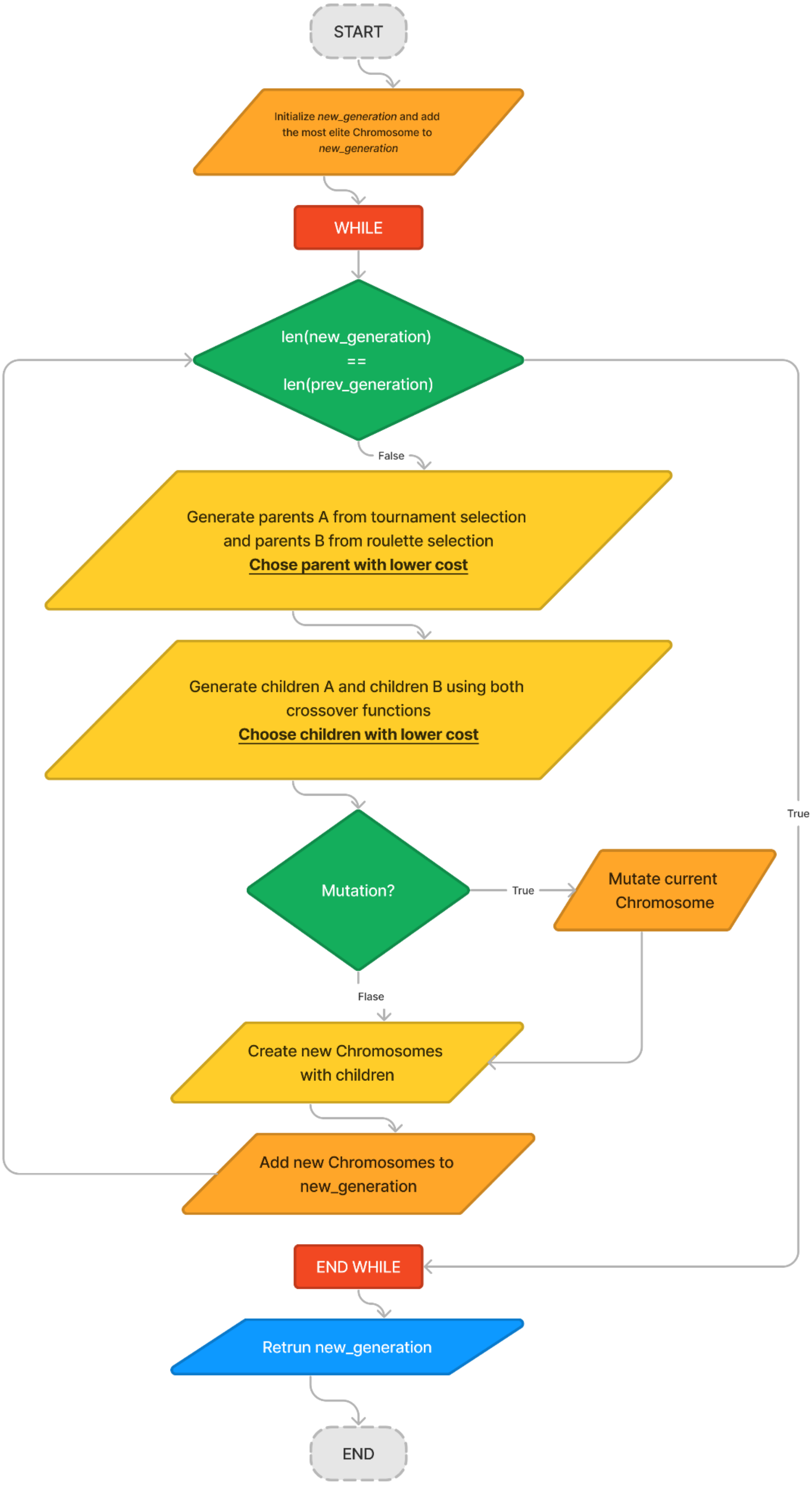
First generation

The initialization of the first generation utilizes two functions. The first function is "*create_random_list*," which takes an array of cities and shuffles them into a random sample of cities. Then, the "*initialization*" function takes the shuffled array of cities and creates the first Chromosome object, representing the first generation.

```
def create_random_list(zoznam_miest):
    shuffled_miest = random.sample(zoznam_miest[1:], len(zoznam_miest) - 1)
    return [zoznam_miest[0]] + shuffled_miest + [zoznam_miest[0]]

def initialization(data, pop_size):
    return [Chromosome(create_random_list(data)) for _ in range(pop_size)]
```

Generating new generation

When generating a new generation in genetic optimization, various techniques of crossover and mutation are employed to create offspring that improves the existing population, gradually converging toward better solutions.


START
Initialize *new_generation* and add the most elite Chromosome to *new_generation*
WHILE
len(new_generation) == len(prev_generation)
False
True
Generate parents A from tournament selection and parents B from roulette selection
Chose parent with lower cost
Generate children A and children B using both crossover functions
Choose children with lower cost
Mutation?
True
Mutate current Chromosome
Flase
Create new Chromosomes with children
Add new Chromosomes to new_generation
END WHILE
Retrun new_generation
END

Steps:
1.Initialize `new_generation`

2.Create new pair of parents using tournament selection

3.Create new pair of parents using roulette selection

4.Choose parents with better fitness

5.Create new pair of children using crossover with 2 points of crossover. In first crossover the middle part is kept

6.Create new pair of children using crossover with 2 points of crossover. In this crossover the beginning and the end are kept

7.If generated number is smaller than `MUTATION_RATE` mutate current chromosome

8.Add new children to `new_generation`

9.Repeat until new_generation is not completed (or `len(population)/2` because each iteration creates 2 new children)

Parent selections, crossovers and mutation are better described in the code.

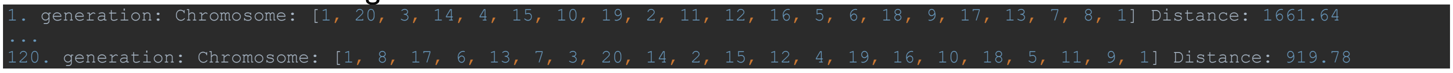
Evaluation of characteristic aspects of the algorithm

With the solution of Traveling salesman problem using genetic algorithm, the following aspects have been considered:

1. **Data representation**
   - Each City is represented with `x, y` coordinates and ID (`num`).
   - Those cities in form of list represent one permutation or Chromosome.
   - This aspect is described in previous part of documentation - Genes representation in more detail.

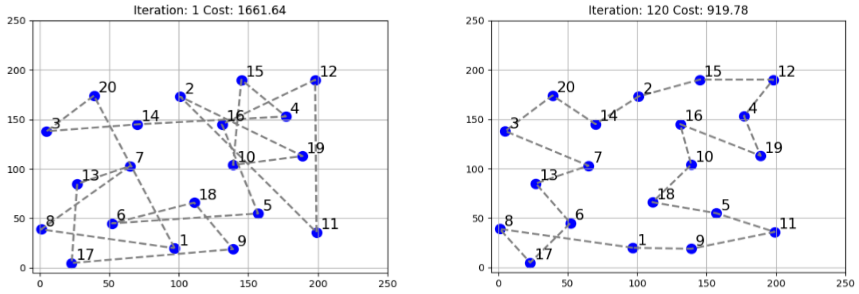
2. **Inputs and variables**
   - Program consists of two files, `main.py` and `config.py.`
   - All inputs that user can change are in the `config.py`. User can change:
     - number of cities,
     - number of generations,
     - population size,
     - mutation rate,
     - minimal distance between cities,
     - how many tickets are generated in tournaments,
     - how many graphs will be shown.
   - In `main.py` there is the algorithm with all functions and algorithms which shouldn't be changed.

3. **Solution representation**
   - The solution consists of two main parts. The first one is output in the program console, the second one is output in the form of graphs.
   - Examples:
     - Program console:



```
1. generation: Chromosome: [1, 20, 3, 14, 4, 15, 10, 19, 2, 11, 12, 16, 5, 6, 18, 9, 17, 13, 7, 8, 1] Distance: 1661.64
...
120. generation: Chromosome: [1, 8, 17, 6, 13, 7, 3, 20, 14, 2, 15, 12, 4, 19, 16, 10, 18, 5, 11, 9, 1] Distance: 919.78
```

     - Graph:



4. **Parent selection**
   - In the program two methods of parent selection are implemented.
     - <u>The Tournament Selection</u>
       - In this method, user can set how many tickets should be created for population. Then randomly select chromosomes are compared and the chromosome with best fitness is return as the parent.
     - <u>The Roulette Selection</u>
       - This method counts together *total_fitness* which is sum of all fitness values of the population then generate random number. Another variable is set to 0 and fitness value of each chromosome is added to this variable. When the number is more than random number the function returns the current chromosome as parent.

5. Crossover

- Like selection, this program uses two different types of crossover functions.
  - Two-Point Crossover – Crossover points at begging.
    - This crossover function is reliant on the random selection of 2 points, which serve as the crossover points. The cities at the crossover points from `parent1` and `parent2` are saved in the variables `base1` and `base2`. The remaining cities from `parent2` are stored in the variable `unused2`. `Child1` is then extended with the cities in `unused2`. Similarly, child 2 is generated using the remaining cities from `parent1`. These children are subsequently returned from the function.
      - ✓ Example: for points 2, 4

        Parent 1: 1, 5, 3, 2, 4, 6, 1 → Child 1: 1, 3, 2, 4, 6, 5, 1
        Parent 2: 1, 2, 6, 4, 3, 5, 1 → Child 2: 1, 6, 4, 3, 5, 2, 1

  - Two-Point Crossover – Crossover point in the middle.
    - This selection process closely resembles the first one, with the key distinction being that `child1` does not include cities between the crossover points. Instead, `child1` is divided into two parts. In the first part, "`child_1_s`," it contains cities from the beginning of the chromosome to the first crossover point. In the second part, "`child_1_e`," it comprises cities from the second crossover point to the end of the chromosome. The remaining cities from `parent2` are stored in "`unused2`." Subsequently, `child1` is reconstructed by combining these three parts: `"child_1_s" + "unused2" + "child_1_e."` The same process is applied to `child2`.
      - ✓ Example for points 2, 4

        Parent 1: 1, 5, 3, 2, 4, 6, 1 → Child 1: 1, 5, 2, 4, 3, 6, 1
        Parent 2: 1, 2, 6, 4, 3, 5, 1 → Child 2: 1, 2, 3, 4, 6, 5, 1

6. Mutation

- The final component of the Genetic algorithm implemented in the assignment is mutation.
  - Users can specify the `MUTATION_RATE` and `MUTATION_DIST`, which determine the rate of mutation and how far apart the cities to be swapped can be.
  - In the mutation process, two random indexes are generated, and the cities corresponding to these indexes in the chromosome to be mutated are identified. If these cities are closer than `MUTATION_DIST`, they are swapped.

PART II. Tabu Search

Integrating the Tabu Search Algorithm proved to be a more straightforward task, as it leveraged certain elements of the previously established Genetic Algorithm (GA). The common functions and classes across both algorithms facilitated the coding process.

Genes representation

In this implementation, we continued to employ the "`Mesto`" and "`Chromosome`" classes, following a similar approach as in the Genetic Algorithm. These classes served their established roles in representing cities and city permutations, respectively. This continuity in class usage facilitated the transfer of knowledge and code from the Genetic Algorithm to the Tabu Search Algorithm, making the implementation process smoother.

First generation

Likewise, the functions "`create_random_list`" and "`initialization`" were recycled for this implementation. The process of generating the first generation remained consistent with the approach used in the GA.

Creating new generation

In this section, we encounter the first significant divergence. The Tabu Search adopts a markedly distinct approach when selecting a new route within the hypothetical decision tree. Notably, it abstains from combining parents or generating offspring. Instead, it introduces the "`get_Neighbours`" function, which furnishes all conceivable neighbours for the current permutation.

Within this function, potential neighbours are produced in two distinct ways, leading to two separate sets, thus ensuring the generation of every possible neighbour. These sets are subsequently amalgamated and filtered to eliminate any potential duplicates.

Subsequently, the best neighbour is designated and stored in the "`best_candidate`" variable. If its fitness surpasses that of the current best solution (`sBest`), it is appended to the "`tabu_list`." This process is encapsulated within a while loop, which iterates until the specified number of iterations is achieved. The total number of iterations is recorded in the "`NUMBER_OF_GENERATIONS`" variable.

Evaluation of characteristic aspects of algorithm

When working with Tabu Search, it's crucial to address the characteristic aspects. Many of these aspects align with those of the Genetic Algorithm. The primary difference that needs to be implemented is the "`get_Neighbours`" function.

1. **get_Neighbours**
   - This function is responsible for producing neighbouring permutations based on the current one. It employs the "`Damocles`" variable to determine which of the two nested for loops to execute.
     - The first for loop generates all possible neighbours by considering all permutations attainable with just one swap.
     - The second for loop generates neighbours in a slightly distinct manner. It involves reversing all elements located between the "`i`" and "`j`" indices.
2. **Tabu size**
   - Tabu size is another variable that change the outcome of algorithm. It is important to choose the right tabu size to optimize the time and effectiveness of solution.

Comparison of achieved results

Genetic Algorithm

Parent Selection

To comparing two potential selections, an identical map is generated in both instances.

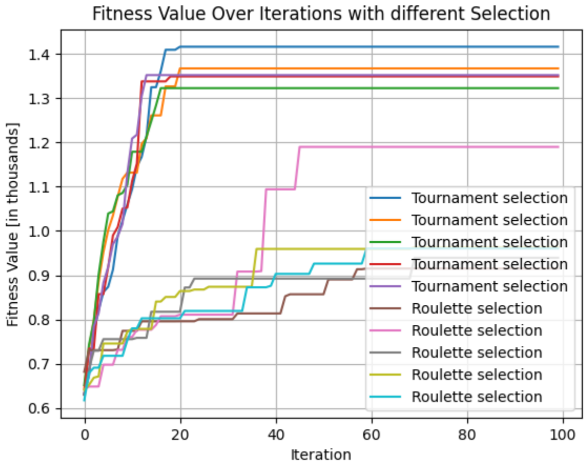


Figure 1Fitness Values with different Selection with 20 Cities

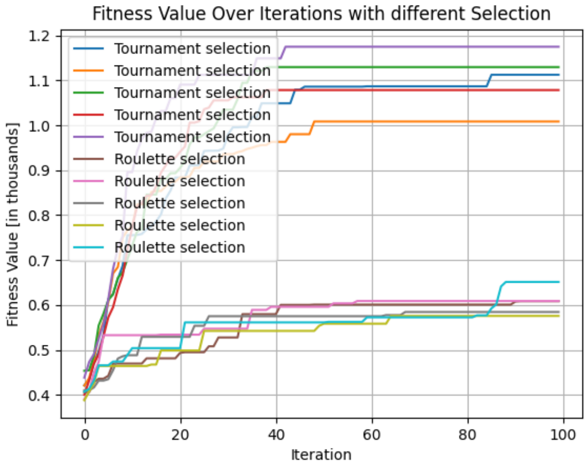


Figure 2 Fitness Values with different Selection with 30 Cities

In this scenario, the Tournament selection method yields superior results. However, in this specific implementation, two pairs of parents are generated, and the better of the two pairs is selected. This approach guarantees the discovery of better solutions.

Crossovers

The same logic was implemented to compare crossover functions.

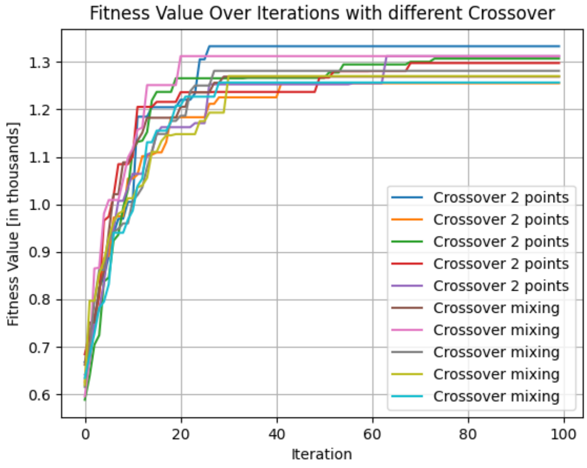


Figure 3 Fitness Value with different Crossover with 20 Cities

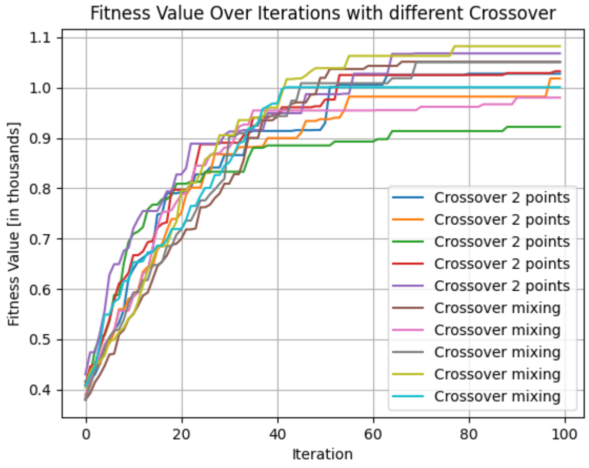


Figure 4 Fitness Value with different Crossover with 30 Cities

In this scenario, it was observed that the crossover function with crossover points at the beginning performed better when 20 cities were implemented. However, when testing with 30 cities, the second crossover method yielded better results. To ensure optimal crossover, both methods are implemented in the algorithm, and the superior one is chosen to proceed to the next phase. This adaptive approach allows for better results depending on the specific problem size.

Mutation

In figure 5 and figure 6 the MUTATION_DIST was set to value 6.

```
MUTATION_DIST = 6
```

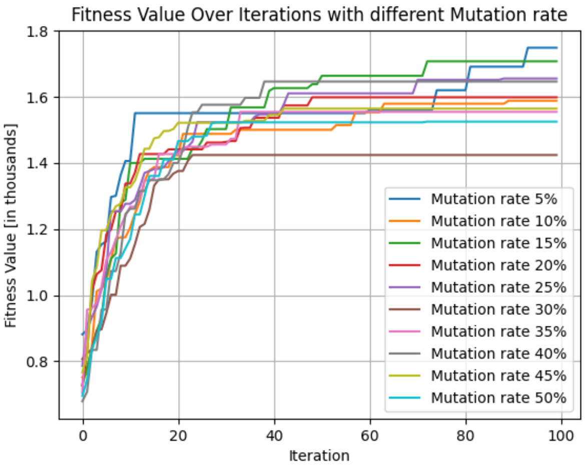


Figure 5 Fitness Value with different Mutations with 20 Cities

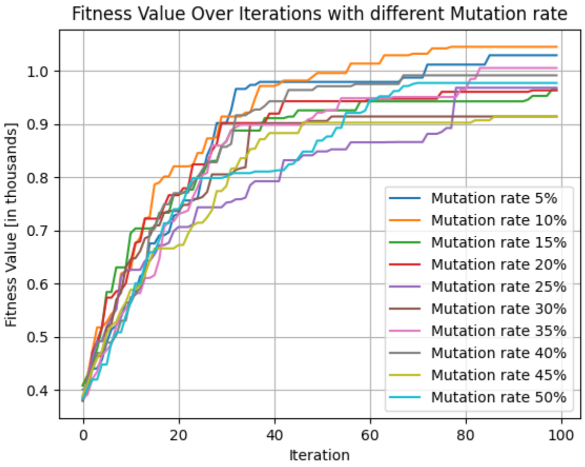


Figure 6 Fitness Value with different Mutations with 30 Cities

In figure 7 and figure 8 the MUTATION_DIST was set to value 1.

```
MUTATION_DIST = 1
```

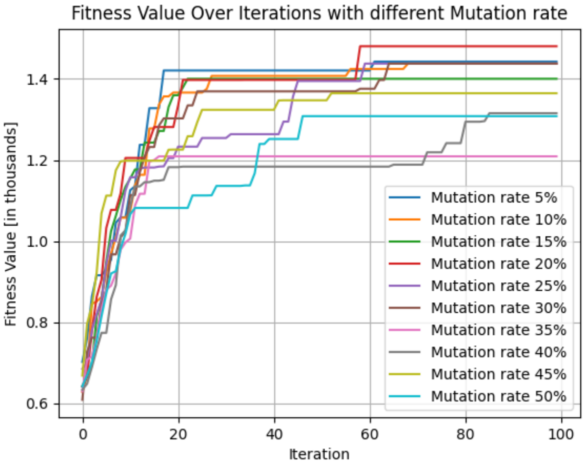


Figure 7 Fitness Value with different Mutations with 20 Cities

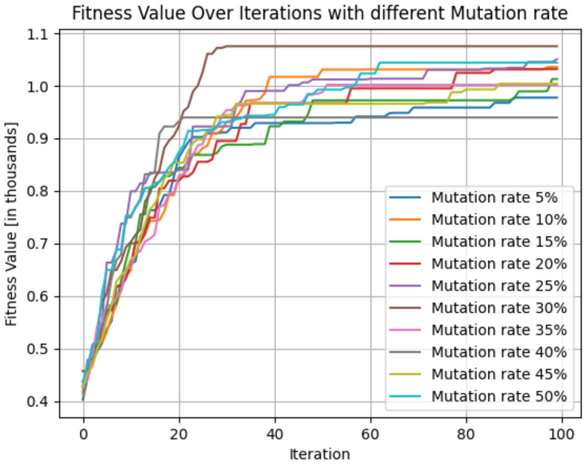


Figure 8 Fitness Value with different Mutations with 30 Cities

As the value of MUTATION_RATE increases, there are minor variations in the results. This implies that the precise value selected for MUTATION_RATE may not be of utmost importance. A general recommendation is to set it within the range of 10 to 20, as this range appears to deliver satisfactory results across a spectrum of scenarios. Additionally, the distance between cities in the mutation process has only a slight impact on the results. However, it seems that with a higher number of cities, a smaller distance between the two cities swapped during mutation leads to better outcomes. In contrast, when testing with 20 cities, the overall fitness was assessed with a larger MUTATION_DIST.

Tabu Search

get-Neighbours

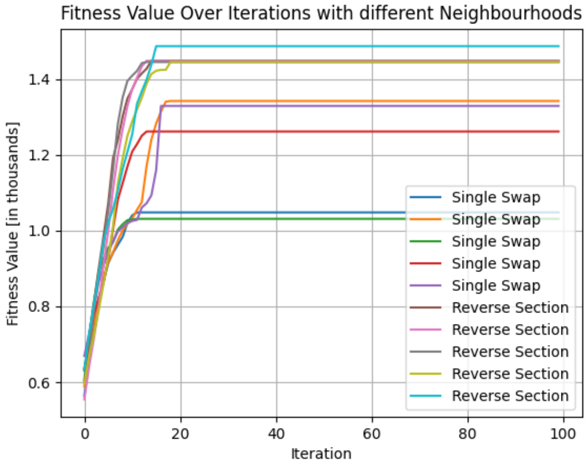


Figure 9 Fitness Value with different Neighborhoods with 20 Cities

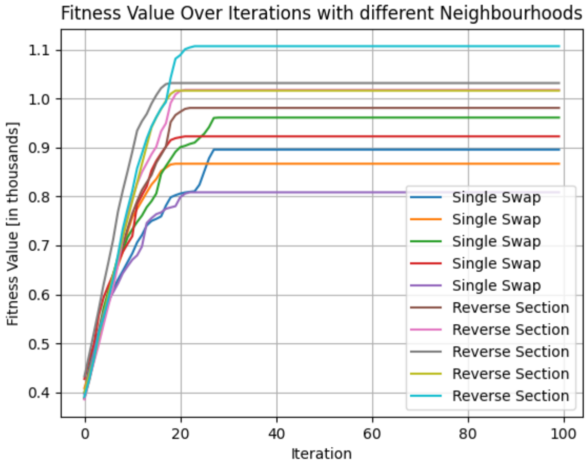


Figure 10 Fitness Value with different Neighborhoods with 30 Cities

The reversed section method was found to yield a better route based on this observation. However, in the implementation, the decision to use either method is made randomly, introducing an element of chance into the process.

Tabu Size

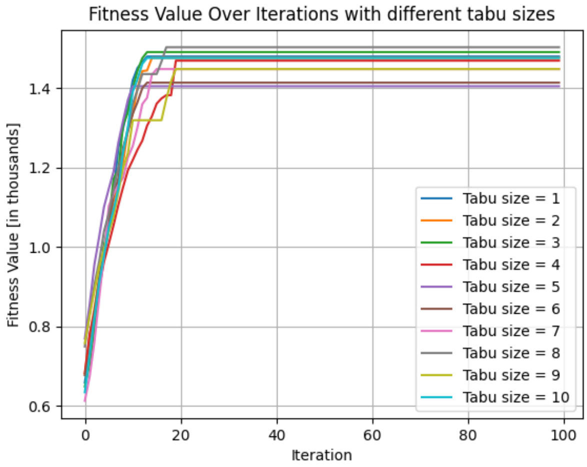


Figure 11 Fitness Value with different Neighborhoods with 20 Cities

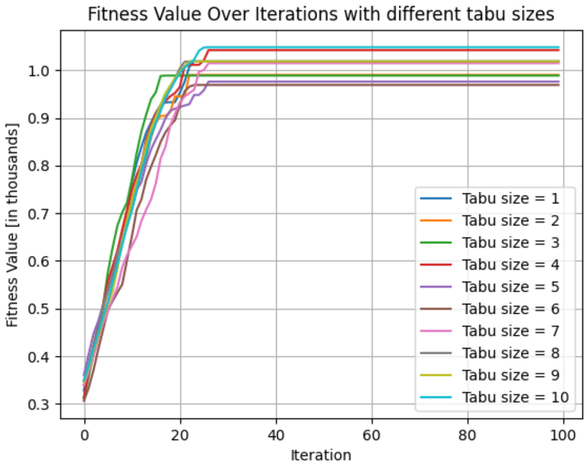


Figure 12 Fitness Value with different Neighborhoods with 30 Cities

In the context of varying Tabu list sizes, the results show only marginal differences. The graph indicates that optimization is possible, but it often involves a trial-and-error approach. The recommended size for this implementation, especially with 20-30 cities, falls within the range of 8 to 15.

Overall fitness of both algorithms

To see which algorithm completes the Traveling salesman problem more efficient with all implemented aspects tests which compares the overall fitness values.

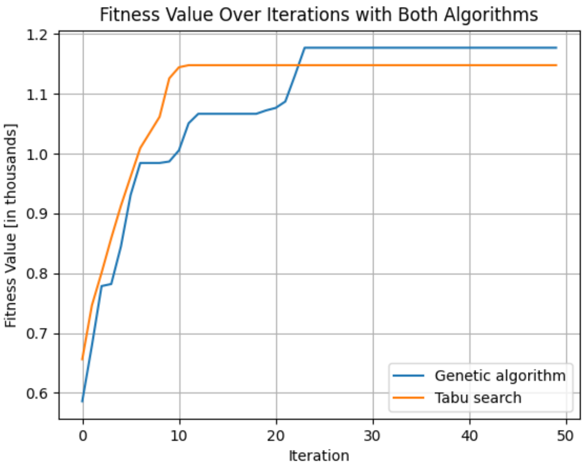


Figure 13 Fitness Value of both algorithms with 20 Cities

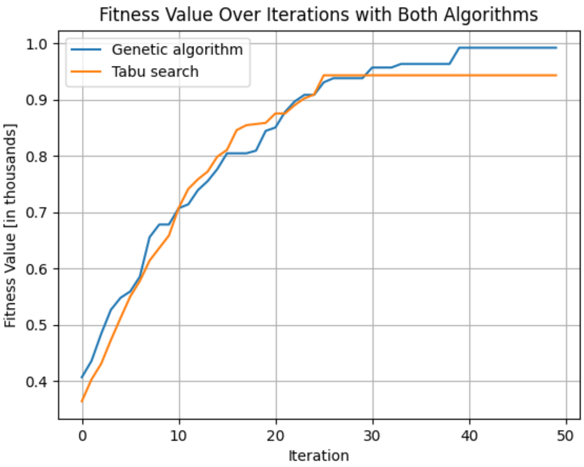


Figure 14 Fitness Value of both algorithms with 30 Cities

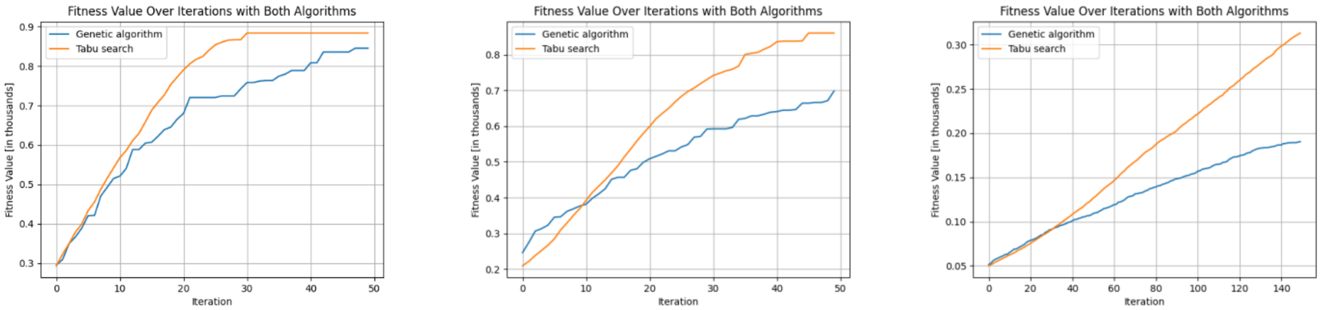


Figure 15 Fitness Value of both algorithms with 40, 50 and 200 Cities

Conclusion

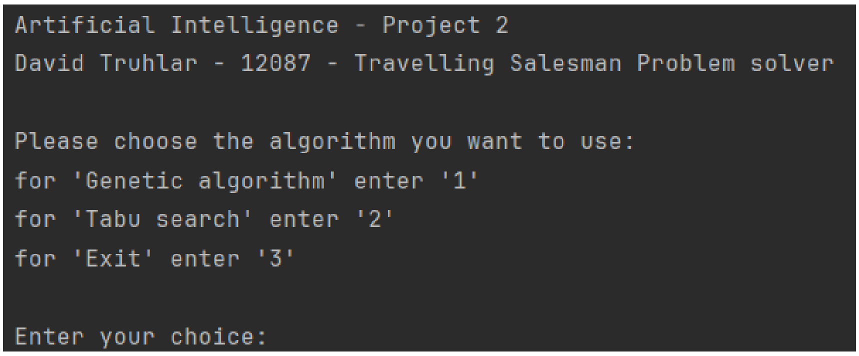
In the analysis of the implemented Genetic Algorithm and Tabu Search, the performance for both algorithms for different city counts in the Traveling Salesman Problem revealed noteworthy trends. In scenarios with fewer cities (20 – 30), the Genetic Algorithm achieved better fitness values while searching for solutions. However, with a higher number of cities (40 – 50), Tabu Search surpassed the Genetic Algorithm in performance. The final test with 200 cities exhibited a clear and consistent advantage for Tabu Search over the Genetic Algorithm. These findings emphasize the importance of correctly configuring variables in config.py to achieve the desired solution efficiently in terms of both time and space complexity.

User Manual

This section is designed to assist users in achieving their desired outcomes when using the program.

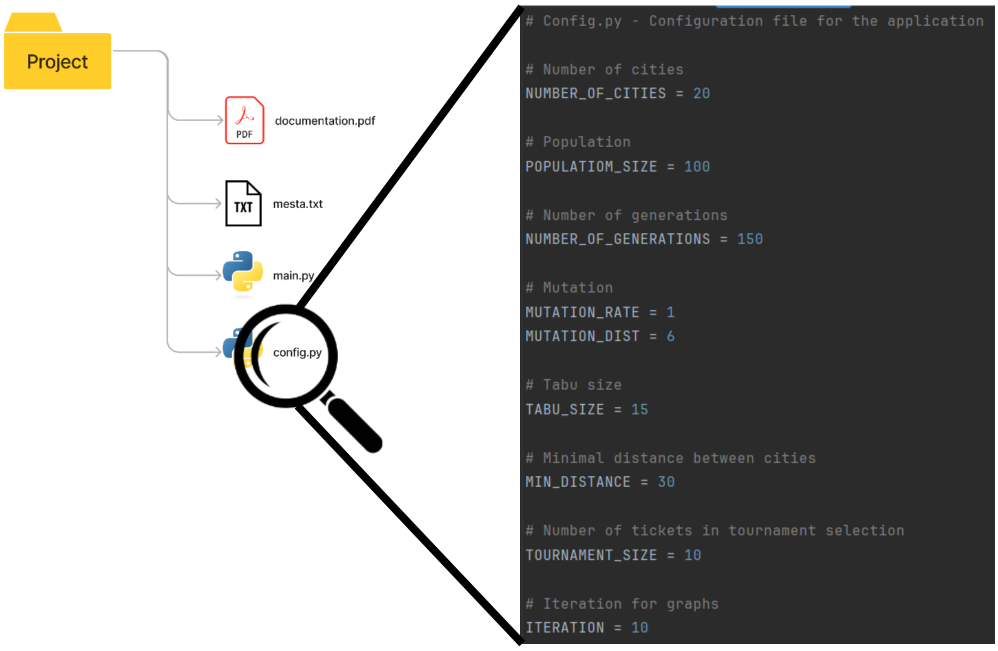
Users have the choice to employ the Genetic Algorithm or Tabu Search to solve his TSP:

- Enter '1' for 'Genetic Algorithm'
- Enter '2' for 'Tabu Search'
- Enter '3' to exit the program



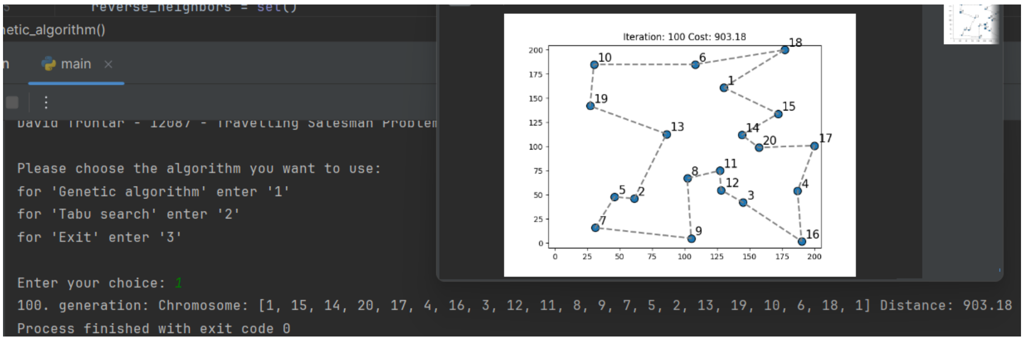
```
Artificial Intelligence - Project 2
David Truhlar - 12087 - Travelling Salesman Problem solver

Please choose the algorithm you want to use:
for 'Genetic algorithm' enter '1'
for 'Tabu search' enter '2'
for 'Exit' enter '3'

Enter your choice:
```

And, for users' convenience, every aspect of these algorithms is fully configurable in the 'config.py' file, allowing to change parameters such as population size, mutation rate, tabu size, and more.



```
# Config.py - Configuration file for the application

# Number of cities
NUMBER_OF_CITIES = 20

# Population
POPULATION_SIZE = 100

# Number of generations
NUMBER_OF_GENERATIONS = 150

# Mutation
MUTATION_RATE = 1
MUTATION_DIST = 6

# Tabu size
TABU_SIZE = 15

# Minimal distance between cities
MIN_DISTANCE = 30

# Number of tickets in tournament selection
TOURNAMENT_SIZE = 10

# Iteration for graphs
ITERATION = 10
```

Output:



```
    reverse_neighbors = set()
David Truhlar - 12087 - Travelling Salesman Problem

Please choose the algorithm you want to use:
for 'Genetic algorithm' enter '1'
for 'Tabu search' enter '2'
for 'Exit' enter '3'

Enter your choice: 1
100. generation: Chromosome: [1, 15, 14, 20, 17, 4, 16, 3, 12, 11, 8, 9, 7, 5, 2, 13, 19, 10, 6, 18, 1] Distance: 903.18
Process finished with exit code 0
```