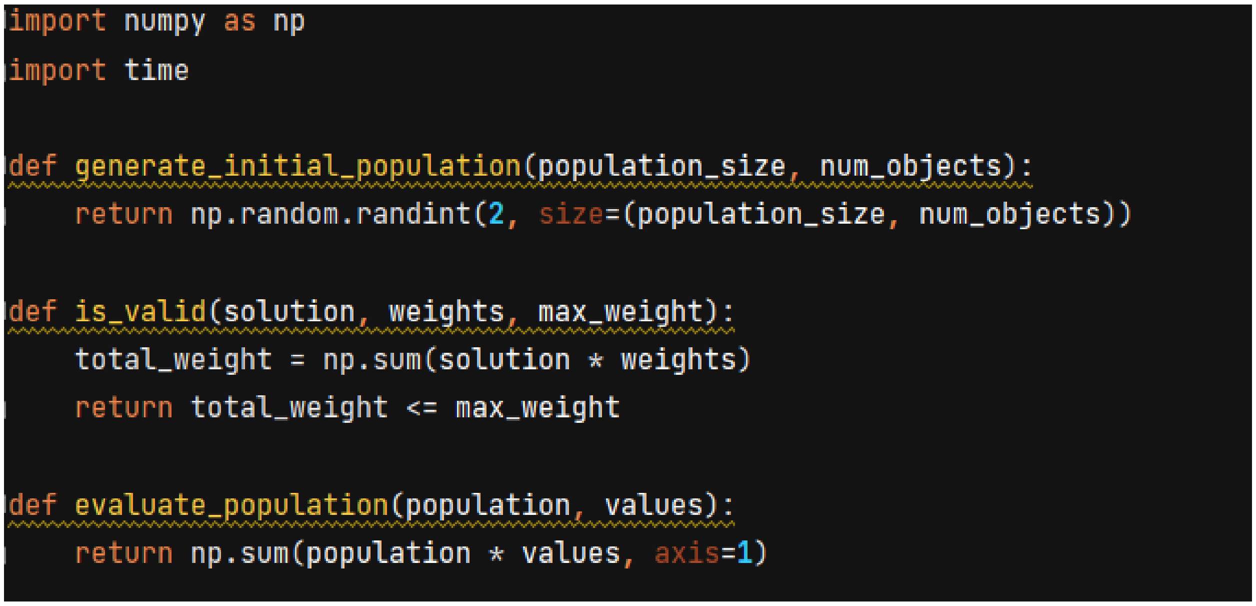
Laborator 3 Neagu Mihnea

Problem: Knapsack Problem

Problem Explanation: The Knapsack Problem is a combinatorial optimization problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a certain limit and the total value is maximized. It gets its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

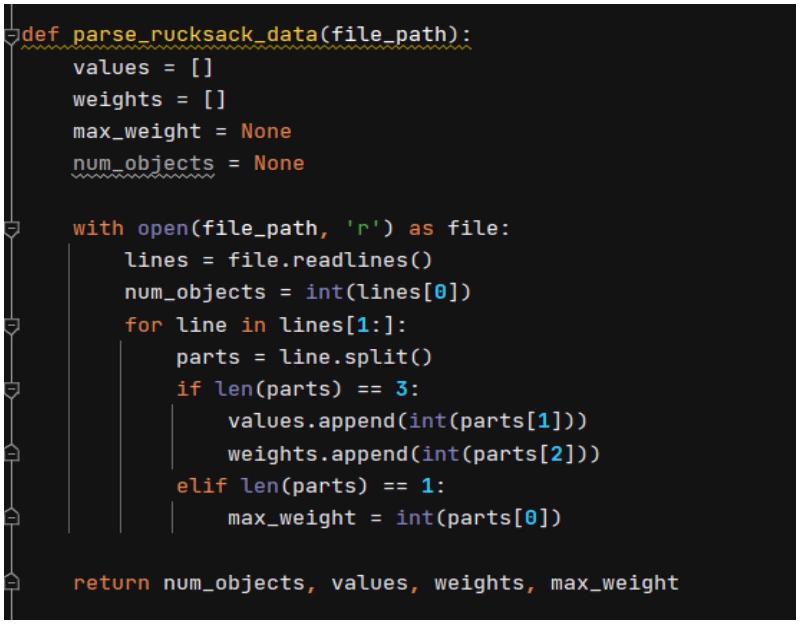
1. Algorithm 1 – Evolutive algorithm Knapsack Problem

Step 1: We create functions for generation a random solution at first(a random population as it is called in genetic algorithms), then an is_valid function that makes sure that the weight of the objects in the solution is not greater than the weight given for the knapsack and another function evaluate_population that calculates the value of our solution and will be used as a fitness function.



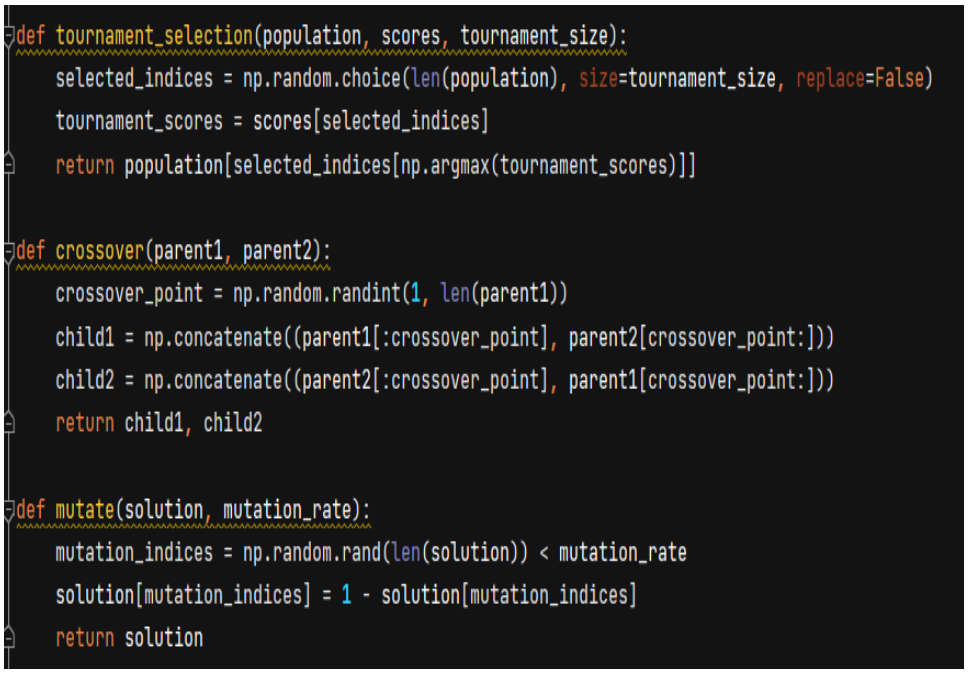
```
import numpy as np
import time

def generate_initial_population(population_size, num_objects):
    return np.random.randint(2, size=(population_size, num_objects))

def is_valid(solution, weights, max_weight):
    total_weight = np.sum(solution * weights)
    return total_weight <= max_weight

def evaluate_population(population, values):
    return np.sum(population * values, axis=1)
```

Step 2 : We will reuse the parse_knapsack function that structures the values in the data.txt file so we can use them further



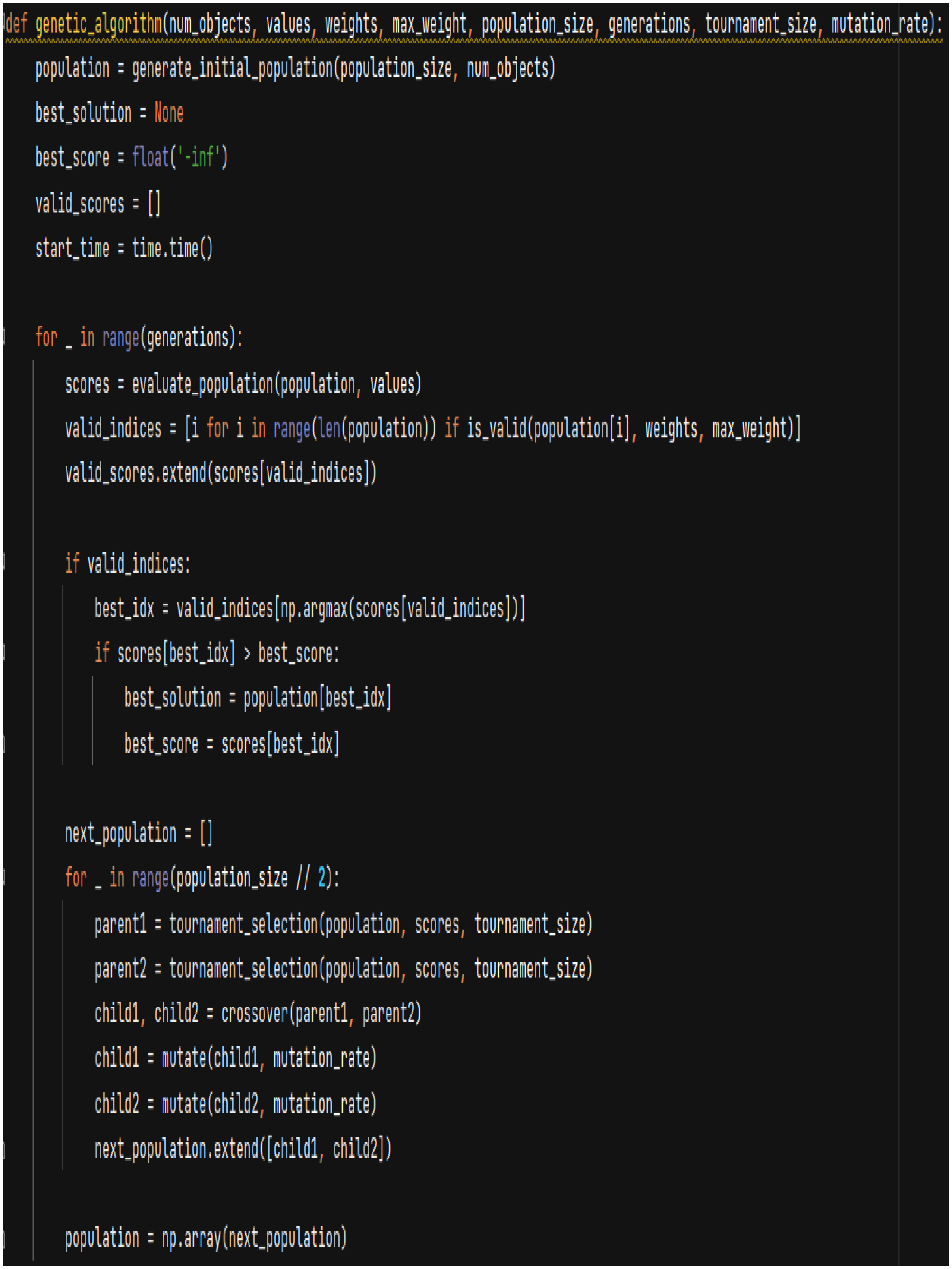
```python
def parse_rucksack_data(file_path):
    values = []
    weights = []
    max_weight = None
    num_objects = None

    with open(file_path, 'r') as file:
        lines = file.readlines()
        num_objects = int(lines[0])
        for line in lines[1:]:
            parts = line.split()
            if len(parts) == 3:
                values.append(int(parts[1]))
                weights.append(int(parts[2]))
            elif len(parts) == 1:
                max_weight = int(parts[0])

    return num_objects, values, weights, max_weight
```

Step 3: Our next 3 three functions are a torunament selection that is used in order to choose the fittest individuals to be part of the population, a crossover function that will reproduce 2 parents and produce an offspring with characteristics from both of them(it cuts the first parent at a given point(crossover_point) and concatenates it with the second parent cut at the same point and then concatenates the rests from both of them in order to obtain a second child). And we also have a mutation function that bit flips a solution in order to obtain a new one, which also uses a mutation_rate parameter that enables mutation to happen at a naturally low rate.

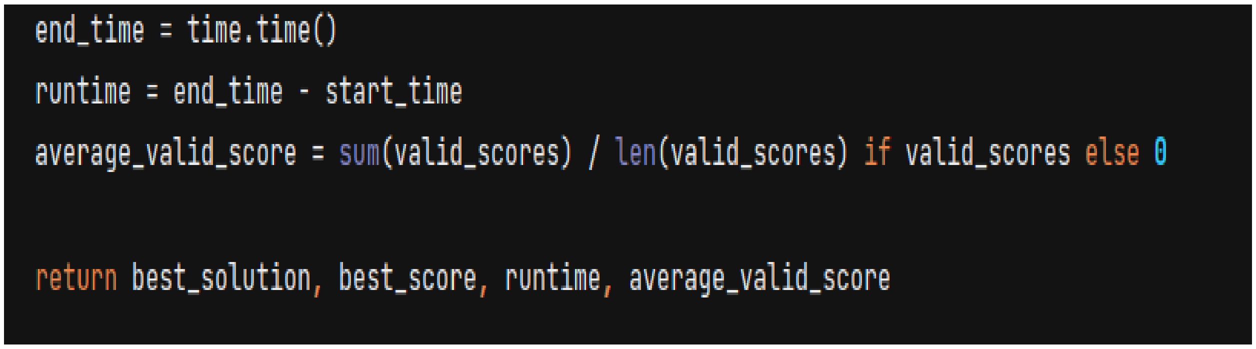


```python
def tournament_selection(population, scores, tournament_size):
    selected_indices = np.random.choice(len(population), size=tournament_size, replace=False)
    tournament_scores = scores[selected_indices]
    return population[selected_indices[np.argmax(tournament_scores)]]

def crossover(parent1, parent2):
    crossover_point = np.random.randint(1, len(parent1))
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return child1, child2

def mutate(solution, mutation_rate):
    mutation_indices = np.random.rand(len(solution)) < mutation_rate
    solution[mutation_indices] = 1 - solution[mutation_indices]
    return solution
```

Step 4: The genetic_algorithm function starts by choosing a random solution and using a given number of generations(that takes the place of iterations) and calculating the score of the respective population. Then it validates it by total weight and if suited it adds it to a valid_scores list. After the next if statement compares all the values added in the valid_scores parameter and find the best one out of them. The next four creates a tournament using the function explained above, tournament_selection. It does so by entering the parents in the tournament and then having the best parent's children advance in the next population, but also giving the offspring the opportunity to mutate. At the end, the runtime is calculated and the average scores of the populations and then they are returned at the end of the function.

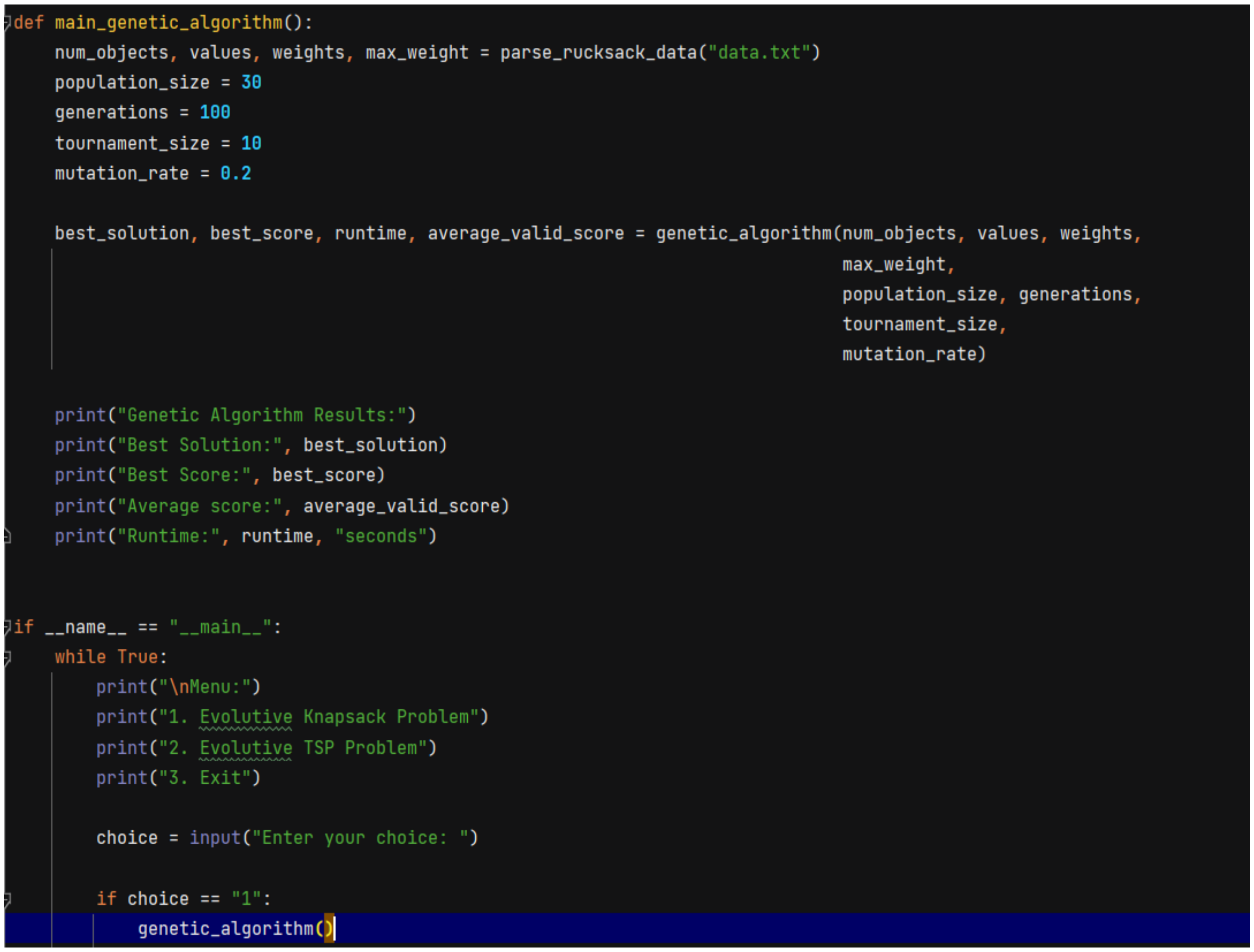


```python
def genetic_algorithm(num_objects, values, weights, max_weight, population_size, generations, tournament_size, mutation_rate):
    population = generate_initial_population(population_size, num_objects)
    best_solution = None
    best_score = float('-inf')
    valid_scores = []
    start_time = time.time()

    for _ in range(generations):
        scores = evaluate_population(population, values)
        valid_indices = [i for i in range(len(population)) if is_valid(population[i], weights, max_weight)]
        valid_scores.extend(scores[valid_indices])

        if valid_indices:
            best_idx = valid_indices[np.argmax(scores[valid_indices])]
            if scores[best_idx] > best_score:
                best_solution = population[best_idx]
                best_score = scores[best_idx]

        next_population = []
        for _ in range(population_size // 2):
            parent1 = tournament_selection(population, scores, tournament_size)
            parent2 = tournament_selection(population, scores, tournament_size)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
            next_population.extend([child1, child2])

        population = np.array(next_population)
```



```
end_time = time.time()
runtime = end_time - start_time
average_valid_score = sum(valid_scores) / len(valid_scores) if valid_scores else 0


return best_solution, best_score, runtime, average_valid_score
```

Step 5: We create another function where we give values to the size of the population and the number of generations which the algorithm will go trough but also the tournament size and the mutation rate. In the end we print out the values wanted, the best solution, best score, average scores and the runtimes when the input given is 1(for the first problem).



```
def main_genetic_algorithm():
    num_objects, values, weights, max_weight = parse_rucksack_data("data.txt")
    population_size = 30
    generations = 100
    tournament_size = 10
    mutation_rate = 0.2

    best_solution, best_score, runtime, average_valid_score = genetic_algorithm(num_objects, values, weights,
                                                                                max_weight,
                                                                                population_size, generations,
                                                                                tournament_size,
                                                                                mutation_rate)

    print("Genetic Algorithm Results:")
    print("Best Solution:", best_solution)
    print("Best Score:", best_score)
    print("Average score:", average_valid_score)
    print("Runtime:", runtime, "seconds")


if __name__ == "__main__":
    while True:
        print("\nMenu:")
        print("1. Evolutive Knapsack Problem")
        print("2. Evolutive TSP Problem")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            genetic_algorithm()
```

2. **Data Table Genetic Algorithm Knapsack**

Mutation rate: 0.2

Problem Instance	Generations	Population size, tournament size	Average value	Best Value	Nr of executions	Average runtime

`num_objects = 5`
`values = np.array([10, 5, 15, 7, 6])`
`weights = np.array([2, 3, 5, 7, 1])`
`max_weight = 10`
	100	5, 2	22.86	31		0.06

	1000	10, 2	22.6	31		0.64
	1000	15, 4	22.46	31		2.15
	10000	5, 2	22.33	31		13.34
	10000	10, 2	22.58	32	10	21.3
Rucsac-20.txt	50	30, 4	483	653		0.03
	100	30, 6	513	658		0.07
	50	50, 4	513	678		0.14
	100	50, 6	513	701		0.68
	10000	80, 20	516	726		6.7
Rucsac-200.txt	50	30, 4	126481	132023		0.04
	100	30, 6	126667	132161		0.1
	50	50, 4	127314	132286		0.18
	100	50, 6	128716	132886		1.05
	10000	100, 10	129873	133717		9.3

Mutation rate: 0.35

Problem Instance	Generations	Population size, tournament size	Average value	Best Value	Nr of executions	Average runtime

`num_objects = 5`
`values = np.array([10, 5, 15, 7, 6])`
`weights = np.array([2, 3, 5, 7, 1])`
	100	5, 2	22.86	31		0.06

	1000	10, 2	22.6	31		0.64
	1000	15, 4	22.46	32		2.15
	10000	5, 2	22.33	32		13.34
	10000	10, 2	22.58	33	10	21.3

`max_weight = 10`						
Rucsac-20.txt	50	30, 4	483	663		0.03
	100	30, 6	513	667		0.07
	50	50, 4	513	695		0.12
	100	50, 6	513	714		0.63
	10000	80, 20	516	798		6.4
Rucsac-200.txt	50	30, 4	126481	132431		0.04
	100	30, 6	126667	132987		0.1
	50	50, 4	127314	133341		0.18
	100	50, 6	128716	134617		1.05
	10000	100, 10	129873	135546		9.3

Data Analysis:

Average and Best Values: Across different configurations, the average values remain relatively consistent, while the best values vary more significantly. Despite higher average values compared to SAHC, the best scores obtained by the Genetic Algorithm are not higher.

Runtime Comparison: The Genetic Algorithm generally shows longer runtimes compared to SAHC, especially for larger generations and population sizes. Increasing these parameters leads to longer runtimes due to more extensive exploration and processing.

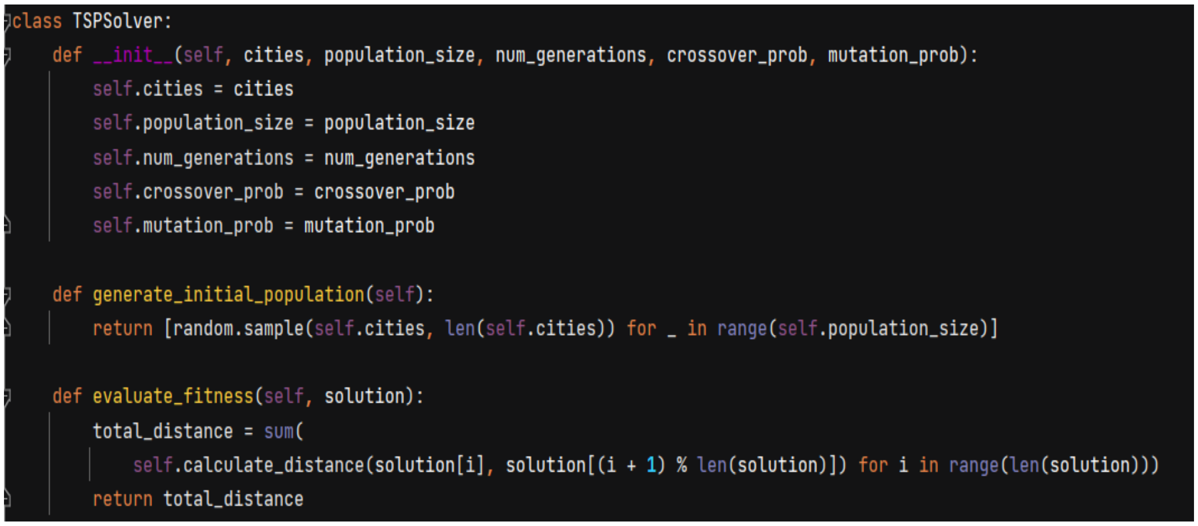
Mutation rate Analysis: Across using the same configurations as before, having a bigger mutation rate seems to make the best scores from the populations better, but that won't always be the case as for a mutation rate of over 0.5 we observe that the best values start to decrease from the one with a mutation rate of only 0.2

Problem: Traveling Salesman Problem(TSP)

Problem Explanation: The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization. Given a set of cities and the distances between each pair of cities, the objective is to find the shortest possible route that visits each city exactly once and returns to the starting city. The problem is named after the scenario faced by a salesman who needs to visit a set of cities to sell products, aiming to minimize the total distance traveled while visiting each city exactly once. The TSP is considered one of the most well-known NP-hard problems, meaning that finding an optimal solution becomes increasingly difficult as the number of cities increases.

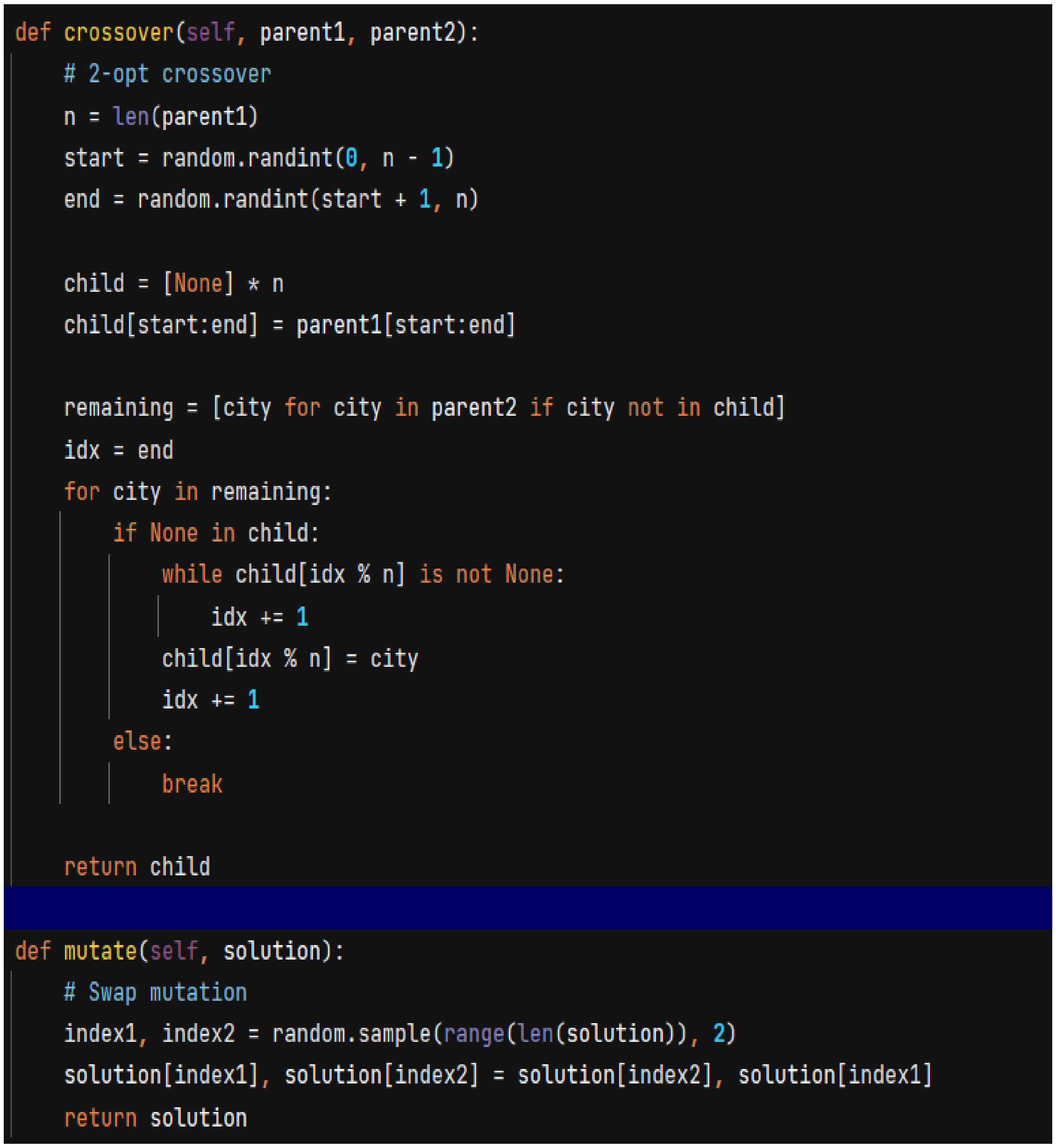
3. Algorithm 2 – Evolutive algorithm TSP

Step 1: We build a class for the TSP problem which has a few parameters like a list of cities, the population size, the number of generations through which we want to achieve the shortest route, and 2 probabilities, one for the crossover probability that the parents would reproduce, and one is the probability that the offspring would go through a mutation. After that, we have a function that gives us a random population to start with, basically a sample of 30(initially) individual solution of routes. Then we have a evaluate function(our fitness function) that calculates the total route that a solution has. This will help us when choosing the best solutions.



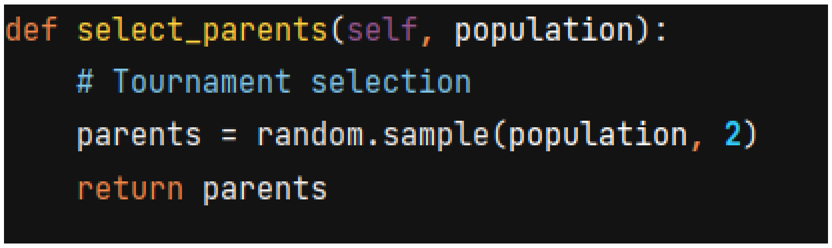
```python
class TSPSolver:
    def __init__(self, cities, population_size, num_generations, crossover_prob, mutation_prob):
        self.cities = cities
        self.population_size = population_size
        self.num_generations = num_generations
        self.crossover_prob = crossover_prob
        self.mutation_prob = mutation_prob

    def generate_initial_population(self):
        return [random.sample(self.cities, len(self.cities)) for _ in range(self.population_size)]

    def evaluate_fitness(self, solution):
        total_distance = sum(
            self.calculate_distance(solution[i], solution[(i + 1) % len(solution)]) for i in range(len(solution)))
        return total_distance
```

Step 2: Generating the descendants in two ways: 1. The first function named crossover uses the 2-opt swap by taking a random number of cities from the first parents and then the n-random number from first city, where n is the total number of cities in a solution(example: for a 76 city solution we would take the first 30 cities from the first parents and then the last 76-30=46 cities from the second one). 2. The second function is a mutation function that would generate a new solution from the same solution, it doesn’t need two solutions as the one before. Naturally, the rate at which this would happen in the evolutive algorithm will be lower than the crossover rate.



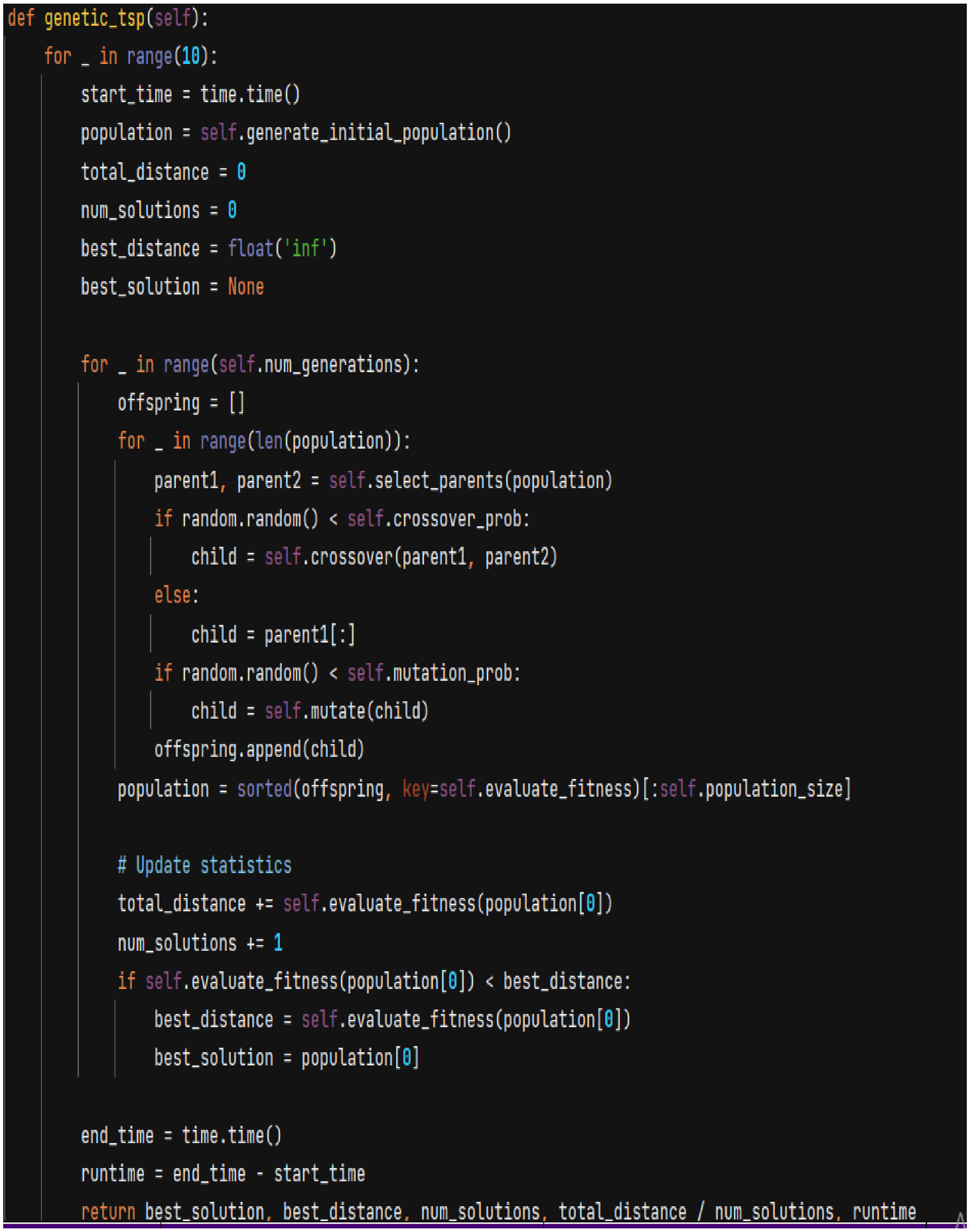
```python
def crossover(self, parent1, parent2):
    # 2-opt crossover
    n = len(parent1)
    start = random.randint(0, n - 1)
    end = random.randint(start + 1, n)

    child = [None] * n
    child[start:end] = parent1[start:end]

    remaining = [city for city in parent2 if city not in child]
    idx = end
    for city in remaining:
        if None in child:
            while child[idx % n] is not None:
                idx += 1
            child[idx % n] = city
            idx += 1
        else:
            break

    return child

def mutate(self, solution):
    # Swap mutation
    index1, index2 = random.sample(range(len(solution)), 2)
    solution[index1], solution[index2] = solution[index2], solution[index1]
    return solution
```

Step 3: In the select_parents function we will use a tournament selection that would return 2 random solutions from the population which then will be crossed over in the eolutive_tsp function. This ensures impartiality and equal chances for each solution to reproduce and take its traits further.



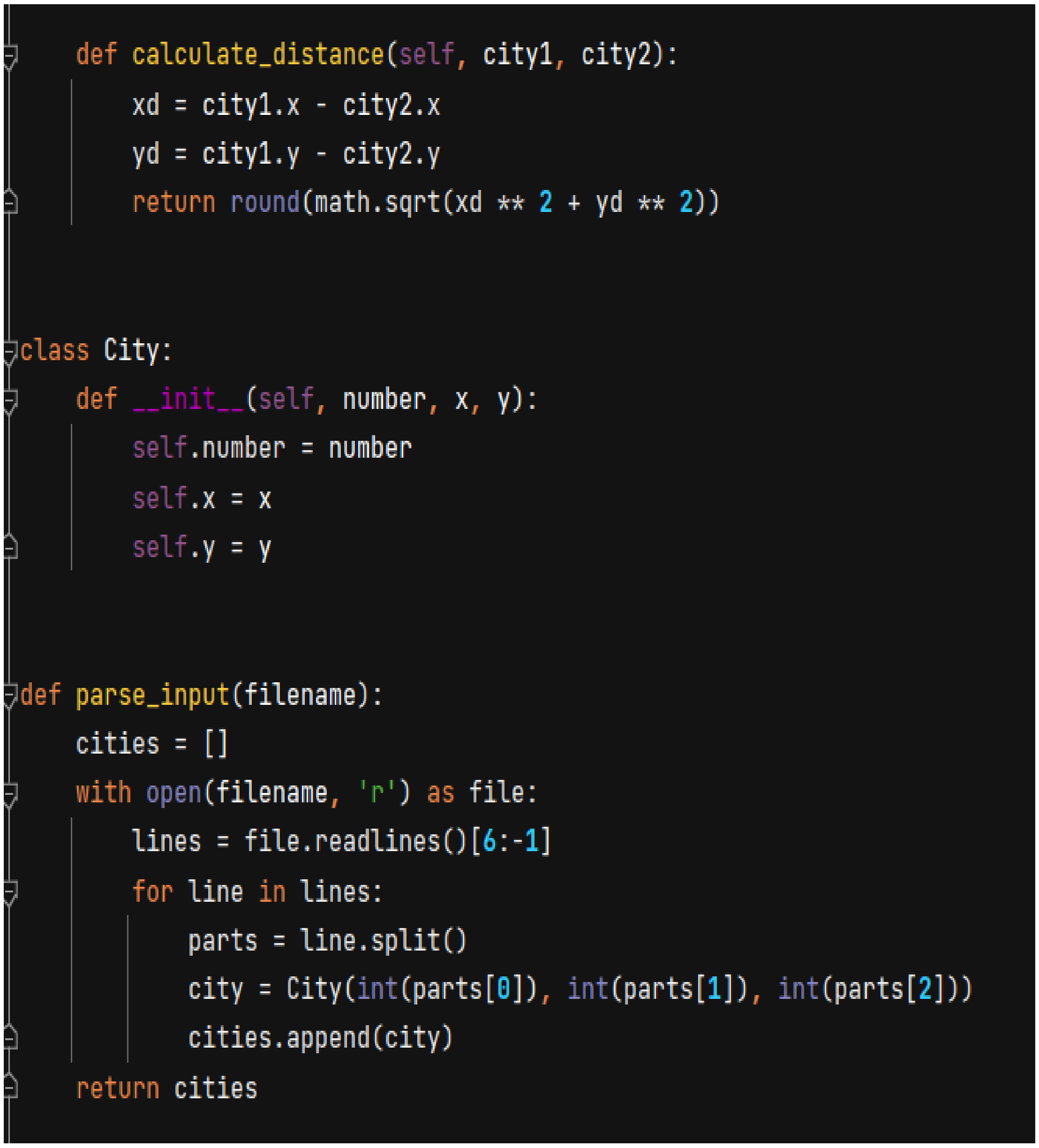
```
def select_parents(self, population):
    # Tournament selection
    parents = random.sample(population, 2)
    return parents
```

Step 4: This is the most important function of our algorithm that start by running the following steps for at least 10 times(10 times in this particular case) so that we would get an average best solution from the searching space. We then iterate through the generations and using the functions from the steps above we choose two parents using the tournament selection and crossinv them over at the crossover_rate and in another case we mutate a single solution at the mutation_rate. We then update our statistics, the best score for a solution and the total_distance and also the number of solutions that we found after the crossover/mutation step. We then return the parts that we are interested in: the best solution, best score, the numver of solutions, the average distance of solutions calculated by the arithmetic average between the total distance and the number of solutions and the average runtime it takes the evolutive algorithm to run.



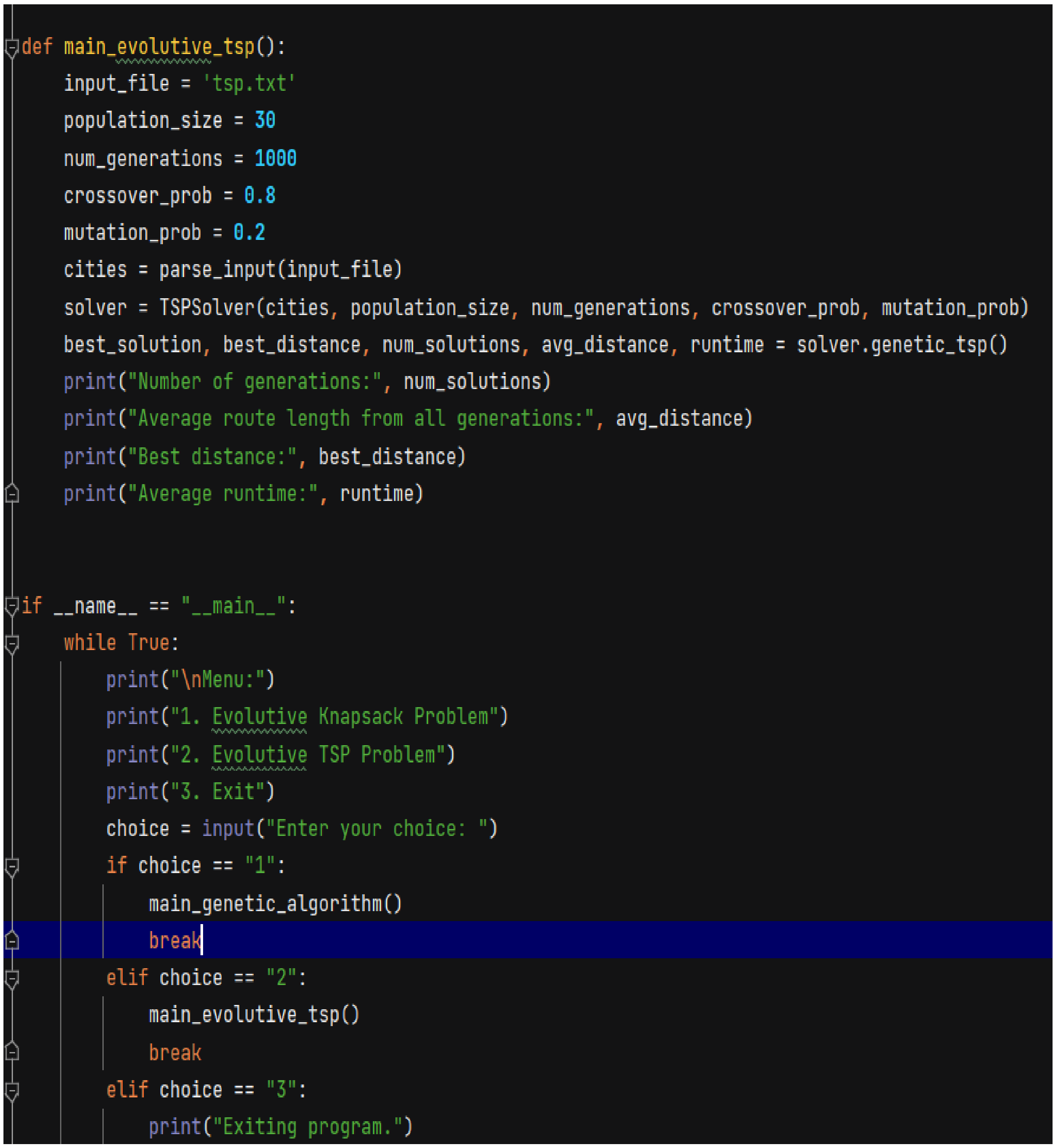
```python
def genetic_tsp(self):
    for _ in range(10):
        start_time = time.time()
        population = self.generate_initial_population()
        total_distance = 0
        num_solutions = 0
        best_distance = float('inf')
        best_solution = None

        for _ in range(self.num_generations):
            offspring = []
            for _ in range(len(population)):
                parent1, parent2 = self.select_parents(population)
                if random.random() < self.crossover_prob:
                    child = self.crossover(parent1, parent2)
                else:
                    child = parent1[:]
                if random.random() < self.mutation_prob:
                    child = self.mutate(child)
                offspring.append(child)
            population = sorted(offspring, key=self.evaluate_fitness)[:self.population_size]

            # Update statistics
            total_distance += self.evaluate_fitness(population[0])
            num_solutions += 1
            if self.evaluate_fitness(population[0]) < best_distance:
                best_distance = self.evaluate_fitness(population[0])
                best_solution = population[0]

        end_time = time.time()
        runtime = end_time - start_time
        return best_solution, best_distance, num_solutions, total_distance / num_solutions, runtime
```

Step 5: The calculate_distance function is used in the fitness function and calculates the euclidian distance between two points in our 2D space(in this case two cities). Then we build a class City which remembers the city number and the coordinates of a city. We then use it in the next function which parses our tsp.txt, the file which stores all the informations about our cities. The first 6 lines have generic information about the problem and the next ones contain information in this format: a(city number) x(x coordinate of the city) y(y coordinate of the city). We save all this information in a list of City objects.



```
    def calculate_distance(self, city1, city2):
        xd = city1.x - city2.x
        yd = city1.y - city2.y
        return round(math.sqrt(xd ** 2 + yd ** 2))


class City:
    def __init__(self, number, x, y):
        self.number = number
        self.x = x
        self.y = y


def parse_input(filename):
    cities = []
    with open(filename, 'r') as file:
        lines = file.readlines()[6:-1]
        for line in lines:
            parts = line.split()
            city = City(int(parts[0]), int(parts[1]), int(parts[2]))
            cities.append(city)
    return cities
```

Step 6: In the main file of the program we create a main_evolutive_tsp function that gives all the information needed for running the program which can be edited in order to test for different values of generations, population sizes and more. We then gives these information to the user when inputting the ‘2’ value when given a choice.



```
def main_evolutive_tsp():
    input_file = 'tsp.txt'
    population_size = 30
    num_generations = 1000
    crossover_prob = 0.8
    mutation_prob = 0.2
    cities = parse_input(input_file)
    solver = TSPSolver(cities, population_size, num_generations, crossover_prob, mutation_prob)
    best_solution, best_distance, num_solutions, avg_distance, runtime = solver.genetic_tsp()
    print("Number of generations:", num_solutions)
    print("Average route length from all generations:", avg_distance)
    print("Best distance:", best_distance)
    print("Average runtime:", runtime)


if __name__ == "__main__":
    while True:
        print("\nMenu:")
        print("1. Evolutive Knapsack Problem")
        print("2. Evolutive TSP Problem")
        print("3. Exit")
        choice = input("Enter your choice: ")
        if choice == "1":
            main_genetic_algorithm()
            break
        elif choice == "2":
            main_evolutive_tsp()
            break
        elif choice == "3":
            print("Exiting program.")
```

Mutation probability : 0.2, Crossover probability: 0.8

Population size: 15,tournament size: 4

Problem Instance	Generations	Average route length	Best route	Nr of executions	Average runtime
Pr76.tsp	50	520959	491017		0.15
	100	523546	469919		0.29
	500	521686	451344		1.6
	1000	525390	459884	10	2.99
	10000	524588	448123		29.9
	100000	525101	444754		302.0
Population size: 30, tournament size: 10					
Problem Instance	Generations	Average route length	Best route	Nr of executions	Average runtime
Pr76.tsp	50	510089	483153		0.169
	100	500928	456720		0.33
	500	500021	451132		1.78
	1000	499182	443599	10	3.29
	10000	498214	434178		33.3
	100000	481723	420878		330

Mutation probability : 0.3, Crossover probability: 0.7

Population size: 15,tournament size: 4

Problem Instance	Generations	Average route length	Best route	Nr of executions	Average runtime
Pr76.tsp	50	520959	487486		0.15

	100	523546	478933		0.29
	500	521686	471521		1.6
	1000	525390	462310		2.99
	10000	524588	461123	10	29.9
	100000	525101	458924		302.0
Population size: 30, tournament size: 10					
Problem Instance	Generations	Average route length	Best route	Nr of executions	Average runtime
Pr76.tsp	50	510089	490182		0.169
	100	500928	487366		0.33
	500	500021	482233		1.78
	1000	499182	470923	10	3.29
	10000	498214	468324		33.3
	100000	481723	451825		330

Mutation probability : 0.05, Crossover probability: 0.95

Population size: 15,tournament size: 4

Problem Instance	Generations	Average route length	Best route	Nr of executions	Average runtime
Pr76.tsp	50	520959	478531		0.15
	100	523546	462184		0.29
	500	521686	446791		1.6
	1000	525390	439623	10	2.99
	10000	524588	431597		29.9
	100000	525101	421623		302.0

Population size: 30, tournament size: 10					
Problem Instance	Generations	Average route length	Best route	Nr of executions	Average runtime
Pr76.tsp	50	510089	471829	10	0.169
	100	500928	446305		0.33
	500	500021	435679		1.78
	1000	499182	424798		3.29
	10000	498214	414256		33.3
	100000	481723	400803		330

Data Analysis:

Average Route Length: As the number of generations increases, the average length of the routes generally decreases. This trend indicates that the algorithm tends to find shorter routes on average with more generations, which aligns with our expectations.

Best Route: Similarly, the length of the best route decreases as the number of generations increases. This implies that the algorithm is progressively able to find better solutions with higher numbers of generations, suggesting that it explores and optimizes the solution space effectively.

Average Runtime: The average runtime of the algorithm is proportional to the number of generations. For instance, with 100 generations, the algorithm takes approximately 0.29 seconds, while with 100,000 generations, it takes about 302.0 seconds. This linear relationship between runtime and the number of generations highlights the computational cost of exploring the solution space exhaustively.

Comparison with Simulated Annealing: While the evolutionary algorithm improves in finding shorter routes with less generations, it may not match the efficiency of Simulated Annealing (SA) for the same problem instance. SA algorithms, leveraging probabilistic techniques like annealing, often excel at finding near-optimal solutions, especially in large solution spaces. Therefore, while the evolutionary algorithm shows promise in route optimization, SA algorithms may outperform it in terms of efficiency and effectiveness, particularly in certain scenarios. The data table showcases this comparison by coloring with red the cells in which SA got better results than the GA and with green the reverse effect.

Mutation rate Comparation: When increasing the mutation and at the same time decreasing the crossover probability of these events happening we see comparing the first and second data tables that the values of the shortest routes get worse(bigger). At the same time, when reducing the mutation rate to a much lesser value of only 0.05 and thus, increasing the crossover probability to 0.95 we see that our route values get better, using the same number of generations, tournament size and other parameters.