Laborator 2 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 - Tabu Search

Step 1: We will reuse the functions from SAHC (Simulated Annealing Hill Climbing) for generating a random solution (the starting one), validating the solution (ensuring its weight is less than W), calculating the score, and generating neighbors.

```python
import time
import numpy as np

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


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


def evaluate_solution(solution, values):
    return np.sum(solution * values)


def get_neighborhood(current_solution):
    neighborhood = []
    num_objects = len(current_solution)

    for i in range(num_objects):
        neighbor = np.copy(current_solution)
        neighbor[i] = 1 - neighbor[i]  # Bit swap
        neighborhood.append(neighbor)

    return neighborhood
```

Step 2: Tabu Class

We create a class for Tabu Search which includes the constructor and the search function. We start by executing the search 10 times and declaring the variables we will use.

```
class TabuSearch:

    def __init__(self, num_objects, values, weights, max_weight, tabu_list_size):
        self.num_objects = num_objects
        self.values = values
        self.weights = weights
        self.max_weight = max_weight
        self.tabu_list = []
        self.tabu_list_size = tabu_list_size

    def search(self, iterations):
        for _ in range(10): # 10 rulari
            best_solution = None
            best_score = float('-inf')
            valid_scores = []
            valid_weights = []
            start_time = time.time()
```

Next, we go through iterations, and if we find a better solution in the neighborhood, we add it to the tabu list, which has a size of 10. If the list becomes full, we start popping from it to keep the best solutions (the most recent ones). Then, we calculate the runtime and the averages.

```
        for _ in range(iterations):
            current_solution = generate_random_solution(self.num_objects)
            current_score = evaluate_solution(current_solution, self.values)

            if is_valid(current_solution, self.weights, self.max_weight):
                valid_scores.append(current_score)
                valid_weights.append(np.sum(current_solution * self.weights))

            if current_score > best_score:
                best_solution = current_solution
                best_score = current_score

            if len(self.tabu_list) >= self.tabu_list_size:
                self.tabu_list.pop(0)

            self.tabu_list.append(current_solution)

        end_time = time.time()
        runtime = end_time - start_time

        # Calculate average score and weight of valid solutions
        average_score = sum(valid_scores) / len(valid_scores) if valid_scores else 0
        average_weight = sum(valid_weights) / len(valid_weights) if valid_weights else 0

        return iterations, average_score, average_weight, best_score, runtime, best_solution
```

Step 3: Reuse parsing function to iterate through data.txt, which will contain data from rucsac20.txt and rucsac200.txt

```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 4: In the main function, create a function called main_tabu_search where we will display all the values to make our data table (number of iterations, average values, average weights, best score, execution time), and then we will create a menu where if the user inputs "1", we will display the desired values.

```python
def main_tabu_search():
    file_path = "data.txt"
    num_objects, values, weights, max_weight = parse_rucksack_data(file_path)
    iterations = 1000
    tabu_list_size = 10

    tabu_search = TabuSearch(num_objects, values, weights, max_weight, tabu_list_size)
    k, avg_score, avg_weight, best_score, runtime, best_solution = tabu_search.search(iterations)

    print("Tabu Search Algorithm Results:")
    print("k (iterations):", k)
    print("Average Score:", avg_score)
    print("Average Weight:", avg_weight)
    print("Best Score:", best_score)
    print("Runtime:", runtime, "seconds")
    print("Best Solution:", best_solution)


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

        choice = input("Enter your choice: ")

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

2. Data Table Tabu Search

Problem Instance	K	Average Value(Score)	Average Weight	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	50	15.657	5.971	43	10	0.0009
	100	14.950	5.557	43		0.0009
	200	17.047	6.46	43		0.003
	1000	15.62	5.86	43		0.012
	10000	16.195	6.006	43		0.113
Rucsac-20.txt	50	450.1	429.1	739		0.0100
	100	414.9	428	788		0.001
	200	409.8	411.66	879		0.002
	1000	416.79	423.52	932		0.015
	10000	411.1	417.43	948		0.15
Rucsac-200.txt	50	123450	105642	152798		0.0009
	100	122077	104623	161310		0.003
	200	122749	104925	162390		0.005
	1000	123472	105532	165327		0.02
	10000	123153	105428	178126		0.2

Data Analysis + Comparison with SAHC:

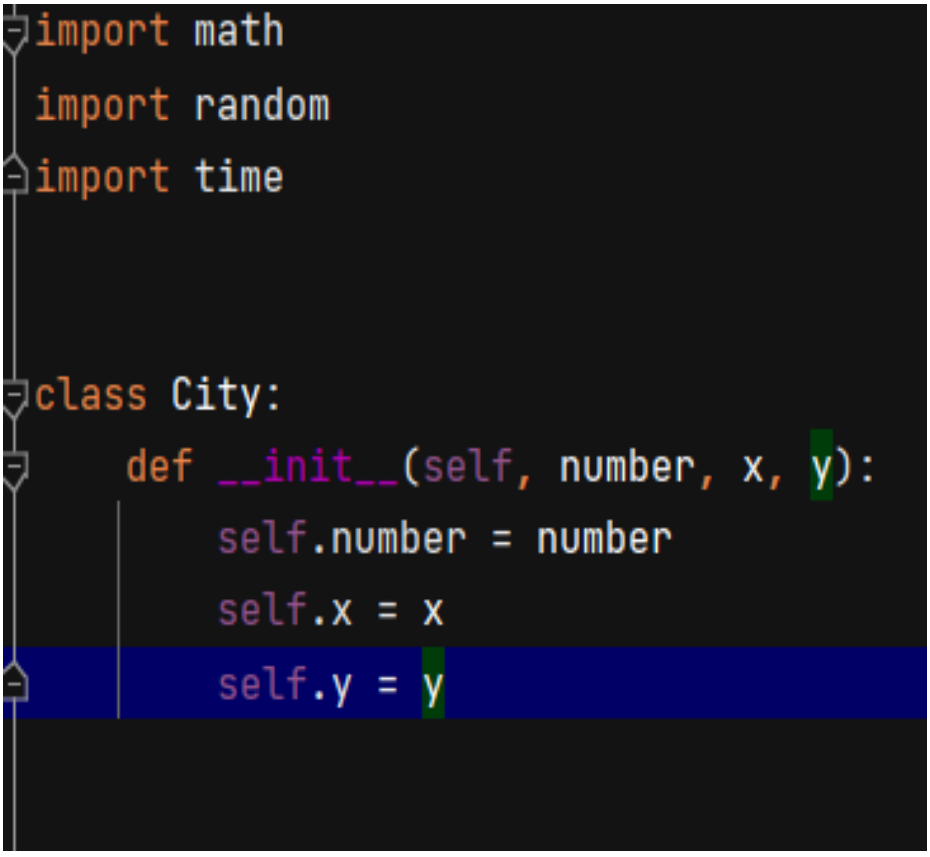
We observe that Tabu Search achieves a higher best score value than SAHC, starting from a number of iterations >= 200. At the same time, it has a much faster execution time compared to SAHC, sometimes even 100 times faster. Therefore, we can conclude that Tabu Search is a better algorithm for exploitation, being able to reach a maximum point more rapidly, and also having characteristics to escape from a local maximum, whereas SAHC is a more exploratory algorithm. (Explanation of table: The cells colored in green are the cells where Tabu Search achieves a higher best value than the table from Lab1 of SAHC.)

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. Simmulated Annealing(SA)

Step 1: Create a class City that keeps track of thenumber of the city and the 2D coordinates of the cities that we will encounter in our problem



Step 2: Create a parsing function that would go through our tsp.txt file that contains for the first 6 lines details about the problem and the next lines have on the first column the number of the city, the second the first coordinate(x) and on the third column the y coordinate. The parsing function will return a list of City type objects. Then, we will create another function that

calculates the euclidean distance between two cities so we know the values of the roads between them.

```
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


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

Step 3: In this step we create 3 new functions, the first one would generate us a random permutation of cities that will represent our starting point. Then we have a swap function so that we can evaluate more solutions. Basically, we would get the starting point permutation and choose 2 cities from it. Now, we will use the 2-opt method of swapping that would take the

cities between the 2 chosen ones(let's name them x and y) and we would reverse the whole road between x and y and paste it back into the permutation so now we would have a new different solution from the initial one. The 3rd and last function of this step is an evaluate one that calculates the total distance of roads in a solution.

```
def initial_solution(cities):
    return random.sample(cities, len(cities))


def swap(cities):
    # 2-opt neighbor generation
    new_cities = cities[:]
    i, j = random.sample(range(len(cities)), 2)
    if i > j:
        i, j = j, i
    new_cities[i:j+1] = reversed(new_cities[i:j+1])
    return new_cities


def evaluate(solution):
    total_distance = 0
    for i in range(len(solution)):
        total_distance += euclidean_distance(solution[i], solution[(i + 1) % len(solution)])
    return total_distance
```

Step 4: We will build our simmulated annealing function by first having a for meant for executing the problem 10 times. We generate a random solution that we will store in a current_solution variable but also in best_solution as for now it is our best solution in terms of shortest route. We also need a T variable that will be our temperature, which we will use as a

searching parameter. We will first start with a big value for it in order to evaluate more solutions but as we progress and go through more iterations the T is gradually reduced which will make our functions become more strict with the evaluation of solutions, thus in the end when the T is at a minimum value, we would have the best solution in the space. We could see this algorithm as a radar that keeps searching but with each step the zone searched becomes smaller and smaller. If we find an evaluation of the new solution better than the current one we swap the current solution with the new one and so on, or using the –delta/T formula for accepting the solution where delta is the subtraction of route lenghts between the two solutions. Then, we use T and alpha to shrink the temperature gradually(alpha is a sub 1 number and the give T the value T*alpha, basically lowering it). In the end we compare the current distance with the best one and give the best_distance the lowest value and the best_solution the current solution. In the end we also calculate the runtime of the function using the time library in Python.

Step 5: We will add a main_SA_TSP function in main that will be called when the user inputs the value '2' and will print out the data needed for our datatable: Nr of iterations, Average length of routes, Best route, Runtime.

```python
def simulated_annealing(cities, max_iterations, T_max, T_min, alpha):
    for _ in range(10):  # 10 rulari
        start_time = time.time()
        current_solution = initial_solution(cities)
        best_solution = current_solution
        best_distance = evaluate(best_solution)
        T = T_max
        total_distance = 0
        num_solutions = 0

        for k in range(max_iterations):
            new_solution = swap(current_solution)
            delta = evaluate(new_solution) - evaluate(current_solution)

            if delta < 0 or random.random() < math.exp(-delta / T):
                current_solution = new_solution

            distance = evaluate(current_solution)
            total_distance += distance
            num_solutions += 1

            if distance < best_distance:
                best_solution = current_solution
                best_distance = distance

            T *= alpha
            if T < T_min:
                break

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

```python
def main_SA_TSP(filename, max_iterations, T_max, T_min, alpha):
    cities = parse_input(filename)
    num_solutions, total_distance, best_distance, runtime = simulated_annealing(cities, max_iterations, T_max, T_min, alpha)
    average_distance = total_distance / num_solutions
    print("Number of iterations:", num_solutions)
    print("Average length of route:", average_distance)
    print("Best route:", best_distance)
    print("Runtime:", runtime)
```

4. Data Table Simmulated Annealing

T_max = **10000**

T_min = **0.00001**

alpha = **0.9999**

Problem Instance	k	Average route length	Best route	Nr of executions	Average runtime
Pr76.tsp	50	529422	519029	10	0.008
	100	582449	548765		0.02
	500	513881	474069		0.08
	1000	500805	439180		0.15
	10000	452766	330146		1.5
	100000	173432	116808		15.4

Data Analysis:

Average Route Length: As the number of iterations (k) increases, the average length of the routes generally decreases. With more iterations, the algorithm tends to find shorter routes on average which is what we expected. **Best Route:** Similarly, the length of the best route decreases as the number of iterations increases. The algorithm is able to find better solutions with higher numbers of iterations. **Average Runtime:** The average runtime of the algorithm also increases with the number of iterations. This is expected, as more iterations require more computational time to explore the solution space and find better solutions.