Project 2: Vehicle Routing Problem

- Janosch Baltensperger (janosch.baltensperger.7118@student.uu.se)
- Domenic Fürer (domenic-luca.furer.2478@student.uu.se)
- Marion Dübendorfer (marion-francine.dubendorfer.0571@student.uu.se)

Representation Method and Capacity Limitations

To represent our solution, we created the dataclass chromosome, which contains two equal length arrays and a fitness value. The first array contains all the stops which means that it is in order-based representation. The second array is in integer representation and contains integers ranging from 0 to 8 depicting the different vehicles. Together they represent a solution in such that each vehicle visits all the stops where its own number is at the same index as the stop. The order of the visits is in the order of the appearance of the stops in the stops array. Hence, this technique is a valid approach to represent a solution. The fitness value is later determined by applying the fitness function to a chromosome.

To address vehicle capacities, we used a penalty cost in the fitness function. For each vehicle, it is checked whether it is overweight or not. If it is, a penalty cost of $(W_{vehicle} - W_{max}) * 1000$ is applied, leading to the fact that overweight solutions are penalised drastically.

Instructions

We recommend to use a Python version > 3.7. There is a dependency of `matplotlib`, which can be installed with `pip install -r requirements.txt`. To run the script, use `python vrp.py` .

Algorithm Implementation

For our project, we decided to use a Genetic Algorithm. The algorithm itself consists of multiple functions.

- A `main` function serves as the entry point for the program. It executes multiple runs of the algorithm and keeps track of the total CPU time and the best result of the experiment.
- The `calculate_map_context` and `calc_dist` function are executed once in the beginning before running `ga_solve`. They read the data from a csv, create a distance matrix, a demands array and store the x and y positions of the point for plotting them. The data is stored in global variable so that each function can access them if necessary without needing to pass them to every function.
- The `plot_map` function is used once in the end to plot the map of the optimal solution
- The `ga_solve` is the main part of the program. It implements a genetic algorithm according to the pseudo code shown in the lecture. We do not need to pass any arguments as all parameters are set as global variables in the beginning of the program.

- Two functions named `gen_chromosome` and `gen_population` are used to create the initial population according to the defined parameter. The permutation array is created by shuffling an array containing all the numbers while the integer array is created by randomly selecting a vehicle.
- The functions `calculate_weights` and `calculate_path_costs_and_weights` are used to calculate either only the weights or the path costs and the weights depending on what is need in other functions. The basically iterate over one array of the chromosome and calculate the route costs and the weight each vehicle has.
- The `evaluate_fitness` and `fun_fitness` functions are used to assign a fitness value to a chromosome. To do so, the evaluate function first calls the `calculate_path_costs_and_weights` function to obtain the individual routes and weights of the corresponding vehicles. Afterwards, by using `fun_fitness` the penalty term is added to each route if the vehicle is overweight.
- The selection function is implement in `select_parent`. It implements the roulette wheel selection by determining the total fitness of all chromosomes and then setting the probabilities accordingly. We also implemented the rank selection at one point. However, it turned out that the algorithm had a worse performance in terms of CPU time and in path costs using rank selection. Therefore, the roulette wheel selection was used in the end.
- The `do_crossover` function is responsible to create a new child from two parents. First, it does the crossover for the permutation array using the order crossover. To do so, it selects two points randomly and assigns the offspring all the values from parent1 in between the two selected points. Afterwards, the unused stops are collected from parent2 and assigned to the offspring in the order they appear in parent2. Thereafter, two children are created. Each contains the newly generated stops array but they differ in the vehicles array in which one contains the vehicles array from parent1 whereas the other has the vehicles array from parent2. Lastly, the fitness of both children is evaluated and only the better one is returned as the final offspring.
- Mutations in the algorithm are applied using the `do_mutation` function. During testing, we realized that our mutation functions are the most helpful to converge to a solution in a reasonable time. Therefore, we implemented different mutations which are `swap_gene_stops`, `swap_gene_vehicles` , `exchange_gene_vehicles`, `two_opt_one_path`, and `two_opt_two_paths`. The number of applied mutations can be influence by setting the `MUTATION_RATE` and the `NO_OF_MUTATIONS` parameters. Initially, we tried to only apply swapping mutations. By using only the simplest mutation, the algorithm also converged to a good solution but needed around 12’000 generations and could not fulfil the time limit. Hence, we implemented additional mutations which help converging way faster. The function starts by making a copy of the old chromosome. This is used in case that the mutation led to a worse fitness value in order to revert the chromosome to the before mutation point. Afterwards, it loops `NO_OF_MUTATIONS` times. In this loop, the function takes a random uniformly distributed number between 0 and 1 and checks it against the `MUTATION_RATE`. If it is smaller, we are doing on of the five mutations mentioned above. Lastly, it checks the

fitness and reverts the chromosome if necessary. The complexity of the five mutations differ quite heavily. First, there is the `swap_gene_stops`. This function chooses two stops randomly and swaps the stops of these points. Similarly, there is the `swap_gene_vehicles`, which also swaps the alleles of random genes in the vehicles array. Due to the fact that the number of stops each vehicle takes is fixed by the initial creation of a chromosome, we also implemented the `exchange_gene_vehicles` function. It takes a random stop and changes the vehicle. To avoid creating unfeasible solutions, a check is made that only vehicles, which can add the demand of the selected stop without being overweight, can be selected. The two other important mutations are the ones using the `two_opt_route` function. This function adjusts a route such that it has no twists in it. The `two_opt_one_path` mutation checks this for a randomly selected route of a vehicle and changes its route if necessary. The `two_opt_two_paths` mutation on the other hand is for checking that two random paths do not overlap. For this, two random vehicles and their routes are selected. Then the two stops with the maximum distance are selected and used as starting points for the new routes. Afterwards the remaining stops of both routes are distributed between the two new routes by adding the currently looked at stop to the closer route. Lastly, the routes are being freed from twists using the `two_opt_route` function and the original chromosome is adjusted.

- The `get_best_chromosome` function is a simple function that finds the chromosome with the highest fitness function inside a population.
- Lastly a function to map the genotype to the phenotype called `print_phenotype` is defined. It goes through one chromosome and prints the total path costs, the weights of the vehicles and the individual routes per vehicle to the console.

Configurations and Results

```
NO_GENERATIONS = 450
POPULATION_SIZE = 45
CROSSOVER_RATE = 0.9
MUTATION_RATE = 0.35
NO_OF_MUTATIONS = 7
OVER_WEIGHT_PENALTY = 1000
KEEP_BEST = True
```

Run #	Total Traveling Cost	CPU Time
1	1144.75	8.1309
2	1128.03	8.2426
3	1152.94	8.4503
4	1190.97	8.2684
5	1138.74	8.1955
6	1212.48	8.0918
7	1143.77	8.5531

Run #	Total Traveling Cost	CPU Time
8	1137.54	12.2919
9	1257.76	8.1339
10 **(best)**	1090.39	8.019
11	1093.71	8.199
12	1219.37	8.3164
13	1227.65	8.5723
14	1174.4	8.5719
15	1139.7	8.4479
16	1167.05	8.4952
17	1248.33	8.552
18	1107.12	8.4179
19	1168.2	8.6139
20	1149.33	8.5706
21	1132.06	8.4842
22	1160.69	8.4887
23	1124.49	8.495
24	1147.89	8.7266
25	1134.89	8.557
26	1176.31	8.6367
27	1200.65	8.6381
28	1182.97	8.8521
29	1167.98	8.6063
30	1137.55	8.5961
	Avg: **1161.92367**	Avg: **8.57384333**

Best result in detail

- Runtime of the algorithm for the best solution: 8.02s
- Absolute difference of optimal solution: 17.39
- Relative difference of optimal solution: 1.62%
- The total costs of the paths are: 1090.39
- Vehicle weights: [94, 100, 90, 99, 94, 100, 98, 82, 82]
- Routes:
  Route #1: 6 45 1 50
  Route #2: 12 10 5 53 40 16 38 32

Route #3: 48 44 24 52 23 18
Route #4: 43 35 9 49 39 47
Route #5: 36 33 2 51 11 15
Route #6: 4 7 42 31 20 46 26
Route #7: 30 22 19 27 13 54 28
Route #8: 25 41 29 21 8
Route #9: 14 17 34 3 37