Capacitated Arc Routing Problem Project

Hao Zheng 11610127
Department of Computer Science and Engineering
Southern University of Science and Technology
11610127@mail.sustc.edu.cn

1. Preliminaries

This project is implementing an evolutionary algorithm to solve Capacitated Arc Routing Problem (CARP), which belongs to the class of NP-hard problem.

1.1. Problem Description

Given a connected undirected graph $G(V, E)$, with non-negative cost and demands on edges. There is a fleet of vehicles that can serve required edges with limited capacity. The objective is to search for a set of routes at minimum cost, which start from, and end at the depot. Of course, each route is passed by a vehicle, serving some of the required edges and without exceeding the vehicle capacity.

1.2. Problem Application

One of the application is garbage collection, in which each route is consist of garbage-collecting edges and pass-by edges. The capacity of vehicles then represent the load of garbage. In order to satisfy all the garbage-collecting tasks within the vehicle capacity, while minimizing the total cost of routes, CARP algorithm can be used in this scenario.

1.3. Algorithm

The algorithm used in this project is genetic algorithm (GA) without crossover operator. In order to generate initial population with less costs, path-scanning algorithm is also implemented. As for the detail of algorithms, I will explain them in the next section.

2. Methodology

2.1. Notation

- Q : the vehicle's capacity
- C : the cost of edge
- D : the demand of edge
- R : one of the routes

2.2. Data Structure

- min_dst: 2×2 matrix, which stores the min cost between each node
- edges: hashMap, mapping the (x, y) Tuple to the edge object
- tasks: hashMap, mapping the (x, y) Tuple to the edge object, but only containing required edges

2.3. Model Design

As is said in Description, the objective is to construct routes that serve all the required $(D \neq 0)$ edges. To express that problem more clearly, the mathematical formulation is needed.

Consider the following notation:

Q	the vehicle's capacity
D_{ki}	the demand of edge in the k^{th} route
ω_i	the i^{th} route
Ω	the number of total routes
e^i_j	the cost of j^{th} edge in the i^{th} route
E_i	the number of total edges in the i^{th} route
τ_{ki}	the i^{th} task served in the k^{th} route
N_k	the number of tasks in the k^{th} route

A mathematical programming formulation for the CARP is given below.

$$Minimize \sum_{i=0}^{\Omega} \sum_{j=0}^{E_i} e^i_j$$

subject to

$$\tau_{k_1 i_1} \neq \tau_{k_2 i_2}, \forall (k_1, i_1 \neq k_2, i_2)$$
$$\sum_{i=0}^{N_k} D_{ki} \leq Q, \forall k = 1, 2, 3..., \Omega$$

GA is an algorithm that apply crossover and mutation to the individuals in population of each generation. Althrouth there's a little difference between my implementation and the original GA, i.e. the crossover is ignored for performance debasement, my implementation roughly follow the thought of GA:

1) Constructing the initial population using path-scanning.

2) Expanding the size of initial population to given size using mutation.
3) For each iteration:
   a. Test each individual in population, whether it'll be removed or not.
   a. Select the "not removed" individuals and apply mutation to them with mutation probability.
   a. Add mutated children to population.
4) Finally, return the valid individual with the least cost.

2.4. Detail of algorithms

To apply less code while explaining the basic idea clearly, the "pseudocode-likely" python code is used in this section.

Before diving into the details, it's necessary to propose several definition:

- non valid generations: for the invalid solutions, it means the passed generations that it's already be a invalid solution; for the valid solutions, it's zero.
- discard probability: the probability that the solution will be discarded, according to the equation:

$$probability = \frac{2 \times load_{exceed}}{\sum load} \times 3^{gen_{non-valid}}$$

- tasks: hashMap, mapping the (x, y) Tuple to the edge object, but only containing required edges

In initialization, I apply path-scanning with 5 rules to generate 5 origin individuals. Then extend them to given population size.

```
def initialize():
    population = set()

    # generate according to 5 rules
    for rule in rules:
        result = path_scanning(rule)
        population.add(result)

    origin_5 = copy(population)

    # extend to size
    while len(population) < size:
        for p in origin_5:
            new = get a solution according
to one of mutations
            if random() > new.discard_prob:
                population.add(new)

    return population
```

As for the path-scanning, I won't give the code of it because it can be retrieved from Internet easily. Roughly, it's finished by selecting the nearest next task from current task. And if the costs is the same between current task and two different tasks, different rules can be used to determine which one to use.

The step, i.e. iteration function, is the core function in the algorithm.

```
def step():
    for individual in population:
        if random() > individual.discard_prob:
            if random() > mutation_rate:
                new = apply all the mutations
to individual, then select the min cost one
                if random() > new.discard_prob:
                    population.add(new)
        else:
            population.remove(individual)

    # get rid of the redundant individuals
    while len(population) > population_size:
        worst = the individual with max cost
        population.remove(worst)

    return the valid individual with least cost
```

All the mutations are the same of partial mutation operators mentioned by Tang *et al.* in [1]. Totally there are 3 kind of mutation:

1) single insertion: select arbitrary task, and insert it into the other position.
2) double insertion: select arbitrary two consecutive task, and insert them into the other position.
3) swap: select two arbitrary tasks, and swap their position.

3. Empirical Verification

3.1. Dataset

I use three datasets for verification: bccm, eglese and gdb. Each of them consists several data files.

3.2. Performance Measure

The performance measurement is done like that:

```
from time import perf_counter() as pc

start_time = pc()
perform testing in all the data for 20 times
end_time = pc()
write the average result of testing into excel
```

By the way, the test environment (CPU) is AMD Ryzen 7 1700 with 8 cores and 16 threads, overclocking to 3.6 MHz.

3.3. Hyperparameters

The most important hyperparameter is roughly population size, so I'll take it as an parameter tuning example.

To test the performance of population size, I take the performance measure with the population size from 40 to 170, then draw the chart.

3.4. Experimental Results

graphicx 1.png 2.png 3.png

3.5. Conclusion

Thus, the 40 is best population size.

References

[1] Tang, Ke, Yi Mei, and Xin Yao. ”Memetic algorithm with extended neighborhood search for capacitated arc routing problems.” IEEE Transactions on Evolutionary Computation 13.5 (2009): 1151-1166.