Capacitated Arc Routing Problem (CARP) Report

Wentao Ning 11610914
Computer Science and technology
Southern University of Science and Technology
11610914@mail.sustc.edu.cn

1. Preliminaries

The capacitated arc routing problem (CARP) has attracted much attention during the last few years due to its wide applications in real life. Since CARP is NP-hard and exact methods are only applicable to small instances, heuristic and metaheuristic methods are widely adopted when solving CARP[1] In this project, I used Genetic Algorithm to solve this NP-hard problem

1.1. Software

In this project, I used Pycharm mainly and Sublime Text 3 cosmetically. I wrote this project in Python3.

1.2. Algorithm

I used genetic algorithm to solve this problem. At first, I used Path-Scanning algorithm to get some initial results. I wrote 5 rules to decide which edge should be chosen when some edges has the same distance from my current position. The I wrote 5 operators to make crossover and mutation of the initial results. I choose about 300 optimal results every time and continue mutation until time over.

2. Methodology

2.1. Representation

There are several variables in my python file.

- **args**: Lists of command-line arguments
- **co**: NumPy 2D adjacent matrix
- **required_edges_num**: Number of required edges
- **vertices_num**: Number of vertices
- **depot_num**: Depot of this dataset
- **capacity**: Capacity of each vehicle

2.2. Architecture

There are also several methods in my file.

- **floyd**: Using Floyd algorithm to calculate distances between points and points
- **path_scanning**: To produce initial solutions of this problem
- **better**: Choose edge when multiple edges are equal distances from the current point using following 5 rules
  - maximize the distance from the task to the depot;
  - minimize the distance from the task to the depot;
  - maximize the term dem(t)/sc(t), where dem(t) and sc(t) are demand and serving cost of task t, respectively;
  - minimize the term dem(t)/sc(t);
  - use rule 1) if the vehicle is less than half-full, otherwise use rule 2)
- **select**: Select the 300 optimal routes.
- **print_list**: Print the routes in given format
- **cal_path_cost**: Calculate the cost of a route
- **check_route**: check if the route is legal
- **mutation**: change the routes by following 5 methods
  - **single insertion**
  - **double insertion**
  - **swap**
  - **2opt**
  - **better 2opt**
- **solve_CARP**: Calculate the cost of a route

2.3. Detail of Algorithm

At first, I use Path-Scanning algorithm to produce initial solution.

$k \leftarrow 0$
copy all required arcs in a list $free$
repeat
$k \leftarrow k+1$; $R_k \leftarrow \emptyset$; $load(k), cost(k) \leftarrow 0$; $i \leftarrow 1$
repeat
$\bar{d} \leftarrow \infty$
for each $u \in free \mid load(k) + q_u \leq Q$ **do**
if $d_{i,beg(u)} < \bar{d}$ **then**
$\bar{d} \leftarrow d_{i,beg(u)}$
$\bar{u} \leftarrow u$
else if $(d_{i,beg(u)} = \bar{d})$ and $better(u, \bar{u}, rule)$
$\bar{u} \leftarrow u$
endif
endfor
add $\bar{u}$ at the end of route R_k
remove arc $\bar{u}$ and its opposite $\bar{u}+m$ from $free$
$load(k) \leftarrow load(k) + q_{\bar{u}}$
$cost(k) \leftarrow cost(k) + \bar{d} + c_{\bar{u}}$
$i \leftarrow end(\bar{u})$
until $(free = \emptyset)$ or $(\bar{d} = \infty)$
$cost(k) \leftarrow cost(k) + d_{i1}$
until $free = \emptyset$

Then, I run this methods thousands of times and sort by cost to get initial solutions

```
costList = list()
    for i in range(20):
        costList.append(path_scanning(rt, cost_matrix, 1))
        costList.append(path_scanning(rt, cost_matrix, 2))
    for i in range(200):
        costList.append(path_scanning(rt, cost_matrix, 3))
        costList.append(path_scanning(rt, cost_matrix, 4))
    costList = select(costList, 300)
    for i in range(2500):
        costList.append(path_scanning(rt, cost_matrix, 5))
    costList = select(costList, 300)
```

initial-solution

Next, I use mutation method to change the initial solutions to get better solution.

```
def mutation(cost_matrix, paths):
    rule = random.randint(1, 5)
    if rule == 1:
        return single_insertion(cost_matrix, paths)
    if rule == 2:
        return double_insertion(cost_matrix, paths)
    if rule == 3:
        return swap(cost_matrix, paths)
    if rule == 4:
        return _2opt(cost_matrix, paths)
    if rule == 5:
        return better_2opt(cost_matrix, paths)
```

mutation

```
for i in range(0, len(costList) - 1):
            tm = mutation(cost_matrix, costList[i])
            if tm:
                costList.extend(tm)
            # print(costList)
        costList = select(costList, 250)
```

mutation-main

Before each mutation, I will check whether the population is mature or not. If it have matured, I will use 2opt method and better 2opt method to jump out of the convergence state!

```
 if sum(costList[0][1]) == sum(costList[int((len(costList) - 1) * 0.9)][1]):
            costList = select(costList, 20)
            for i in range(0, len(costList) - 1):
                tm = better_2opt(cost_matrix, costList[i])
                if tm:
                    costList.extend(tm)
                tm = _2opt(cost_matrix, costList[i])
                if tm:
                    costList.extend(tm)
```

check-mature-main

After that, I will sort the population by cost and print the least cost route.

What's more, I use multiprocessing in this project. I run 8 times at the same time. Then I choose the best solution to print.

```
worker = []
    worker_num = 8
    seed = int(sys.argv[5])
    create_worker(worker_num)
    Task = [seed + i + random.randint(60, 300) for i in range(0, 24, 3)]
    # print('Task', Task)
    for i, t in enumerate(Task):
        worker[i % worker_num].inQ.put(t)
    result = []
    for i, t in enumerate(Task):
        result.append(worker[i % worker_num].outQ.get())
    result = sorted(result, key=lambda x: sum(x[1]))
    print_list(result[0])
    # print_list(result[7])
    finish_worker()
```

multiprocessing-main

3. Empirical Verification

3.1. Design

Since CARP is NP-hard and exact methods are only applicable to small instances, heuristic and metaheuristic methods are widely adopted when solving CARP.[1]

3.2. Data and data structure

Data: Data is the 7 given datasets. For example, "gdb10.dat", "egl-s1-A.dat", "egl-e1-A.dat", "val7A.dat", "val4A.dat" and so on.

Data Structure: I used list, 2D NumPy matrix, dict, tuple and so on.

3.3. Performance

The time complexity is O(n3) and it mainly results from floyd algorithm.

3.4. Result

I got a good result in CARP OJ system and this is my results.

RunTime(s)	Cost	Dataset
119.04	5313	egl-s1-A
59.51	279	val7A

RunTime(s)	Cost	Dataset
59.44	402	val4A
118.82	3687	egl-e1-A
58.80	275	gdb10
59.20	173	val1A
58.80	316	gdb1

3.5. Analysis

This project is an application of genetic algorithm. Through this project, I have a better understanding of genetic algorithm and I gained a lot by reading relative papers.

Acknowledgments

References

[1] K. Tang, Y. Mei and X. Yao, "Memetic Algorithm With Extended Neighborhood Search for Capacitated Arc Routing Problems", IEEE Transactions on Evolutionary Computation, vol. 13, no. 5, pp. 1151-1166, 2009.