Carp: Project 2 of CS303 Artificial Intelligence

11912039 Xinying Zheng : Department of Computer Science and Engineering

December 3, 2021

1 Preliminaries

1.1 Problem Description

1.1.1 Introduction

Capacitated Arc Routing Problem, also called CARP Problem, is a classical NP-hard arc routing problem. In Carp, we have an undirected connected graph, with some edges having tasks need to be served. Each edge, no matter it requires to be served or not, has a cost value when passing it. What we need to do is implementing an algorithm which can find a feasible solution containing multiple routes serving the required edges with the smallest cost. [1]

As a constrained optimization problem, its input and constraints can be described as following. Input:

- G(V, E) : an undirected graph.
- C$ost(e) > 0$ and $demand(e) >= 0$ for each edge.
- Depot node v_0: the node where each route starts and ends.
- capacity Q: the maximum load for each route

Constraints:

- Each route needs to begin at v_0 and end at it.
- Each task needs to be served for exactly once.
- The sum of demand of each route cannot exceed Q.

The solution presentation:

- A list of routes, each of which contains only the served edges in it with the total cost.

To specify, our project need the format as following:

s 0,(1,8),(8,3),(3,2),(2,7),(7,4),(4,2),(2,1),0,0,(1,5),(5,2),(7,5),(5,6),(6,11),(11,1),0,0,(1,4),(4,11),(11,12),(12,10),(10,9),(9,8),(8,12),(10,1),0,0,(1,9),(9,3),(3,4),(4,6),0
q 275

1.1.2 Software and Hardware

The project is written with Python 3.9.7, with pycharm as code editor. The configuration of the test platform is Debian 10 with a CPU2.2GHz*2, 8-core total.

1.2 Problem Application

As one of the three types of arc path problems, it is of wide application in our life. For example, route planing for garbage collection, snow removal in winter, street cleaning is some classical scenario of this problem.

2 Methodology

2.1 Notation

- v_0: depot where each route starts and ends
- c: capacity for each route
- distance [x][y]: minimum distance between x and y.
- edge.start/edge.end: although each edge is undirected, when we serve it, we can serve it from both end. So we manually give each edge a direction, each edge have two versions with different direction. To avoid duplicate, when we have served one of it, we also denote another as served.
- length: denote the length of that exactly array, which was determined by that array itself.

2.2 Data Structure

- class *graph*: store the input
  - edges[length]: all edges
  - vertices[length]: all vertices
  - depot : v_0
  - required-edges[length]: edges with positive demand
  - capacity : c
- class *edge*: store the information of each edge
  - start
  - end
  - cost
  - demand
  - rev: also a edge which refer to the reverse direction of the edge
- load[length]: store load of the route of current solution
- cost[length]: store cost of the route of current solution
- distance[length][length]: symmetric matrix store the shortest distance between two nodes.
- result[[length]]: a list of list, the outer list contains the routes which is denoted by the inner list. And the route was denoted by the inner list containing the serving edges. Eg. [[ed1,ed2,ed3,...].....]

2.3 Model design

I use the **Path Scanning** algorithm to generate initial solution with some improve strategies, the steps are as follows.

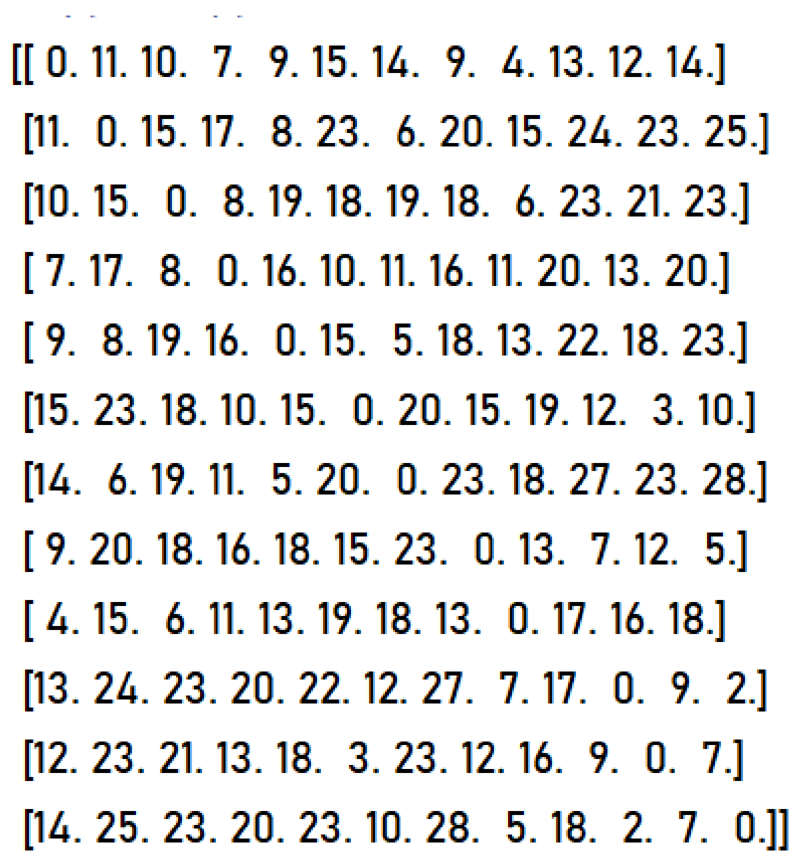
- Read data from input and generate the data structure to store it.
- Use Floyd algorithm to calculate the shortest distance between two nodes, since we use this statistic frequently so we calculate it in the beginning.
- Do the path scanning to generate initial solution and apply the move operator to probe potential improvements. Update cost and result only the newly found cost is small then current one.
- Repeat previous step until execution time exceed the requirements.

The main ideas of step two and three are as follows:

2.3.1 Find Min

We first generate a matrix called distance with the size of number of vertices * number of vertices and initial the elements in it with infinite which denote unreachability between two nodes. Secondly, we traverse all edges in the set. For each edge e, we set distance[e.start][e.end] = e.cost and distance[e.end][e.start] = e.cost. Last but not the least, For each node, we traverse the matrix and determine whether bring in this node will decrease the cost travel from one node to another, if so, update the cost, thus we can get a distance matrix for each dataset.

For example, figure 1 is the distance matrix for dataset $gdb10$



```
[[ 0. 11. 10.  7.  9. 15. 14.  9.  4. 13. 12. 14.]
 [11.  0. 15. 17.  8. 23.  6. 20. 15. 24. 23. 25.]
 [10. 15.  0.  8. 19. 18. 19. 18.  6. 23. 21. 23.]
 [ 7. 17.  8.  0. 16. 10. 11. 16. 11. 20. 13. 20.]
 [ 9.  8. 19. 16.  0. 15.  5. 18. 13. 22. 18. 23.]
 [15. 23. 18. 10. 15.  0. 20. 15. 19. 12.  3. 10.]
 [14.  6. 19. 11.  5. 20.  0. 23. 18. 27. 23. 28.]
 [ 9. 20. 18. 16. 18. 15. 23.  0. 13.  7. 12.  5.]
 [ 4. 15.  6. 11. 13. 19. 18. 13.  0. 17. 16. 18.]
 [13. 24. 23. 20. 22. 12. 27.  7. 17.  0.  9.  2.]
 [12. 23. 21. 13. 18.  3. 23. 12. 16.  9.  0.  7.]
 [14. 25. 23. 20. 23. 10. 28.  5. 18.  2.  7.  0.]]
```

Figure 1: the distance matrix for dataset $gdb10$

2.3.2 Path-Scanning

I adopted path-scanning strategy [2] to generate a relative high-quality initial solution. Using the idea of greedy, each time we choose the node that has the shortest distance from current end until current route cannot hold the load anymore, we come back to depot and start a new route.

The load of route i is:

$$load[i] = \sum_{e \in route\ i} e.demand$$

The cost of route i is:

$$cost[i] = \sum_{e \in route\ i} e.cost + distance[depot][e_0.start] + \sum_{i=1}^{n-1} distance[e_i.end][e_{i+1}.start] + distance[e_n.end][depot]$$

where n is the number of edges in the route i

One thing that need to note is that, when there are multiple nodes have the same shortest distance, we randomly choose one metric from the five to judge which task we should choose.

- maximize the distance[depot][edge.end]

- minimize the distance[depot][node.end]
- maximize edge.demand/edge.cost
- minimize edge.demand/edge.cost
- use rule 1 when load smaller than half of the capacity, otherwise use rule 2

2.3.3 Single-insertion

if we only use path-scanning, we are likely to miss some better solutions that are just close to the result we have found. So we use local search method, it can also be called as *moveoperator*, to explore neighborhood around current solution. Actually, there are a lot of move operators that have been proposed with both small steps and big steps. For small step's operator, we have:

- Flip : reverse each task.
- Single insertion: choose one task and find a best position to insert it.
- Double insertion: choose two adjacent tasks and find a best position to insert it.
- Swap: exchange two tasks
- 2-opt
  - applied to one route, a part of route is selected and change its direction
  - applied to two routes, the route is first cut into two subroutes and then reconnect them, just like a cross over operator.

For big step's operator, we have:

- Ms Operator: Since I didn't implement it, the detailed can be referred from [3]

In my algorithm, I just use single insertion as move operator.

2.4 Detains of Algorithm

In this section, I give out Pseudo-code for several main function.

2.4.1 find-min()

Use Flyod algorithm to calculate the shortest distance between any two nodes. Pseudo-code refers to Algorithm 1. The complexity is O(n^3)

Algorithm 1 find-min()

```
Input: Edge set, Node set: $edges$, $nodes$
Output: Distance matrix: $distance$.
  $distance \leftarrow$ matrix with elements equals to infinity;
  for edge in edges do update matrix[edge.start][edge.end]
  end for
  for i in nodes do
      for j in nodes do
          for k in nodes do
              if $distance[j][k] > distance[j][i] + distance[i][k]$ then update distance[j][k]
              end if
          end for
      end for
  end for
```

Algorithm 2 better()

```
Input: task1, task2: $edge_1$, $edge_2$
Output: chosen task: $edge$.
    $a \leftarrow$ randomly generate a number in the range[0,4]
    if a equals 1 then
        if task1 has smaller distance from depot then
            edge $\leftarrow$ edge1
        else
            edge $\leftarrow$ edge2
        end if
    end if
    if a equals 2 then
        if task1 has larger distance from depot then
            edge $\leftarrow$ edge1
        else
            edge $\leftarrow$ edge2
        end if
    end if
    if a equals 3 then
        if task1 has larger demand/cost value then
            edge $\leftarrow$ edge1
        else
            edge $\leftarrow$ edge2
        end if
    end if
    if a equals 4 then
        if task1 has smaller demand/cost value then
            edge $\leftarrow$ edge1
        else
            edge $\leftarrow$ edge2
        end if
    end if
    if a equals 5 then
        if $load\ of\ current\ route > c/2$ then
            use rule 1
        else
            use rule 2
        end if
    end if
```

2.4.2 better()

As I mentioned in the previous section, there are five metrics to use when we meet with multiple tasks to choose. Pseudo-code refers to Algorithm 2.

2.4.3 path-scanning()

Adopting the idea of greedy, path-scanning generating a feasible route. Pseudo-code refers to Algorithm 3.

Algorithm 3 path-scanning()

```
Input: all edges with positive demand: $free$
Output: solution: $result$.
    $result \leftarrow$ empty list
    $load \leftarrow$ empty list
    while $free$ is not empty do
        $end \leftarrow$ depot
        $route \leftarrow$ empty list
        while True do
            $distance \leftarrow infinity$
            $choose \leftarrow none$
            for $edge$ in free do
                if current route can hold edge and edge have smaller distance from end then
                    update $distance$ and $choose$
                else if current route can hold edge and edge have equal distance from end then
                    update $distance$ and $choose$ with better(edge,choose)
                end if
            end for
            if choose is not none then
                update $load$
                $route$.add($choose$)
                remove $choose$ and its reverse from $free$
                $end = choose.end$
            end if
        end while
        add $route$ to $result$
    end while
```

2.4.4 single-insertion()

In this algorithm, we randomly choose a task and find a best position for it. The cost for changing the position of task can be calculated as follows, assume origin route is [[t1,t2,t3],[t4,t5,t6]......] and we need to move t2 to the second route [[t1,t3],[t4,t5,t2,t6].....].

$$alter = -distance[t1.end][t2.start] - distance[t2.end][t3.start] + distance[t1.end][t3.start]$$

$$-distance[t5.end][t6.start] + distance[t5.end][t2.start] + distance[t2.end][t6.start]$$

In the algorithm, we use $tail_0$ and $head_0$ to denote the tasks before and after the choose task in the original route. we use $tail_1$ and $head_1$ to denote the tasks before and after the choose task in the route that the task is going to insert. We only do the insertion when the cost alter is negative, that is decrease the cost.

Algorithm 4 single-insertion()

```
Input: target routes: $origin$
Output: route after operation: $after$.
  $task \leftarrow$ randomly choose a task from the route
  $min - alter \leftarrow$ Infinity
  $position \leftarrow$ none
  for route in origin do
    if route can hold this task then
      for each insertable position in route do
        alter=+distance[$tail_0$.end][$head_0$.begin]-distance[$tail_0$.end][task.begin]-distance[$head_0$.start][task.end]+distance[$tail_1$.end][task.begin]+distance[task.end][$head_1$.start]-distance[$tail_1$.end][$head_1$.begin]
        if $alter < min - alter$ then update min-alter, record the position
        end if
      end for
    end if
  end for
  if $min - alter < 0$ then
    insert the task to the new position
  end if
```

3 Empirical Verification

3.1 Dataset

In the debug stage, I generate some graphs which has around 10 nodes and half of its edges need to be served.

Besides that, by surveying in the Internet, I got to know that there are lots of datasets for Carp Problems including KSHS,GDB,BBCM,EGL,EGL-Large,BMCV Instances.

I choose kshs1-kshs6 as big datasets whose best solution is around 10000 around. Some results is listed in the 3.4 section.

3.2 Performance measure

The performance of my algorithm is measured in my local machine, the confiuration is:

- Intel(R) Core(TM) i5-8265U CPU @1.60GHz
- python version 3.9.7

I measure the performance in the following aspect and searched for the best solution to be the comparison.

- The best solution it can find within certain time.

3.3 Hyperparameters

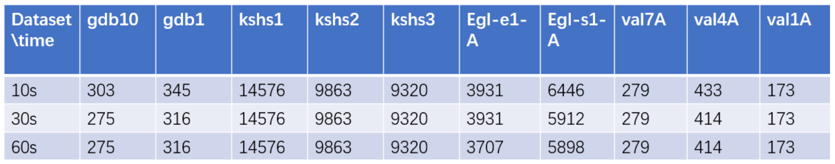
There are not so much hyperparameters I used in my program. Parameters that may affect performance is as following.

- Which metric to choose in the better() method, that is, when there are a list of candidate task, which we should choose. Here I use a random number, since these can increase the search field for my algorithm. If we choose one rule only, we are likely to be trapped in the same solution.
- Number of times we applied move operator. To achieve a local optimal, I applied single insertion for multiple times until the solution cannot improve anymore the maximum times is set as 100.
- Number of iteration times. Here I use the execution time as the constraints, only stop searching when time is exceeding the requirements.

Later if I modify my algorithm and applied big step move operators, I should first apply small step move operator to get a local maximum, then apply big step, finally apply small step operator. In this way, I am more likely to jump out local maximum and get better solution.

3.4 Experimental result

Some results are as following. I list the best solution I can find within certain time.



Dataset \time	gdb10	gdb1	kshs1	kshs2	kshs3	Egl-e1-A	Egl-s1-A	val7A	val4A	val1A
10s	303	345	14576	9863	9320	3931	6446	279	433	173
30s	275	316	14576	9863	9320	3931	5912	279	414	173
60s	275	316	14576	9863	9320	3707	5898	279	414	173

Figure 2: the best solution that can be found within certain time

3.5 Conclusion

3.5.1 Advantages

- Use path-scanning to get a relatively good solution, and in many small dataset, path-scanning can even find the best solution. In big dataset, the solution is also not so bad.
- The move operator can improve the solution at many times.
- A relatively clear code structure.
- High encapsulation used in my code

3.5.2 Disadvantages

- Use only small step move operator and can be easily trapped in local optimal.
- I use path-scanning to generate initial solution, although it can generate good solution, it also shrink the search field. Bad initial solution can sometimes get better final solution after local search algorithm.
- Not fully test my code in the local. I did wrote other move operators, but it still have bugs until deadline.

3.5.3 Improvements

- Including more small step move operators like swap,flip,2-opt and so on.
- Including big step move operator like MS operator.
- Adopting multithreading
- Adopting genetic algorithm frame and randomly generate initial population
- Testing my code in more dataset

References

[1] B. L. Golden and R. T. Wong, “Capacitated arc routing problems,” *Networks*, vol. 11, no. 3, pp. 305–315, 1981.

[2] Á. Corberán and G. Laporte, *Arc routing: problems, methods, and applications.* SIAM, 2015.

[3] 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.