CARP

Li Jiajun 11912021
Department of Computer Science and Engineering
Southern University of Science and Technology
11912021@mail.sustech.edu.cn

1. Preliminaries

1.1. Introduction

CARP is the abbreviation of Capacitated Arc Routing Problem. This is one of the combinatorial optimazation problems. The problem can be described as follow: There is a connected graph whose edges have two parameters– cost and demand. Meanwhile, there are some transport vehicles starting from a given vertex called depot, and each vehicle has a certain capacity to satisfy the demand of edges whenever the vehicle passes an edge. After a vehicle runs out of its capacity, it would return to the depot. The goal is to find the best routes for the vehicles so that all demands can be satisfied while achieving minimun cost expense.

1.2. Application

The CARP problem can be observed in our daily life. For instance, a large quantity of garbage is created in a city everyday, and bin wagons would collect garbage everyday. The amount of garbage of each street can be viewed as the demand of the edge, while the length or width of the streets could be used to estimate the cost of edges. The effort to collect all garbage is highly similar to solving the CARP. Likewise, many logistics and transportation activities can be viewed as CARP.

1.3. Software

The project uses Python with editor Pycharm. Operating on Windows 10.

1.4. Algorithm

The project applied Dijkstra to find the routes with minimun cost, and used Path-Scanning algorithm to calculate the conclusion.

2. Methodology

This section demonstrates the design idea of the whole problem.

2.1. Representation

- cost_graph: A two dimensional matrix storing the costs between any two connected vertices.
- demand_graph: A two dimensional matrix storing the demands of every edge.
- dis: A two dimensional matrix storing the minimun cost between any two vertices.
- free: A list of edges with unsatisfied demands.

2.2. Architecture

A list of routes that fulfills the demands of all edges can be generated by the following means. Firstly the program reads the problem file and builds cost_graph and demand_graph. Then Dijkstra is used to acquire the minimu distance between every pair of vertices and generate a corresponding matrix named dis. Meanwhile we scan the demand_graph to generate the list free which stores all edges with unsatified demands, and the edges are sets with two elements. Finally we invoke the path_scanning algorithm to get the final routes. Path_scanning scans every edges in free and uses simple logical metrics to decide whether an edge is available to go through or not. To acquire the best solution, which is a solution with the least total cost, the path_scanning algorithm applies different decision criterias to generate different solutions and compares the solutions' total cost to choose the best one.

The objective of CARP can be represented by the following equations. The overall goal is to minimize the total cost $TC(s)$

$RC(R_k)$ is the route cost of route R_k, and the relation between $TC(s)$ and $RC(R_k)$ can be described by the following equation: $TC(s) = \sum_{k=1}^{m} RC\,(R_k)$, suggesting that it is the sum of each route's cost that form the final result.

2.3. Detail of Algorithm

This section shows the pseudocode of the functions. The code fragments are presented in the following pages.

Algorithm 1 make_free

Input: demand_graph, vertices_num
Output: free

```
initialize free
for i = 1; i < vertices_num + 1; i++ do
    for j = 1; j < vertices_num + 1; j++ do
        if demand_graph[i][j] != 0 and (i, j) or (j, i) are not in free then
            add (i, j) and (j, i) to free
        end if
    end for
end for
```

Algorithm 2 dijkstra

Input: cost_graph
Output: dis

```
initialize heap
add start node into heap
while heap is not empty do
    curnode = heap.poll
    for each child node i ∈ curnode do
        if child node i not visited then
            distance of i = current distance + cost_graph[curnode][i]
            node i visited
            add node i to heap
        else
            if distance of i > currrent distance + cost_graph[curnode][i] then
                update distance of i
                remove i in heap
            end if
        end if
    end for
end while
```

3. Empirical Verification

3.1. Dataset

Apart from the dataset given, I created my own dataset for testing and debugging. Since the official given dataset is too complex, making debugging difficult, I used my own simpler dataset to test the accuracy of the algorithm to verify whether I have missed any edges during my calculation.

3.2. Performance measure

The performance is measured as its time cost. The cost can be measured by implementing the time package to measure the time spent throughout the whole calculating process. The total cost is another key descriptor to evaluate the performance, which is calculated by adding the cost value of all visited edges.

Algorithm 3 path_scanning

Input: depot, cost_graph, demand_graph, vertices_num, capacity, dis, mode
Output: result, cost

```
free = make_free(demand_graph, vertices_num)
initialize result, cost
while free is not empty do
    initialize route
    cap = capacity, start = depot
    repeat
        dis2 = 9999999
        for every arc ∈ free do
            if demand of arc < cap then
                if dis[start][arc[0]] < dis2 then
                    dis2 = dis[start][arc[0]]
                    arc2 = arc
                else if dis[start][arc[0]] == dis2 then
                    if mode = 1 then
                        if dis[arc[1]][start] < dis[arc2[1]][start] then
                            arc2 = arc
                        end if
                    end if
                    if mode = 2 then
                        if dis[arc[1]][start] > dis[arc2[1]][start] then
                            arc2 = arc
                        end if
                    end if
                    if mode = 3 then
                        if demand of arc / cost of arc < demand of arc2 / cost of arc2 then
                            arc2 = arc
                        end if
                    end if
                    if mode = 4 then
                        if demand of arc / cost of arc > demand of arc2 / cost of arc2 then
                            arc2 = arc
                        end if
                    end if
                    if mode = 5 then
                        if less than half full then
                            if dis[arc[1]][start] < dis[arc2[1]][start] then
                                arc2 = arc
                            end if
                        else if more than half full then
                            if dis[arc[1]][start] > dis[arc2[1]][start] then
                                arc2 = arc
                            end if
                        end if
                    end if
                end if
            end if
        end for
        add arc2 to route
        update free, cost, cap and start
    until free is empty or dis2 = 9999999
    add route to result
    update cost
end while
```

3.3. Hyperparameters

Although there are no hyperparameters in my algorithm, potentially applicable parameters can still be introduced if the algorithm can be improved. For instance, there is room for improvement about the implementation of local search algorithm. Once we apply local search algorithm like the simulated annealing, one significant parameter is the final cooling temperature that ends the calculating procedure. This parameter ought to be small to ensure good performance, while it is not supposed to be too low to pose negative impact on the total time cost.

3.4. Experiment results

I imported the time package and measured the time cost of every task provided. There are seven official tasks and the corresponding information is given in TABLE 1.

TABLE 1. **PERFORMANCE TEST**

test case	vertices	edges	time	cost
egl-e1-A	77	98	0.478	4382
egl-s1-A	140	190	2.315	6145
gdb1	12	22	0.009	374
gdb10	12	25	0.017	309
val1A	24	39	0.040	195
val4A	41	69	0.103	459
val7A	40	66	0.102	330

3.5. Conclusion

The path_scanning algorithm is an effective method to solve CARP. However, the efficiency of this algorithm is largely based on the determination metric, which is adjustable. In my case, I utilized five metrics to complete the task independently and finally choose the best among them. This fact implies that creating more metrics and logical conditions would be helpful to find the best solution, as long as the number of conditions is large enough. However, more conditions lead to larger time cost, which may pose great influence when dealing with large dataset. Therefore, local search algorithms can be introduced to improve the efficiency. Ideas like genetic algorithm or tabu search may be useful in face of huge datasets.

Acknowledgments

Many thanks to the teacher and classmates who enlightened me.

References

[1] Johnson, D. B. (1973). A note on Dijkstra's shortest path algorithm. Journal of the ACM (JACM), 20(3), 385-388.

[2] Tang, K., Mei, Y., Yao, X. (2009). Memetic algorithm with extended neighborhood search for capacitated arc routing problems. IEEE Transactions on Evolutionary Computation, 13(5), 1151-1166.