Project 2: Capacitated Arc Routing Problems

Yuejian Mo 11510511
Department of Biology
Southern University of Science and Technology
Email: 11510511@mail.sustc.edu.cn

1. Preliminaries

This project is an implementations of Path-Scanning and Dijkstra's algorithm to solve the capacitated arc routing problems(CARP). CARP is the most typical form of arc routing problem, which has many application in real world, such as urban waste collection, post delivery. [1] In the case of waste collection, some stresses of city has waste, cars with limit capacitated will carry the these waste start from depot and return again when car is full. A solution with less cost will be perfect.

CARP can be represented formally as follows: a mixed graph $G = (V, E, A)$, with a set of vertices denoted by V, set of edges denoted by E and a set of arcs denoted by A, is given. There is a central depot vertex $dep \ni V$, where a set of vehicles are based. A subset $E_R \supseteq E$ composed of all the edges required to be served and a subset . The objective of CARP is to determine a set of routes for the vehicles to serve all task with minimal costs while satisfying [1]: a) Each route must start and end at dep b) The total demand serviced on each route must not exceed the vehicle's capacity Q; c) Each task must be served exactly once(but the corresponding edge can be traversed more than once).

CARP is NP-hard, various heuristics and meatheuristics are used, such as Augement-Merge, Path-Scanning. Here I successfully implemented Path-Scanning.

1.1. Software

This project is written by Python 3.7 with editor Atom and Vim. Numpy library and sys library are used.

1.2. Algorithm

Using Dijkstra's algorithm to calculate the closest pathway between two vertex. Path-Scanning is used to find out task sequence for CARP. I defined three group of functions in order to find out optimal service sequence, including one function to generate cost graph and demand graph between vertex and vertex from provided txt file, two functions to generate shortest distance and pathway dictionary by Dijkstra algorithm, other are control flow and output format function. Function $Better$ break the balanced status.

2. Methodology

CARP is NP-hard. Path-Scanning provide a reasonable heuristic method to reduce the total cost. Firstly, edge with required was copy to a new list. Car begin its service from depot(vertex 1). Then the shortest pathway away from now service vertex will be choose as new service vertex while the car is not full and here still are required edges. Repeat above produce to clean up all required edges.

2.1. Representation

Some main data are maintain during process: **capacity**,**graph_dm** , **graph_ct**. Others data would be specified inside functions, shortest_dist should noticed.

- **capacity**: The car's capacity, generated from input file.
- **graph_dm**: The graph of edge with demand and their demand, generated from input file.
- **graph_ct**: The graph of edge with cost and their cost, generated from input file.
- **shortest_dist**: A dictionary of shortest distance and pathway between two vertexes.

2.2. Architecture

Here list all functions in given code:

- **generateGraph**: Generate cost and demand graph from input file.
- **dijkstra**: Calculate all vertexes closest distance and pathway away from specify source.
- **genDijkstraDist**: Generate each two vertexes closest distance and pathway from **dijkstra** function.
- **better**: Second rule to choose one vertexes from two vertex.
- **pathScan**: Path-Scanning algorithm to generate serve sequence.
- **s_format**: Standard output function.
- **__name__**: Main control function.

The CARP_solver would be executed in test platform.

2.3. Detail of Algorithm

Here describes some vital functions.

- **generateGraph**: Generate global cost and demand graph from input file.

Algorithm 1 generateGraph

Input: $input\_file\_name$
Output:
1: open $input\_file\_name$ as $file$ {open file and read line by line}
2: $capacity \leftarrow$file.$readline$
3: {Read each line information until arrive edge information}
4: **for** each edge e **do**
5: split each line string into an array $line$ with 4 elements.
6: **if** ($line$[3] larger than 0) **then**
7: $graph\_dm$[($line$[0], $line$[1])]= $line$[3]
8: $graph\_dm[(line[1], line[0])] = line[3]$ {Only add edge to dictionary $graph_dm$when edge has demand. Both of two direction will be created.}
9: **end if**
10: $graph\_ct[(line[0], line[1])] = line[2]$
11: $graph\_ct[(line[1], line[0])] = line[2]$
12: **end for**

- **dijkstra**: generate closest distance and pathway away from source

Algorithm 2 dijkstra

Input: $source$
Output: $dist$, $prev$
create vertex set Q
2: create distance set $dist$
create path set $prev$
4: **for** each vertex v in $graph\_ct$ **do**
$dist[v] \leftarrow INFINITY$
6: $prev[v] \leftarrow UNDEFINED$
add v to Q
8: **end for**
$dist[source] \leftarrow 0$
10: **while** (Q is not empty) **do**
$u \leftarrow$ vertex in Q with min $dist[u]$
12: remove u from Q
for for each neighbor v in u **do**
14: $alt \leftarrow dist[u] + graph\_ct(u, v)$
if alt larger than $dist[u]$ **then**
16: $dist[v] \leftarrow alt$
$prev[v] \leftarrow u$
18: **end if**
end for
20: **end while**
return $dist$, $prev$

- **genDijstraDist**: generate closest distance and pathway between two vertex into a dictionary $shortestDist$

Algorithm 3 genDijstraDist

Input:
Output: $shortestDist$
create dictionary $shortestDist$
for each edge's first vertex in $graph\_dm$ as $source$ **do**
3: $dist$, $prev \leftarrow$ dijkstra($source$)
for each edge's first vertex in $graph\_dm$ as $target$ **do**
$shortestDist[source, target] \leftarrow dist$
6: **end for**
end for
return $shortestDist$

- **better**: choose better in pointers

Algorithm 4 better

Input: now, pre
Output: $Ture$ or $False$
return $graph\_dm[now] < graph\_dm[pre]$

- **pathScan**: Path Scanning algorithm to determine serve routes

Algorithm 5 pathScan

Input: $shortest\_dist$
Output: R, $cost$

```
create successive routes R
copy required edge from graph_dm to free
k, cost ← 0
while free is not empty do
    k ← k + 1
    cost_k, load_k ← 0
    rest successive routes R_k each time
    serve from origin vertex: source ← 1
    while free in not empty do
        rest shortest distance: d ← ∞
        rest candidate serve edge: e_candidate ← −1
        for each edge e in free do
            if load_k + graph_dm[e] ≤ capacity then
                dist_now ← shortest_dist[source, e.start]
                if dist_now ≤ d then
                    d ← dist_now
                    e_candidate ← e
                else if dist_now = d ∩ better(e,e_candidate
                ) then
                    e_candidate ← e
                end if
            end if
        end for
        if e = ∞ then
            BREAK
        end if
        add e_candidate to R_k
        load_k = load_k + graph_dm[e_candidate]
        cost_k = cost_k + graph_ct[e_candidate]
        i = e_candidate.end
        remove e_candidate and its opposite edge from
        free
    end while
    add back home distance: cost_k = cost_k +
    shortest_dist[i, 1]
    add R_k to R
    cost = cost + cost_k
end while
return R, cost
```

3. Empirical Verification

Empirical verification is confirmed in public test platform. Both right output format and reasonable routes are produced.

3.1. Design

Dijkstra's algorithm return successfully shortest distance and pathway between two vertex. Path-Scanning run correctly. However, Path-Scanning don't provide most optimal routes in spite of the size of test data. Function $better$

3.2. Data and data structure

Dictionaries and lists are used widely rather than matrix. Because the input graph is sparse, dictionary are always used to store graph. Global variable $graph\_dm$ and $graph\_ct$ are dictionary. Lists always store routes and edges information.

3.3. Performance

Following table show different performance with different dataset.

Dataset	Run Time(s)	Cost
gdb1	0.65	370
gdb10	0.63	309
val1A	0.68	212
val4A	0.92	450
val7A	4	334
egl-e1-A	0.96	4201
egl-s1-A	5.02	6383

3.4. Result

Solutions are accepted for all data set test. Larger data set require more time.

3.5. Analysis

Because Path-Scanning is kind of greedy algorithm with heuristics, most of solutions are not optimal. The space cost is close to $O(n)$. First time cost $O(n^2)$ is in building Dijkstra's distance, which only run once. Path-Scanning without function $Better$ just cost $O(n)$. Total cost time most vary on function $Better$. For example, if $Better$ just return random result, it will cost less time than comparison method. Although complex and well-design $Better$ cost more time, it's solution is less-cost-routes.

Acknowledgment

Thanks TA Yao Zhao who explain question and provide general method to solve it. And I also thanks for Kebing Sun discuss algorithm and point the output formation error.

References

[1] Ke Tang, Yi Mei, and Xin Yao, "Memetic Algorithm With Extended Neighborhood Search for Capacitated Arc Routing Problems," IEEE Transactions on Evolutionary Computation, vol. 13, no. 5, pp. 1151–1166, Oct. 2009.