CS303 Artificial Inteligence Project Report: Capacitated Arc Routing Problems(CARP)

Zitong Huang

12012710@mail.sustech.edu.cn

December 9, 2022

1. Introduction

1.1. Arc Routing Problem

Arc Routing Problems(ARP) are a category of general routing peoblem(GRP)[3]. Different from another kind of GRP called Node Routing Problems(NRP), ARP's main serve object is edges in graph.

1.2. Application

Arc Routing Problem is widly used in daily life:

> Arc routing problems can be applied to garbage collection, school bus route planning, package and newspaper delivery, deicing and snow removal with winter service vehicles that sprinkle salt on the road, mail delivery, network maintenance, street sweeping, police and security guard patrolling, and snow ploughing. Arc routings problems are NP hard, as opposed to route inspection problems that can be solved in polynomial-time.[3]

In these project, a sub-problem of ARP called CARP is mainly discussed. A further explaination of CARP will be show later.

1.3. Purpose

This report is mainly write to explain the secand project of CS303 course. And, this report will give a better explanation of some algorithm used, theory, fomuler and method used in project code.

2. Preliminary Problem Description

2.1. Overview of CARP

The CARP(capacitated arc routing problem) is a typical form of the arc routing problem. It is a combinatorial optimization problem with some constrains needed[2].

2.2. Describe

In natural language, CARP can be describe as follows:

Given a graph with serial edges to be served. A velhicle group get task to serve these edges. Each velhicle have the **same** capacitate. At the processing of serving edges, if one velhicle's total served amount reach its capacity, it should return to a given vertice (called depot) for at least one time, so that it can serve other edges continuely. Each edge have a cost, represent the time a velhicle would cost to travel through. Each edge need to be served have a demand, represent the demand a velhicle will get after serving it. CARP is to get some routes to minimize the total cost of this velgicle group.

In mathmetic, CARP can be describe as follows:

> …consider an undirected connected graph $G = (V, E)$, with a vertex set V and an edge set E and a set of required edges (tasks) $T \subseteq E$. A fleet of identical vehicles, each of capacity Q, is based at a designated depot vertex $v_0 \in V$. Each edge $e \in E$ incurs a cost $cc\ ee$ whenever a vehicle travels over it or serves it (if it is a task). Each required edge (task) $\tau \in T$ has a demand $d(\tau) > 0$ associated with it.[2]

CARP is proved as a NP-hard Problem in 1981 by B. L. Golden and R. T. Wong[1], which means it cannot be solved in polynomial time.

2.3. Formulation

Variables use in this report is defined below:

- $G(V, E)$: undirected graph with vertices set V and edge set E
- s: start vertice of the graph.
- t: end vertice of the graph.
- d: depot vertice of the graph.
- W_{ij}: least cost when a velhical travel from vertice i to vertice j.
- E_d: edges need to be served by vehical group
- C:max serve a velhical can offer one time.
- $Cost(G, d, C)$:total cost a velhical take after served all demand edges.

Then, a carp problem can be defined as a optimized problem—Target function is *Cost*(*G,d,C*), the objective of an agent is to minimize *Cost*(*G,d,C*)

Other specific name need to be assert is show below.

- *graph*: the graph in CARP, contains some edges needed to be served. *graph*[*i, j*] represente the cost when vehicle travel from vertices[i] to vertices[j].If i and j is not connected directly, *graph*[*i, j*] will be set as ∞.
- $d[i,j]$: Shortest distance between i-th vertice and j-th vertice.
- *distance*: the graph after doing Floyd.

$$distance[i,j] = d[i,j]$$

- *middle_vertice*: vertice choosed as "jump board" in Floyd algorithm
- *connect*ed: if vertice i and vertice j have direct path, or, distance[i, j] < ∞
- V: the number of vertices in graph
- V_i: the i-th vertices
- E: the number of edges in graph.
- E_i: an edge in graph.
- E_{ij}: an edge between V_i and V_j
- S_{edge}: start vertice of an edge
- E_{edge}: end vertices of an edge
- *route*: element in routes, represent edges between two passes through the depot of a vehicle.
- *total_routes* : a result of a CARP. Contains some route.
- *demand edges*: edges consider need to be served in algorithm
- *served amount*: when a velhicle is at one moment when do serving, the total serve amount it served.
- *capacity*: the max served amount of one velhicle. It is always true that

$$served\_amount \leqslant capacity$$

- *time*: total time procedure cost
- terminate_time: max time procudure can cost. It is always true that time $\leqslant$ terminate time

2.4. Solution-mark format

A solution of a CARP is marked as *routes*. A *routes* contains serial *route*. Each *route* marked as a list

$$route = [0, E_1, E_2 \ldots E_i, 0]$$

means a vehicle travels from the depot, pass through E_1, $E_2 \ldots E_i$, then back to the depot.

Thus, *total_routes* can be marked as

$$total\_routes = [route_1, route_2, route_3 \ldots route_i]$$

3. Procedure

3.1. Introduction

Solve this problem need such procedure: data reading, distance calculate and routes decide. When dealing with routes decide, a path-scanning algorithm optimized by multi-processing and merge-split is used.

3.2. Data Reading

As the give document described, online jugement parse parameters for 2 parts.

3.2.1. First Part

At the first part, terminal parse basic imformation to procedure. The format of first parameter are shown below:

- 1st line: NAME : <string> – the name of instance
- 2nd line: VERTICES: <number> – number of vertices
- 3rd line: DEPOT:<number> – the depot vertex
- 4th line: REQUIRED EDGES:<number> – number of required edges
- 5th line: NON-REQUIRED EDGES : <number> – the number of non-required edges
- 6th line: VEHICLES : <number> – the number of vehicles
- 7th line: CAPACITY : <number> –the vehicle capacity
- 8th line: TOTAL COST OF REQUIRED EDGES: <number> – the total cost of all tasks

A sample input is followed:

```
NAME : gdb1
VERTICES : 12
DEPOT : 1
REQUIRED EDGES : 22
NON-REQUIRED EDGES : 0

VEHICLES : 5
CAPACITY : 5
TOTAL COST OF REQUIRED EDGES : 252
```

While processing data, it is easy to found velhicle amount will not affect our result. Proof is not list here since it's not hard by commutative law of multiplication.

3.2.2. Secand Part

In secand part, imformation of graph is parsing to the procidure.

Input data is constitute by serial rows, each row represent an edge in graph. One row is divided to 4 columns, each column represent start_point of the edge, end_point of the edge, demand, cost, respectively. A sample input is show below:

```
NODES          COST          DEMAND
1    2          13               1
1    4          17               1
 . . .
10   11         11               1
END
```

3.3. Distance Calculate: Shortest Path Algorithm—Floyd

3.3.1. Introduce

When doing CARP, a shortest path algorithm is needed to compute distance between two vertice. Algorithm used here is Floyd algorithm.

Main procedure of Floyd can be represented by upon:

1. Initial the graph, set $d[i, j]$ as the shortest distance between vertice i and vertice j. If there is no path bewteen i and j, set $d[i, j]$ be ∞
2. choose a jump-vertice k
3. For jump-vertice k, choose two different vertices i, j from graph. if $d[i, k]$ + $d[k, j]$ less than $d[i, j]$, set $d[i, j]$ be $d[i, k]$+ $d[k, j]$
4. repeat step 3 until all vertices in graph have been choosen as i or j at least once.(except vertice k)
5. repeat step 2,3 until all vertices in graph has been choosen as k as least once.

The time conplexy is

$$O(V^3)$$

while V is the number of vertices

3.3.2. Formulation

Floyd can be expree as following:

$$distance[i, j] = Min(distance[i, k] + distance[k, j])$$

where k is vertice connected to both i and j.

3.3.3. Pseudo-code

Pseudo-code is list below:

```
function floyd(graph)
begin
    for i=V_0 -> V_i
    begin
        for j=V_0 -> V_i
        begin
            for k=V_0 -> V_i
            begin
                if graph[i, j] > graph[i, k] + graph[k, j]
                then graph[i,j]= graph[i,k] + graph[k,j]
            end
        end
    end
    return graph
end
```

3.4. Routes Decide

3.4.1. Path-Scanning

Introduction

In Yao's work, a Path-Scanning algorithm is used to when initial the population while doing memetic algorithm. The core concept of Path-Scanning is greedy. For each edge Path-Scanning served, algorithm choose the nearest k edges as candinates. For elements in candinates, algorithm choose next-serve edge by following rules:

1. maximize the distance from the head of task to the depot
2. minimize the distance from the head of task to the depot
3. maximize the term $\mathrm{dem}(t) / \mathrm{sc}(t)$, where $\mathrm{dem}(t)$ and $\mathrm{sc}(t)$ are demand and serving cost of task t, respectively;
4. minimize the term $\mathrm{dem}(t) / \mathrm{sc}(t)$
5. use rule 1 if the vehicle is less than half˜full, otherwise use rule 2

As a greedy algorithm, each round of Path-Scanning has a time complexy:

$$O(E^2), \text{while } m \text{ is number of edges.}$$

In this project, since merge-split operator(going to introduced next) failed to be used in final code, 3-rd and 4-nd is not used in order to keep the randomness.

Formulation

Path-Scanning can be

Pseudo-code

Pseudo-code in this project is show following:

function path_scanning(*demand_edges*)
    routes = []
    while *demand_edge* not null
    begin
        route = []
        last_vertice ← depot
        while *served_amount* ⩽ *capacity*
        next_edges ← **nearest** edge from last_vertices in *demand_edges*
        begin
            if *served_amount* ⩽ *capacity* / 2
            begin
                next_edge ← **furthest** edge from depot in *next_edges*.
            end
            else if *served_amount* > *capacity* / 2
            begin
                next_edge ← **nearest** edge from depot in *next_edges*.
            end
        end
        route ← *route* + *next_edge*
        remove *nect_edge* from *demand_edge*
        last_vertices ← $E_{\text{next\_edge}}$
        routes = *routes* + *route*
    end
return *routes*

3.4.2. Merge-Split Operator

Merge-Split Operator is first intruduced by X.Yao in 2009[2], by using it as memetic algorithm's local-search operator. With Merge-Split used, Yao had improved the performance of memetic algorithm to a new level.

A basic procedure of Merge-split is following:

1. Initial *r* by using path-scanning. The route is not neccesary to be the best result.
2. Choose k routes randomly, save edges in these rooutes.
3. Do path-scanning again. Append result return to remained routes.

function merge-split(route)
    begin
        split_point = random(1, length(route))
        subroute1 = route[1...split_point]
        subroute2 = route[split_point+1...length(route)]
        merge_distance = distance(subroute1[end], subroute2[start])
        merged_route = subroute1 +merge_distance + subroute2
        return merged_route
    end

3.5. Working flow

1. Data reading
2. Calculate distances between each vertices. Store the information is distance.
3. while time < terminate_time / 2, do path_scanning to get result routes
4. while time > terminate_time / 2, do merge-split to updat routes exist.
5. In result routes, choose least cost routes.

4. Experiments

4.1. Environment

4.1.1. Hardware Environment

CPU: Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz
GPU: NVIDIA GeForce RTX 2060

4.1.2. Software Environment

Operation System: Ubuntu 20.04 LTS
Python: Python 3.9.12
IDE: Visual Studio Code 1.72.2 with python extension
Numpy: 1.16.1
Other package used in python : math, time, random

4.2. Dataset

A dataset is built by myself in order to check if algorithm work well when meet some edge conditions. Example is list below

```
NAME : test
VERTICES : 2
DEPOT : 1
REQUIRED EDGES : 2
NON-REQUIRED EDGES : 0

VEHICLES : 1
CAPACITY : 1
TOTAL COST OF REQUIRED EDGES : 1

NODES       COST         DEMAND
1   2        13              1
```

```
END
```

Since test data all have solution, no-result-condition is not considered here.

4.3. Solution analysed

For some reson, Merge-Split operator is not used in submitted code cause some unsolved bugs.

Perfomance difference before and after introduce mer-split operater is show followed.

Data used in following is data given, since it's hard to show data in report.

Before

gdb10	gbd1	egl-s1-A	egl-e1-A	val7A	val4A	val1A
275	316	5412	3781	280	410	173

After

gdb10	gbd1	egl-s1-A	egl-e1-A	val7A	val4A	val1A
275	316	5308	3726	283	417	173

5. Conclusion

5.1. Evaluate algorism

5.1.1. Advantage

Algorithm use multi-processing to improve performance. By multi-processing, algorithmm is easier to get its upper bound. Also, with the introduce of merge-split, algorithm's upper bound is higher than normal path-scanning algorithm.

5.1.2. Defect

Since the only random operator is merge-split, algorithm's upper bound is still not enough to get best solution. Also, the ratio of path-scanning and merge-split is not perfect, for small data, path-scanning will do lots of duplicate compute, with is a waste of time.

5.2. Space to improve

The ratio of path-scanning and merge-split can be improved to get a better belance. Also, a memetic algorithm is better to introduce randomness, to get best solution.

Bibliography

[1] Bruce L. Golden and Richard T. Wong. Capacitated arc routing problems. *Networks*, 11(3):305–315, 1981.

[2] Ke Tang, Yi Mei, and Xin Yao. Memetic algorithm with extended neighborhood search for capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 13(5):1151–1166, 2009.

[3] Wikipedia. Arc Routing — Wikipedia, the free encyclopedia. 2004. [Online; accessed 22-July-2004].