CARP Report

Yilin Zhang 11510280

School of Computer Science and Engineering
Southern University of Science and Technology
11510280@mail.sustc.edu.cn

November 22, 2018

1. Preliminaries

Capacitated Arc Routing Problem (CARP), a NP hard problem, can be discribed as follows:

Consider an undirected connected graph $G = (V, E)$, with a vertex set V, 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 $c(e)$ whenever a vehicle travels over it or serves it. Each required edge (task) $\tau \in T$ has a demand $d(\tau) > 0$ associated with it.

The objective of CARP is to determine a set of routes for the vehicles to serve all the tasks with minimal costs while satisfying:

- Each route must start and end at v_0;
- The total demand serviced on each route must not exceed Q;
- Each task must be served exactly once, but the corresponding edge can be traversed more than once.

This project is to design and implement heuristic search algorithms to give a feasible solution with high-quality of CARP.

1.1. Software

This project is written in Python 3.6.3 using Editor *Visual Studio Code*. The libraries being used includes *argparse*, *time*, *sys*, *numpy* and *random*.

The name of the executable CARP solver is *CARP_solver.py*. Other submodule files I used in a folder named *submodule*.

2. Methodology

I mainly used *path_scanning* (PS) and *dijkstra* search methods to determine routes in this project.

2.1. Representation

Because all information is given by **.dat* file, we need to read file and parse it to usable data structure first. I used a dictionary to store general informations such as **NAME**, **VERTICES**, **DEPOT** *etc*. As for the graph, I used a 2-dimensional matrix denotes the cost and the other 2-dimensional matrix denotes the demand.

The format of the solver call is as follows (the test environment is Unix-like):
python CARP_solver.py <CARP instance file> -t <termination> -s <random seed>

As above, three parameters **FILE_PATH**, **TERMINATION** and **RANDOM_SEED** are given by commend line, which means parameter parsing is essential, too.

- **Constant:**
  - **TERMINATION:** specifies the termination condition of the algorithm.
  - **RANDOM_SEED:** specifies the random seed used in this run. In case that your solver is stochastic, the random seed controls all the

stochastic behaviors of your solver, such that the same random seeds will make your solver produce the same results.
- **FILE_PATH:** path of CARP instance file.
- **INFO:** a dictionary stores all informations about file and graph including *NAME, VERTICES, DEPOT, REQUIRED EDGES, NON-REQUIRED EDGES, VEHICLES, CAPACITY, TOTAL COST OF REQUIRED EDGES*.
- **gra_cost:** a 2-dimensional matrix stores the cost of each edge.
- **gra_demand:** a 2-dimensional matrix stores the demand of each edge, $gra\_demand[i, j] = 0$ denotes there is no edge (i, j) or a edge (i, j) without demand.

- **Variables:**
  - **dis:** a 2-dimensional matrix stores closest distences between any two points.
  - **previous:** a 2-dimensional matrix records the previous vertex of closest distences between any two points which can directs the closest path of them. It is not used in this project but I still retained it for further extension.
  - **R:** a 2-dimensional list stores the determination of rounds.
  - **q:** a int number, denotes the total cost of the solution. It is used to evalute the solution.

2.2. Architecture

The executable file is *CARP_solver.py*, which contains parameter parsing, flie reading, and result printing. There are two modeules *dijkstra,py* and *path_scanning.py* in the *submodule* folder, offer functions dijkstra and path_scanning, respectively.

- **dijkstra:**
  - **extract_min:** find the vertex with minimum distance to start point from given set of candidate vertexes.
  - **dijkstra:** implement dijkstra algorithm, return **dis** and **previous**.
- **path_scanning:**
  - **initial_method:** use a determine rule to excute path scanning algorithm.
  - **initial:** calling **initial_method** function by different rules, choose the best result (with minimum **q**) to return.

2.3. Algorithm

First, use dijkstra algorithm get the closest distence between any two vertexes, since dijkstra procedure is an exact algorithm and has been well known for years, I omit its detailed steps in this report. Once we get the matrix **dis**, we can use path scanning algorithm to get fessible solutions, and choose the best one of them as the final results.

Traditional path scanning has many steps: PS starts by initializing an empty path. At each iteration, PS finds out the tasks that do not violate the capacity constraints. If no task satisfies the constraints, it connects the end of the current path to the depot with the shortest path between them to form a route, and then initializes a new empty path. If a unique task satisfies the constraints, PS connects that task to the end of the current path (again, with the shortest path between them). If multiple tasks satisfy the constraints, the one closest to the end of the current path is chosen. If multiple tasks not only satisfy the capacity constraints but are also the closest to the end of the current path, five rules are further adopted to determine which to choose:

Table 1: *Experiments*

Name	$\|V\|$	$\|E\|$	*quality*	Time(s)
1	12	22	1.35	0.01
2	12	25	1.17	0.01
3	24	39	1.27	0.02
4	40	66	1.31	0.05
5	41	69	1.26	0.05
6	77	98	2.90	0.13
7	140	190	4.58	0.68

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 $dem(t)/sc(t)$, where $dem(t) and sc(t)$ are demand and serving cost of task t, respectively;
4. minimize the term $dem(t)/sc(t)$;
5. use rule 1 if the vehicle is less than half-full, otherwise use rule 2.

Based on it, I add two additional rules, or more precisely, there is only one rule, that is, randomly choose the candidate tasks, which may lead the best result sometimes. One manifestation is randomly choose by **RANDOM_SEED**, and the other is choose by the sequence of edges (always choose the first one that be found).

3. Empirical Verification

3.1. Experiments

There are given several example files to test, all solution of them has been verify that is fessible. The quality of solutions as *table 1*, where *quality* is q/q_{best}. According this table, the quality getting worse as the data set gets bigger, but the excute time is very short all the way.

3.2. Data and data structure

Data being used in this project is **VERTICES**, **DEPOT** and **CAPACITY**. Data structure used in this project includes dictionary, array, list, and set.

3.3. Performance

The performance can be simply measured by actual running time as *table 1*. And also, theoretic running time, computed by time complexity of functions is about $O(|V| * |E|^2 + |task|^2)$.

3.4. Analysis

Here the code I stated is just based on the path scanning method, a classical method, which is not a good enough algorithm today, but still is a really effective method to get a much better result than random function. It is widely be used as other iterative heuristic algorithm's initial result such as MAENS[1], MA[2], *etc*.

References

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

[2] ——, *Competitive memetic algorithms for arc routing problems*, Annals of Operations Research, 131(2004), pp. 159-185.