Report for Project CARP

Name ID: Yutong Wang 11611808
Department: Computer Science and Technology
Institution: Southern University of Science and Technology
Email: 11611808@mail.sustc.edu.cn

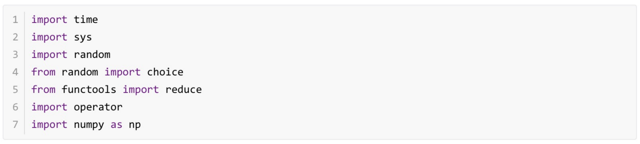
***Abstract*—The capacitated arc routing problem (CARP) is an NP-hard problem in vehicle routing with applications where a service must be provided by a set of vehicles on specified roads [1]. CARP has a great number of applications in the real life such as urban garbage collection, sprinkling salt on roads in winter, mail delivery, network maintenance and so on. In the second project of SUSTech CS303 Artificial Intelligence, we are asked to discover methods to determine a set of routes for the vehicles to serve all the tasks with minimal costs. To address this problem, well-designed greedy algorithm and huristic methods are necessary.**

I. Preliminaries

In this part, I will briefly describe the software and algorithm I used for the project. For the Algorithm part, I introduce the general idea of my implementation, the detail of which will be discussed soon in Section Methodology.

A. Software

This project is written in Python(version 3.6.6) using the IDE PyCharm. The library being used includes sys, time, random, functions, operator and numpy.



```python
import time
import sys
import random
from random import choice
from functools import reduce
import operator
import numpy as np
```

Fig 1. Libraries used.

B. Algorithm

My algorithm to solve CARP mainly includes two parts. The frist part is to use *Path Scanning* method to find an initial solution and iterate this process with enough times to make this initial solution good enough. In this part, I use two methods to optimize the greedy algorithm to get a good initial solution. And the second part is to add some heuristic algorithms to make the initail silution better. Due to time constraints, I've just implement the *local search* heuristic mentioned, which is a small part of the *MEANS* algorithm proposed by Ke Tang et al. in 2009 [2]. Details will be illustrated in the next section.

II. Methodology

This part includes the representation, the architecture of codes and details of my algorithm.

A. Representation

1. Representation of some global variables in the code:

- **vertices** : The number of vertices.
- **capacity** : The capacity of one vehicle.
- **cost_matrix** : A 2d array with size $(vertices+1)\times(vertices+1)$ to store the cost between any two connected vertices.
- **demand_matrix** : A 2d array with size $(vertices+1)\times(vertices+1)$ to store the demand between two connected vertices which is a required task.
- **min_dist** : A 2d array with size $(vertices+1)\times(vertices+1)$ to store the minimum distance between any two vertices no matter whether they are connected or not. The minimum distances are calculated by *Floyd's Algorithm*.
- **free_set** : A set to store all the required edges(aka. tasks).
- **actual_total_cost** : *type integer* The total cost of all the routes in one solution.
- **actual_total_cost_list** : A list to store the actual_total_cost of all the solutions.
- **all_routes** : A list to store all the routes of one solution.
- **all_routes_list** : A list to store all_routes of all the solutions.

2. Notations in the paper:

Notation	Meaning
L	number of routes in the best initial solution
R_{ini}	routes list of the initial best solution
C_{ini}	total cost of the initial best solution
Q	capacity of a vehicle
L	load of the vehicle
v_0	depot
(v_{from}, v_{to})	an edge
$c(e)$	cost of edge e
$d(e)$	demand of edge e

Table 1. Notations table.

B. Data Structure

The main data structure in the algorithm to represent the problem is an undirected graph which is implemented by 2d matrices. The relative matrices includes *min_dist*, *cost_matrix*, and *demand_matrix*.

List, aka. *actual_total_cost_list* and *all_routes_lista*, is frequently used in the algorithm to store the history solutions. With the advantage of indexing, it is very convenient to iterate all the solutions and find out corresponding cost and routes.

Set, aka. *free_set*, is also an important data structure which is used to avoid serving two required edges more than one times.

C. Functions Explanation

Here list all functions in the codes with their usage.

- ***comb*** : With two parameters n and k, it can calculte the value of C_n^k.
- ***set_globals*** : Analyze the provided file, extrat the data information and then assign the value to corresponding global variables.
- ***extract_digit*** : Extract digit from a string, which is every line in the given file.
- ***set_cost_n_demand_matrix*** : Assign value to *cost_matrix* and *demand_matrix*.
- ***set_free_set*** : Assign value to *free_set*.
- ***set_min_dist*** : Calculate the minimum distance between any two vertices and assign value to *min_dist*.
- ***find_v_2_all*** : Given one vertice *v* and find the minimum distance between v and every other vertices in the graph.
- ***path_scanning*** : The path scanning algorithm to find the best initial solution.
- ***path_scanning2*** : The path scanning algorithm used for *local search* heuristic algorithm.
- ***main*** : The main function in which some of the above functions are invoked. And after finding the best initial solution, the *local search* heuristic algorithm is implemented.

D. Model Design

- **Step 1**:
  Open file and read lines. Store every lines in a list and every element in the list is of type *string*.
- **Step 2**:
  Pass the list as a parameter to invoke the function *set_globals*. In this *set_globals* function, all the global variables mentioned above in *Representation in Code* are initialized according to the list.
  The function *extract_digit* is used to extract the data value from the string and *set_cost_n_demand_matrix*, *set_free_set*, *set_min_dist* are all invoked in the *set_globals*, with the aim of initializing all the global variables.
- **Step 3**:
  Iterations of *Path Scanning Algorithm* begin. In each time, calling the function *path_scanning* and store the new solution — routes and corresponding total cost in the lists. After 4000 iterations, a best initial solution with minimum total cost C_{ini} can be found.
- **Step 4**:
  Randomly choose two routes in the list R_{ini} and iterate C_L^2 times. In each iteration, break all of the tasks order and invode the function *path_scanning2* to do the *path scanning algorithm* again to get a new solution. If the new solution's cost is smaller than before, then update the solution. This step is also called *local search heuristic algorithm*.

E. Detail of Algorithm

In this part, more details for the main algorithms will be discussed. The main algorithms I used in this project are *path scanning algorithm* and *local search heuristic algorithm*.

Algorithm 1 *path scanning*

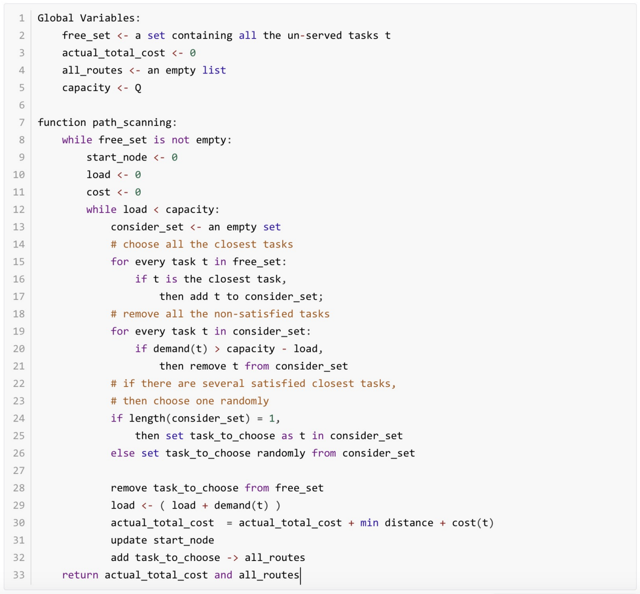
In traditional path scanning algorithm, one route starts from v_0 and each time it chooses the closest task e with $d(e)$ smaller than or equal to $(Q - L)$. If there is no task e with $d(e)$ smaller than or equal to $(Q - L)$, the current route returns back to v_0. If there are more than one closest satisfied tasks, then use some rules to select one task.

As is mentioned, I use two approaches to optimize this path scanning algorithm.

Approach 1 When there are more than one closest satisfied tasks, I randomly choose one task from them and don't care any of the rules.

Approach 2 Every time I check the tasks which haven't been served yet, I only check the closest task rather than all the satidfied tasks with $d(e)$ smaller than or equal to $(Q - L)$. If the closest task's $d(e)$ is larger than $(Q - L)$, the current route returns back to v_0.

The **pseudo-code** is as followed:

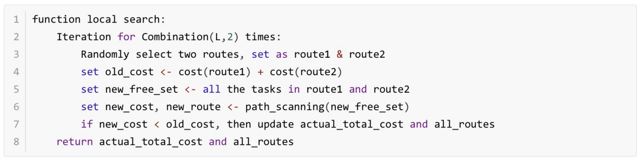


```
Global Variables:
    free_set <- a set containing all the un-served tasks t
    actual_total_cost <- 0
    all_routes <- an empty list
    capacity <- Q

function path_scanning:
    while free_set is not empty:
        start_node <- 0
        load <- 0
        cost <- 0
        while load < capacity:
            consider_set <- an empty set
            # choose all the closest tasks
            for every task t in free_set:
                if t is the closest task,
                    then add t to consider_set;
            # remove all the non-satisfied tasks
            for every task t in consider_set:
                if demand(t) > capacity - load,
                    then remove t from consider_set
            # if there are several satisfied closest tasks,
            # then choose one randomly
            if length(consider_set) = 1,
                then set task_to_choose as t in consider_set
            else set task_to_choose randomly from consider_set

            remove task_to_choose from free_set
            load <- ( load + demand(t) )
            actual_total_cost  = actual_total_cost + min distance + cost(t)
            update start_node
            add task_to_choose -> all_routes
    return actual_total_cost and all_routes
```

Algorithm 2 *local search heuristic algorithm*

Since there are several routes in the solution all_routes, I randomly choose two routes and put all the tasks in these two routes in a new $free\_set$ without order and then do *path scanning algorithm* to these tasks. In this way, I can get a new solution of all_routes and $actual\_total\_cost$ and compare with the previous one to update the solution, making the cost smaller.

The **pseudo-code** is as followed:

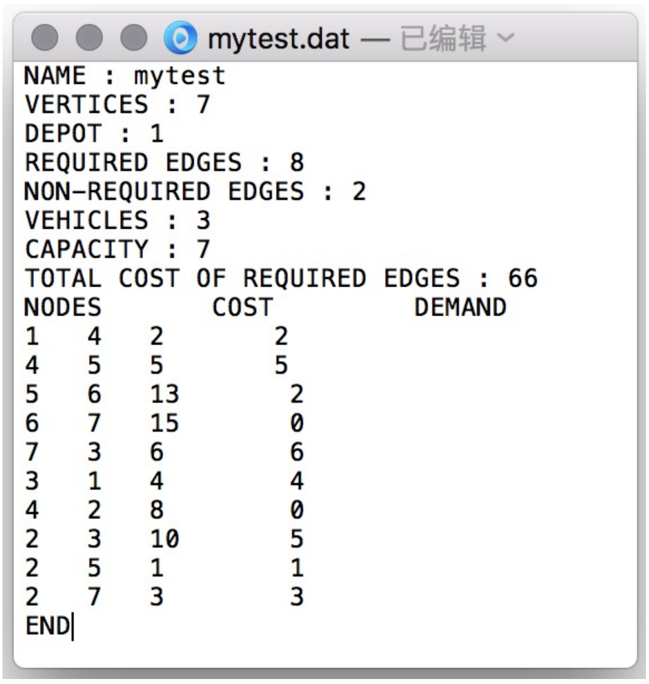


```
function local search:
    Iteration for Combination(L,2) times:
        Randomly select two routes, set as route1 & route2
        set old_cost <- cost(route1) + cost(route2)
        set new_free_set <- all the tasks in route1 and route2
        set new_cost, new_route <- path_scanning(new_free_set)
        if new_cost < old_cost, then update actual_total_cost and all_routes
    return actual_total_cost and all_routes
```

III. Empirical Verification

A. Dataset

Except for the data that carp platform supply, I only conduct one small dataset *mytest.dat* for debugging errors and checking the feasibility of my algorithm. The reason to conduct it is that this small dataset(Fig 2) is extract from the familiar example(Fig 3) and the correct answer is known. So it is pretty easy to implement.



```
NAME : mytest
VERTICES : 7
DEPOT : 1
REQUIRED EDGES : 8
NON-REQUIRED EDGES : 2
VEHICLES : 3
CAPACITY : 7
TOTAL COST OF REQUIRED EDGES : 66
NODES       COST        DEMAND
1   4   2       2
4   5   5       5
5   6   13       2
6   7   15       0
7   3   6        6
3   1   4        4
4   2   8        0
2   3   10       5
2   5   1        1
2   7   3        3
END
```

Fig 2. mytest.dat

B. Performance measure

I use the terminal on Mac Pro to test the performance of the code. And I use total cost and corresponding running time to measure the performance. For total cost, the smaller the better. For running time, the faster the better.

C. Hyperparameters

The hyperparameter is the iteration times of the optimized path scanning algorithm, which is chosen to be 4000. With the hyperparameter being larger, the time cost increases but the total cost of the CARP problem doesn't become better. So I choose 4000 to be the iteration time.

D. Experimental results

1. Set the iteration time as 2000, and compare the total cost of different algorithms: *completely randomly choose, traditional path scanning, optimized path scanning with Approach 1, optimized path scanning with Approach 1 & 2, optimized path scanning + local search.*

Dataset: egl-s1-A.dat

Algorithm	Total cost
completely randomly choose	12800
traditional path scanning	6390
optimized path scanning with Approach 1	5800
optimized path scanning with Approach 1 & 2	5460
optimized path scanning + local search	5398

Table 2. Total cost of different algorithms.

2. Vary iteration times of path scanning and compare the total cost and running time of the algorithm *optimized path scanning + local search*. The result shows that with the iteration times increases, the result of total cost is stable around 5400.

Dataset: egl-s1-A.dat

Iteration times	Total cost	Running time(s)
10	5799	2.43
200	5505	3.39
500	5527	4.95
1000	5397	7.36
2000	5406	13.12
5000	5448	27.97
10000	5397	53.17

Table 3. Total cost with different iteration times.

E. Conclusion

The two approaches to optimize the greedy algorithm aka. *path scanning algorithm* contribute a lot to the total cost of the CARP problem. As can be seen from the experimental result in table 2, the total cost reduce from 6390 to 5460. Besides, randomness in the algorithm is also an import part. With the iteration times increases, the randomness can make the result of cost consistent with normal distribution. So a best initial solution can be achieved.

As for the local search heuristic algorithm, it can optimize the cost but the improvement seems not big enough. As a result, I will work on better heuristic algorithm in the future to find better solution.

References

[1] B. L. Golden and R. T. Wong, Capacitated arc routing problems, Networks, vol. 11, no. 3, pp. 305315, 1981.

[2] K. Tang, Y. Mei, and X. Yao, "Memetic Algorithm with Extended Neighborhood Search for Capacitated Arc Routing Problems", *IEEE Trans. Evol. Comput.*, VOL.13, NO.5, OCT. 2009.