Path-Scanning Algorithm for the Capacitated Arc Routing Problem

Feng Chenchen, 12011914 *CSE, Southern University of Science and Technology*

***Abstract*— The Capacitated Arc Routing Problem (CARP) is a NP-hard combinatorial optimization problem. When given an undirected graph, the objective need to find a minimum cost set of tours that services a subset of edges with positive demand under capacity constraints. In this paper, path-scanning algorithm with local searching function is implemented based on Python to solve this problem. Based on path-scanning as main framework, the algorithm adopts random path scanning and flip to solve the optimal solutions of NP-hard problems.**

***Index Terms*— capacitated arc routing problem; path-scanning; local search; order crossover; string manipulation;**

I. Introduction

THE Capacitated Arc Routing Problem applies to those edges of a given connected graph that need service. There is a fleet that starts at some point in the network, and all the vehicles in the fleet are the same. Each side must be provided by one vehicle, and the service must be completed at once. All sides are allowed to be passed as many times as desired. Each vehicle starts from the parking lot and returns to the starting point after the service. You want to find the shortest possible path to complete all the services.

The arc path problem can be traced back to Euler's seven-hole bridge problem in 1936, and the classic arc path problem in China is the Chinese post road problem proposed by Professor Guan Meigu (1962), which requires the postman to traverse each side of the undirected graph at least once to find the minimum cost. The arc path problem can be roughly divided into three categories: Chinese post road problem, rural post road problem, capacitated arc routing problem. In this report, we study the capacitated arc routing problem.

CARP is a NP-hard problem, which was first proposed by Golden (1981), and a large amount of literature is based on this problem. Since Goldeng proposed it in 1981, CARP has been widely applied in daily life, especially in municipal services, such as road sprinkler path planning, garbage recovery vehicle path planning, road de-icing vehicle path planning, and school bus transfer path problems.

To characterize the path-scanning algorithm, We can characterize it in terms of six rules when multiple tasks are the closest to the end of current path.

Rule 1: Maximize the distance from the task to the depot.

Rule 2: Minimize the distance from the task to the depot.

Rule 3: Maximize demand(t)/cost(t), where demand(t) and cost(t) are demand and distance cost of task t.

Rule 4: Minimize demand(t)/cost(t), where demand(t) and cost(t) are demand and distance cost of task t.

Rule 5: Use rule 1 if the vehicle is less than half- full, otherwise use rule 2.

Rule 6: Select a node connected to it randomly.

As for the practical implications of carp, I think it can be applied to mail route planning or urban design planning. CARP was developed to solve the route problem of choosing the shortest route to send or dispatch all mail, so it makes sense to apply it to route planning. I want to talk mainly about applying it to urban planning. The essence of a city is the organic combination of many functions, such as schools, hospitals, shopping malls, hotels and so on. We can give different buildings different demand weights, and find the best geographical location of different buildings by using CARP. So as to facilitate the daily travel of citizens as much as possible.

For this project, my idea is to apply the graph theory knowledge I have learned into reality. Although we have learned a lot of knowledge in the theory class, we have few opportunities to realize it. With the help of this assignment, I can apply the knowledge I have learned to solve the practical problems we often encounter in real life, which I think is quite meaningful.

II. PRELIMINARY

The problem can be formulated as a Minimize problem, which is specified by a tuple (I, c_e, $\delta(S)$, $E(S)$, $\delta_R(S)$, $E_R(S)$, $even$, x_e, y_e). The goal of the problem is to minimize the cost.

Minimize

$$\sum_{p\in I}\sum_{e\in R} c_e x_e + \sum_{p\in I}\sum_{e\in E} c_e y_e$$

s.t.

$$\sum_{p\in I} x_e = 1 \quad \forall e \in R$$

$$\sum_{e\in R} d_e x_e = 1 \quad \forall p \in I$$

$$x_p(\delta_R(S)) + y_p(\delta(S)) \geq 2x_f \quad \forall p \in I$$

$$x_p(\delta_R(S)) + y_p(\delta(S)) = even \quad \forall p \in I$$

$$x_E \in \{0,1\} \quad y_e >= 0$$

TABLE I
PARAMETER INTERPRETATION IN THE FORMULATION

Symbol	Meaning
$I = 1, 2, ..., K$	set of vehicles
c_e	the cost of edge e which is passed
$\delta(S)$	subset of edge, one vertice in $V - S$, another vertice in $S \subseteq$V
$E(S)$	subset of edge, both of vertices in $S \subseteq$V
$\delta_R(S)$	subset of demand edge, one vertice in $V - S$, another vertice in $S \subseteq$V
$E_R(S)$	subset of demand edge, both of vertices in $S \subseteq$V
$even$	a positive even number
x_e	two element variable, when serving edge e, it's 1, else 0
y_e	number of times passed edge e but not served

III. METHODOLOG

A. General workflow

The proposed method divides into steps 1,2 and 3, each involving dijkstra algorithm, path-scanning algorithm and flip. After obtaining qualified paths through path-scanning, the path distance of these paths is calculated. If the path distance is less than 1.05 times of the shortest path, the path distance is flipped and recalculated. If the path distance is less than the shortest path, the shortest path is updated.

B. Detailed algorithm design

1) Dijkstra: This algorithm mainly calculates the distance between each vertex in the graph and saves it as a two-dimensional array for use in path-scanning algorithm.

Algorithm 1 Dijkstra(G,w,s)

Initialize-Single-Source(G,s)
$S = \emptyset$
$Q = G.V$
repeat
  u = Extract-Min(Q)
  $S = S \cup u$
  for each $vertexv \in G.Adj[u]$ **do**
    Relax(u, v, w)
  end for
until $Q = \emptyset$

2) Path Scanning: Every route starts from the depot. Let Sbe the demand side set, which is closest to the end of the current node, has not been served, and does not exceed the capacity of the current route. If $S$is empty, the shortest path found by dijkstra is used to return to the repository. If $S$is not empty, the required edge is randomly selected in S or according to the five rules as the next edge of the route to be served, and the current node is updated to the end of the selected edge.

rule 1:maximize the distance from the task to the depot.

rule 2:minimize the distance from the task to the depot.

rule 3:maximize the term dem(t)/sc(t), where dem(t) and sc(t) are demand and serving cost of task t, respectively.

rule 4:minimize the term dem(t)/sc(t), where dem(t) and sc(t) are demand and serving cost of task t, respectively.

rule 5:use rule 1 if the vehicle is less than half- full, otherwise use rule 2.

In my implementation, I set the rule 2 as the highest priority. Then, if the two required edge have the same distance, I will randomly choose one edge to go. Because once I choose to follow only one rule, my path choice is dead, and many potentially better paths will not be taken.

At the same time, I am not completely random. I will assign good or bad values to different edges according to their distance from the current node. Good edges have a higher probability of being selected. This method not only enables me to have complete path selection, but also enhances the availability of the randomly generated path, which can greatly improve the probability of finding the optimal solution in the random path.

Algorithm 2 Random Path-Scanning

$k \leftarrow 0$
copy all required arcs in a list $free$
repeat
  $k \leftarrow k + 1; R_k \leftarrow \emptyset; load(k), cost(k) \leftarrow 0; i \leftarrow 1$
  repeat
    $\overline{d} \leftarrow \infty$
    for each $u \in free | load(k) + 1_u \leq Q$ **do**
      if $d_{\underline{i},beg(u)} < \overline{d}$ **then**
        $\overline{d} \leftarrow d_{\underline{i},beg(u)}$
        $\overline{u} \leftarrow u$
      else if $d_{\underline{i},beg(u)} = \overline{d}$and $better(u, \overline{u}, rule)$ **then**
        $\overline{u} \leftarrow u$
      end if
    end for
    add $\overline{u}$ at the end of route R_k
    remove arc $\overline{u}$ and its ooposite $\overline{u} + m$ from $free$
    $load(k) \leftarrow load(k) + q_{\overline{u}}$
    $cost(k) \leftarrow cost(k) + \overline{d} + c_{\overline{u}}$
  until $free = \emptyset$ or $(\overline{d} = \infty)$
  $cost(k) \leftarrow cost(k) + d_{i1}$
until $free = \emptyset$

3) Flip: To simplify the problem, we can change the path-handling problem to a string-handling problem, because ultimately what we want is the string representing the path and the cost of the path, so we can directly manipulate the string representing the path. What flip does is treat the path as a string, flip the edge (a, b) to the edge (b, a), change the string, and then calculate the length of the corresponding path from the string. If the cost is less, update the minimum path string to the string after the flip, if the cost is more, discard the flip updated string.

In order to save time, I set a threshold 1.05 and only flip a string whose path cost is less than the threshold. In doing so, I saved a lot of time in a random path that would have never been possible to generate a minimum path cost through flip.

Algorithm 3 Flip(parts)

```
ori_score = cal_cost(parts)
for each part ∈ parts do
  i = 0
  repeat
    swap parts[i] and parts[i+1]
    if cal_cost(parts) ≥ ori_score then
      back_swap(parts[i], parts[i+1])
      continue
    end if
    i+ = 2
  until i ≥ len(parts)
end for
```

4) Local Search: Apart from recombination, local search plays an important role in any hybrid or memetic forms. Firstly, the child C_1 is improved using a local search with a probability r_{LS}. Three move operators have been used to perturb the solution, i.e., $single\ insertion, double\ insertion$, and $swap$. The best chromosome C_2 with the shortest distance is the outcome of this phase. It is very important to highlight that if the top Q is still the same after a generation, a roulette wheel selection is used to identify C_1.

The best result S from splitting C_1 identified from phase 1 is now improved further using a larger search domain, wherein tasks of two vehicle trips are redistributed among them using five rules of path scanning respectively resulting in five candidate part solutions. The best candidate part-solution is inserted into the rest of the S to result in the new solution S_C . Since there are C_2 N combinations possible, the following condition is enforced, i.e., if I:N!/2(N-2)! $\leq$ 50,l_{times} = I, else l_{times} = 50, where l_{times} denotes the number of attempts

Algorithm 4 Local Search Algorithm

```
Perform singleinsertion operator to C1 ⟹ C_1^1
Perform doubleinsertion operator to C1 ⟹ C_1^2
Perform swap operator to C1 ⟹ C_1^3
Keep the best one of C_1^1, C_1^2 and C_1^3 ⟹ C_2
Apply split method to of C_2 ⟹S = (S_1, S_2, ..., S_N)
l_times = min[I,50]
for i=1 to l_times do
    Apply path-scanning to each pair of solutions to generate
    five different part solutions
    Select the best one of them ⟹ S_C
    Combine S_C with the rest solutions ⟹ C_3
    Apply split method to evaluate C_3
    Update the solution if C_3 is better than C_2, C_3 ⟹C_new
end for
if FoundC_new = true then
    Apply local search again on C_new
end if
```

C. Analysis

To simplify the path-finding problem, I converted it to a string problem. In this problem, I looked for the path through the path-scanning algorithm and converted it into string output. Then the string is used as input and the corresponding distance cost is calculated by cal_cost algorithm.

At this point, it's easy to continue optimizing the path. We only need to conduct operation and mutation of the string generated by path-scanning, then calculate the path cost of the new string through cal_cost, and update the answer if the cost is smaller. At this point, the subsequent optimization becomes a string processing issue.

IV. EXPERIMENT

A. Setup

At first I take to sample data the teacher gived us as a my set, including egl-e1-A, egl-s1-A, gdb1, gdb10, val1A, val4A, val7A sample files. These files contain both small and large pictures, of which the optimal solution of the small picture is more than 200, and the optimal solution of the large picture is more than 5,000. After that, I will present my performance on this data set in tabular form.

Later, I found the data set of the paper on Github, including all the bccm, eglese, gdb and kshs data sets. However, because the format of these data sets was different from the format required by the project, I selected only a few files and modified the format, and tested my code as input, and the effect was OK.

data set source: https://github.com/edydfang/SUSTech_CS303_Artificial_Intelligence/tree/main/Lab02_CARP/datasets

- Software:Pycharm 2022.1
- Hardware:Intel(R) Core(TM) i7-10710U CPU @ 1.10GHZ 1.61GHZ
- Python:3.10
- Numpy:1.21.5

B. Result

For the NP-Hard problem of finding the optimal path in finite time, the most important thing is undoubtedly the size of time consumption and path cost.

In order to test the maximum possibility of a good path when the random probability is equal to what, I tested the shortest path found with different probabilities under the condition of 60s and the picture is gdb10.

Through the experiment, I found that the threshold value of randomly taking other paths was 0.3, which was in line with my expectation. Because it has to be less than 0.5, it is more likely to choose a solution with higher probability (the node closest to the current node) when choosing a path. Then the threshold should not be too small, so that the path selection is more likely to take paths that have not been taken.

Another interesting point is that I only flip paths that are less than a certain multiple of the minimum path cost. It is like the survival of the fittest in nature, where only the good individuals have a chance of inheriting their genes and the poor ones have no chance of inheriting them. Plugging into

TABLE II
EXPERIMENTAL RESULTS OF DIFFERENT DATA SETS

File	10s Quality Quality Diff	30s Quality Diff	60s Quality Diff	90s Quality Diff	Optimal Solution
egl-e1-A	3959 11.58%	3931 10.79 %	3852 8.56%	3848 8.45%	3548
egl-s1-A	6029 20.12%	6070 20.96%	5873 17.03%	5917 17.91%	5018
gdb1	316 0%	316 0%	316 0%	316 0%	316
gdb10	289 5.09%	283 2.90%	283 0%	275 0%	275
val1A	184 6.35%	178 2.89%	179 2.89%	174 0.57%	173
val4A	431 7.75%	423 5.75%	421 5.25%	420 5.00%	400
val7A	304 9.74%	304 9.74%	294 6.13%	297 6.40%	277

TABLE III
EXPERIMENTAL RESULTS OF DIFFERENT RANDOM THRESHOLDS

tests	threshold=0.2	threshold=0.3	threshold=0.4	threshold=0.5
test1	284	285	283	289
test2	289	283	284	289
test3	283	275	283	277
test4	275	283	285	283
test5	283	285	285	284
test6	284	285	283	283
test7	285	277	283	285
average	283.29	281.85	283.71	284.28

the project means that only the path that costs less will have a chance to flip, rather than letting the path that costs more take up the flip's time. This is given because after a path is flipped, the path cost is rarely dramatically reduced. Setting this coefficient can effectively reduce the useless path turnover and save valuable time to solve this problem in a limited time.

C. Analysis

I think my algorithm performs well on small graphs and can be further optimized on large graphs. The overall results barely met my expectations, but due to time constraints, I didn't have time to inherit and mutate the path string. In the future, for example, in the winter vacation, I can use the genetic knowledge learned in the experiment class to inherit and mutate the path string.

In the previous section, I have analyzed the effects of different parameters on the experimental results, so I will not repeat the details here.

Although my theoretical experiment is $O(n^3)$, my code is still fast. I think the first is to remove some of the meaningless parts of the code, such as flipping paths that cost much. It's also partly because I use break and continue in the loop, so it doesn't always execute.

V. CONCLUSION

A. Characteristics and Realization of Expectation

My algorithm has a good effect on solving the problem of small graph. The optimal solution of the graph can be found after 60 seconds of operation, or the optimal solution is usually less than 5% larger than the optimal solution of the graph. However, in the large graph, the performance is not so good. The optimal solution found after 60 seconds is often 10 to 15 percent larger than the optimal solution of the graph. I think this is because the graph is large and it takes time to find a complete path, so the number of paths that can be found within the effective time is much smaller than that of the small graph.

I think the experimental results barely meet my expectations, but due to the lack of time, there are still many areas and details that can be further improved. However, I was happy to see that the code I wrote was finally able to solve NP-Hard problems that could not be solved manually. The effort paid off.

B. Further Improvement

Randomness can be improved by reducing the probability that a passed edge will pass again. There's a higher probability of traversing different combinations of paths in the graph.

Genetic algorithm can be added to the path string for inheritance and variation. So the problem is just like the string generation problem, the only difference is that you need to add a string legitimacy check.

REFERENCES

[1] José M. Belenguer, Benavent E (2003) A cutting plane algorithm for the capacitated arc routing problem. Computers Operations Research 30(5):705-728.

[2] Baldacci R , Maniezzo V (2006) Exact methods based on node-routing formulations for undirected arc-routing problems. Networks 47(1):52-60.

[3] Diego Pecin, Eduardo Uchoa (2019) Comparative Analysis of Capacitated Arc Routing Formulations for Designing a New Branch-Cut-and-Price Algorithm. Transportation Science 53(6):1673-1694.

[4] Hertz, A., Laporte, G., and Mittaz, M. (2000). A tabu search heuristic for the capacitated arc routing problem. Operations research, 48(1):129–135.

[5] Polacek, M., Doerner, K. F., Hartl, R. F., and Maniezzo, V. (2008). A variable neighborhood search for the capacitated arc routing problem with intermediate facilities. Journal of Heuristics, 14(5):405–423.

[6] Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2010). An improved ant colony optimization based algorithm for the capacitated arc routing problem. Transportation Research Part B: Methodological, 44(2):246–266.

[7] Chen, L., Gendreau, M., H'a, M. H., and Langevin, A. (2016a). A robust optimization approach for the road network daily maintenance routing problem with uncertain service time. Transportation research part E: logistics and transportation review, 85:40–51.

[8] Laporte G . Arc Routing: Problems, Methods, and Applications[M]. Society for Industrial and Applied Mathematics, 2015.

[9] Belenguer J M , Benavent E . The Capacitated Arc Routing Problem: Valid Inequalities and Facets[J]. Computational Optimization and Applications, 1998, 10(2):165-187.