Capacitated Arc Routing Problems Report

Huihui Huang
Department of Computer Science and Engineering
Southern University of Science and Technology
12010336
12010336@mail.sustech.edu.cn

I. INTRODUCTION

A. Problem Description

The capacitated arc routing problem (CARP) is one of the most typical form of the arc routing problem which can be described as follows: a mixed graph $G = (V, E, A)$, with a set of vertices denoted by V, a set of edges denoted by E and a set of arcs denoted by A. There is a central depot vertex $dep \in V$, where a set of vehicles are based. A subset $E_r \subseteq E$ composed of all the edges required to be served are called edge tasks. A subset $A_r \subseteq A$ composed of all the arcs required to be served are called arc tasks. Each edge or arc in the graph is associated with a demand, a serving cost, and a deadheading cost. Both the demand and the serving cost are zero for the edges and arcs that do not require service. A solution to the problem is a routing plan that consists of a number of routes for the vehicles, and the objective is to minimize the total cost of the routing plan.

B. Problem Applications

The CARP has many applications in the real world, such as urban waste collection, post delivery, sanding or salting the streets [1], [2], etc.

C. Purpose

The project is separated into two parts. The goal of the first part is to find one solution of CARP subject to the following constraints : [3]

- Each route starts and ends at the depot.
- Each task is served in exactly one route.
- The total demand of each route must not exceed the vehicle's capacity Q.

The goal of the second part is to try the best to minimize the total cost of the solution.

II. PRELIMINARIES

A. Terminology

The list of all my terminologies used in my report is given below.

- **required edge**: The edge that should be served.
- **non − required edge**: The edge that does not need to be served.
- **serving cost**: The cost of a vehicle traveling along the edge or arc if serve it.
- **deadheading cost**: The cost of a vehicle traveling along the edge or arc without serving it.
- **depot**: Source and destination of all routes.

B. Notation

The list of all my notations used in my report is given below.

- **floyd**: A two-dimensional array which stores the shortest path between any two vertices.
- **psize**: Size of the population
- **bestPopulation**: A list of $psize$ best solutions found by each generation.
- $\mathbf{c_{ij}}$: The cost of edge [i, j].
- $\mathbf{d_{ij}}$: The demand of edge [i, j].
- **required**: A list of all edges need to be served.
- **temptRequired**: A deep copy of $required$.
- **capacity**: The capacity of a vehicle.
- **depotPosition**: Position of the depot.
- **task**: A tuple with the start position and the end position.
- **path**: Contain the tasks that served by the vehicle each time it starts from the depot and ends at the depot.
- **routes**: One of the solutions composed of $path$.
- **load**: The total load of each $path$ when finish the services.
- **candidates**: A set of tasks that mostly satisfy the current chosen rule.
- **evaluation**(): A function to get the value calculated by different rules of the current task.
- **curRuleResult**: The value of current task calculated from $evaluation()$.
- **ruleResult**: Record the maximum or minimum value of the $curRuleResult$ obtained based on the requirements of the five rules.
- **totalCosts**: Sum of the cost of $required - edges$.
- **mother**: The $routes$ for generating the new $routes$ in the crossover algorithm.
- **father**: The another $routes$ for generating the new $routes$ in the crossover algorithm.
- **son**: The $routes$ that generated by the crossover algorithm.
- **combine**(): A function to combine tow mutated $routes$.

III. METHODOLOGY

A. Workflow

The CARP is a path planning problem with constraints. Therefore, the first step is to find some feasible solutions and

then improve them as better as possible.

In the first step, $path\ scanning\ algorithm$ is a fundamental method. It is a method with a greedy algorithm that always chooses the task with the best value according to the evaluation rule. However, it may meet the problem that there is more than one task have the same best value, so I design to randomly choose one of the best tasks.

In the second step, to improve the solutions, I use $crossover\ algorithm$ to produce a new generation with 6 times the size of the father population. And then choose the best $psize$ solutions for the new population. And each time I get a new population I will choose the solution with the smallest costs as the $bestSolution$. Fig.1 may show the workflow more clearly.

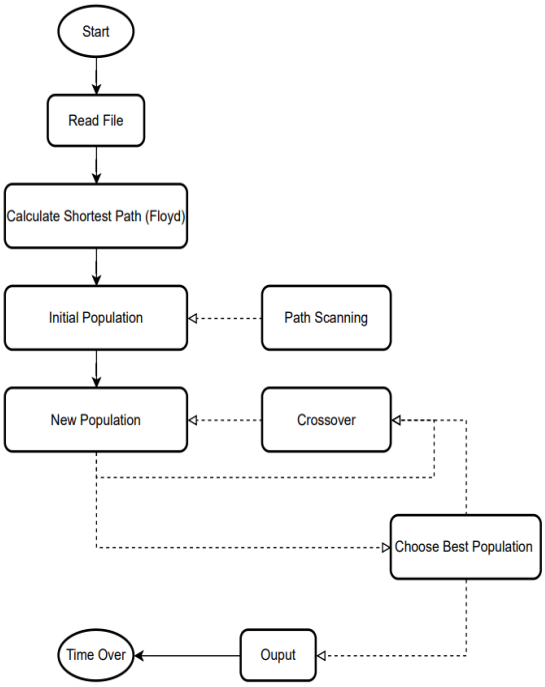

Fig. 1. Workflow of the Program

B. Detailed algorithm

1) Path Scanning Algorithm: As shown in algorithm 1. This method is based on the procedure proposed by Golden et al. Let $temptRequired$ be a copy of the set of required edges. Each time, select one $required-edge$ in $temptRequired$ to be the next edge to be serviced according to the current rule by $evaluation()$ and extend the current $path$. And if there are more than one edges that satisfy the rule, randomly choose one.In a loop, When the $load$ is greater than $capacity$, the $path$ is terminated and added to $routes$. And when $temptRequired$ is empty which means that all the required edges have been served then return the $routes$ and $totalCosts$.

Five rules are used to determine the next $required-edge$ in the $evaluation()$ function.

- Maximize the ratio c_{ij} /d_{ij}.
- Minimize the ratio c_{ij} /d_{ij} .

Algorithm 1 *Path Scanning Algorithm*

Input: $depotPosition$, $floyd$, $required$
Output: $routes$, $totalCosts$

$temptRequired \leftarrow copy\ of\ required$
$routes \leftarrow empty\ tuple$
while $temptRequired \neq \emptyset$ **do**
  $path \leftarrow empty\ tuple$
  $start \leftarrow depotPosition$
  $load \leftarrow 0$
  $totalCosts \leftarrow 0$
  while load $<capacity$ **do**
    $candidates \leftarrow empty\ tuple$
    $ruleResult \leftarrow +\infty$
    for each $task \in temptRequired$ **do**
      if $load+demand\ of\ this\ task<capacity$ **then**
        $curRuleResult \leftarrow evaluation()$
        if $curRuleResult<ruleResult$ **then**
          $ruleResult \leftarrow curRuleResult$
          $candidates \leftarrow empty\ tuple$
          add this $task$ to $candidates$
        else if $curRuleResult = ruleResult$ **then**
          add this $task$ to $candidates$
    random choose a $task$ in the $candidates$
    remove that $task$ in the $temptRequired$
    remove the opposite of that $task$ in $temptRequired$
    calculate the $load$
    calculate the $totalCosts$
  add the $path$ to the $routes$
return $routes$ and $totalCosts$

- Maximize the return cost from j to the depot.
- Minimize this return cost.
- If the vehicle is less than half-full, then apply rule3, otherwise rule4.

2) Crossover Algorithm: As shown in algorithm 2. Crossover is the most difficult part of my program. The difficulty is how to combine two genes from the father and mother to create a new route and insert it into the original solution without any faults. In the $combine()$ function, firstly combine the two fragments to create a new route R1 and remove duplicate tasks in R1. After that, we need to check whether it is a feasible route (whether it exceeds the $capacity$). If so, remove tasks randomly until R1 is feasible then return. In the algorithm, after the combination, I will get two solutions and finally return the solution with the smaller costs.

C. Analysis

1) Analysis on Path Scanning Algorithm: The time consumption of this algorithm depends on how many tasks there are and how many paths are required to complete all the tasks. Suppose that there are n tasks, and only one path is required to complete them. When only one task is found after each traversal, then the number of traversals is $n+(n-1)+(n-2)...+2+1$. So the final time complexity is

Algorithm 2 $Crossover$

Input: $mother$, $father$
Output: son

$r1 \leftarrow random\ int\ in\ range\ 0\ to\ length\ of\ mother$
$r2 \leftarrow random\ int\ in\ range\ 0\ to\ length\ of\ father$
$path1 \leftarrow r1\ in\ the\ mother$
$path2 \leftarrow r2\ in\ the\ father$
$r11 \leftarrow random\ int\ in\ range\ 0\ to\ length\ of\ path1$
$r22 \leftarrow random\ int\ in\ range\ 0\ to\ length\ of\ path2$
$gene11 \leftarrow from\ 0\ to\ r11\ in\ path1$
$gene12 \leftarrow from\ r11\ to\ the\ end\ in\ path1$
$gene21 \leftarrow from\ 0\ to\ r22\ in\ path2$
$gene22 \leftarrow from\ r22\ to\ the\ end\ in\ path2$
$son1 \leftarrow combine(mother,\ gene11,\ gene22)$
$son2 \leftarrow combine(father,\ gene21,\ gene12)$
$son \leftarrow one\ has\ less\ cost\ between\ son1\ and\ son2$
return son

$O(n^2)$. What's more, the final result of this algorithm depends on the five rules mentioned above, as well as the judgment of conditions and the selection of thresholds when choosing different rules.

2) Analysis on Crossover Algorithm: The time consumption of this algorithm mainly depends on the size of genes to be crossed and whether the region to be crossed can get a feasible result. If a feasible result cannot be obtained, it will continue to keep crossing, which is very time-consuming and unpredictable. And when the genes to be crossed involve many tasks, it is also time-consuming to iterate through the results to verify whether the results are feasible.

IV. Experiments

A. Setup

1) Software and Hardware: The project is developed using PyCharm as an integrated development environment based on Python language. The local testing platform is Windows 10 Professional Edition with Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz of 8 cores and 16 threads.

2) Dateset: All data sets from the platform: $egl-e1-A$, $egl-s1-A$, $gdb1$, $gdb10$, $val1A$, $val4A$, $val7A$ And I also find some data sets on the internet: $egls1B$, $egl-s1-C$, $egl-s2-A$, $egl-s2-B$, $egl-s2-C$, $egl-s3-A$, $egl-s3-B$, $egl-s3-C$, $egl-s4-A$, $egl-s4-B$, $egl-s4-C$.

3) Experimental Design: When choosing the candidate $task$ I design the following three sets of experiments only based on $path\ scanning\ algorithm$:

- Mode 1: Each time, select the nearest of all valid tasks.
- Mode 2: Each time, randomly select the nearest of all valid tasks.
- Mode 3: Follow the above five rules as shown in algorithm 1.

B. Results

I used two data sets to test the results, one is the $gdb10$ data set with a small number of edges and the other is the $egl-s1-A$ data set with a large number of edges. Each experiment was given ten minutes and run eight times to calculate the mean and optimal results without $crossover$. The result is shown in TABLE I and finds that the $mode$ 3 has the best performance.

TABLE I
MODE COST RESULT OF $gdb10$

	Mean Value	Optimal Value
mode 1	281	281
mode 2	275	275
mode 3	275	275

TABLE II
MODE COST RESULT OF $egl-s1-A$

	Mean Value	Optimal Value
mode 1	12047	12047
mode 2	9625	9428
mode 3	7229	7075

Finally, I chose $mode$ 3 and ran the program with $crossover$ for eight times, each time for ten minutes. The optimal value of the final result is in TABLE III.

TABLE III
FINAL RESULT

gdb10	275
gdb1	316
egl-s1-A	5465
egl-e1-A	3758
val7A	291
val4A	412
val1A	173

C. Analysis

1) Comment on the experimental results: The experiment meets my expectations. $mode$ 1 selects the nearest task every time, so that the result is the same every time, and may jump into the local optimal case. $mode$ 2 randomly selects the nearest task, so different results can be obtained each time, increasing the possibility of obtaining the optimal solution. But because it was greedy to choose the closest task, the smaller graph performed well, but the larger one does poorly. $mode$ 3 takes a holistic approach, so it works well on both small and large graphs.

2) Results related to Methodology: As we all know, the role of path scanning is to find the best possible results with as many varieties as possible, to serve as the $mother$ and $father$ of $crossover\ algorithm$. However, $mode$ 1 does not take randomness into account, and the result is very simple. Although $mode$ 2 increases the randomness to a certain extent, it chooses the next task nearest to the current task every time,

which is easy to cause the local optimal solution, so it cannot get a good solution either. But $mode3$ not only takes many directions into account but also adds a certain amount of randomness. In this way, not only can the relatively optimal solution be obtained, but also the diversity of results can be improved, which is conducive to $crossover\ algorithm$.

V. Conclusion

A. The experience I learned

In this project, I have learned what is the constraint satisfaction problem and how to solve it. In addition, I learned not only how to obtain a solution from CARP, but also how to optimize the solution through $crossover\ algorithm$. Besides, most of the previous assignments had a unique solution, but this project let me know that the solution of real engineering is not unique, and there are many standards to measure the solution.

B. Advantages of my algorithm

With regard to $path\ scanning\ algorithm$, it mainly takes into account multiple perspectives for evaluation to prevent local scanning and adds randomness to improve diversity. As for the $crossover\ algorithm$, I do not simply mutate the existing population directly, but collected the population of the optimal solution each time and carry out the mutation based on the population of the optimal solution. This will make offspring more efficient and increase the likelihood of finding an optimal solution.

C. Disadvantages of my algorithm

The disadvantage in $path\ scanning\ algorithm$ mainly lies in the absence of good integration of the five rules, but in the implementation of a separate rule each time, followed by the selection of the optimal one from the results. However, the main problem of $crossover\ algorithm$ is that each variation is too random, and there is no fixed variation direction that can promote the result toward the optimal solution, so it is difficult to get a convergent solution.

D. Possible directions of improvements

In my point of view, the valuation function can be improved to make a multifaceted judgment based on the overall situation to prevent entering into the local optimal solution. In addition, a simulated annealing algorithm can be added to my program to help jump out of the local optimal solution and local search algorithm to increase the diversity of solutions.

References

[1] H. Handa, D. Lin, L. Chapman, and X. Yao, "Robust solution of salting route optimisation using evolutionary algorithms," in Proc. IEEE Congr. Evol. Comput. 2006, Vancouver, BC, Canada, pp. 3098–3105.

[2] H. Handa, L. Chapman, and X. Yao, "Robust route optimization for gritting/salting trucks: A CERCIA experience," IEEE Comput. Intell. Mag., vol. 1, no. 1, pp. 6–9, Feb. 2006.

[3] Tang K, Mei Y, Yao X, "Memetic algorithm with extended neighborhood search for capacitated arc routing problems[J]," IEEE Transactions on Evolutionary Computation, 2009, 13(5): 1151-1166.