Heuristic Search For Capacitated Arc Routing Problem

12011702 ZHANG Zhentao

Abstract—The Capacitated Arc Routing Problem (a.k.a CARP), has been proved as an NP-hard problem. The objective is to find a minimum cost set of tours that services a subset of edges with positive demand under capacity constraints. This problem has a wide range of application but is comparatively neglected. In this project, our goal is to find a feasible solution of CARP meanwhile as optimal as we can using heuristic search.

Index Terms—CARP, Heuristic Search, NP-hard Problem, Greedy Strategy, Genetics Algorithm

1 Introduction

Unlike traditional uncapacitated arc routing problem, which can be specialized as Chinese postman problem for one instance, the capacitated arc routing problem, assigns a demand value to a subset of edges E in graph, with the constraint of the actor taking missions in one travel round not exceeding an assigned capacity value C.

1.1 Difficulty of CARP

An NP-completeness proof is a strong indication that the problem is intractable. A problem is NP-hard if the existence of a polynomially bounded algorithm for it implies the existence of a polynomially bounded algorithm for all NP-complete problems. The proof for an approximate, restricted version of the CARP is NP-hard has been published decades ago, indicating the problem cannot be solved in polynomial time and increases in exponential way with the increment of the nodes and edges in this problem.

1.2 Realistic Application Scene

The potential application research area of CARP problem is wide and covers different aspects of subjects like finance, city designing, logistic services and so on. For example, a logistic company may have a number of depots with vehicles to collect or send cargo in specific places which lie randomly in a given map, and we need to satisfy **all those demands** with as less cost on traffic (fee for fuel, driver, etc.) as possible.

1.3 Purpose

As we can see in section 1.1, CARP is an NP-hard problem, which means we are not capable of receiving a feasible result in linear time. One instinctive thought is to use brute force, which enumerate all the possible route that met with condition of capacity limitation for a single travel. Since we need to at least get all the permutations of edges that has missions, the time complexity is lower bounded by $O(n!)$. And since we have undirected maps, the time complexity is bounded by $O(2^n n!)$. That's an incredible cost for calculating, even with parameter n is 20, it needs about 4×10^{15} minutes to find the optimal result.

Therefore, in this project, our initial goal is to find a feasible solution and then based on that solution, we can dig into the problem deeper and find more optimized solutions.

2 Preliminary

2.1 Notations and Overall Description

In the project, CARP is operated on a map that only has undirected edges. Let's Consider an undirected connected map $G = (V, E)$, with a set of vertices V and a set of edges E. In the set of edges, there's a special subset $T \subseteq E$ that has demands (tasks) on them that waits for serve. Vehicles, which is responsible for service, has capacity Q and is originally placed at designated depot $v_0 \in V$. For all the edges $e \in E$, it features a cost $c(e) > 0$, which is added to the overall cost during the whole serving process in the map. And moreover, for every edge $\tau \in T$, a demand value $d(\tau) > 0$ is assigned.

The goal of CARP problem is to find a set of route that serves all the demand edges while try to mimimize the cost of edges it has travelled through with the restriction of following rules: 1. each route must start and end at v_0. 2. Each demand must be served and for only once. 3. The total demand serviced on each route should not exceed Q.

2.2 Problem Formulation

The solution S of this problem can be written as a set of **routes** R. A route is defined as a set of edges τ with demand.

To be more accurate:

$$s = (R_1, R_2, \cdots, R_n)$$

, where $R_i = (\tau_{i0}, \tau_{i1}, \cdots, \tau_{il_i})$, l_i means the number of tasks contained in route R_i. Specifically, $\tau_{ij} = (head_{ij}, tail_{ij})$, which means the edge has two end point $head_{ij}, tail_{ij}$.

Formally the problem is to minimize the total cost of s. And let's notate it as $TC(s)$, and it results from the algebra summing of route cost, which is denoted as $RC(R)$.

$$TC(s) = \sum_{i=1}^{n} RC(R_i)$$

And for route cost $RC(R_i)$, it's calculated as: $RC(R_i) = \sum_{j=1}^{l_i} c(\tau_{ij}) + \sum_{j=2}^{l_i} dc(tail_{ij-1}, head_{ij}) + dc(v_0, head_{i1}) + dc(tail_{il_i}, v_0)$. ($dc$ is the shortest path cost between these two nodes).

Most importantly, following rules are required to obey, in formula:

$$\sum_{j=1}^{l_i} d(\tau_{ij}) \leq Q \ (\forall \ 1 \leq i \leq n) \ (1)$$

$$\sum_{i=1}^{n} l_i = |T| \ (2)$$

$$\tau_{i_1 k_1} \neq \tau_{i_2 k_2} \ \forall i_1, k_1 \neq i_2, k_2 \ (3)$$

$$\tau_{i_1 k_1} \neq inv(\tau_{i_2 k_2}) \ \forall i_1, k_1 \neq i_2, k_2 \ (4)$$

3 METHODOLOGY

In this project, our methodology is mainly based on two algorithms, one is modified path-scanning, which applies the idea of greedy strategy. In addition, genetic algorithm is applied in order to enhance the randomization of output.

3.1 General workflow

Before we get started to find a solution, we need to first read and process all information of map from the dat file. After that, we are set to use **floyd algorithm** to find shortest path between every two nodes in the map. All the shortest path between every two nodes are stored in a 2-D array, which could be accessed in $O(1)$ time and reduce duplicated calculation in running time.

When the shortest path calculation is finished, we are about to use **modified path scanning** (see Algorithm 2), which uses greedy strategy and features some random factor to produce different result.

And at the same time, **genetic algorithm** is contained to assist the above method to add some uncertainty which is way more noticable in bigger scale map.

3.2 Algorithm Design

3.2.1 Getting shortest path

In a undirected weighted graph, there're many famous algorithms to find the shortest path between every two nodes. In this project we use floyd algorithm (see Algorithm 1), which is easy to achieve.

Algorithm 1 Floyd Algorithm

```
procedure FLOYD(G(V,E)) return res
    for i,j in map do
        if i and j is not connected then res[i][j] = infinity
        else res[i][j] = cost(i,j)
        end if
    end for
    for k in V do
        for i in V do
            for j in V do
                if res[i][k]+res[k][j]¡res[i][j] then
                    res[i][j] = res[i][k]+res[k][j]
                end if
            end for
        end for
    end for
```

Although floyd algorithm without development takes $O(n^3)$, it's still acceptable because the problem scale is limited and we have plenty of time and the time spent on this algorithm is short compared to the process of attempting to finding better solution.

3.2.2 Greedy strategy

After we get the shortest path, we now start to find a solution, at least feasible. And we use **modified path scanning**, which is based on **greedy** strategy.

The algorithm can be described as we always choose the node which is the nearest from current node. And from that node, we set it as the start node, randomly select an end node in the map and check if it forms a demanded edge with the start node. If it forms, then we add it to route and if not simply continues finding another end node. If no demanded edges are found at the end of this iteration, we choose the second nearest node as the start node. It's worth noticing that we have to control the sum of demand value cannot exceed Q in one route, If no single demand are able to be added in current route, we simply start another. During teh whole process, if the current node is depot and we have executed demand in the route, we need to finish this route, empty the demand and start new one.

Algorithm 2 Greedy Strategy

```
procedure PATHSCANNING(G(V,E), shortestPath)
    assign solution an empty list
    curPosition = depot
    while has edge to serve do
        Assign route an empty list
        curDemand = 0
        curPosition = depot
        while curDemand < Q do
            sorted=nodes sorted by distance from current
            if curPosition = Depot and curDemand>0 then
                break
            end if
            for v in sorted do
                Randomly select other node t
                if (v,t) forms demanded edge then
                    Add (v,t) to route
                    Remove demand on (v,t)
                    curPosition = t
                    curDemand += d(v,t)
                    break
                end if
            end for
            break
        end while
        Append route to solution
    end while
```

3.2.3 Genetic Algorithm

With the above modified path-scanning, a good performance in small scale map is easy, but for larger scaled map, we still need GA to achieve stronger performance.

A typical genetic algorithm contains an original generation, cross of chromosomes and mutation of genes. It's generally the same here, but with slightly difference.

In typical genetic algorithm, the original generation contains series of solutions, which provides source for selection. However, in this project, we use **one** feasible solution as the **original generation**. Within this type of structure, we can define each route in this solution as **chromosomes** and the tasks in each route as **genes**. Operations during producing new generations are described as follows:

1. Sort chromosomes (routes) based on the fitness value of it. The fitness of chromosome is calculated as $\frac{1}{cost}$.

2. Randomly choose two chromosomes which has high fitness.

3. For cross and exchange of chromosomes, select a random fragment in each chromosome, and stitching them together. (Shown in Fig.1).

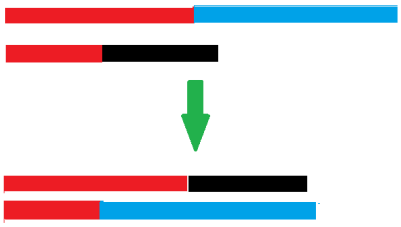

Fig. 1: Cross between two chromosomes

4. After we finished the above process, we assign a probability for the gene on chromosome to mutate. The mutation could be implemented in several ways including select one gene (edge) and reverse it, i.e. mutate the edge into its inverse edge(fig.2), or select two gene and exchange their positions(without reverse) (fig. 3).

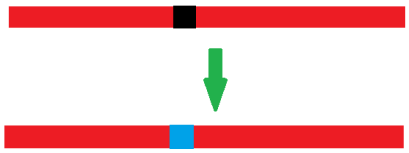

Fig. 2: Mutation for specific edge

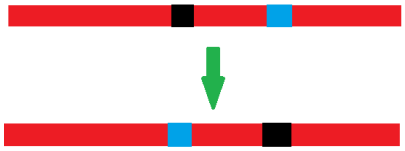

Fig. 3: Mutation for two gene

5. check whether the newly produced chromosome violates the rule, which is introduced in section 2. If not, replace the origin choromosome by the new one.

6. Do the above steps 1-5 for certain rounds.

3.3 Analysis

The modified path-scanning above do have enough power to handle small scaled map. But when it comes to bigger map, the path-scanning itself may stuck in a trap of local optimal and hard to jump out. In fact, many better solutions could be found through genetic algorithm introduced above. It can effectively add some randomization and uncertainty to solution generation, which is appreciated.

For the time complexity, the above modified path scanning could averagely be done in $O(|T|^3)$, and the genetic algorithm, can be done in $O(|V|)$ time. All of them are in linear time complexity and thus indicating acceptable running time.

4 Experiments

4.1 Setup

4.1.1 Hardware and Software Properties

CPU: Intel Core i5-10210U CPU @ 1.60GHz

GPU: No independent GPU. Integrated Intel UHD 128MB GPU

RAM: 8GB

OS: Windows 10 Home Edition

Development Tools: PyCharm 2022.1

Programming Language: Python 3.9

Dependencies: Numpy 1.21.5

4.1.2 Sample Map Test

Take three typical characterized map as test map.

Val1A: Middle-sized map, containing 24 vertices and 39 demanded edges.

gdb10: Small-sized map, containing 12 vertices and 25 demanded edges.

egl-s1-A: Huge-sized map, containing 140 vertices and 75 demanded edges.

4.2 Efficiency Test

To achieve more efficient solution generation, in this project, we use **multithreading**. We start 8 thread at the same time and half of them are allocated for method of greedy startegy and another half of them are allocated for genetic method. In multithreading, we need to apply a thread lock for global variable modification to ensure the threading security in case of dead lock or premature read/write.

Different time can be assigned to control the thread running time and the result is as listed as followed:

	gdb10	val1A	egl-s1-A
10s	275	173	6023
30s	275	173	5987
60s	275	173	6025
600s	275	173	5941

TABLE 1: Test on different maps and time limit

More over, we test the efficiency of genetic algorithm with different generations number limit on map *egl-s1-A*.

generation	egl-s1-A
10	6140
50	6013
500	5975

TABLE 2: Test on Efficiency of Genetic Algorithm

To present the data more clearly, a curve plot is given as followed (fig. 4).

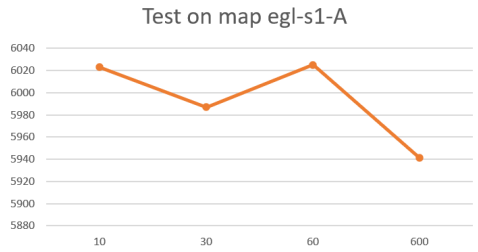

Fig. 4: Test on map egl-s1-A

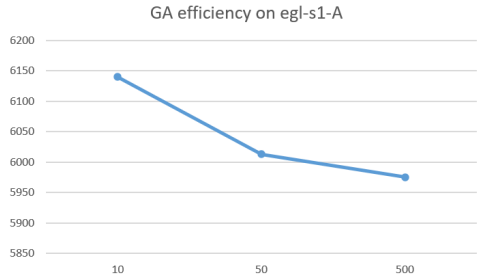

Fig. 5: GA efficiency on egl-s1-A

4.3 Analysis

From the above test data, one thing we can determine and confirm is that the modified path scanning algorithm using greedy strategy is effective in solving small sized map. Even 10 seconds is enough for all the tested middle and small scaled map. This is mostly because the number of edges with tasks are not large so that with a mechanism of random choice in the modified path scanning, we are able to always find the one with best result, i.e. the smallest cost.

However, for large scaled map like *egl-s1-A*, although an acceptable result could given by this algorithm, its efficiency is not as high as genetic algorithm. In genetic algorithm, the mechanism of cross and mutation offers a irreplaceable role in providing potential source for better solution. Just like what we see in the plot above (fig. 5), GA could converge in a smooth rate in the early and middle stage of evolving.

As we have mentioned before, the GA applied is closely connected to the selection of parent chromosomes and fitness estimation.

What we current use is directly based on cost of route, which is straight forwarding but may not performs good enough. This could results in premature converge for evolving, as the current generation cannot provide valuable chromosome and mutation is also an undecidable behavior, leading the GA performs worse in later stage.

5 Conclusion

As an NP-hard problem, CARP could be solved completely with knowledge of artificial intelligence.

In this project, the algorithm with have tried both have advantages and disadvantages. For greedy strategy, the idea is great and convenient to implement compared to genetic algorithm, featuring perfect performance in solving relatively small scaled maps, but its efficiency in solving large map is low and thus needing assist in genetic algorithm.

For genetic algorithm, the efficiency for larger map may be better than that for greedy strategy, but the implement is hard and an excellent fitness calculation and the combination of cross and mutation is even a tough mission for this project.

Generally, the result is acceptable in this project and one thing I have learned in this project is the input of command arguments. With the argument of time and seed in command, we don not need to modify the source code and this is convenient since when we want to test multiple file, we can use a bash and all the test file and time limit could be managed in it.

References

[1] Capacitated arc routing problems Bruce L. Golden,Richard T. Wong Autumn (Fall) 1981 https://doi.org/10.1002/net.3230110308