CS303 Project-2 CARP Report

Jiajian Yang
Southern University of Science and Technology
12012711@mail.sustech.edu.cn

1. Introduction

1.1. Background

The research of path problem can be divided into two directions: vehicle routing problem (VRP) with point as service object and arc routing problem (ARP) with arc as service object. Unlike the former, the basic feature of ARP is that the fleet starts from a warehouse and works on all the edges that need to be serviced, rather than serving at the vertex. Arc path problems can be roughly divided into three categories: China Post Path Problems, Rural Post Path Problems and The Capacitated Arc Routing Problem. Since Golden and Wong (1981) proposed the Capacity Constrained Arc Routing Problem (CARP), CARP has been widely used in daily life, especially in municipal services, such as road sprinkler path planning, garbage recycling vehicle path planning, road deicing vehicle path planning and school bus pickup path planning.

1.2. Problem Formalization

CARP can be described as follows: consider an undirected connected graph $G = (V, E)$, with a vertex set V and 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 (if it is a task). 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: a) Each route must start and end at v_0; b) The total demand serviced on each route must not exceed Q; c) Each task must be served exactly once (but the corresponding edge can be traversed more than once).

Thus, we have a solution to CARP as:

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

m is the number of routes (vehicles). The kth route $R_k = (0, \tau_{k1}, \tau_{k2}, \cdots, \tau_{kl_k}, 0)$, where τ_{kt} and l_k denote the tth task and the number of tasks served in R_k, and 0 denotes a dummy task which is used to separate different routes. The cost and the demand of the dummy task are both 0 and its two endpoints are both v_0 (the depot).

Moreover, since each task here is an undirected edge and it can be served from either direction, so each task in R_k must be specified from which direction it will be served. Specifically, $\tau_{kt} = (head(\tau_{kt}, tail(\tau_{kt}))$, where $head(\tau_{kt})$ and $tail(\tau_{kt})$ represent the endpoints of τ_{kt}, and τ_{kt} is served from $head(\tau_{kt})$ to $tail(\tau_{kt})$.

2. Methodology

2.1. Notations

2.1.1. Notation.

- N = 9999, a constraint factor used in program;
- NAME: <string>, the name of the instance;
- VERTICES : <number>, the number of vertices;
- DEPOT : <number>, the depot vertex;
- REQUIRED EDGES : <number>, the number of required edges (tasks);
- NON-REQUIRED EDGES : <number>, the number of non-required edges;
- VEHICLES : <number>, the number of vehicles;
- CAPACITY : <number>, the vehicle capacity;
- TOTAL COST OF REQUIRED EDGES : <number>, the total cost of all tasks;
- TIME: <number>, maximum time that can be used;
- SEED: <number>, random seed given;

2.1.2. Data Structure.

- map: <3D array>, a three-dimensional array which stores all edges, cost and arcs demand
- shortestPath:<2D array>, a two-dimensional array which stores the shortest path between any two vertices;

2.2. Simple Path-scanning

Firstly we implement a simple way to get the solution by repeating to add the path with the shortest distance to the end of current path, not yet serviced and compatible with vehicle capacity. This method doesn't rely on any random factors, and need not to do any complex searches, so the time consuming is short. The whole complexity is $O(n^2)$, costing by Dijkstra to find the shortest path between each pair of vertices.

2.3. Random Path-scanning

To reach a better solution, we try to generate many solutions by Random Path-scanning (RPS) and select the least-cost solution when meeting the time limit. To avoid falling into local optimum, we used two strategies when generating solutions. The first is randomly choosing a task as the first step in a route. And the second is when the vehicle pass the depot, discharge immediately which can be achieved by calculating the shortest path.

The *better* strategy in path-scanning algorithm is also aim to increased randomness and avoid falling into local optimum.

Algorithm 1: Random Path-scanning

Input: map
Output: solution$_list$
$k \leftarrow -1$
copy all required arcs in a list $free$
$Route \leftarrow []$
while $free$ *is not empty* **do**
  $k \leftarrow k+1$
  repeat
    $r \leftarrow random(0,1)$
    if *this is the first step and r <0.5* **then**
      randomly choose a task as the first task to serve
    end
    else
      choose the closest task $arcnest$ in $free$ which will not over the CAPACITY
      if *There are two or more tasks have the shortest distance* **then**
        use $better()$ strategy to choose the better $acrnext$
      end
    end
    $Route$[k] append $arcnext$
    pop $arcnext$ from $free$
  until *there are no tasks satisfy the capacity or the shortest route from* $arcnext$ *and the current position pass the* $depot4$;
end

2.4. Argument-merge

Compared with $Clarke$ and $Wright$ (CWH) for the CARP [1], a preliminary phase called $Augment$ is added. The initial routes are sorted in non-increasing order of costs. Recall that we represent a route as a list of required arcs. Starting from the longest route, each route $R_k = ((i,j)), k = 1,2,\cdots,t1$, is compared with each shortest route $R_p, p = k+1, k+2, \cdots, t$ such that the sum of their loads fits vehicle capacity. If the unique edge serviced by R_p lies on SP_{1i} or SPj_1, it can be transferred in R_k and R_p can be replaced by an empty trip. The cost of R_k does not change, but a saving equal to the cost of R_p is incurred. The Augment phase can strongly reduce the total cost and the number of routes before starting the Merge phase.

Algorithm 2: better

Input: map
Output: solution$_list$
$k \leftarrow -1$
copy all required arcs in a list $free$
$Route \leftarrow []$
while $free$ *is not empty* **do**
  $k \leftarrow k+1$
  repeat
    $r \leftarrow random(0,1)$
    if *this is the first step and r <0.5* **then**
      randomly choose a task as the first task to serve
    end
    else
      choose the closest task $arcnest$ in $free$ which will not over the CAPACITY
      if *There are two or more tasks have the shortest distance* **then**
        use $better()$ strategy to choose the better $acrnext$
      end
    end
    $Route$[k] append $arcnext$
    pop $arcnext$ from $free$
  until *there are no tasks satisfy the capacity or the shortest route from* $arcnext$ *and the current position pass the* $depot4$;
end

This nontrivial heuristic is detailed in Algorithm 7.3, where F_k and L_k denote the first and last node of route R_k. The Augment phase can be implemented in $O(nt^2)$, while the Merge phase is dominated by the sort line 24 in $O(t^2logt)$. The whole complexity is then $O(t^2(n+logt))$, or $O(m^2n)$ if all edges are required.

2.5. Simple MEANS

For getting a better solution in a limited time, we used the *memetic algorithm* (MA) introduced in MAENS[2]. In general, MA can converge to high-quality solutions more efficiently than their conventional evolutionary counterparts. MA is an outstanding algorithm which can help the solution jump out of the local optimum. Offsprings also can inherit good genes from their fathers. It makes it easier for solutions to evolve in a better direction.

We implement a simple MEANS in this project. At each iteration of MAENS, crossover is implemented by applying the sequence based crossover (SBX) operator to two parent individuals randomly selected from the current population. Each pair of parent individuals leads to a single offspring individual. [1]

Actually, the most difficult part is how to combine two genes from father and mother to create a new route and insert it to the original solution without any faults. 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, pop tasks randomly until R1 is feasible. Then replace the original route in the original solution with R1 to create a new immature solution Son. Check Son, there probably are some tasks not in Son, put them in a set $lack$.

For each task in $lack$, try to insert it in any route of Son to check weather it is feasible after insertion. If so, insert it at the best position in that route and then try next task in $lack$.(best: least deadheading cost) When $lack$ is empty, it means Son is a new feasible solution and can be returned.

Algorithm 3: MEANS

Input: A CARP instance, psize, opsize, ubtrial
Output: A feasible solution S_{bf}

//Initialization Set the current population $pop = \emptyset$
while $|pop| < psize$ **do**
  Set the trail counter ntrial = 0
  repeat
    Generate an initial solution S_{init}
    $ntrial \leftarrow ntrial + 1$
  until S_{init} *is not a clone of any solution* $S \in pop$ *or* $ntrial = ubtrial$;
  if S_{init} *is a clone of some* $S \in pop$ **then**
    break
  end
  $pop \leftarrow pop \cup S_{init}$
end
$psize = |pop|$
// Main Loop: stopping criterion is not met Set an intermediate population $pop_t = pop$
for *i = 1* $\rightarrow$ $opsize$ **do**
  Randomly select two different solutions $S1$ *and* $S2$ *as the parents from pop*
  Apply the crossover operator to $S1$ *and* $S2$ *to generate* S_x
  Sample a random number r from the uniform distribution between 0 and 1
  if S_x *is not a clone of any* $S \in pop$ **then**
    $pop_t = pop_t \cup S_x$
  end
  Sort the solutions in pop_t *using stochastic ranking*
  Set pop = the best $psize$ *solutions in* pop_t
end
return *the best feasible solution* S_{bf} *in pop*

3. Experiments

3.1. Hardware and Software

CPU: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30GHz
GPU: NVIDIA GeForce MX350
Python: 3.9.12
Numpy: 1.23.2

TABLE 1. SUMMARY OF THE PARAMETERS OF MAENS

Name	Description	Value
$psize$	Population size	30
$ubtrail$	Maximum trail for generating initial solution	50
$opsize$	No. of offspring generated in each generation	6*psize
G_m	Maximum number of generations	500

3.2. Test Dataset

We select a few from gdb, val and egl BENCHMARK TEST SET and run the program on those tests.

Throughout the experiments, MAENS adopted the same parameters. The algorithm was terminated when the time limit was reached (300 seconds). Table I summarizes the parameter settings of MAENS used in the experiments. All the experiments were conducted for 30 independent runs, and the best and average results obtained are reported in this paper. Table II demonstrates the result by these four method in terms of costs of solutions.

4. Analysis

4.1. Analysis

We implement four methods and test for its efficiency respectively.

As we can see, the algorithm with random selecting ideas (RPS and $MAENS$) perform better than $SimplePS$ and $Argument - merge$. For small graphs, $MAENS$ and RPS can both get the optimal solutions, and for the biggest graphs ($egls1A$ and $egl - e1 - A$), $MAENS$ and RPS can get a relatively good solution, while the other method perform worse.

The disadvantages of my program is that it runs slowly on small graphs and for big datasets, it can give a solution that does not deviate much from the optimal solution if you give him enough time.

5. Conclusion

5.1. Problem

In the process of completing the project, I did encounter many problems. There were problems with python programming, such as how to write code elegantly. How to cleverly design data structures that make the program easy to manage and read. I would also have to consider how to jump out of local optimal. The version management is also a big problem because I realised many method to have a comparison. This would be a huge undertaking. However, the result is relatively good.

References

[1] G. CLARKE AND J.W. WRIGHT, Scheduling of vehicles from a central depot to a num-ber of delivery points, Operations Research, 12 (1964), pp. 568–581

TABLE 2. RESULTS ON SET OF THE BENCHMARK SET OF BEULLENS ET AL. IN TERMS OF COSTS OF SOLUTIONS.

Methods	**egl-e1-A**	**egl-s1-A**	**gdb1**	**gdb10**	**val1A**	**val4A**	**val7A**
$Path - scanning$	4201	6446	370	309	212	504	370
$ImprovedPath - scanning$	3774	5608	316	275	173	410	290
$Argument - merge$	5060	7076	395	339	197	492	352
$MEANS$	3589	5272	316	275	173	418	292
$Optimal$	3548	5018	316	275	173	400	273

[2] 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.