Carp Report

Yifei Xu 11611209
Southern University of Science and Technology

1. Preliminaries

The capacitated arc routing problem (CARP) has attracted much attention during the last few years due to its wide applications in real life. Since CARP is NP-hard and exact methods are only applicable to small instances, heuristic and metaheuristic methods are widely adopted when solving CARP.

1.1 Introduction[1]

The capacitated arc routing problem (CARP), which is the most typical form of the arc routing problem, is considered in this paper. It 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 (i.e., directed edges) denoted by A, is given. There is a central depot vertex dep ∈ V , where a set of vehicles are based. A subset ER ⊆ E composed of all the edges required to be served and a subset AR ⊆ A composed of all the arcs required to be served are also given. The elements of these two subsets are called edge tasks and arc tasks, respectively. Each edge or arc in the graph is associated with a demand, a serving cost, and a deadheading cost (the cost of a vehicle traveling along the edge/arc without serving it). 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 subject to the following constraints:

1) each route starts and ends at the depot;
2) each task is served in exactly one route;
3) the total demand of each route must not exceed the vehicle's capacity Q.

In my project, I try to simply recur ***MEANS***[1] created by Xin.Y, Ke.T and Mei.Y

1.2 Application

There are various applications of the carp problem such as Chinese postman problem. In real life, garbage companies can use this to find an optimal way to schedule their garbage trucks.

2. Methodology

This part is to introduce the details of algorithm and the architecture of my program.

2.1 Notation

- *Q* : the capacity of a vehicle
- *V* : vertices of the graph
- *E* : edges of the graph
- *S* : solutions of this problem
- *u* : an edge
- *beg(u)* : the beginning of edge u
- *end(u)* : the end of edge u
- *Shortest(a, b)* : the cost of the shortest path from a to b
- c_u : the cost of edge u
- q_u : the load of edge u
- *load(k)* : the total load of vehicle k
- *cost(k)* : the total cost of vehicle k
- *psize* : population size
- *ubtrial* : maximum trials for generating initial solutions
- *opsize* : # of offsprings generated in each generation
- *pls* : probability of carrying out local search (mutation)

2.2 Data Structure

- *Worker*
  Pipeline processors
- *Problem*
  Store the information of a carp dataset.
  e.g. The shortest path from a to b.
- *Solution*
  Store the information of a solution.
  e.g. The routes of a solution.
- *Route*
  Store the information of a route.
  e.g. The demand of a specific route.
- *Edge*
  Store the information of an edge.
  e.g. The end point of this edge.

2.3 Model Design

As following are the steps of solving a carp problem.

1. Read and resolve the data in datasets.
2. Generate several initial solutions by path scanning[2].
3. Select two solutions S_1 and S_2 randomly and apply the crossover operator(the sequence based crossover) to them to get an offspring S_x.
4. Decide whether apply local search(single insertion, double insertion and swap) to S_x to generate S_{ls}.
5. Resort solutions and keep the top *psize* solutions.
6. Repeat step 3,4,5 until stopping criterion is met.
7. Return the first solution.

2.4 Details of Algorithms

Path_Scanning : This method is to generate an initial solution of a carp problem. I use random-select rule to determine which edge the vehicle should take if there is/are the closest required edges. Otherwise, the vehicle should go back to the depot.

Input: *G :* graph
Output: *S :* solution

$k \leftarrow 0$ # the ID number of vehicles
copy all required arcs(both directions) in a list *free*
repeat
  $k \leftarrow k + 1$
  $R_k \leftarrow \varnothing$
  $load(k), cost(k) \leftarrow \varnothing$
  $i \leftarrow 1$
  repeat
    $d_{min} \leftarrow Infinite$
    $U \leftarrow \varnothing$
    $U' \leftarrow \varnothing$
    for each u $\in$ *free* **do**
      if $shortest(i, beg(u)) < d$ **then**
        $d \leftarrow shortest(i, beg(u))$
        $U \leftarrow \{u\}$
      else if ($shortest(i, beg(u)) = d_{min}$) **then**
        $U \leftarrow U \cup u$
    for each $u \in U$ **do**
      if $load(k) + q_u \leq Q$ **then**
        $U' \leftarrow U' \cup u$
    if $U' = \varnothing$ **then**
      break
    else
      select an edge u from U' randomly
      add edge u at the end of route R_k
      remove arc u and its opposite u' from *free*
      $load(k) \leftarrow load(k) + q_u$
      $cost(k) \leftarrow cost(k) + d + c_u$
      $i \leftarrow end(u)$
  until *free* = $\varnothing$ or d = **Infinite**
  $cost(k) \leftarrow cost(k) + shortest(i, 1)$
until *free* = $\varnothing$
return $\sum R_i$ and their cost and load

---

local_search **:** This method is to find the best local solution using 3 operators(single insertion, double insertion, swap).

Input: S_x *:* solution
Output: *S :* solution

S_1 = Apply single insertion to S_x
S_2 = Apply double insertion to S_x
S_3 = Apply swap to S_x
return the best solution of $\{S_1, S_2, S_3\}$

---

crossover : This method is to enlarge the population set of solutions and create more choices.

Input: S_1 *:* solution, S_2 *:* solution
Output: S_x *:* offspring solution

Select R_1 and R_2 randomly from S_1 and S_2.
Divide R_1 and $R2$ into (R_{11}, $R1_2$) and (R_{21}, R_{22}) randomly
$R_3 \leftarrow (R_{11}, R_{22})$
// Recombine a new routes of the offspring
$Routes \leftarrow$ {routes of S_1}$\cup$\{R_3\} - \{R_1\}
$U_{lack} \leftarrow$ lacked required edges in *Routes*
$U_{duplicate} \leftarrow$ redundant required edges in *Routes*
// Remove duplicated edges
$Routes \leftarrow Routes - U_{duplicate}$
for each arc u in U_{lack} **do**
  Insert u in the position where causes the lowest extra cost of the route in *Routes*.
return S_x with *Routes*

Means[1] : This method is to find the best solution of a carp problem based on the steps I mentioned in **2.3 Model Design** and also in the essay[1] in the reference**.**

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

Set the current population $pop = \varnothing$;
while | *pop*| < *psize* **do**
Set the trial counter *ntrial* = 0;
repeat
  Apply ***path_scanning*** to generate an initial solution *Sinit*;
  $ntrial \leftarrow ntrial + 1;$
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
$pop \leftarrow pop \cup S_{init};$
$psize = |pop|;$
// Main Loop:
while stopping criterion is not met **do**
  Set an intermediate population $popt = pop$
  for $i = 1 \rightarrow opsize$ **do**
    Randomly select two different solutions S_1 and S_2 as the parents from *pop*; Apply ***crossover*** to S_1 and S_2 to generate S_x
    Sample a random number *r* from the uniform distribution between 0 and 1;
    if $r < Pls$ **then**
      Apply ***local_search*** to S_x to generate S_{ls};
      if S_{ls} is not a clone of any $S \in popt$ **then**
        $popt = popt \cup S_{ls}$
      else if S_x is not a clone of any $S \in popt$ **then**
        $popt = popt \cup S_x$
    else if S_x is not a clone of any $S \in popt$ **then**
      $popt = popt \cup S_x$
  Sort the solutions in *popt* using according to the total cost
  Set pop = {the best *psize* solutions in *popt* };
return the best feasible solution S_{bf} in *pop*

3. Empirical Verification

This part is to introduce the performance of this algorithm on different datasets.

3.1 Dataset

- val datasets
- gdb datasets
- egl datasets

3.2 Performance Measure

I. **Performance**

Fig.1 is the performance on provided datasets.
Fig.2 is the performance on other datasets.

Dataset	Average Cost	Best Cost	The cost of the 1st	Running time (s)
val1A	173	173	173	60
val4A	407	402	400	60
va7A	279	279	279	60
gdb1	316	316	316	60
gdb10	275	275	275	60
egl-e1-A	3707	3618	3548	120
egl-s1-A	5450	5336	5018	120

Fig.1 Performance on provided datasets

Dataset	Average Cost	Best Cost	Running time (s)
gdb2	345	345	60
gdb3	275	275	60
val5A	428	427	60
egl-e2-A	5170	5138	120
egl-e3-A	6190	6160	120

Fig.2 Performance on different datasets

II. **Test Environment**

CPU:	Intel Core i7
Frequency:	2 GHz
# Cpu:	1
# Core:	4
Python version:	3.7

3.3 Hyperparameters

Hyperparameter	Value
psize	2000/50
ubtrial	30
opsize	20
pls	0.2

Firstly I set psize as 2000 to generate 2000 initial solutions as this step is random. Then choose the top 50 solutions in solutions set. As for opsize, if it is too large, it will cause a low running speed. According to practice and Means[1] essay, 20 is suitable.

3.4 Experimental

From Fig.1 and Fig.2, for the datasets gdb, this algorithm can find the best solution in most cases. But for datasets egl and val, the algorithm is able to find relatively good solutions but the best solutions.

3.5 Conclusion

Advantages : The process of generating initial solutions is fast and random rule in path-scanning is much better than the other five rules.

Disadvantages : Because of the limited time, I have not implemented MS operator and only use local search. As a result, the solution may be trapped in the local best. So it is necessary to implement large step searching.

4. References

[1] 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.
[2] B. L. Golden, J. S. DeArmon, and E. K. Baker, “Computational experiments with algorithms for a class of routing problems,” Comput. Oper. Res., vol. 10, no. 1, pp. 47–59, 1983.