CARP Solving Project Report

Passing Scanning and Evolutionary Computation

Ou Yulin

May, 2022

Abstract

This project is to use pass-scanning and evolutionary-computation to solve CARP. In the implementation, we first do pass scanning to generate the base case, then run evolutionary computation to optimize the result.

1. Introduction

The capacitated arc routing problem (CARP) is one of the most typical form of the arc routing problem. CARP has many practical applications in our real world, such as Urban road planning, garbage disposal and express delivery. Because CARP is an NP-hard problem, this project try to find a relatively better solution in limited time using local-search. The method in this project mainly refers to MAENS[1]. In the implementation, pass-scanning is used to generate the base solution, and evolutionary-computation is used to optimize it. In the part of evolutionary-computation, four kinds of traditional operators are used to do local search, and merge-split operator is used when the searching process is stuck in local optimums.

2. Preliminary

Here is the description of CARP. Given a graph $G = (V, E)$, where V denotes all the vertices of the graph, E denotes all the edges of the graph. There exists a depot vertex $depot \in V$, where a set of vehicles are based. All the edges are associated with a passing cost, denoting the cost for vehicles passing this edge. A subset $E_t \subseteq E$ composed of all the edges required to be served. All the edges are associated with a serving demand, denoting the capacity taken for vehicles to serve this edge, and for edges $e \notin E_t$, the serving demand of them should be 0. Each vehicle has the same capacity c, they travel from the depot and go back to the depot at the end, and serve the demanding edges during the travel. One demanding edge must be entirely served in one service, so a vehicle can only serve a demand edge whose demand is not exceeding the current capacity of the vehicle.

A solution for CARP is the a set of routes for the vehicles, the each vehicle goes along a route and serves the selected demanding edges. Here we represent the routes as edge-task sequences $SeqList$. In one Seq, the demanding edges are listed in order, a vehicle starts from the depot, picking the shortest path to the start of the first demanding edge, going along the edge and serves it, then picking the shortest path from the end of this edge to the start of the next edge, iterating for all the edges in the sequences, and finally picking the shortest path from the end of the last edge to the depot. The shortest paths can be determined from the distance matrix Dis. Dis can be generated by running $Floyd$ at the beginning. So for $Seq_i \in SeqList$, the cost of it is:

$$Cost(Seq_i) = Cost_{from\ and\ to\ depot} + \sum_{edges\ in\ Seq_i} (Dis[to(e_j)][from(e_{j+1})] + Cost(e_j))$$

So the total cost the *SeqList* should be:

$$Cost(SeqList) = \sum Seq_i$$

In CARP, we want the total cost of the *SeqList* to be as smaller as possible, so we reach the representation of CARP:

$$\min Cost(SeqList)\ under\ s.t.$$

$$s.t.:$$

$$\forall\ Seq_i \in SeqList,\ Load(Seq_i) \leq c$$

$$\bigcup_{Seq\ in\ SeqList} Seq_i = E_t$$

3. Methodology

3.1. Top Function

In this project, pass-scanning is used to generate the base solution, and evolutionary-computation is used to optimize it, so the top function goes like algorithm 1.

Algorithm 1: Top-function

input : all the information of the problem $information$, including depot $root$, capacity c, edges to be cleaned $cleanList$, adjacency matrix of the graph $matrix$
output: the solution $SeqList$, the cost of the sequences $totalCost$

```
1 SeqList, totalCost ← Pass-scanning(information);
2 SeqList, totalCost ← Evolutionary-computation(SeqList, information);
3 return SeqList, totalCost
```

3.2. Pass Scanning

In pass-scanning, we first do *Floyd* to generate the shortest path matrix *distance*. In the body part of the algorithm, we iteratively build sequences from the set of demanding edges *CleanList*, until all the demanding edges are put in one sequence. During the building process of one sequence, we apply *FindEdge* function on *CleanList* to get the most suitable edge-task to do next.

In *FindEdge*, we traverse the *CleanList* to select a satisfying edge-task. The principles for the selection is as follow:

- The demand of the edge-task should be equal to or less than the left capacity of the vehicle;
- If multiple edge-tasks satisfy the above requirements, we select the edge-task with the shortest distance from the current spot of the vehicle to the start of the edge;
- If multiple edge-tasks share the same minimum value, we select the edge-task with the longest distance to the depot if the load of the vehicle is less than half-full, or the edge-task with the shortest distance to the depot if the load of the vehicle is more than half-full;
- If there are still multiple choices, select one edge-task randomly.

If no edge-tasks can be added to the sequence, return *None*.

The pass-scanning algorithm can give a basic solution of CARP, and it goes like algorithm 2.

Algorithm 2: Pass-scanning

input : all the information of the problem $information$, including depot *root*, capacity c, edges to be cleaned *cleanList*, adjacency matrix of the graph *matrix*

output: the solution *SeqList*, the cost of the sequences *totalCost*

```
distance ← Floyd(matrix);
initialize SeqList as an empty list;
while cleanList is not empty do
    initialize Seq as an empty list;
    while True do
        edge ← FindEdge(information, current stage);
        if edge is None then break;
        else append edge to Seq and remove edge from cleanList;
    end
    append Seq to SeqList;
end
return SeqList and total cost of SeqList
```

3.3. Evolutionary Computation

In evolutionary-computation, every time we apply the local search to all the solutions in the population to optimize them independently, then evaluate the solutions to select the best solutions as *BestPop*, and use them to generate the next population.

To start with the algorithm, we initialize *BestPop* by filling it with the copies of the *SeqList*. When generating the population, as the format of the CARP is hard to do hybridization between solutions, so the population is generated by simply coping the solutions in *BestPop*.

Then, we apply *Optimize* to every solution in the population. In the function *Optimize*, we use local-search operators on the given solution for certain times to find the better solution. The four operators are as follow:

- *Single insertion*
  Select an edge-task from one sequence, and insert it into another spot in one sequence or a new vehicle sequence with the consideration of the direction of the edge-task.
- *Double insertion*
  Instead of selecting one edge-task, we select two contiguous edge-tasks from one sequence, and insert it just the way we do in single insertion.
- *Swap*
  Exchange the positions of two edge-tasks in the same sequence or in different sequences. Direction reversing may needed for swapping.
- $2-opt$
  Select a sub-sequence from a sequence, and reverse the direction of the sub-sequence.

There is another local-search operator implemented in this project called $Merge-Split$. In this operator, we select multiple sequences and break them down to edge-tasks, collecting these edge-tasks to

form a *CleanList*. Then, we apply pass-scanning to this *CleanList*, and output a sub-*SeqList*. We then merge the original and the sub-*SeqList* to get the new solution. Different from the above four operators which take small steps in the solution space, $Merge - Split$ makes a big difference to the solution, and take a large step in the solution space. Such an operation is very likely to generate a solution with much higher cost. $Merge - Split$ also runs much slower than other operators, as it needs to run pass-scanning. If we treat it as common operators, the solutions generated by it can be easily abandoned by the population very quickly, and it also leads to worse efficiency for *Optimize*. For these reasons, we use it as a disturbance in this project. When the best solution remains unchanged for several iterations, we apply $Merge - Split$ to all the solutions in *BestPop* to get out from the local optimum.

The terminating criterion is the time limit. At the end of the algorithm, we return the best solution in our whole searching history. So our evolutionary-computation goes like algorithm 3:

Algorithm 3: Evolutionary-computation

input : the sequence list generated from pass-scanning *SeqList*, all the information of the problem $information$, including depot *root*, capacity *c*, edges to be cleaned *cleanList*, adjacency matrix of the graph *matrix*

output: the solution *SeqList*, the cost of the sequences *totalCost*

```
initialize BestPop as an empty list;
fill BestPop with copies of SeqList to the expected size;
while terminating criterion is not met do
    generate Population by BestPop;
    apply Optimize(solution, information) to all the solutions in Population;
    evaluate all the solutions in Population and select the top expected-size solutions to be
     BestPop;
    if the best solution in BestPop remains unchanged for certain iterations then
        save the best solution;
        apply MergeSplit(solution, information) for all solutions in BestPop;
    end
end
return the best solution and the total cost of it
```

4. Experiments

In the experiments, five different methods are tested on the Online-Judge, and here we show the results.

	PS	**Sim-Opt**	**Full-Opt**	**EC**	**EC with MS**
egl-e1-A.dat	4188	4163	3965	3965	3808
egl-s1-A.dat	6446	6411	5553	5454	5362
gdb1.dat	370	370	348	348	323
gdb10.dat	309	307	289	289	289
val1A.dat	212	202	185	185	182
val4A.dat	349	342	313	312	309
val7A.dat	506	499	442	434	434

Table 1: Testing result from OJ

The above five methods are described below:

- PS
  Only pass-scanning is used, can be seen as the base line.
- $Sim - Opt$
  Run *Optimaze* on a single solution, and local-search operators every time will only make changes inside one sequence. Used to test the solution of pass-scanning. The result is very similar to pass-scanning, denoting that the way pass-scanning lining up edge-tasks inside every sequence is quite good.
- $Full - Opt$
  Run *Optimaze* on a single solution, but local-search operators can work on multiple sequences in one time. It makes a great difference to pass-scanning, denoting that the way pass-scanning group up edge-tasks into sequences can be optimized. $Full - Opt$ is very likely to be trapped into local optimum.
- EC
  Use evolutionary-computation to do local-search operators. The solution does not improve a lot. The local optimum traps are still hard to come across.
- $EC\ with\ MS$
  Add $Merge - Split$ to adjust the solutions. It gives out the best solution.

From the comparison of the above results, we can see that $Merge - Split$ does bring optimization to the solution. In my local tests, $EC\ with\ MS$ can come to around 5250 on egl-s1-A.dat when running for several more minutes. But the process is time taking, as everytime $Merge - Split$ is applied, we have to spend a few iterations to decrease the costs. So the efficiency of the algorithm can be improved. Another problem is that CARP with smaller graph is hard to optimize using the operators in my implementation, so they actually need other ways of local-search.

I also try hyper-parameter adjusting in $EC\ with\ MS$. For example, if the size of $BestPop$ is small, the cost will drop rapidly at the beginning, but soon be trapped in local optimums, and it is hard for they to get out. If the size of $BestPop$ becomes bigger, the cost will drop slower at the beginning, but less likely to be trapped. The same phenomenon happens on the times *Optimize* runs. If *Optimize* runs for too many times in one iterations, though the costs may drop fast at the beginning, it is more likely to be trapped. But less times of running lead to slower dropping, which is quite annoying with $Merge - Split$. So there are many trade-offs in the adjusting of hyper-parameters, and a good choice can lead to better performance.

5. Conclusion

In this project, I use local-search to solve the NP-hard problem CARP. I learnt the skills to use operators doing local-search, built the process of evolutionary-computation, and tried some methods to avoid the traps of local optimums. I also tried adjusting the hyper-parameters of the local-search, and saw how those hyper-parameters affect the running of the local-search. I think I obtained a better understanding of the theoretical knowledge in class by working on this project.

References

[1]K. Tang, Y. Mei, and X. Yao, ≪Memetic algorithm with extended neighborhood search for capacitated arc routing problems≫, IEEE Transactions on Evolutionary Computation **13**, 1151–1166 (2009) 10.1109/TEVC.2009.2023449.