Heuristic solution of Capacitated Arc Routing Problem

Yi Xiang 11912013
Computer Science and Engineering
Southern University of Science and Technology
11912013@mail.sustech.edu.cn

Contents

1. Preliminary

1.1. Problem Description

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 $c(e)$ when ever a vehicle travels over it or serves it (if it is a task). Each required edge $\tau \in T$ 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. However, the corresponding edge can be traversed more than once.[1] CARP is a NP-hard problem. It was first developed by Golden (1981). A large amount of literature is currently based on the study of this problem.

1.2. Problem Applications

The CARP problem has applications in many places, such as municipal garbage collection, winter road snow removal, road sprinkling, street sweeping, transmission line detection, etc. [2] The solution of CARP problem is of great significance to urban planning problems, and also promotes the development of heuristic solution of constrained problems.

2. Methodology

2.1. Notation

The main notations used are list as the following tables.

Symbol	Meaning
V	The vertices set in the graph.
D	The vertex represented by the depot.
Q	Vehicle capacity.
C	The total demand for all tasks.
E	The edges set in the graph.
T	The time limit for the problem.
S	The random seed used in the problem.

TABLE 1. MAIN SYMBOLS THAT APPEAR

2.2. Data Structure

There are not many complex data structures are applied. The *graph* is a 2-dim matrix which specfic the min-distance for each two vertex in the graph. The *edge* is a Map which mapping the graph's edges to their demand and cost. This may be easy to calculate the demand and cost of a path. The *solution* is a list for a solution. The first dimension is for the routes, and the second dimension is for the edges which contain the order for serving the tasks.

2.3. Model Design

First, I read and process the parameters passed in from the command line, resolve the data address, read the time

Function	Meaning
getSample()	Return the instance reading from the data file.
createSolution()	Return the initial solution for the subsequent calculations.
calDistance(instance)	Generate the graph from the instance, and then calculate the min-distance for each two vertices.
show_result(solution, cost)	Print the answer into the console according to the standard format provided.
work()	The body that executes all functions. Do not return anything.
localSearch(state)	According to the existing status, performing local-search to obtain the new status. Return the new status.
swap(state)	Performing the swap operation to obtain the new state.
flip(state)	Performing the flip operation to obtain the new state.
insert(state)	Performing the single-insert operation to obation the new state.
opt(state)	Performing the 2-opt operation to obation the new state.
checkValid(solution)	Check whether the solution is valid. Return a boolean.
calCost(solution)	Calculate the cost for the solution. Return an integer.

TABLE 2. ALL FUNCTIONS THAT APPEAR

limit and start timing, and obtain and apply the seed of random numbers. I pass the input data and obtain the key information for the problems like V, D, Q, C, E as *vertices, depot, capacity, total demand* and *edges*. All the subsequent calculations will be based on the above information.

Then, I apply a randomly path scanning and a greedy path scanning to obtain an initial solution for the heuristic solution. The randomly path scanning is to choose the next task completely randomly, and the greedy approach is to choose the closest task. The latter one may find a nice solution actually, in a very short time. However, it is not enough. So we may apply a heuristic algorithm.

In this part, I use simulated annealing algorithm to get better results. Each round I took one of **flip, swap, insert, or 2-opt** to generate a new state based on the current state. As for which method to take, it is a matter of random probability to choose. The selection of hyperparameters will be further explained in the following chapters. In each new state I will calculate the **cost** directly, and compare it to the cost of the old state. If it is smaller than the old state, which means that it may be an better solution, then accept it. If is bigger than the old state, this accepts the new state with some probability $e^{\frac{\Delta cost}{T}}$, since it may lead me to a better solution in the following loops.

It is worth noting that a series of illegal solutions may be generated during each state transition (**an invalid solution means that its capacity may overflow, or other conflicts with the problem constraints**). Such a solution would not get the grade, although its cost might be very low. However, it also plays an important role in the simulated annealing algorithm, because it is also possible to transform into a very good solution through a series of changes. Therefore, I will keep this illegal solution as part of the current state of **Local Search**, and how to use it will be shown in the algorithm details section in the next section.

2.4. Detail of Algorithms

For more detail of the algorithms, I may start from the algorithm for finding the initial solution.

Input: Instance
Output: The initial solution

```
1:  function CREATESOLUTION
2:      free ←instance.edges where edges.demand ≠ 0
3:      routes ← an empty list
4:      pointer ← instance.depot
5:      cost ← 0
6:      while free is not empty do
7:          cap ← 0
8:          route ← an empty list
9:          while True do
10:             if free is empty then
11:                 break
12:                 tmp ← free
13:                 sort(tmp) according to the distance between the edge and the pointer
14:                 flag ← False
15:                 for edge in tmp do
16:                     if The edge(task) can be accept then
17:                         flag ← True
18:                         route ← edge
19:                         remove edge from free
20:                         update pointer ← edge.end
21:                     end if
22:                 end for
23:             end if
24:             if flag is False then
25:                 break
26:             end if
27:         end while
28:         routes.add(route)
29:         update pointer ← instance.depot
30:     end while
31:     return routes
32: end function
```

Algorithm 1 The algorithm to get the initial solution

More specifically, if you use a random algorithm, you don't need to sort line 13 and just randomly select the next edge in the remaining task. Since this method is not used in the final submission, it is not covered in the report.

After processing the initial solution, I will try to obtain a better solution based on the initial solution. Here I use the simulated annealing algorithm, the details of the algorithm will be shown below.

Input: The initial solution
Output: The better solution

```
1: function LOCALSEARCH(state)
2:     t ← 0
3:     while True do
4:         T ← 0.99^t
5:         new_state, new_cost = localSearch(state)
6:         if checkValid(new_state) then
7:             if The current state is not valid then
8:                 if The new state is better then
9:                     accept the new state
10:                    state ← new_state
11:                end if
12:            end if
13:        else
14:            temp_state ← state
15:        end if
16:        if cost - new_cost ≥ 0 or random() ≤ e^((cost−new_cost)/T) then
17:            state ← new_state
18:        end if
19:        if time limit exceed then
20:            break
21:        end if
22:    end while
23:    if checkValid(state) then
24:        return state
25:    else
26:        return temp_state
27:    end if
28: end function
```

Algorithm 2 The algorithm of Simulated annealing

Specifically, the function *random()* on line 16 will randomly generate a floating point number in the range $(0, 1)$. This will represents accepting a new solution with a certain probability. In lines 6 to 15, there is a process of selecting an invalid solution, and sometimes the invalid solution can be transformed into a good calid solution, so I will keep it and replace it when appropriate. In lines 23 through 26, I make a judgment call to make sure that the final result is valid.

Then is the core function *localSearch()*. Since it is divided into four parts, I will describe the principle of each part in the pseudo-code of the four functions. These four pseudocodes represent four operators, **swap, flip, single-insert and 2-opt.** In particular, the pseudocode function *choice()* below means a random selection from the list in the argument. The function *randint(i, j)* means an random integer in range $[i, j)$.

The **swap** operator is to randomly choose two edges from two routes in the solution, and then exchanges them. After that we can get a new state from this state.

Input: The current state
Output: The new state

```
1: function SWAP(state)
2:     route1 ← choice(state)
3:     route2 ← choice(state)
4:     edge1 ← choice(route1)
5:     edge2 ← choice(route2)
6:     edge1, edge2 ← edge2, edge1
7:     new_state ← updated routes
8:     return new_state
9: end function
```

Algorithm 3 The algorithm of swap

The **flip** operator is to randomly choose an edge from a route in the solution. Then reverse the edge. For example, the edge (1, 0) will becomes (0, 1) after flip operator.

Input: The current state
Output: The new state

```
1: function FLIP(state)
2:     route ← choice(state)
3:     i ← randint(0, |route|)
4:     edge ← route[i]
5:     new_edge ← (edge[1], edge[0])
6:     new_state ← updated routes
7:     return new_state
8: end function
```

Algorithm 4 The algorithm of flip

The **insert** operator is to randomly choose an edge from one route from the solution, and then randomly insert it into another route chosen randomly, at random position. After that we can obtain a new state from the old state.

Input: The current state
Output: The new state

```
1: function INSERT(state)
2:     route1 ← choice(state)
3:     route1 ← choice(state)
4:     i ← randint(0, |route1|)
5:     edge ← route[i]
6:     j ← randint(0, |route2|)
7:     insert edge into route2 at position j
8:     delete edge in route1
9:     new_state ← updated routes
10:    return new_state
11: end function
```

Algorithm 5 The algorithm of insert

The **2-opt** operator is to randomly choose a sequence of edges of one route. Then reverse them all. For example, there is a route $\{(1, 9), (3, 2), (1, 4), (6, 0)\}$. Then reverse it from position 1 to position 2. The result is $\{(1, 9), (2, 3), (4, 1), (6, 0)\}$.

Input: The current state
Output: The new state

```
1: function 2-OPT(state)
2:     route ← choice(state)
3:     left ← randint(0, |route|)
4:     right ← randint(0, |route|)
5:     assert left ≤ right
6:     for edge in route[left:right] do
7:         perform flip operator on edge
8:     end for
9:     reverse route[left:right]
```

10: new_state ← updated routes
11: **return** new_state
12: **end function**
Algorithm 6 The algorithm of 2-opt

3. Empirical Verification

3.1. Dataset

The dataset I use is from a open source data set for CARP problem. This data set is consists of four parts. **bccm**[3], **eglese**[4][5][6], **gdb**[7] **and kshs**[8]. The test data provided by the platform is part of the above data set.

The data set are all in the same format. They are composed of the following parts: the description of the file, including the data name, the number of vehicles, the number of nodes, the number of demand side, the number of non-demand side and other basic information. Next is the start and end node information of each edge and their cost and demand. The format is nearly same as the data set provided from the plantform, so it is easy to deal with them.

3.2. Performance measure

Regarding how I use data sets to evaluate the performance of my programs, I choose to directly use my programs to compute the solutions to these data sets. Given time constraints time limit and random seeds to make the test more meaningful. Consider the efficiency of the test and the requirements of the question. I choose the lower bound of the time limit for each test case in the performance measure part. That is, the time limit for each test cases is $60seconds$. And after testing, I obtain a table of the cost for each test cases. These results are listed in the appendix and compared with the exact solution. From the comprehensive results, my algorithm has achieved a good balance in efficiency, robustness and accuracy.

The following is my introduction to the experimental environment. This project is written in *Python* with editor *Pycharm*. The main testing platform is *Windows 10 Home Edition* (version 20H2) with Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz of 4 cores and 8 threads, the memory is 16GB. And the develop platform is *Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-151-generic x86_64)* with Intel(R) Xeon(R) Gold 6278C CPU @ 2.60GHz with 8 cores and 16 threads, the memory is 16GB.

Both of the *python* version are 3.9 and the *numpy* module's version is 1.20.1 .

3.3. Hyperparameters

The only hyperparameters I use in the algorithm is the different possbility to choose the four operators, swap, flip, insert and 2-opt. In the process of selecting the hyperparameters, I calculated the number of task edges affected by each operator on average. Finally, in order to make the expectation of task edges affected by each operator approximate to the same, I selected a group of hyperparameters. The calculation of this group of hyperparameters is as follows:

$$\begin{aligned} E(opt) &= acc \times n \\ E(opt_1) = E(opt_2) &= E(opt_3) = E(opt_4) \\ \sum acc &= 1 \end{aligned} \tag{1}$$

3.4. Experimental results

The experimental results are shown in the appendix. In such table we can find that in the smaller test data, my algorithm can obtain the lower bound result sometimes. However, when occur large data set. My algorithm does not work well. In all data sets, I obtain 12 times lower bound totally, and other result are close to the lower bound. Compared with existing research, there is still a big gap.

3.5. Conclusion

In this project, I mainly studied the design and solution of heuristic algorithm deeply. In the experimental results, I can see that sometimes accurate solutions can be obtained from small data sets, while in large data sets, I can only approach the exact solutions. This may be due to the fact that the acquisition of the new state is not very rational, and the situation may not have been particularly well considered. For example, I didn't add double-insert and double-2-opt operators. The advantage of heuristic solution algorithm is that it can spend less time to get a relatively reliable solution, but its disadvantage is that the solution is not the global optimal, may be only the local optimal solution. In the experiment, I learned the simulated annealing algorithm in depth. The defect in my design lies in that every time I get a new state, I have to calculate cost completely from scratch, which is very uneconomical. The direction of future improvement lies in adding more reasonable operators and optimize the process of cost calculation.

Acknowledgments

I would like to thank Mr. Zhao and Professor Yuan Bo for their wonderful classes and careful guidance. I would also like to thank the student teaching assistants for providing me with an excellent test platform.

References

[1] He Rui, Project_Carp, Southern University of Science and Technology, November 10, 2021. Accessed on: November 10, 2021. [Online]. Available: https://sakai.sustech.edu.cn/access/content/group/ee81f5ca-f01a-47f4-bbb6-ebeb1d96c0cf/Projects/Project_Carp.zip

[2] PENG Jin-huan, MA Hui-min.Review of Literature about Capacitated Arc Routing Problem,2015,38(01):63-66.DOI:10.13714/j.cnki.1002-3100.2015.01.019.

[3] E. Benavent, V. Campos, A. Corberán and E. Mota (1992). The Capacitated Chinese Postman Problem: Lower Bounds. Networks 22 (7), pp. 669-690.

[4] L.Y.O. Li (1992). Vehicle Routeing for Winter Gritting. PH. D. Thesis, Dept. of Management Science, Lancaster University.

[5] L.Y.O. Li and R.W. Eglese (1996). An Interactive Algorithm for Vehicle Routeing for Winter-Gritting. Journal of the Operational Research Society 47, pp. 217-228.

[6] J.M. Belenguer and E. Benavent (2003). A Cutting Plane Algorithm for the Capacitated Arc Routing Problem. Computers and Operations Research 30 (5), pp. 705-728.

[7] B.L. Golden, J.S. DeArmon and E.K. Baker (1983). Computational Experiments with Algorithms for a Class of Routing Problems. Computers and Operations Research 10 (1), pp. 47-59.

[8] M. Kiuchi, Y. Shinano, R. Hirabayashi and Y. Saruwatari (1995). An exact algorithm for the Capacitated Arc Routing Problem using Parallel Branch and Bound method. Abstracts of the 1995 Spring National Conference of the Oper. Res. Soc. of Japan, pp. 28-29.

[9] Y. Yong, Study in Multi-Vehicle Capacitated Arc Routing Problem (MVCARP).College of Computer Science of Chongqing University,2008. •

Appendix

The following tables are the results for the test date set performance. And the meaning of the symbols are shown below.

- data: the name of the data set
- n: the number of the vertices
- e: the number of required edges
- LB: the lower bound of the data set (The lower bound is obtain from the paper[9]).
- ps_cost: the cost calculate by the path-scanning algorithm
- cost: the cost calculate by my algorithm in 60 seconds

The performance for the gdb data set. There are 11 data's cost obtain the lower bound of the data set.

data	n	e	LB	ps_cost	cost
gdb1	**12**	**22**	**316**	**370**	**316**
gdb2	**12**	**12**	**339**	**402**	**339**
gdb3	**12**	**22**	**275**	**339**	**275**
gdb4	**11**	**19**	**287**	**350**	**287**
gdb5	13	26	377	486	387
gdb6	12	22	298	390	324
gdb7	12	22	325	368	330
gdb10	27	46	275	309	289
gdb11	**27**	**51**	**395**	**465**	**395**
gdb12	12	25	450	666	498
gdb13	22	45	536	589	558
gdb14	**13**	**23**	**100**	**123**	**100**
gdb15	**10**	**28**	**58**	**64**	**58**
gdb16	**7**	**21**	**127**	**143**	**127**
gdb17	**7**	**21**	**91**	**95**	**91**
gdb18	**8**	**28**	**164**	**192**	**164**
gdb19	**8**	**28**	**55**	**69**	**55**
gdb20	9	36	121	139	123
gdb21	8	11	156	176	158
gdb22	11	22	200	209	201
gdb23	11	33	233	248	240

TABLE 3. THE PERFORMANCE FOR GDB DATA SET.

Since I cannot find the lower bound for the kshs data set. So I just list my performance in the test.

data	n	e	ps_cost	cost
kshs1	8	15	16068	14729
kshs2	10	15	12048	9863
kshs3	6	15	12199	9524
kshs4	8	15	15248	12378
kshs5	8	15	15027	11151
kshs6	9	15	12913	10305

TABLE 4. THE PERFORMANCE FOR EGL DATA SET.

The performance for the bccm data set. There are 1 data's cost obatin the lower bound of the data set.

The performance for the eglese data set. There are no data's cost obatin the lower bound of the data set.

data	n	e	LB	ps_cost	cost
val1A	24	39	173	212	180
val1B	24	39	173	229	185
val1C	24	39	235	340	257
val2A	24	34	227	287	235
val2B	24	34	259	337	267
val2C	24	34	455	568	467
val3A	24	35	81	98	83
val3B	24	35	87	110	94
val3C	24	35	137	182	144
val4A	**41**	**69**	**400**	**504**	**400**
val4B	41	69	412	526	430
val4C	41	69	428	595	476
val4D	41	69	520	710	568
val5A	34	65	423	550	428
val5B	34	65	446	559	450
val5C	34	65	469	567	481
val5D	34	65	571	806	627
val6A	31	50	223	281	235
val6B	31	50	231	282	233
val6C	31	50	311	424	327
val7A	40	66	279	370	305
val7B	40	66	283	357	307
val7C	40	66	333	430	355
val8A	30	63	386	508	389
val8B	30	63	395	523	417
val8C	30	63	517	647	601
val9A	50	92	323	372	330
val9B	50	92	326	417	340
val9C	50	92	332	433	349
val9D	50	92	382	483	432
val10A	50	97	428	507	440
val10B	50	97	436	526	450
val10C	50	97	446	560	472
val10D	50	97	524	638	576

TABLE 5. THE PERFORMANCE FOR BCCM DATA SET.

data	n	e	LB	ps_cost	cost
egl-e1-A	77	51	3517	4201	3552
egl-e1-B	77	51	4436	3517	4671
egl-e1-C	77	51	5453	7331	6020
egl-e2-A	77	72	4994	6345	5245
egl-e2-B	77	72	6249	8221	7141
egl-e2-C	77	72	8114	11103	8806
egl-e3-A	77	87	5869	7031	6293
egl-e3-B	77	87	7646	10330	8330
egl-e3-C	77	87	10119	13241	11164
egl-e4-A	77	98	6372	7840	6764
egl-e4-B	77	98	8809	10837	9585
egl-e4-C	77	98	11276	14545	12334
egl-s1-A	140	75	4992	6446	5388
egl-s1-B	140	75	6210	8359	6848
egl-s1-C	140	75	8310	10486	9518
egl-s2-A	140	147	9780	12810	11084
egl-s2-B	140	147	12886	17068	14839
egl-s2-C	140	147	16221	20706	18258
egl-s3-A	140	159	10025	12794	11396
egl-s3-B	140	159	13554	18681	15478
egl-s3-C	140	159	16969	20786	18992
egl-s4-A	140	190	12027	16381	13862
egl-s4-B	140	190	15933	21127	18089
egl-s4-C	140	190	20179	25715	23315

TABLE 6. THE PERFORMANCE FOR EGLESE DATA SET.