Tabu search algorithm for the capacitated arc routing problem

YingZhou 11610701
Computer Science and Engineering
SUSTech
11610701@mail.sustc.edu.cn

1. Preliminaries

The Capacitated Arc Routing Problem (CARP) is normally defined on an undirected connected graph $G = (V, E)$. The set V of n nodes contains one depot (node 1). The set E of m edges includes a subset E_R of t required edges (or tasks), need to be seviced by a vehicle. Each edge e in E has a traversal cost c_e. A nonnegative demand q_e is associated with each task. The goal is to find a minimum cost set of routes to complete all tasks, the number of routes being a decision variable. In a route, some of these edges are serviced, while there exists deadheading edges are traversed but not services. The total service demand of a route by a vehicle is limited to Q. [1] The CARP has significant application in urban refuse collection,postal deliveries, snow clearance, etc. [2]

2. Methodology

2.1. Representation

There are four code files including *CARP_solver.py*, *Graph.py*, *RandomPS.py* and *Tabu_Search.py* in CARP project.

- CARP_solver.py
  - name: the CARP problem set name
  - vertices: the number of vertices V in Graph
  - depot: the vertex of vehicles start and finish
  - required_edges: the number of required edges
  - capacity: the limit capacity of a vehicle
  - graph: a data structure built by problem set
- Graph.py
  - vertices: the number of vertices V in Graph
  - adj_list: a collection of lists represents graph
  - adj_matrix: a matrix represents graph
  - mul_sp: a matrix represents shortest path between any two vertices in graph
- RandomPS.py
  - carp: a CARP object storages information of CARP problem
- Tabu_Search.py
  - S: initialized with initial solution, representing the current solution for iteration
  - S_BF: it used to record the best feasible solution during iteration
  - N: the number of required edges
  - P: the penalty term in objective function
  - graph: a data structure built by problem set
  - tabu_list: a list used to ban searched solution for a tenure time

2.2. Data Structure

- CARP_solver.py
  - dictionary
  - NumPy array
- Graph.py
  - graph
  - heap
  - list
  - NumPy array
- RandomPS.py
  - list
  - NumPy array
- Tabu_Search.py
  - dictionary
  - list

2.3. Model Design

According to the comparison of different algorithms in the *Arc routing*, TSA has a good performance in considering the quality of the solution and the speed of searching. [1].

After several attemps, I decided the structure as follows: Random path scanning (PS) method is used to obtain initial solutions proposed by Pearn [3], then an optimized tabu seach algorithm (TSA) based on [2] is implemented for imporving initial solution. Both two parts are seperately using eight processes at the same time to get the best solution.

2.4. Detail of Algorithm

2.4.1. Random Path Scanning. The *Random Path Scanning* (PS) was proposed by Pearn [3]. He modified *Path Scanning* by selecting one of five criteria at random to use when several requied edges are incident. Throwing away five criteria, I randomly pick one edge when more than one requied edges are incident in the experiment. However, I surprisingly found that it outperforms selecting criteria. The method is explained as follows.

Algorithm 1 Random Path Scanning

1: **function** RANDOM_PATH_SCANNING
2: $k = 0$ ▷ the index of route
3: copy all required arcs in a list *free* ▷ include both directions of an arc
4: **repeat**
5: $R_k \leftarrow \emptyset; load(k), costk \leftarrow 0; i \leftarrow 1$
6: **repeat**
7: $\bar{d} \leftarrow \infty$
8: **for** each u$\in free$ **do**
9: **if** $d_{i,beg(u)} < \bar{d}$ **then**
10: $\bar{d} \leftarrow d_{i,beg(u)}$
11: $\bar{u} \leftarrow u$
12: **else if** $(d_{i,beg(u)} = \bar{d})$ **then**
13: **if** $random.randint(0, 1) = 1$ **then**
14: $\bar{u} = u$
15: **end if**
16: **end if**
17: **end for**
18: add $\bar{u}$ at the end of route R_k
19: remove arc $\bar{u}$ from *free*
20: remove reverse $\bar{u}$ from *free*
21: $load[k] = load[k] + q_{\bar{u}}$
22: $cost[k] = cost[k] + \bar{d} + c_{\bar{u}}$
23: $i = end(\bar{u})$
24: **until** $\bar{d} = \infty$ or $load(k) + q_u > Q$
25: $cost(k) \leftarrow cost(k) + d_{i1}$
26: $k \leftarrow k + 1$
27: **until** $free = \emptyset$
28: **end function**

The loop (6-24) builds successive routes R_k. When searching for next arc, if the demand of the nearest required service in *free* exceeds the capacity Q that the vehicle can load, the vehicle will back to depot directly, which means a vehicle in a route will not as full as possible. The lines(13-14) means when traversing a minimum edge is incident, it will randomly be selected or discarded.

2.4.2. Heuristic Tabu Search.

1) **Neighbourhood moves**
the TSA is based on three types of neighbourhood move consisting of *Single Insertion*, *Double Insertion* and *Swap*. I add one more type called *Merge Split*.

a) *Single Insertion*
In a single insertion move, an edge from a route is removed from its current route and a insertion is made in any other route between any two serviced edges no mather they are adjacent or not, including the beginning and end of the route connecting with depot. The trail insertion considers both directions for the edge inserted in the new route.

b) *Double Insertion*
In a double insertion move, the operation is similar except that a candidate consists of two connected requeired edges in one route no mather whether there is a deadheading path between two connected edges in a route.

c) *Swap*
In a swap operation, swapping any two edges from any two different routes.

d) *Merge Split*
The merge split move is picking any two routes and using random path scanning to build another completely new routes set, which substitude the original two routes.

2) **Objective function**
The objective function to be minimised by the TSA, and it is described as follows:

for a successive route i:

$$f(i) = cost(i) + P * w(i)$$
$$w(i) = max(x(i) - 1, 0)$$

for a CARP problem solution:

$$F = max(f(i))\forall i \in routes$$

The parameter P is set 1 initially, and is then halfed if all solutions are feasible for 5 consecutive iteration; it is doubled if all solutions are infeasible for 5 consecutive iteration. Since time limition of this project, the solution gets far awy is not affordable, so the parameter P will be set to 2 if P exceeds 64 and current solution will be set to best feasible solution for the current record.

3) **Tabu list**
Tabu list, a set of solution or any else that be banned and seen as inaddmissible moves as neighbour. It can avoid falling into local optimum on finding global optimal iterations. In this project, I put the cost of searched optimal neighbors in the tabu list for two reasons: The probability of more than one solutions identical costs can be ignored; recording a set of banned solution is memory costing and it is time consuming when judging a solution is banned or not.

Algorithm 2 Tabu_solver

function RUN(time_limit)
  Set starting time, $t = time.time()$
  Set iteration counter, $k = 0$
  Set current solution S to be an initial solution and let $f(S)$ to be the objective function value of S
  Set the best feasible solition $S_{BF} = S$ and let $f(S_{BF}) = f(S)$
  Set number of consecutive iterations that solution is feasivle, $k_F = 0$
  Set number of consecutive iterations that solution is infeasible, $k_I = 0$
  Set tabu tenure, $t = N/2$, N is the number of required edges
  Set penalty parameter, $P = 1$
  Empty *tabu_list*
  repeat
    Update tabu list and remove expired solutions
    Find neighbourhood move $S\prime$ from S, set $f(S\prime) = \infty$
    for each neighbourhood move $s \notin$ *tabu_list* **do**
      if $f(s) < f(S\prime)$ **then** $S\prime = s$ and $f(S\prime) = f(s)$
      end if
      if s is feasible and $f(s) < f(S_{BF})$ **then** $S_{BF} = s$ and $f(S_{BF}) = f(s)$
      end if
    end for
    Set $S = S\prime$ and $f(S) = f(S\prime)$
    add $S\prime$ to *tabu_list*
    $k = k + 1$
    if $S\prime$ is feasible **then** $k_F = k_F + 1$
    else $k_I = k_I + 1$
    end if
    if $k_F = 5$ **then** $P = P/2$
    end if
    if $k_I = 5$ **then** $P = P * 2$
    end if
    if $k_F = 5$ or $k_I = 5$ **then** $k_F = 0$ and $k_I = 0$
    end if
  until $time.time() - t > time\_limit - 1$
end function

3. Empirical Verification

3.1. Dataset

- *gdb* instances from Golden et at. (1983)
  The 23 *gdb* instances have 7 to 27 nodes and 11 to 55 edges without no required edge, but do not use instances 8 and 9 due to inconsistencies graph.
- *val* instances from Benavent et al. (1992)
  The 34 *val* instances include 24 to 50 nodes and 34 to 97 edges without no required edge.
- *egl* instances from Li (1992) and Li and Eglese (1996)
  The 24 larger *egl* instances have 77 to 140 nodes and 98 to 190 edges with 51 to 190 required edges.

3.2. Hyperparameters

All the random seed is set by int(time.time()).

In this section, I first evaluate *Random Path Scanning* by comparing randomly picking edge or selecting criteria when multiple minimum edges are incident. Because in larger data set *egl-e-A*, it dose not exceed 0.01 s to run one time *Random Path Scanning*, so I measure the performance separately in 1s, 5s and 10s to make sure there is enough time for both of them to demonstrate effectiveness.

Secondly, *Randomly Path Scanning* with randomly picking edge as my initial solution, time setting depends on the size of data sets. For example, the smaller data sets *gdb* and *val* are set to 0.01s, 0.1s and 1s; and the larger instance are set to 0.1s, 1s and 5s. Since the randomness of *Random Path Scanning*, I run code in each test case with different time setting 10 times and record the average and maximum of the costs.

Finally, I measure the overall performance of the project under the setting below: due to the randomness of initial solution, I implemented code in each instance with different time setting five times to get the maximum and minimum costs compared with the optimal solution. Similarly, time setting depends on the size of data set. In specific, the smaller test cases *gdb* and *val* are set to 30s and 60s, while *egl* are set to 60s ans 120s.

3.3. Performance Measure

The performances are measured with fixed time. The closer to the optimal solution, the better performance. Test environment as follow: The code is written in Python, compiled using Python 3.6.5 :: Anaconda. Only NumPy package is extra imported. Ubuntu 18.04.1, 16 processors, each processor with 2 threads, 32G RAM, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz.

3.4. Experimental Result

TABLE 1: Random Path Scanning in egl-e1-A instance

Randomly Pick	1s	5s	10s
criteria(avg)	3921.00	3888.00	3850.00
edge(avg)	3741.20	3722.30	3717.05

TABLE 2: The performance of RPS in multiple instances

File	$Cost_{avg}$	$Cost_{max}$	Opt	time
	329	324.9		0.01s
	319	316.3		0.1s
gdb1	**316**	**316**	316	1s
	369	359.6		0.01s
	358	352.4		0.1s
gdb2	349	345.8	339	1s
	297	287.4		0.01s
	281	279.6		0.1s
gdb3	**275**	**275**	275	1s
	185	181.2		0.01s
	179	175.2		0.1s
val1a	**173**	**173**	173	1s
	196	192.9		0.01s
	189	186.9		0.1s
val1b	184	181.3	173	1 s
	276	271.7		0.01s
	270	263.2		0.1s
val1c	260	255.9	245	1s
	3877	3801.7		0.1s
	3754	3732.6		1s
egl-e1-A	3724	3721.3	3548	5s
	4759	4696.4		0.01s
	4695	4645.0		1s
egl-e1-B	4618	4604.1	4498	5s
	5736	5656.0		0.01s
	5562	5501.5		1s
egl-s1-A	5491	5451.6	5018	5s

TABLE 3: The performance of RPS + TSA

File	$Cost_{max}$	$Cost_{min}$	Opt	time
	345	**339**		30s
gdb2	**339**	**339**	339	60s
	179	**173**		30s
val1b	179	**173**	173	60s
	254	247		30s
val1c	254	247	245	60s
	3612	3564		60s
egl-e1-A	3612	3568	3548	120s
	4567	4567		60s
egl-e1-B	4567	4553	4498	120s
	5315	5151		60s
egl-s1-A	5209	5135	5048	120s

3.5. Conclusion

As table 1 shows, randomly picking edge when there are multiple edges are incidents obviously better than randomly picking criteria under 20 independent repeat tests with same random seed in *egl-e1-A* instance in 1 second, 5 seconds, 10 seconds respectively. In the case of eight processes running at the same time, the lowest cost is selected. Running a randomly picking criteria algorithm costs 0.00732s, while running a randomly picking edge algorithm costs 0.00427s under 160 independent repeat tests.

Table 2 displays the performance of *Random Path Scanning* in different data sets. In smallest dataset *gdb*, it approximately obtains the optimal solution in 1s. As nodes and edges increase, the combination of edges are also increased. Therefore, it is hard to get closer to the optimal solution by greedy algorithm for nearest task. The costs of RPS solution are from 10 to several hundred more than optimal solutions from *val* to *egl*.

According to table 3, it mainly measures the performance of *Random Path Scanning* algorithm with *Tabu Search* algorithm. Since finding solution neighbors by four operations totally cost about 1s depending on the size of instances, it can calculate the number of iteration by time. In small data set, 30s is enough to get a solution that is close to the optimal solution to no more than 5. Even though TSA significantly improves the initial solution, it is still about 100 more than the optimal solution. It slightly worse than *deterministic tabu search*, mainly because I do not have enough time to implement Frederickson's heuristic method. However, adding a *Merge Split* operation which increases the variability of neighbor solutions improves the performance of code through experiment.

TSA is one of the fastest mataheuristics since its neighbor operations are simple. But it seems that its performance deeply depends on neighbor moves. Without *Merge Split*, it only can reach 5300+ in *egl-s1-A*.

References

[1] H. Eiselt, M. Gendreau and G. Laporte, Arc routing. Montreal: Centre de recherche sur les transports, Universite de Montreal, 1992.

[2] Brandao J, Eglese R. A deterministic tabu search algorithm for the capacitated arc routing problem. Computer & Operations Research. 2008 Apr 1;35(4):1112-26.

[3] Pearn WL. Approximate solutions for the capacitated arc routing problem. Computers & Operations Research. 1989 Jan 1;16(6):589-600.