Capacitated Arc Routing Problem

A Lab Report of
CS303
Artificial Intelligence

BY

Xiangyi Yan

11510706@mail.sustc.edu.cn

UNDER THE GUIDANCE OF

Ke Tang
AND
Yao Zhao

Department of Computer Science and Engineering

Southern University of Science and Technology

Shenzhen, China, November 2017

Abstract

The capacitated arc routing problem (CARP) is a difcult optimisation problem in vehicle routing with applications where a service must be provided by a set of vehicles on specied roads. A heuristic algorithm based on tabu search is proposed and tested on various sets of benchmark instances. The computational results show that the proposed algorithm produces high quality results within a reasonable computing time. Some new best solutions are reported for a set of test problems used in the literature.

Code & Notable Files:

`https://github.com/yanxiangyi/CARP`

Introduction

The capacitated arc routing problem (CARP) may be described as follows: consider an undirected connected graph G=(V, E), with a vertex set V and edge set E and a set of required edges E $\supseteq$ R. A eet of identical vehicles, each of capacity Q, is based at a designated depot vertex. Each edge of the graph (v_i, v_j) incurs a cost c_{ij} whenever a vehicle travels over it or services a required edge. When a vehicle travels over an edge without servicing it, this is referred to as deadheading. Each required edge of the graph (v_i , v_j) has a demand q_{ij} associated with it. A vehicle route must start and nish at the designated depot vertex and the total demand serviced on the route must not exceed the capacity of the vehicle, Q. The objective of the CARP is to nd a minimum cost set of vehicle routes where each required edge is serviced on one of the routes.

In the instances tested, the objective is to minimise the total cost incurred on the routes and does not include any costs relating to the number of routes or vehicles required.

The graph or network on which the problem is based may be directed or undirected or mixed, but in this paper only undirected graphs are considered. A good introduction to and survey of the CARP and other arc routing problems may be found in Dror.

In the CARP, each edge in the graph may model a road that can be travelled in either direction and each vertex corresponds to a road junction. Applications of the CARP arise in operations such as postal deliveries, household waste collection, winter gritting, snow clearance and others, though in most practical applications there are additional constraints that must also be considered, including time window constraints or restrictions on certain turns.

The CARP is NP-hard. Even when a single vehicle is able to service all the required edges, the problem reduces to the rural postman problem (RPP) which has been shown

to be NP-hard by Lenstra and Rinnooy Kan. Additionally, Golden and Wong showed that even 1.5-approximation for the CARP is NP-hard. Exact methods for the CARP have only been able to solve relatively small examples to optimality.

Data Set Source:

`http://logistik.bwl.uni-mainz.de/benchmarks.php`

Basic Data Structures and Algorithms

Data Structures

Graph

A (simple) graph is an ordered pair G = (V, E), where V is a nite set of nodes (vertices, points) and E is a nite set of edges (lines, arcs, links). An edge is an ordered pair of dierent nodes from V , (s, t), where s is the source node and t is the target node. This is a directed edge and in that case G is called a directed graph.

An edge can be dened as a 2-element subset of V, s, t = t, s, and then it is called an undirected edge. The nodes s and t are called the ends of the edge (endpoints) and they are adjacent to one another. The edge connects or joins the two endpoints. A graph with undirected edges is called an undirected graph. In our approach, an undirected edge corresponds to the set of two directed edges (s, t), (t, s) and the representative is usually (s, t) with s < t.

The order of a graph G = (V, E) is the number of nodes |V|. The degree of a node in an undirected graph is the number of edges that connect to it.

A graph G'= (V', E') is a subgraph of a graph G = (V, E) if V'is a subset of V and E'is a subset of E.

Algorithms

Dijkstra's Algorithm

The shortest distances between two vertices of two required edges are calculated by Dijkstras algorithm and stored in the shortestPath matrix. First get all the vertices which are the end of required edges. Second iterate the vertices and set the vertex as the source in Dijkstras algorithm.

for *each vertex* $v \in V_G$ **do**
    $dist[v] \leftarrow \infty$;
    $parent[v] \leftarrow NIL$;
end
$dist[s] \leftarrow 0$;
$Q \leftarrow V_G$;
while $Q \neq \emptyset$ **do**
    $u \leftarrow Extract - MinQ$;
    for *each edge* $e = (u, v)$ **do**
        if $dist[v] > dist[u] + weight[e]$ **then**
            $dist[v] \leftarrow dist[u] + weight[e]$;
            $parent[v] \leftarrow u$;
        end
    end
end
$H \leftarrow (V_G, \emptyset)$;
for *each vertex* $v \in V_G,\ v \neq s$ **do**
    $E_H \leftarrow E_H \cup \{(parent[v], v)\}$;
end
return $H, dist$

Algorithm 1: Dijkstra's Algorithm

Initialization: Path Scanning

Path Scanning with 5 Rules

This method is based on the procedure proposed by Golden et al. Each route is extended by one required edge at each step using a variety of selection rules.

Each route starts at the depot vertex. Let S be the set of required edges closest to the end of the current route that are not yet served and do not exceed the capacity of the current route. If S is empty then complete the current route using the shortest deadheading path from the end of the current route to the depot vertex and start a new route. If S is not empty, exclude from S any edges that would close the route unless that would make S empty. Select a required edge in S to be the next edge in the route to be serviced according to the current rule and extend the current route to the vertex at the end of the selected edge.

Five rules are used to determine the next required edge, e, in the route to be serviced:

1: Maximize the ratio c_{ij} $/d_{ij}$, d_{ij} being the demand of edge [i, j].

2: Minimize the ratio c_{ij} $/d_{ij}$.

3: Maximize the return cost from j to the depot.

4: Minimize this return cost.

5: If the vehicle is less than half-full, then apply rule3, otherwise rule4.

Algorithm

$k \leftarrow 0$;
copy all required arcs in a list $free$;
while $(free \neq \emptyset)$ **do**
  $k \leftarrow k + 1; R_k \leftarrow \emptyset; load(k), cost(k) \leftarrow 0; i \leftarrow 1;$
  while $(free \neq \emptyset) \vee (\bar{d} \neq \infty)$ **do**
    $\bar{d} \leftarrow \infty$;
    for $u \in free \mid load(k) + q_n <= Q$ **do**
      if $d_{i,beg(u)} < d$ **then**
        $\bar{d} \leftarrow d_{i,beg(u)}$;
        $\bar{u} \leftarrow u$;
      end
      if $d_{i,beg(u)} = d \wedge better(u, \bar{u}, rule)$ **then**
        $\bar{u} \leftarrow u$;
      else
    end
    add $\bar{u}$ at the end of route R_k;
    remove arc $\bar{u}$ and its opposite $\bar{u} + m$ from $free$;
    $load(k) \leftarrow load(k) + q_{\bar{u}}$;
    $cost(k) \leftarrow cost(k) + d + c_{\bar{u}}$;
    $i \leftarrow end(\bar{u})$;
  end
  $cost(k) \leftarrow cost(k) + d_{i1}$;
end

Algorithm 2: Path-Scanning for one priority rule

Path Scanning with Randomizations

The improved Path Scanning Algorithm is called PS with Random Task (PSRT) determines at each iteration the set F of feasible unserved edges that are nearest to the last node of the emerging route. then draws one edge at random among these conditions. Note that the five rules of PS are no longer used in this version. The main interest of these randomizations is to allow multiple executions to get better solutions. The tests show that 4 to 11 executions are enough to do better that PS and its five runs.

Path Scanning with Ellipse Rule

The ellipse rule is defined as follows. Let ned be the number of edges with positive demand in the network, td the total demand to be collected, tc the total cost assigned to edges with positive demand, a real parameter, and $[v_h, v_i]$ the last serviced edge on the route. If the remaining capacity of the vehicle is less than or equal to td/ned (i.e., the average demand on the arcs), then the next edge to be serviced $[v_p, v_j]$ must be the nearest edge to $[v_h, v_i]$ ($v_i = v_p$, if the edges are adjacent) satisfying the condition:

$$SP(v_i, v_p) + c_{pj} + SP(v_j, v_0) <= tc/ned + SP(v_i, v_0)$$

where for example, $SP(v_i, v_p)$is defined as the shortest path cost between the nodes v_i and v_p , and v_i and v_0 are the $foci$ of the ellipse. If no feasible edge (i.e., an edge with demand less than or equal to the remaining capacity of the vehicle) satisfies (1) then the vehicle returns directly to the depot. In general, as increases, the ellipse rule is invoked earlier in the construction of a vehicle route. When enforced, condition 1 limits the addition of arcs to the route to those that are within an ellipse. In general, tc/ned will be less than $SP(v_i, v_0)$ because it represents the average cost (distance) of an arc in the network while $SP(v_i, v_0)$ must include at least one arc and generally will include multiple arcs. However, tc/ned may be greater than $SP(v_i, v_0)$ if node v i is near node $v0$. In Fig. 1, we present 3 ellipses for various values of tc/ned where $c_1 + c_2 = tc/ned + SP(v_i, v_0)$. We set $SP(v_i, v_0) = 1$ and set tc/ned equal to 0.1, 0.5, and 1.0. As tc/ned increases, the area of the ellipse increases and its shape approaches its limit which is a circle. As this occurs, more unserved arcs are likely to be eligible for inclusion on the current route.

Optimization: Tabu Search

Neighbourhood moves

The TSA is based on three types of neighbourhood move. The rst two are insertions (single and double) and the third is a swap.

In a single insertion move, a candidate edge (only required edges can be candidates) is removed from its current route and a trial insertion is made in any other route between any two serviced edges that are not adjacent, but are linked by a deadheading path of edges. The trial insertion considers both directions for the edge to be traversed when inserted in the new route. In a double insertion move, the operation is similar except that a candidate consists of two connected required edges in one route.

The swap move is similar. Two candidate edges are selected from two different routes, each containing deadheading paths. The candidate edges are removed from their original routes and inserted in the other routes between any two serviced edges that are not adjacent, but are linked by a deadheading path of edges.

In this implementation, the frequencies of the different types of move may change according to the phase of the algorithm. Parameters, F_{SI}, F_{DI} and F_{SWAP} denote the frequencies of the single insertion, double insertion and swap moves, respectively. For example, the value of F_{SWAP} implies that the swap move is only tested every F_{SWAP} iterations.

The trial move chosen depends on the effect of the move on the objective function dened in the next section.

Objective function

The objective function to be minimised by the TSA includes the total cost of each route, i, plus a penalty cost $P_w(i)$, where P is a penalty term and $w(i) = max(x(i)Q, 0)$, and where $x(i)$ is the sum of the demands on the edges serviced in route i. For any candidate solution,S,the objective function is denoted by $f(S)$.

The parameter P is set at 1 initially, and is then halved if all the solutions are feasible for 10 consecutive iterations; it is doubled if all the solutions are infeasible for 10 consecutive iterations. A similar use of P to direct the search from feasible to infeasible regions and vice versa has also been used before by e.g. Hertz and Beullens.

Admissibility of moves

A conventional tabu list is constructed to prevent the reversal of accepted moves for the next t iterations, where t is the length of the tabu list. The length of the tabu list is xed and after some experimentation has been set to $N/2$ in Phase 1 of the TSA and $N/6$ in Phase 2, where N is the number of required edges. The tabu restriction may be overridden if the move will produce a solution that is better than what has been found in the past. This is referred to as the aspiration criterion.

In the TSA, a trial move to solution S is regarded as admissible:

(i) if it is not currently on the tabu list;

(ii) or if it is tabu and infeasible, but the value of $f(S)$ is less than the value of the best infeasible solution yet found;

(iii) or if it is tabu and feasible, but the value of $f(S)$ is less than the value of the best feasible solution yet found.

Outline of TSA

The operation of the TSA can now be summarised in the following steps:

1. Initialize:

Set current solution S to be an initial solution and let $f(s)$ be the total cost of S Set best solution S_B = S Set best feasible solution $S_{BF} = S$, if S is feasible; otherwise set $f(S_{BF}) = \infty$ Set iteration counter, $k = 0$ Set number of iterations since best solution after Intensication step, $k_B = 0$

Set number of iterations for applying Intensication step, $k_L = 8N$ Set number of iterations since best feasible solution, $k_{BF} = 0$ Set number of consecutive iterations that solution is feasible, $k_F = 0$ Set number of consecutive iterations that solution is infeasible, $k_I = 0$ Set number of iterations since best solution in total, $k_{BT} = 0$ Set tabu list to be empty Set tabu tenure, $t = N/2$ Set frequency parameters, $F_{SI} = 1$, $F_{DI} = 5$, $F_{SWAP} = 5$ Set penalty parameter, $P = 1$.

2. Find neighbourhood move (S') from S, using all required edges as a candidate list: Set $f(s') = \infty$.

2.1. If k is a multiple of F_{SI} perform trial single insertion moves For each admissible move, s, do:

If $f(s) < f(s')$, then

(1) S'= s and $f(s') = f(s)$;

(2) If s is feasible and $f(s) < f(S_{BF})$, or $f(s) < f(S_B)$ go to Step 3. Repeat until all the potential moves have been explored.

2.2. If k is a multiple of F_{DI} perform trial double insertion moves For each admissible move, s, do:

If $f(s) < f(s')$, then

(1) $S' = s$ and $f(s') = f(s)$;

(2) If s is feasible and $f(s) < f(S_{BF})$, or $f(s) < f(S_B)$ go to Step 3. Repeat until all the potential moves have been explored.

2.3. If k is a multiple of F_{SWAP} perform trial swap moves For each admissible move, s, do:

If $f(s) < f(s')$, then

(1) $S' = s$ and $f(s') = f(s)$;

(2) If s is feasible and $f(s) < f(S_{BF})$, or $f(s) < f(S_B)$ go to Step 3. Repeat until all the potential moves have been explored.

3. Improve changed routes:

For each of the two routes that have been changed, apply the route improvement procedure using Fredericksons heuristic to try to nd further cost reductions if possible, resulting in solution S''.

4. Update:

Update tabu list.

If $S''$is feasible and$f(S'') < f(S_{BF})$ then

(1) apply Fredericksons heuristic to each route individually (except the two routes that have just been updated in Step 3) to get a new solution S'''

(2) set $S_{BF} = S'''$, $S''= S'''$and set $k_B = k_{BF} = k_{BT} = 0$.

If $f(S'') < f(S_B)$ then set $S_B = S''$and $k_B = k_{BT} = 0$.

Set $k = k + 1$, $k_B = k_B + 1$, $k_{BF} = k_{BF} + 1$, $k_{BT} = k_{BT} + 1$.

If k is a multiple of 10, then if $k_F = 10$, then set $P = P/2$ else if $k_I = 10$ then set $P = 2P$; If $k_F = 10$ or $k_I = 10$ then recalculate $f(S_B)$ with the new value of P and set $k_F = k_I = 0$.

5. Change of parameter values:

If $k_B = k_L$ /2, then set $F_{SI} = 5$, $F_{DI} = 1$, $F_{SWAP} = 10$.

6. Intensication: If $k_B = k_L$, then

(1) if $f(S_{BF}) < \infty$ then set S = S_{BF} else set S = S_B ,

(2) set $k_B = 0$, $P = 1$, $k_F = 0$, $k_I = 0$, $F_{SI} = 1$, $F_{DI} = 5$, $F_{SWAP} = 5$, $k_L = k_L + 2N$,

(3) recalculate $f(S_B)$with the new value of P, and empty tabu list.

(Note that the changes to the parameter values and emptying the tabu list at this point means that the sequence of solutions following S is different to the sequence generated previously.)

7. Stopping criterion:

If ($k >= 900\sqrt{N}$ and $k_{BF} >= 10N$) or $k_{BT} = 2k_L$ then stop, otherwise go to Step 2.

Empirical Verification

Environment

- **CPU**: Intel i5 Dual Core 3.1GHz
- **OS**: MacOSX High Sierra
- **IDE**: Pycharm
- **Python Version**: 2.7.12

Benchmark Results

Benchmark Cost	Test Time: 60s	Test Time: 600s	Lowerbound
egl-e1-A	3700	3701	3515
egl-s1-A	5380	5363	5018
gdb1	316	316	316
gdb10	275	275	275
val1A	173	173	173
val4A	406	406	400
val7A	289	285	279

Table 1: Results on benchmark data sets

Reference

- A. Kapanowski and L. Galuszka, **Weighted graph algorithms with Python**
- A. Corberan and G. Laporte, **Arc Routing: Problems, Methods, and Applications**
- Luis Santos, Joao Coutinho-Rodrigues, John R. Current, **An improved heuristic for the capacitated arc routing problem**
- Jose Brandao, Richard Eglese, **A deterministic tabu search algorithm for the capacitated arc routing problem**