What makes the dynamic capacitated arc routing problem hard to solve

Tong, Hao; Minku, Leandro; Menzel, Stefan; Senhoff, Bernhard; Yao, Xin

DOI:
10.1145/3512290.3528756

Algorithm 1: VND-CARP

```
Generate tour by Frederickson heuristic.
Apply SHORTEN and CUT to obtain an initial solution.
while True do
    Set $i = D/Q$, $s_{best} = s$;
    while True do
        Set number of neighbours: $c = 1$.
        Set best value of a neighbour: $f_{best} = f(s)$.
        while $c \leq M_i$ do
            Select $i$ routes in $s$, merge them into a giant tour.
            Apply SWITCH and then CUT on this tour.
            Apply SHORTEN on each new tour.
            New resulting solution: $s^{'}$;
            if $f(s^{'}) < f_{best}$ then
                Set $best_s = s^{'}$;
                Set $f_{best} = f(s^{'})$ ;
            c = c + 1;
        if $f_{best} < f(s)$ then
            Set $s = best_s$;
        else
            i = i - 1
        if $i \geq 1$ then
            break;
    if $f(s) \geq f(s_{best})$ then
        break;
```

Algorithm 2: ILMA

Initialization: $nc - 3$ chromosomes;
Add one chromosome by Path_Scanning;
Add one chromosome by Augmnt-Merge;
Add one chromosome by Ulusoy's split;
while *Stop criterion is not met* **do**
    Select two chromosomes P_1 and P_2 by binary tournament selection;
    Apply ordered crossover operator to P_1 and P_2 to generate O_x;
    Set $O = O_x$;
    if $rand() < P_{ls}$ **then**
        apply local search to O_x to generate O_m;
        if O_m *is not existed in pop* **then**
            $O = O_m$;
    Evaluate O to get $f(O)$;
    if $f(O) == f(P_1)$ **then**
        Replace P_1 by O;
    else if $f(O) == f(P_2)$ **then**
        Replace P_2 by O;
    else if $f(O) == f(P) \&\& P! = P_1 \&\& P! = P_2$ **then**
        Discard O;
    else if $f(O)$ *is not used in current pop* **then**
        Randomly choose a P from $[nc/2, nc]$;
        Replace P with O ;
    Resort Population;
    if *Replacement criteria are met* **then**
        replace $nrep$ chromosomes with randomly generated chromosomes;

Algorithm 3: The pseudo code of Simulation System

Input: Executable solution s, Time of change: t, Previous graph G

Set Events probability: $\{p_1, p_2, p_3, p_4, p_5\}$;
Set probability for broke down roads recovering: p_{bdrr};
Set probability for congest roads recovering and becoming better: p_{crr}, p_{crbb};
Determine the stopping point for each vehicles according to s, t, G;
Update graph, and remove all served tasks.
Randomly select $p_1 \times 100\%$ vehicles to break down (**Event 1**).
Update the graph.
/** Cost Impact ****/**
for *each edge* e_i **do**
  if $e.change == 0$ **then**
    $r_2 = rand(), r_3 = rand()$
    if $r_2 < p_2$ *and* $r_3 < p_3$ **then**
      Event 2 happens: $e_i.cost = Inf$, $e_i.change == 2$;
    if $r_2 < p_2$ *and* $r_3 > p_3$ **then**
      Event 3 happens: Increase cost of e_i, $e_i.change == 3$;
  else if $e_i.change == 2$ *and* $rand() < p_{recover}$ **then**
    Recover edge e_i, $e_i.change == 0$;
  else if $e.change == 3$ **then**
    if $rand() < p_{congestion\_recover}$ **then**
      Recover edge e_i, $e_i.change == 0$;
    else if $rand() < p_{congestion\_better}$ **then**
      Decrease cost of e_i
    else
      Increase cost of e_i
  else
    continue;
/** Demand Impact ****/**
for *each edge* e_i **do**
  if $e.change = 1$ **then**
    continue;
  else
    $r_4 = rand(), r_5 = rand()$;
    if $Edge.demand > 0$ *and* $r_4 < p_4$ **then**
      Event 4 happens: $e_i.change = 4$;
    if $Edge.demand == 0$ *and* $r_5 < p_5$ **then**
      Event 5 happens: $e_i.change = 5$;

Output: The new graph G_1

Algorithm 4: Build auxiliary graph for DCARP

Input: Individual : $I = \{t_1, t_2, ..., t_N\}$
Stop points for outside vehicles: $V = \{v_1, v_2, ..., v_K\}$
Remain capacity for outside vehicles: $CP = \{cp_1, cp_2, ..., cp_K\}$

Generate $N + 1$ Nodes (Index from 0 to N) for the auxiliary graph G^*.
for *each outside vehicle* k **do**
  for *each node pair:* $Node_i$ *and* $Node_j$ **do**
    Use vehicle k to serve $\{t_{i+1}, t_{i+2}, ..., t_j\}$;
    Sub-route: $r_{ijk} = \{v_k \to t_{i+1} \to t_{i+2}, \to ..., \to t_j \to depot\}$;
    Calculate the total demand d_{ijk} of r_{ijk};
    if $d_{ijk} > cp_k$ **then**
      $continue$;
    else
      Calculate the cost of r_{ijk}: c_{ijk};
      Assign an edge e_{ijk} between $Node_i$ and $Node_j$ with weight equal to c_{ijk};

Output: An auxiliary graph G^*

Algorithm 5: A* based optimal split scheme

Input: Individual : $I = \{t_1, t_2, ..., t_N\}$

Build an auxiliary graph G^* for DCARP;
$expandNode = Node_0$; openNodeSet = {}; pathSet = {};
while *True* **do**
    if *expandNode == target* **then**
        Shortest path P: path correspond to $expandNode$;
        Minimal cost C: $f_{expandNode}$ correspond to $expandNode$;
        $break$;
    Select *rootPath* (i.e. path from $Node_0$ to $expandNode$) from pathSet;
    for *each successor of expandNode* **do**
        newPath = *rootPath* + $expandNode \rightarrow successor$;
        Remove all edges correspond to vehicles being used in *newPath* for $successor$;
        Calculate the h_{succ} and g_{succ};
        Set $f_{succ} = h_{succ} + g_{succ}$;
        if *successor == target* **then**
            Repair f_{succ};
        Add the $successor$ into openNodeSet;
        Add the path correspond to $successor$ into pathSet;
    Remove the $expandNode$ from openNodeSet, and the *rootPath* from pathSet;
    Select the node in openNodeSet with minimal f as $expandNode$;
The shortest path from $Node_0$ to $target$ in G^*: $P = \{p_1, p_2, ..., p_M\}$;
Each p_m represents an edge e_{ijk}, which denotes a sub-route r_{ijk};
Obtain the solution S by splitting the I by P.

Output: Solution $S = \{r_1, r_2, ..., r_M\}$, Minimal cost: C

Algorithm 6: Greedy split scheme

Input: Individual : $I = \{t_1, t_2, ..., t_N\}$

Build an auxiliary graph G^* for DCARP;

for *each edge* e_{ijk} *in* G^* **do**

  Calculate the UDC: UDC_{ijk};

$expandNode = Node_0$; *newPath* = $Node_0$

while *True* **do**

  if *expandNode == target* **then**

    Greedy path: *newPath*, $P = \{p_1, p_2, ..., p_M\}$;

    Calculate the greedy cost of greedy path: C;

    break;

  rootPath $\leftarrow$ *newPath*;

  Select the $Node_X$ with minimal UDC from all $successors$ for $expandNode$;

  newPath = *rootPath* + $expandNode \rightarrow Node_X$;

  Remove all edges correspond to vehicles being used in *newPath*;

  $expandNode \leftarrow Node_X$;

Each p_m represents an edge e_{ijk}, which denotes a sub-route r_{ijk};

Obtain the solution S by splitting the I by P.

Output: Solution $S = \{r_1, r_2, ..., r_m\}$, Greedy cost: C

Algorithm 7: The hybrid local search framework

```
Input: The update Map (update graph data)
       Dynamic State:
         1). Stop locations of outside vehicles;
         2). Remaining capacities of outside vehicles;
         3). Remaining tasks.
1  Initialize the solution archive $SA \leftarrow \varnothing$;
2  Re-construct the solution $S_0$ with explicit routes;
3  Add initial solution into archive $SA = SA \cup S_0$;
4  Set global best solution $S_{gb} = S_0$ for each solution $S_i$ in $SA$ do
5      Local best solution $S_{lb} = S_i$;
6      while true do
           // The following loop (line7-line10) run in parallel
7          for each neighborhood move $Move_j$ do
8              Solution $S_{mj}$ = $Move_j(S_{lb})$
9              if improved AND archive is not full then
10                 Add $S_{mi}$ into archive: $SA = SA \cup S_{mi}$;
11         Update best solution $S_{lb}$ from $S_{mj}$;
12         if No improved move OR exceed time limitation then
13             break;
14     if $S_{lb}.cost < S_{gb}.cost$ then
15         $S_{gb} \leftarrow S_{lb}$;
16     if exceed time limitation then
17         break;
Output: The global best solution $S_{gb}$
```

Algorithm 8: Pseudo code of the instance generator.

Input: Static instance, initial solution;
The full capacity of vehicles: Q;
A configuration of dynamic event: C_{Event};
Configurations of state factors: C_{OV}, C_{RQ};
Output: A DCARP instance

if $C_{Event} == ND$ **then**
    if *ND-N* $==$ *few* **then**
        $p = 20\%$
    else
        $p = 80\%$
    Uniformly random select p of tasks in the remaining tasks and save them into a set Set_{ND}.
    if *ND-V* $==$ *small* **then**
        $dm = \frac{Q}{|Set_{ND}|}$
    else
        $dm = \frac{4Q}{|Set_{ND}|}$
    Add demand dm to each task in Set_{ND}.
if C_{Event} == *NT* **then**
    Save all available edges (not task) in a list $List_{NT}$;
    if *NT-P* $==$ *close* **then**
        Sort $List_{NT}$ in ascending order according to the max distance of two nodes to the depot.
    else
        Sort $List_{NT}$ in descending order according to the max distance of two nodes to the depot.
    if *NT-N* $==$ *few* **then**
        $p = 20\%$
    else
        $p = 80\%$
    Select the front p of edges from $List_{NT}$ as the new tasks and save into Set_{NT}.
    if *NT-V* $==$ *small* **then**
        $dm = \frac{Q}{|Set_{ND}|}$
    else
        $dm = \frac{4Q}{|Set_{ND}|}$
    Add demand dm to each task in Set_{NT}.