A Route Clustering and Search Heuristic for Large Scale Multi-Depot Capacitated Arc Routing Problem

Yuzhou Zhang, Yi Mei, Senior Member, IEEE, Shihua Huang, Xin Zheng and Cuijuan Zhang

Abstract—The capacitated arc routing problem (CARP) has attracted much attention for its many practical applications. The large scale multi-depot CARP (LSMDCARP) is an important CARP variant which is very challenging due to its vast search space. To solve LSMDCARP, we propose an iterative improvement heuristic named Route Clustering and Search Heuristic (RoCaSH). In each iteration, it first (re)decomposes the original LSMDCARP into a set of smaller single-depot CARP sub-problems using route cutting off and clustering techniques. Then, it solves each sub-problem using the effective Ulusoy’s split operator and local search. On one hand, the route clustering helps the search for each sub-problem by focusing more on the promising areas. On the other hand, the sub-problem solving provides better routes for the subsequent route cutting off and clustering, leading to better problem decomposition. The proposed RoCaSH was compared with the state-of-the-art MDCARP algorithms on a range of MDCARP instances, including different problem sizes. The experimental results showed that RoCaSH significantly outperformed the state-of-the-art algorithms, especially for the large scale instances. It managed to achieve much better solutions within a much shorter computational time.

Index Terms—Capacitated arc routing problem, multi-depot, scalability, route clustering, search heuristic

I. INTRODUCTION

As a complex combinatorial optimization problem, the Capacitated Arc Routing Problem (CARP) [1] is well-known for its wide applications in the real world, such as winter gritting [2], street salting [3], mail delivery [4], urban waste collection [5], and snow removal [6], [7]. The aim of the problem is to find an optimal set of routes for a fleet of vehicles based at a depot to serve the edges in a given network subject to specific constraints so that the total cost is minimized. There have been extensive research on CARP, and a large amount of competitive approaches have been proposed, e.g. [8]–[17].

Multi-depot CARP (MDCARP) [18] is an important CARP variant, which consists of multiple depots located at different regions of the graph. It reflects a variety of real-world considerations, such as garbage collection, road sprinkling and street sweeping in the large districts, in which more than one depot are involved. Compared with traditional single-depot CARP, MDCARP has a much larger and more complex solution space, as one has to allocate the tasks (i.e. edges) to the depots.

In real world, the problem size of CARP or MDCARP can usually be very large (e.g., thousands of streets in a city need to be served for waste collection). The existing approaches to MDCARP were only examined on the small and medium instances. For example, in [19] two algorithms (i.e., HACA and MDMA) were tested on the mdgdb set in which the number of required edges varies from 11 to 55. Though certain works (e.g., [20]–[27]) on large scale CARP (LSCARP) have been proposed, these works only considered the problems with a single depot.

As an efficient strategy for large scale problems, the Divide and Conquer (DC) strategy aims to decompose the whole problem into several sub-problems, and then tackle each sub-problem independently or cooperatively. Finally, all the obtained sub-solutions are concatenated together to form a complete solution to the original problem. The DC strategy has achieved great success in a range of problems, such as multi-objective optimization [28], [29], continuous optimization [30]–[32], vehicle routing [33], [34], job shop scheduling [35], [36] and other large-scale optimization problems [37], [38]. It has also been successfully applied to large scale CARP with a single depot (e.g., [22][33]). Intuitively, we can use the DC strategy in large scale MDCARP (LSMDCARP), and divide the problem into several single-depot CARP sub-problems (each required edge, i.e., task, is associated with a depot). However, there is no work to solve MDCARP with the DC strategy so far, not to mention LSMDCARP. It is challenging to find the appropriate assignment.

To address the above issues, in this paper, we propose a route clustering and search heuristic (RoCaSH) for solving LSMDCARP. The proposed RoCaSH employs a novel route clustering scheme and a route cut-off operator to achieve an effective allocation of the tasks to the depots. Then, the smaller single-depot CARP sub-problem associated to each depot is solved by the Ulusoy’s split operator and local search, which have been effective for solving LSCARP. To verify the effectiveness of RoCaSH, it was examined on the MDCARP datasets with a wide range of problem sizes. The results showed that the proposed RoCaSH significantly outperformed the state-of-the-art algorithms for MDCARP, especially for the instances with large problem sizes.

The rest of this paper is organized as follows. First, MDCARP is described in Section II. Then, the proposed RoCaSH is introduced in Section III. Afterwards, Section IV
provides the experimental studies and discussions. Finally, the conclusions and future work are given in Section V.

II. BACKGROUND AND RELATED WORK

A. Multi-Depot Capacitated Arc Routing Problem

In MDCARP, an undirected connected graph $G(V,E)$ is given, where $V$ is the vertex set, and $E$ is the edge set. The vertex set $V$ contains two types of vertices, the depot vertices $V(D) \subseteq V$ and non-depot vertices $V(N) \subseteq V$, i.e., $V = V(N) \cup V(D)$. Each edge $e \in E$ has three non-negative properties: a demand $dem(e)$, a service cost $sc(e)$ and a deadheading cost $dc(e)$ (which is the cost caused by traversing without service). An edge is called a task if its demand is positive (i.e., it is required to be served). Let the set of all the tasks be denoted as $T \subseteq E$. The tasks are to be served by a fleet of identical vehicles with limited capacity of $Q$. The goal of MDCARP is to find a set of least-cost routes for the vehicles to serve all the tasks subject to the following constraints:

- Each vehicle starts from a depot $v_d^{(D)} \in V(D)$, and returns to the depot;
- Each task is served exactly once by a vehicle;
- The total demand served by each vehicle cannot exceed the vehicle’s capacity $Q$.

Note that the conventional CARP is a special case of MDCARP, where there is a single depot, i.e., $|V(D)| = 1$.

We give the problem formulation under the task representation [39]. Specifically, each task $t = (u,v)$ in $T$ is assigned two IDs $x_1$ and $x_2$, each representing one of its directions. Each task ID has a head vertex and a tail vertex. For example, the head and tail vertices of $x_1$ are $hv(x_1) = u$ and $tv(x_1) = v$, while the head and tail vertices of $x_2$ are $hv(x_2) = v$ and $tv(x_2) = u$. The two task IDs are inverse to each other, i.e. $x_1 = inv(x_2)$ and $x_2 = inv(x_1)$. Each task ID is also associated with a demand, a service cost, and a deadheading cost, which are set based on their corresponding task. For example, $dem(x_1) = dem(x_2) = dem(t)$, $sc(x_1) = sc(x_2) = sc(t)$, and $dc(x_1) = dc(x_2) = dc(t)$. Given a graph with $|T|$ undirected tasks, we will obtain $2|T|$ task IDs.

In addition, we denote the depot loop for each depot using a separate ID $x_d^{(D)}(d = 1,\ldots,|V(D)|)$, where $hv(x_d^{(D)}) = tv(x_d^{(D)}) = v_d^{(D)}$, $dem(x_d^{(D)}) = sc(x_d^{(D)}) = 0$, $inv(x_d^{(D)}) = x_d^{(D)}$. Overall, we will have $2|T| + |V(D)|$ task IDs.

Under the above representation, the $k$th route associated with depot $v_d^{(D)}$ can be represented as:

$$R_{d,k} = (x_{d,k,1}, x_{d,k,2}, \ldots, x_{d,k,R_{d,k}}),$$

where $|R_{d,k}|$ indicates the number of the tasks in $R_{d,k}$. A solution $S$ consists of all the routes and can be represented as:

$$S = (R_1, \ldots, R_{|V(D)|}),$$

where $R_d = (R_{d,1}, \ldots, R_{d,R_d})$ consists of the routes associated with $v_d^{(D)}$. $|R_d|$ is the number of the routes associated with the depot $v_d^{(D)}$. The total cost of $S$ is calculated as follows:

$$tc(S) = \sum_{d=1}^{|V(D)|} tc(R_d),$$

where

$$tc(R_d) = \sum_{k=1}^{|R_{d,k}|} \sum_{i=1}^{\phi} \left[ sc(x_{d,k,i}) + \varphi(tv(x_{d,k,i}), hv(x_{d,k,i+1})) \right],$$

where $\varphi(u, v)$ indicates the total deadheading cost of the edges on the shortest path from vertex $u$ to $v$, which can be pre-calculated by Dijkstra’s algorithm [40].

Then, we give the problem formulation of MDCARP as follows.

$$\min \ tc(S),$$

s.t. $x_{d,k,1} = x_d^{(D)}, \forall d = 1,\ldots,|V(D)|, k = 1,\ldots,|R_d|,$

$$x_{d,k,|R_{d,k}|} = x_d^{(D)}, \forall d = 1,\ldots,|V(D)|, k = 1,\ldots,|R_d|,$$

$$|V(D)| = |R_d|,$$

$$x_{d,k,i} \neq x_{d,k,i}, \forall d \neq d', k \neq k' \lor i \neq i',$$

$$x_{d,k,i} \neq inv(x_{d,k,i}), \forall d \neq d' \lor k \neq k' \lor i \neq i',$$

$$\sum_{i=1}^{|R_{d,k}|} dem(x_{d,k,i}) \leq Q, \forall d = 1,\ldots,|V(D)|,$$

$$1 \leq k \leq |R_d|,$$

Eq. (5) is the objective function, which is to minimize the total cost calculated by Eq. (3). Eqs. (6) and (7) indicate that each route starts and ends at the same depot. Eqs. (8), (9) and (10) ensure that all the tasks in $T$ must be served exactly once. Eq. (11) guarantees that the capacity constraint is satisfied.

B. Related Work

CARP has attracted a lot of attention since 1981 when it was presented by Golden [1]. The early approaches to CARP can be divided into two types, i.e., mathematical programming (e.g., [17], [20], [41]-[43]) and heuristic algorithms (e.g., [10], [13], [44], [45]). However, most of these studies aimed at the small and medium scale problems (e.g. the tested instances have no more than 190 tasks).

In 2008, Brandão and Eglese [46] generated a large EGL-G dataset. In the new dataset, the number of tasks is up to 375, which is much larger than that of the previously used datasets (e.g., gdb, val and egl). This raises the issue of scalability of the algorithms, which leads to the emergence of studies for Large Scale CARP (LSCARP) [15], [20], [22], [24], [26], [46]. In 2017, Tang et al. [27] addressed the scalability issue on two much larger datasets, i.e., Hefei and Beijing, in which the size of the instances is over 3000.

The Divide-and-Conquer (DC) strategy is an effective technique to decompose large problems into smaller ones and solve them independently or cooperatively. For LSCARP, there have been a number of effective algorithms proposed based on the DC strategy. For example, Mei et al. [22] proposed
a cooperative co-evolution algorithm called RDG-MAENS, in which the entire evolutionary process is divided into a number of cycles. At the beginning of each cycle, the routes of the best-so-far solution are grouped together based on a Route Distance Grouping (RDG) decomposition scheme. The decomposition scheme can guarantee that the optimal solution under the decomposition can be no worse than the current solution. Tang et al. [27] proposed an individual search algorithm based on a hierarchical decomposition, which is called SAHiD. In SAHiD, the tasks are grouped together in an hierarchical way to form the so-called “virtual tasks”. The search is repeated by heuristically breaking and forming virtual tasks.

In contrast with CARP, MDCARP has another level of challenge, which is to assign tasks/routes to the depots. Since introduced in 2000 [18], MDCARP received a number of research interests over past years (e.g., [19], [47]–[52]). In [47], Zhu et al. proposed a Hybrid Genetic Algorithm (HGA) to tackle the MDCARP, in which a route is considered as an individual, and the population is essentially a set of routes. In [19], a memetic algorithm was proposed for MDCARP by Kansou and Yassine, named MDMA, in which each depot involves a set of routes. The initial individuals are created by repetitively assigning a randomly selected task to its nearest depot using the best insertion strategy. Then, the individuals undergo the process of evaluation, selection, crossover and local search, and the best solution is returned after a certain number of iterations. MDMA is essentially an approach to the basic CARP without particular consideration of multiple depots except in the initialization process. Moreover, there have been a lot of work on different variants of MDCARP. In [49], a MILP model was presented for the asymmetric MDCARP, and the valid inequalities tightening its LP relaxation were taken into account. In [51], a memetic algorithm was developed for the MDCARP with multi-objectives, and a hybrid ant colony optimization algorithm was proposed for multi-depot periodic open capacitated arc routing problem in [52].

When the problem size grows, the existing approaches to MDCARP become less effective, as they tend to solve the problem as a whole. For large scale optimization problems, the DC strategy has achieved great success in a wide range of problems, such as single-objective [31], [53], [54] and multi-objective continuous optimization [28]–[30], [32], [55], job shop scheduling [35], [36], [56], vehicle routing [33], [34], timetabling [57], [58], and LSCARP [21], [22], [24], [27].

As the counterpart of CARP, vehicle routing problem (VRP) serves the vertices [59] rather than the edges. There have been extensive studies on multi-depot VRP and large scale VRP with the DC strategy. Lim et al. proposed a single-stage approach to multi-depot VRP, in which the assignment of the customers and the routing for each depot are combined, so that the routing information can be fed back to the future customer assignments [60]. Oliveira et al. proposed a cooperative co-evolutionary algorithm, which divides all the customers among the depots, and then solves the VRP sub-problem associated to each depot (along with its allocated customers) by a parallel evolution strategy [33]. Lalla-Ruiz et al. decomposed the multi-depot VRP into sub-problems by a partial optimization meta-heuristic under special intensification conditions (called POPMUSIC), in which the sub-problem is built from the selected seed part [61]. Besides, there have been various approaches to the extended versions of multi-depot VRP [62]–[65] and DC approaches to large scale VRP with a single depot, e.g., [66]–[68].

However, the existing approaches for VRP cannot be directly applied to LSMDCARP due to two reasons: (1) they are designed for solving either multi-depot or large scale VRP, but not both; (2) CARP tends to be much larger and harder to be solved than the VRP counterpart [11]. For example, a CARP with $|T|$ tasks may be converted to a corresponding VRP with $2|T| + 1$ vertices.

In summary, all the existing studies for MDCARP focus on the small scale instances, and solve the problem as a whole. So far, there has been no effective approach to address the scalability issue for LSMDCARP. On other hand, although the DC strategy has achieved great success in tackling LSCARP effectively, these approaches are not directly applicable to LSMDCARP. For example, they consider only a single depot, [22], [24], [27], and cannot effectively allocate the tasks/routes among different depots. In this paper, we investigate the MDCARP, and propose a route clustering and search heuristic (RoCaSH) for it. The proposed RoCaSH employs a novel route clustering scheme and a Cutting Off Operator (RCO) to allocate the tasks to the depots effectively, and then uses the Ulusoy’s split operator as well as an iterative local search procedure to boost the optimization performance of the sub-problem associated to each depot. The details of RoCaSH is described in Section III.

### III. Route Clustering and Search Heuristic

It is natural for us to use the DC strategy to decompose the LSMDCARP into several smaller single-depot CARPs due to the structure of the problem. However, there are three key problems which must be solved. The first is the development of the decomposition scheme. It is well-known that an appropriate decomposition scheme is very important for the complex problem, as good scheme can gather the elements which are intense and peel off those loose ones. The second is the design of the optimization method for the CARP sub-problems obtained by the decomposition scheme. The third is how to design effective co-operations among the optimization process of the CARP sub-problems. We propose the RoCaSH to LSMDCARP to address the above three issues.

#### A. The Overall Framework

The overall framework of RoCaSH is described in Algorithm 1. The flowchart is given in Fig. 1 (along with how the solution changes at each step). It is an individual-based search algorithm, which hybridizes a route clustering scheme to decompose the problem into smaller sub-problems and a local search to solve each sub-problem separately. The route clustering and local search are interleaved with each other. On one hand, the route clustering can help the local search improve the solution more efficiently. On the other hand, the
improved solution by the local search can give feedback to further improve the route clustering.

In Algorithm 1, a solution \( S \) is first initialized heuristically (line 2). The solution is composed of a set of \(|V(D)|\) sub-solutions \( (S_1, \ldots, S_{|V(D)|}) \), where \( S_d (d = 1, \ldots, |V(D)|) \) is a set of routes associated with a depot \( v_d^{(D)} \). During the search process, the average rank \( \text{rank}(S) \) of the links in \( S \) is calculated in line 5. Then, for each depot, the routes are first cut into sub-routes by the route cutting off operator \([69]\), which tends to cut the poor (long distance) links rather than the promising (short distance) links (lines 6–11). Then, the sub-routes of all the depots are re-clustered to the depots by the Ulusoy’s splitting procedure \([8]\) and local search (lines 13–18). The whole algorithm stops when the maximal number of iterations or the maximal number of iterations without improvement is reached. Note that throughout the entire process, the order of the routes in the (sub-)solutions are always retained. In other words, we maintain the routes as lists rather than sets in the algorithm. This way, we expect to improve the search efficiency without losing much useful information. In the following, we will describe the initialization, route cut off operator, route clustering scheme, and local search in more detail.

### B. Initialization

The pseudo-code of the solution initialization is shown in Algorithm 2. At first, each task \( t \in T \) is allocated to a random depot, and a single-task route \( (x^{(D)}_d, x(t), x^{(D)}_d) \) is formed, where \( x(t) \) is a random direction (i.e. task ID) of \( t \). Then, all the single-task routes are re-clustered to the depots by the route clustering scheme. After that, the giant route concatenated by all the single-task routes undergoes the Ulusoy’s splitting procedure and local search for each depot.

### C. Route Cutting Off Operator

It has been shown that grouping the routes can ensure that the optimal solution under the new grouping is no worse than the current solution \([22]\). However, in practice it may not be sufficiently effective, especially when the routes contain tasks that are connected by poor links (i.e. with long distance), and should have been served in different routes or even allocated to different depots. To address the above issue, we employ the route cutting off operator \([69]\) before reclustering the routes to cut the routes into sub-routes. The route cutting off operator

---

**Algorithm 1. The Route Clustering and Search Heuristic**

1: procedure RoCaSH (The input instance with task set \( T \), depot set \( V_D \), maximum iterations \( \text{MaxIter} \), maximum iterations without improvement \( \text{MaxNoImp} \))
2: Initialize \( S = (S_1, \ldots, S_{|V(D)|}) = \text{init}(T, V(D)) \);
3: \( \text{iter} = 1, \text{noimp} = 0 \);
4: while \( \text{iter} \leq \text{MaxIter} \) and \( \text{noimp} \leq \text{MaxNoImp} \) do
5:  Calculate the average rank \( \text{rank}(S) \) of the links in \( S \);
6:  for \( d = 1 \to |V(D)| \) do
7:    \( R_d = () \);
8:    for each route \( R \) in \( S_d \) do
9:      Cut \( R \) into sub-routes using the route cutting off operator based on \( \text{rank}(S) \), and add the obtained sub-routes to the end of \( R_d \) in order;
10:  end for
11:  ClusterRoute\( (R_1, \ldots, R_{|V(D)|}) \);
12:  for \( d = 1 \to |V(D)| \) do
13:    Build a giant route by merging all the routes \( R \in R_d \) in order;
14:    Apply 2-OPT on the giant route to obtain better one;
15:    Split the giant route with Ulusoy’s splitting procedure \([8]\) to obtain the sub-solutions \( S'_d \);
16:    Improve \( S'_d \) with the local search;
17:  end for
18:  Set \( S' = (S'_1, \ldots, S'_{|V(D)|}) \);
19:  if \( S' \) is better than \( S \) then
20:    \( S = S' \), \( \text{noimp} = 0 \);
21:  else
22:    \( \text{noimp} = \text{noimp} + 1 \);
23:  end if
24:  \( \text{iter} = \text{iter} + 1 \);
25: end while
26: return the best feasible solution \( S \);
27: end procedure
Algorithm 2 The solution initialization

1: procedure Init (task set \( T \), depot set \( V^{(D)} \))
2: for \( d = 1 \rightarrow |V^{(D)}| \) do
3: Initialize a route list \( R_d = () \);
4: end for
5: for each task \( t \in T \) do
6: Randomly select an ID (direction) \( x(t) \);
7: Randomly select \( d \in \{1, \ldots, |V^{(D)}|\} \);
8: Create a route \( R(t) = (x_d^{(D)}), x(t), x_d^{(D)} \);
9: \( R_d = (R_d, R(t)) \);
10: end for
11: ClusterRoute(\( R_1, \ldots, R_{|V^{(D)}|} \));
12: for \( d = 1 \rightarrow |V^{(D)}| \) do
13: Build a giant route by merging all the routes \( R \in R_d \) in order;
14: Split the giant route with Ulsoy’s splitting procedure to obtain the sub-solutions \( S_d \);
15: Improve \( S_d \) with the local search with 2-OPT;
16: end for
17: return \( S = (S_1, \ldots, S_{|V^{(D)}|}) \);
18: end procedure

Algorithm 3 Route cutting off operator

1: procedure RCO (a route \( R \), the average rank \( \text{rank}(S) \), cutting probabilities \( \lambda \) and \( \theta \))
2: Set good links \( L_g = \emptyset \), poor links \( L_p = \emptyset \);
3: for each link \( (t_1, t_2) \in R \) do
4: Sort all the links \( \{(t_1, t')|t' \in T\} \) from shortest to longest, and get the rank of \((t_1, t_2)\);
5: end for
6: for each link \((t_1, t_2) \in R \) do
7: if the rank of \((t_1, t_2)\) is better than \( \text{rank}(R) \) then
8: \( L_g = L_g \cup (t_1, t_2) \);
9: else
10: \( L_p = L_p \cup (t_1, t_2) \);
11: end if
12: end for
13: Randomly sample \( r_1 \in U[0, 1] \);
14: if \( r_1 < \lambda \) then
15: Randomly select a link \( L_g \) and split \( R \) from it;
16: end if
17: Randomly sample \( r_2 \in U[0, 1] \);
18: if \( r_2 < \theta \) then
19: Randomly select a link from \( L_p \) and split \( R \) from it;
20: end if
21: return the resultant sub-routes;
22: end procedure

where the distance between two routes \( R_1 \) and \( R_2 \) is defined as

\[
\delta(R_1, R_2) = \frac{1}{4}(\varphi(tv(R_1), tv(R_2)) + \varphi(tv(R_1), hv(R_2))
+ \varphi(hv(R_1), tv(R_2)) + \varphi(hv(R_1), hv(R_2))),
\]

(13)

and split from it;

D. Route Clustering Scheme

The pseudo code of the route clustering scheme is given in Algorithm 4. Given an ordered list of routes, the scheme uses three criteria to re-cluster the routes. First, the routes associated to the depots are reset. That is, each depot \( v_d^{(D)} \) is associated with a single empty route \( R_d = (t_d^{(D)}, t_d^{(D)}) \), and all the given routes are unassigned. Then, for each unassigned route \( R \), the distance from the route to each depot is defined as the average distance from the route to the routes already assigned to that depot, i.e.,

\[
\Delta_d(R) = \frac{1}{|R_d|} \sum_{R_d \in R_d} \delta(R, R_d),
\]

(12)

splits the routes according to the quality of the links. After the cut, the sub-routes tend to maintain the better links (i.e., with shorter deadheading costs than the other links with shared end-nodes). The subsequent route clustering scheme is then conducted on the sub-routes rather than the original entire routes. This way, we expect to achieve more fine-grained allocation of the tasks to the depots.

Algorithm 3 describes the route cutting off operator. It takes a route \( R \), the average rank \( \text{rank}(S) \) (calculated in line 5 of Algorithm 1), and two parameters \( \lambda \) and \( \theta \) as inputs. First, for each link, we sort the links from the starting task to all the other tasks from shortest to longest, and get the rank of the link (ties are broken by sharing the rank). Then, the links of \( R \) are grouped into good and poor links based on their ranks relative to \( \text{rank}(S) \). Afterwards, the route \( R \) has a (smaller) probability \( \lambda \) of being split by breaking a good link and a (larger) probability \( \theta \) of being split by breaking a poor link. As a result, a route may be split into at most 3 sub-routes (when breaking both a good link and a poor link).

More details of RCO can be found from [69].

In RCO, the most time consuming step seems to be the sorting process in line 4, whose time complexity is \( O(|T| \log(|T|)) \). However, in practice one can calculate the rank of each task externally and store the ranks globally. Therefore, the complexity of line 4 can be reduced to \( O(1) \) in practice. Overall, the complexity of the RCO operator is \( O(|T|) \).

The pseudo code of the route clustering scheme is given in Algorithm 4. Given an ordered list of routes, the scheme uses three criteria to re-cluster the routes. First, the routes associated to the depots are reset. That is, each depot \( v_d^{(D)} \) is associated with a single empty route \( R_d = (t_d^{(D)}, t_d^{(D)}) \), and all the given routes are unassigned. Then, for each unassigned route \( R \), the distance from the route to each depot is defined as the average distance from the route to the routes already assigned to that depot, i.e.,
Algorithm 4 The three-criteria route clustering procedure

1: procedure ClusterRoute (route lists \( \mathcal{R}_1, \ldots, \mathcal{R}_{V(O)} \))
2: Set the unassigned route list \( \mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_{V(D)}) \);
3: for \( d = 1 \rightarrow |V(D)| \) do
4: Construct an empty route \( R_{d,0} = (x_d^{(D)}, x_d^{(D)}) \);
5: \( \mathcal{R}_d = (R_{d,0}) \);
6: end for
7: while \( \mathcal{R} \neq \emptyset \) do
8: Set candidate routes \( \Pi_1 = \emptyset \); \( ^\triangleright \) first criterion
9: for each route \( R \in \mathcal{R} \) do
10: for \( d = 1 \rightarrow |V(D)| \) do
11: \( \Delta_d(R) = \frac{1}{|\mathcal{R}_d|} \sum_{R_d \in \mathcal{R}_d} \delta(R, R_d) \);
12: end for
13: Calculate the gap \( \theta_1(R) \) between the smallest and second smallest values in \( \{ \Delta_d(R) \mid d = 1, \ldots, |V(D)| \} \);
14: if \( \theta_1(R) \geq 10\% \) then
15: \( \Pi_1 = \Pi_1 \cup R \);
16: end if
17: end for
18: if \( \Pi_1 \neq \emptyset \) then
19: Select the route with max gap \( R^* = \arg \max_{R \in \Pi_1} \theta_1(R) \);
20: Select the closest route list \( d^* = \arg \min_d \Delta_d(R^*) \);
21: \( \mathcal{R}_{d^*} = (\mathcal{R}_{d^*}, R^*) \), \( \mathcal{R} = \mathcal{R} \setminus R^* \);
22: goto line 7;
23: end if
24: \( \Pi_2 \leftarrow \emptyset \); \( ^\triangleright \) second criterion
25: for each route \( R \in \mathcal{R} \) do
26: for \( d = 1 \rightarrow |V(D)| \) do
27: \( \sigma_d(R) = \text{var}(\delta(R, R_d) \mid R_d \in \mathcal{R}_d) \);
28: Calculate the gap \( \theta_2(R) \) between the smallest and second smallest values in \( \{ \sigma_d(R) \mid d = 1, \ldots, |V(D)| \} \);
29: if \( \theta_2(R) \geq 40\% \) then
30: \( \Pi_2 = \Pi_2 \cup R \);
31: end if
32: end for
33: end if
34: if \( \Pi_2 \neq \emptyset \) then
35: Select \( R^* = \arg \max_{R \in \Pi_2} \theta_2(R) \);
36: Select the closest route list \( d^* = \arg \min_d \sigma_d(R^*) \);
37: \( \mathcal{R}_{d^*} = (\mathcal{R}_{d^*}, R^*) \), \( \mathcal{R} = \mathcal{R} \setminus R^* \);
38: goto line 7;
39: end if
40: for each route \( R \in \mathcal{R} \) do \( ^\triangleright \) third criterion
41: Calculate the closest distance \( \Delta_{\min}(R) = \min_d \Delta_d(R) \);
42: end for
43: \( R^* = \arg \min_{R \in \mathcal{R}} \Delta_{\min}(R) \);
44: Select the closest route list \( d^* = \arg \min_{R} \Delta_d(R^*) \);
45: \( \mathcal{R}_{d^*} = (\mathcal{R}_{d^*}, R^*) \), \( \mathcal{R} = \mathcal{R} \setminus R^* \);
46: end while
47: remove the empty routes \( R_{d,0} \) \( (d = 1, \ldots, |V(D)|) \);
48: end procedure

E. Local Search

After the route clustering, merge and Ulusoy’s split, the solution is further improved by the local search. In the local search, we use the single-insertion (SI), double-insertion (DI) and swap operators, which are commonly used move operators for CARP. If the best neighbour is better than the current solution, then it replaces the current solution, and the search continues. Otherwise, the search stops, and the current solution is returned. Since the three employed operators have small search step size, the search can only be done within a limited neighborhood of the incumbent solution. To enlarge the search space, before the Ulusoy’s split, we apply another local search process on the giant route (using the 2-OPT operator to distinguish from the three basic move operators, to avoid getting stuck into the same local optimum).

F. Complexity Analysis

Overall, the complexity of RoCaSH consists of that of the RCO operator, route clustering and local search. The complexities of the RCO operator and route clustering are \( O(|T|^2) \) and \( O(|R|) \), respectively. The local search is the most time consuming step, with a complexity of \( O(L_{ls} |T|^2) \). The preprocessing stage contains the calculations of the shortest distance between each pair of the nodes by Dijkstra’s algorithm [40] (complexity of \( O(|V|^3) \)) and the rankings of the task (complexity of \( |T|^2 \log(|T|) \)). However, in practice the complexity of the preprocessing stage can be mostly ignored when comparing with the search process. Therefore, the overall complexity of RoCaSH is \( O(\text{MaxIter} \times L_{ls} \times |T|^2) \), where \( \text{MaxIter} \) is the maximal number of iterations, and \( L_{ls} \) is the expected steps during the local search.

IV. EXPERIMENTAL STUDIES

To verify the effectiveness of the proposed RoCaSH, we carried out experiments on a range of MDCARP datasets with different problem sizes. In the experiments, RoCaSH was compared with the Hybrid Genetic Algorithm (HGA) [47] and the Multi-Depot Memetic Algorithm (MDMA) [19], which are the two existing state-of-the-art algorithms for MDCARP. Note that another Hybrid Ant Colony Algorithm (HACA) was also proposed in [19]. However, it showed much worse performance than MDMA in the experimental studies. Thus, we omitted the comparison with HACA in our experimental studies.

HGA and MDMA focused mainly on the small-scale MDCARP instances, and thus are not divide-and-conquer methods. However, to the best of our knowledge, there is no existing study particularly for large scale MDCARP. The comparison with HGA and MDMA can also demonstrate the effectiveness of the proposed decomposition strategy.

A. Datasets

So far, there is no benchmark dataset specifically for MDCARP. Instead, the existing studies for MDCARP extended the single-depot CARP instances to multi-depot ones. For example, in [19], the \( gdb \) small scale CARP instances are converted into MDCARP instances. Specifically, each \( gdb \)
instance is converted into a double-depot mdgdb instance by switching the last vertex into the second depot.

In our experimental studies, we consider a wide range of instances with different problem sizes, and particularly focus on the large problem instances. Specifically, we consider the mdgdb instances generated in [19], as well as convert the larger val, egl, EGL-G, Hefei and Beijing instances into multi-depot ones following the idea in [19].

The gdb set consists of 23 small size instances with 11–55 edges. The val set contains 34 instances with 34 to 97 edges. The egl set consists of 24 instances with 98–190 edges, which are generated from the network of roads in Lancashire, U.K. All of the 24 instances are based on two graphs and constructed by varying required edges and capacity constraints. Likewise, The EGL-G dataset consists of 10 instances, which are also derived from the road network of Lancashire, UK, but with 375 edges. The dataset contains two groups G1 and G2, each with 5 instances. The instances belonging to the same group have the same task set, and different vehicle capacities. The Hefei and Beijing datasets were generated by Tang et al. from the maps of the two cities in China [27]. The Hefei dataset contains ten instances sharing a graph with 1212 edges, and the Beijing dataset also consists of ten instances sharing a graph with 3584 edges. The tasks for the instances of each dataset are randomly selected from the edge set, while the numbers of tasks are set from 10% to 100% of the number of edges.

The MDCARP instances are generated from the single-depot CARP instances based on the following steps.

- First we set the number of depots for the generated datasets based on the minimum number of vehicles required for the problems. Specifically, the mdgdb instances have 3 depots, and the mdegl and mdEGL-G instances have 4 and 8 depots respectively. The number of depots is set to 5 for the mdHefei and mdBeijing instances.
- Then, we set the depots uniformly among the vertices. The depot set contains the first vertex, as well as the vertices whose indices are \( k \cdot \left\lfloor \frac{\left| V \right|}{m-1} \right\rfloor \), where \( m \) is the number of depots, and \( k = 1, \ldots, m-1 \). For example, if there are 50 vertices (index from 1 to 50) and 3 depots, then the depots contain the nodes 1, 25 and 50.
- To be consistent with [19], the depot with the maximum index is replaced by the last vertex for the mdgdb dataset.

**B. Parameter Settings**

The parameter settings of the proposed RoCaSH are listed in Table I. The maximal number of iterations and maximal iterations without improvement are set to 5000 and 1000 respectively. For the RCO operator (Algorithm 3), the probabilities of cutting off the good and poor links are set to 0.1 and 0.5.

The stopping criteria of the compared HGA and MDMA are set according to the settings used in the original papers. Specifically, for HGA, the maximal number of iterations is 10000, and the maximal iterations without improvement is 2000. For MDMA, the maximal number of iterations is 5000. All the other parameters of the compared algorithms are set the same as the original papers [19], [47]. Furthermore, we replace the original pure random initialization of HGA, which leads to very poor initial solutions, with the heuristic initialization of RoCaSH to remove the biased effect of the initialization on the final performance in the comparison.

In the experiments, all compared algorithms are coded with C++ and run on Intel Core i5-7500 with 3.4 GHz. For each instance, each of the compared algorithms were run 30 times independently.

**C. Results and Discussions**

First, we examine the performance of RoCaSH on the mdgdb, mdval, mdegl and mdEGL-G datasets. These datasets have their problem sizes are from 11 edges to 375 edges. The results are given in Tables II–V. In each table, the columns headed “|V|”, “[E]”, “[T]” and “Q” indicate the number of vertices, edges, tasks and the capacity of vehicle. For each algorithm, the columns “Average”, “Std” and “Time” are the average and standard deviation of the total costs and average runtime (in seconds) over 30 independent runs. The minimal results are marked in bold. In addition, statistical test is carried out between RoCaSH and each other compared algorithm with Wilcoxon rank sum test under the significance level of 0.05. If an algorithm is significantly worse (better) than RoCaSH, then it is marked with “−” (“+”). At the bottom of the table, the “Mean” row provides the mean values of the results over all the instances in the test set, and the “B-C-W” row indicates the number of instances on which RoCaSH performed significantly better (than) C, comparable with (“C”), and significantly worse than (“W”) the corresponding algorithm.

From Table II, one can see that RoCaSH significantly outperformed HGA (MDMA) on 17 (10) out of the 23 mdgdb instances. Moreover, RoCaSH obtained the minimal mean value over all the instances in the mdgdb set (e.g. 251.15 for RoCaSH, 251.43 for MDMA and 259.99 for HGA). It is notable that the runtime of RoCaSH on most instances is below one second. This indicates that RoCaSH can converge much faster than the compared algorithm.

To give a more intuitive comparison between RoCaSH and other compared algorithms, some representative instances are selected from the mdgdb set, and the convergence curves of RoCaSH, MDMA and HGA on them are shown in Fig. 2. In the figure, the x-axis is the runtime, and the y-axis is the mean total cost of the best-so-far solutions over the 30 runs. Note that the 30 runs may have different runtime (e.g., one run reached the local optimum very quickly and stopped early with the stopping criterion of 1000 iterations without improvement, while another run reached the 10000 maximal number of iterations). For each algorithm, the length of its convergence curve is decided by the run with the longest runtime. For the shorter runs, we simply use its final total cost to calculate the mean total cost at any time later than its stopping time.

To better distinguish the difference between the algorithms, each curve starts from the total cost obtained after the first iteration, i.e., iteration zero is ignored in the figures, as the algorithms produce almost the same initial solutions.
TABLE II
The average performance over 30 independent runs of the compared algorithms on the mdgdb dataset. For each instance, the minimal mean total cost is marked in bold. Under Wilcoxon rank sum test with significance level of 0.05, an algorithm is marked with “-” (“+”) if it is significantly worse (better) than RoCaSH.

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>HGA</th>
<th>MDMA</th>
<th>RoCaSH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V</td>
<td>E</td>
<td>T</td>
<td>Q</td>
<td>Average</td>
<td>Std</td>
<td>Time (s)</td>
</tr>
<tr>
<td>mdgdb1</td>
<td>12</td>
<td>22</td>
<td>22</td>
<td>5</td>
<td>317.17</td>
<td>5.80</td>
<td>2.68</td>
</tr>
<tr>
<td>mdgdb2</td>
<td>12</td>
<td>26</td>
<td>26</td>
<td>5</td>
<td>330.00</td>
<td>6.76</td>
<td>2.86</td>
</tr>
<tr>
<td>mdgdb3</td>
<td>12</td>
<td>22</td>
<td>22</td>
<td>5</td>
<td>272.97</td>
<td>8.44</td>
<td>2.60</td>
</tr>
<tr>
<td>mdgdb4</td>
<td>11</td>
<td>19</td>
<td>19</td>
<td>5</td>
<td>291.83</td>
<td>11.58</td>
<td>2.30</td>
</tr>
<tr>
<td>mdgdb5</td>
<td>13</td>
<td>26</td>
<td>26</td>
<td>5</td>
<td>390.63</td>
<td>14.34</td>
<td>2.87</td>
</tr>
<tr>
<td>mdgdb6</td>
<td>12</td>
<td>22</td>
<td>22</td>
<td>5</td>
<td>305.63</td>
<td>9.50</td>
<td>2.63</td>
</tr>
<tr>
<td>mdgdb7</td>
<td>12</td>
<td>22</td>
<td>22</td>
<td>5</td>
<td>325.80</td>
<td>9.50</td>
<td>2.63</td>
</tr>
<tr>
<td>mdgdb8</td>
<td>27</td>
<td>46</td>
<td>46</td>
<td>27</td>
<td>375.73</td>
<td>8.44</td>
<td>4.12</td>
</tr>
<tr>
<td>mdgdb9</td>
<td>27</td>
<td>51</td>
<td>51</td>
<td>27</td>
<td>324.57</td>
<td>9.25</td>
<td>3.95</td>
</tr>
<tr>
<td>mdgdb10</td>
<td>12</td>
<td>25</td>
<td>25</td>
<td>35</td>
<td>295.00</td>
<td>0.00</td>
<td>2.57</td>
</tr>
<tr>
<td>mdgdb11</td>
<td>22</td>
<td>45</td>
<td>45</td>
<td>50</td>
<td>406.60</td>
<td>2.39</td>
<td>3.26</td>
</tr>
<tr>
<td>mdgdb12</td>
<td>13</td>
<td>23</td>
<td>23</td>
<td>10</td>
<td>454.73</td>
<td>9.80</td>
<td>2.57</td>
</tr>
<tr>
<td>mdgdb13</td>
<td>10</td>
<td>28</td>
<td>28</td>
<td>41</td>
<td>544.70</td>
<td>4.14</td>
<td>2.83</td>
</tr>
<tr>
<td>mdgdb14</td>
<td>7</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>100.40</td>
<td>1.23</td>
<td>2.57</td>
</tr>
<tr>
<td>mdgdb15</td>
<td>7</td>
<td>21</td>
<td>21</td>
<td>37</td>
<td>60.73</td>
<td>1.31</td>
<td>2.12</td>
</tr>
<tr>
<td>mdgdb16</td>
<td>8</td>
<td>28</td>
<td>28</td>
<td>27</td>
<td>126.77</td>
<td>1.94</td>
<td>2.68</td>
</tr>
<tr>
<td>mdgdb17</td>
<td>8</td>
<td>28</td>
<td>28</td>
<td>27</td>
<td>92.53</td>
<td>0.85</td>
<td>2.50</td>
</tr>
<tr>
<td>mdgdb18</td>
<td>9</td>
<td>36</td>
<td>36</td>
<td>27</td>
<td>162.10</td>
<td>1.30</td>
<td>2.93</td>
</tr>
<tr>
<td>mdgdb19</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>27</td>
<td>55.20</td>
<td>0.60</td>
<td>2.00</td>
</tr>
<tr>
<td>mdgdb20</td>
<td>11</td>
<td>22</td>
<td>22</td>
<td>27</td>
<td>127.87</td>
<td>2.16</td>
<td>2.33</td>
</tr>
<tr>
<td>mdgdb21</td>
<td>11</td>
<td>33</td>
<td>33</td>
<td>27</td>
<td>163.40</td>
<td>2.35</td>
<td>3.01</td>
</tr>
<tr>
<td>mdgdb22</td>
<td>11</td>
<td>44</td>
<td>44</td>
<td>27</td>
<td>205.03</td>
<td>1.05</td>
<td>3.15</td>
</tr>
<tr>
<td>mdgdb23</td>
<td>11</td>
<td>55</td>
<td>55</td>
<td>27</td>
<td>242.27</td>
<td>1.84</td>
<td>4.08</td>
</tr>
</tbody>
</table>

Mean
B-C-W
259.99
17-6-0
2.83
251.43
10-8-5
3.58
251.15

Fig. 2. Convergence curves on some mdgdb instances.

From Fig. 2, one can see that the convergence curves of RoCaSH are below those of other two compared algorithms on most instances excluding the mdgdb1. Especially, on mdgdb16 and mdgdb23 instances, there are obvious gaps the convergence curves of RoCaSH and others.

From Table III, it is clear to see that RoCaSH performed much better than HGA on the mdval set. It performed slightly poorer than MDMA in terms of the mean values over all 34 instances (i.e. 331.18 for RoCaSH, while 329.53 for MDMA). However, there is no statistical significant difference (13 wins and 13 losses). In addition, the mean runtime for RoCaSH to converge is much less than those of MDMA and HGA (1.06 sec for RoCaSH and 12.37 sec for MDMA). From Table IV, we can see RoCaSH significantly outperformed HGA and MDMA on the most (17 out of 24) mdegl instances. For the mean total cost over all 24 instances, RoCaSH reached the minimal value among all the three compared algorithms. RoCaSH was also much less time consuming than that of MDMA. Though HGA had slightly shorter runtime, it showed very poor performance in terms of total cost.

The convergence curves of the compared algorithms on some representative mdegl instances are shown in Fig. 4. From the figure, it can be seen that the convergence curves of RoCaSH are below those of the other compared algorithms and the runtime of RoCaSH is also shorter than that of MDMA.

From Table V, we can see RoCaSH outperformed HGA on
10 \textit{mdEGL-G} instances and MDMA on 8 \textit{mdEGL-G} instances. There is no instance on which RoCaSH obtained worse results. Moreover, RoCaSH obtained much better mean total cost over the 10 instances than the other algorithms. The mean runtime of RoCaSH is slightly longer than that of HGA, but still much shorter than that of MDMA.

Fig. 5 shows the convergence curves of the three compared algorithms on selected representative \textit{mdEGL-G} instances. From the figure, it can be seen that the convergence curves of RoCaSH are below those of both compared algorithms on all the four selected instances. For three instances (i.e. \textit{mdEGL-G1-E}, \textit{mdEGL-G2-C} and \textit{mdEGL-G2-E}), the advantage was shown from the very beginning of the search process.

D. The performance on large scale problems

As the problem size increases, it becomes harder to set the number of iterations for the compared algorithms to lead to similar runtime. Therefore, in the experiments on the large \textit{mdBeijing} and \textit{mdBeijing} datasets, we set the same time budget for the compared algorithms to make fair comparison. Specifically, the time budget for an instance is set as 0.2 × |T| seconds, where |T| is the number of the instance. For example, if an instance contains 1210 tasks, the time budget is set to 0.2 × 1210 = 242 seconds.

The results are shown in Tables VI and VII. From the tables, we can clearly see that RoCaSH achieved significantly better results than HGA and MDMA on all the \textit{mdBeijing} and \textit{mdBeijing} instances.

Figs 6 and 7 show the convergence curves of the three compared algorithms on selected representative \textit{mdBeijing} instances. Since the average results obtained by HGA are far worse than RoCaSH, it is difficult to put the curves of three compared algorithms together. Therefore, the figures only consist of two curves of MDMA and RoCaSH for each representative instance. From the figures, one can see that the convergence curves of RoCaSH are obviously below those of MDMA on all the eight selected instances. On some instances (e.g., \textit{mdBeijing-1}, \textit{mdBeijing-10}), RoCaSH showed the poor ability at beginning of the search process, but it converged very fast and caught up very soon.
In summary, from the analysis of the results on different datasets with a wide range of problem sizes (i.e. mdgdb, mdval, mdeg, mdEGL-G, mdHefei and mdBeijing), it can be seen that RoCaSH can tackle MDCARP well, especially for the large and complex instances, RoCaSH is less likely to be stuck into poor local optima, and can use the allowed computational budget more effectively than HGA and MDMA. From Figs 6 and 7, we can see that RoCaSH has a clear superiority on the large MDCARP test instances.
E. Scalability

To investigate the scalability of RoCaSH, we give the scatter plot of the ratio between the performance of RoCaSH and other compared algorithms (i.e., HGA, MDMA) versus the number of tasks on the test instances. The plot is shown in Fig. 8, where the x-axis is the number of tasks in the instance, and the y-axis is the ratio between the mean total costs of RoCaSH and the capacity constraints, and propose more effective problem.

The experimental studies on a wide range of MDCARP and other compared algorithms. For example, on mdegl-s4-C, the mean total costs of HGA, MDMA and RoCaSH are 12980.37, 11766.07 and 11737.13, thus the ratios of RoCaSH versus HGA and MDMA are 1.05 and 1.04, respectively.

From Fig. 8, it can be seen that for large instances (e.g., more than 1000 tasks), the ratio values are smaller than 1, indicating that RoCaSH performed better than HGA and MDMA. HGA or MDMA performed slightly better than RoCaSH (the ratios are between 1 and 1.05) only on some small instances (problem size smaller than 200). This verifies the scalability of RoCaSH on LSMDCARP instances.

V. CONCLUSIONS AND FUTURE WORK

The large scale multi-depot CARP (LSMDCARP) is a challenging problem due to its large solution space. This paper solves LSMDCARP based on a divide-and-conquer strategy, and proposed a Route Clustering and Search Heuristic (RoCaSH). Specifically, to address the challenge of allocating the tasks among the depots. We developed novel route cutting and clustering schemes in RoCaSH. Then, for the tasks associated to each depot, we employ the Ulusoy’s splitting procedure combined with the local search for solving single-depot LSCARP. The route clustering and search components interact with each other to solve the problem effectively. The experimental studies on a wide range of MDCARP instances demonstrate that the proposed RoCaSH significantly outperformed the state-of-the-art approaches in terms of both effectiveness and efficiency.

In the future, we will further investigate the interactions between tasks and routes under the problem constraints (e.g., the capacity constraints), and propose more effective problem.
decomposition methods. For example, we can focus more on the tasks and routes that are on the boundaries of multiple depots, i.e. which have similar distances to multiple depots, and re-cluster the routes based on the membership of the routes to different depots. We will also consider bi-level decomposition schemes that employ the divide-and-conquer strategy not only for allocating tasks/routes between different depots, but also for decomposing the sub-problems for each single depot (as known as large scale single-depot CARP [22]).

ACKNOWLEDGMENT

This work was supported by Anhui Provincial Natural Science Foundation (No. 1808085MF173) and Natural Science Key Research Project for Higher Education Institutions of Anhui Province (Nos. KJ2019A0554, KJ2016A438).

REFERENCES

algorithms versus the number of tasks on four datasets.

Fig. 7. Convergence curves on the mdBeijing instances.

Fig. 8. The ratio between the mean total costs of RoCaSH and other compared algorithms versus the number of tasks on four datasets.


