Automated Heuristic Design Using Genetic Programming Hyper-Heuristic for Uncertain Capacitated Arc Routing Problem

ABSTRACT

Uncertain Capacitated Arc Routing Problem (UCARP) is a variant of the well-known CARP. It considers a variety of stochastic factors to reflect the reality where the exact information such as the actual task demand and accessibilities of edges are unknown in advance. Existing works focus on obtaining a robust solution beforehand. However, it is also important to design effective heuristics to adjust the solution in real time. In this paper, we develop a new Genetic Programming-based Hyper-Heuristic (GPHH) for automated heuristic design for UCARP. A novel effective meta-algorithm is designed carefully to address the failures caused by the environment change. In addition, it employs domain knowledge to filter some infeasible candidate tasks for the heuristic function. The experimental results show that the proposed GPHH significantly outperforms the existing GPHH methods and manually designed heuristics. Moreover, we find that eliminating the infeasible and distant tasks in advance can reduce much noise and improve the efficacy of the evolved heuristics. In addition, it is found that simply adding a slack factor to the expected task demand may not improve the performance of the GPHH.

CCS CONCEPTS

•Mathematics of computing → Combinatorial optimization; **•Theory of computation → Evolutionary algorithms;** *Routing and network design problems;*

KEYWORDS

genetic programming, hyper-heuristic, uncertain capacitated arc routing problem

ACM Reference format:
. 2017. Automated Heuristic Design Using Genetic Programming Hyper-Heuristic for Uncertain Capacitated Arc Routing Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 8 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Capacitated Arc Routing Problem (CARP) [10] aims to determine a set of routes to serve the edge tasks with minimal costs subject to some predefined constraints. Many applications in the real world can be modelled as CARP such as street watering, snow removal [4] and waste collection [1]. Researches on CARP are helpful for improving the work efficiency in related service industries. Lots of methods, ranging from exact methods [2, 3] to heuristics [9, 20, 26] and meta-heuristics [5, 18, 21], have been proposed to solve CARP. Most of them show promising performance in both solution quality and running time [15], even for large-scale CARP instances [14].

Existing works mainly focus on static CARP [5, 15], i.e., the CARP that is formulated based on a static network with deterministic parameters such as task demands and deadheading costs along edges. However, the static CARP has a large gap from the practice, in which problem parameters are often stochastic and unknown before edges are served or traversed [16]. For example, in waste collection, the amount of garbage on streets may vary weekly and can only be known exactly after the service [23]. The CARP models with stochastic parameters is closer to reality.

Fleury et al. [7, 8] proposed a CARP model with stochastic demands, namely Stochastic CARP (SCARP), along with a Memtic Algorithm (MA) for solving it. Their work pushed the researches about CARP to a more realistic area, and several new algorithms for SCARP have been proposed since then (e.g. [6, 11]). To include more real-world factors, Mei et al. [16] proposed the Uncertain CARP (UCARP) with four stochastic parameters, i.e., presence of tasks, demands of tasks, presence of paths and traversal costs of paths. They gave the definition of robustness measures for solutions to UCARP and provided some preliminary solutions. Then Wang et al. proposed a MA [24] and an estimation of the distribution algorithm [23] for solving UCARP, respectively.

There are two obvious shortcomings in the existing works for solving UCARP. First, most existing approaches are not flexible in real-time adjustment. They optimise robust solutions based on expected values, and employ a straightforward recourse operator to apply the solutions to the realised scenarios. That is, if the actual demand of the current task is greater than expected, making it infeasible to complete the current service (so-called *route failure*), the vehicle will go back to the depot to refill, and then come back to continue the remaining service of the task. Such a recourse operation can lead to huge increase of the cost of final solutions in some realised scenarios. Second, most existing approaches to UCARP assume that the actual demand of the next task is known as soon as the vehicle completes serving the previous task in the same route (e.g. [23, 24]). With such an assumption, they can find more robust solutions by completely avoiding route failures. However, this assumption is not consistent with reality, in which the exact demand of a task cannot be known until the vehicle arrives its place, or even after finishing its service. How to efficiently solve UCARP without such an unrealistic assumption is a challenging problem.

UCARP can be seen as a form of dynamic routing problem. Pillac et al. [17] gave a comprehensive review on dynamic vehicle routing including a variety of methods to address it, including periodic reoptimisation and memory-based approaches. Ritzinger et al. [19] gave another survey which mentioned *routing policy* learning. Routing policy is a *heuristic* telling the vehicle what the next task to go as soon as it becomes idle. Due to the computational efficiency, it is particularly effective in dynamic problems where immediate response is required. Weise et al. [25] proposed a Genetic Programming Hyper-Heuristic (GPHH) to evolve routing policy for static CARP and tested on the uncertain CARP with unknown task set, i.e. stochastic presence of tasks. However, their heuristic design does not consider the stochastic demand, and thus cannot well cope with the *route failure* situation when the actual demand of the current task exceeds the remaining capacity. So far, there is no study on evolving heuristics for a general UCARP with various stochastic factors such as the problem defined in [16].

The goal of this paper is to develop a new GPHH approach for the general UCARP defined in [16]. To be specific, we have the following research objectives in this paper:

- Develop a new framework (*meta-algorithm*) for evolving heuristics, i.e. routing policies, for the UCARP with stochastic presences of tasks and edges as well as stochastic task demands and deadheading cost of edges;
- Develop a GPHH approach for evolving heuristics for the UCARP;
- Analyse the proposed meta-algorithm and the GP-evolved heuristics on the UCARP benchmark instances.

The rest of this paper is structured as follows. Section 2 presents the backgrounds including formal definition of UCARP and related work. Section 3 describes our approach for UCARP, especially the new meta-algorithm. Section 4 shows the experimental studies. Section 5 gives the conclusion and future work.

2 BACKGROUND

In this section, we first present the mathematical definition of UCARP and then discuss existing works on UCARP.

2.1 Problem Definition

A UCARP instance can be represented by a connected graph $G = \{V, E\}$, where V is the set of vertices and E is the set of edges. Each edge is indirect but can be seen as two arcs with opposite directions. Each edge e is associated with three features: a demand $d(e) \geq 0$, a serving cost $sc(e) \geq 0$ and deadheading cost $dc(e) \geq 0$. Deadheading cost means the cost of travelling along the edge without serving it. Edges with non-zero demand and serving cost are called *tasks*, which are represented by $T \subseteq E$. There is a depot node $v_0 \in V$. The capacity of vehicle is Q, which is smaller than the total demand of all the tasks. Hence, multiple routes are needed. This can be achieved by using multiple vehicles, each with one route, or one vehicle with several routes (refills). In the serving process, each edge can be traversed many times, but each task should be served only once, on either direction.

The differences between UCARP and static CARP lie in the stochastic factors about demand and deadheading cost. First, the demand of each task is unknown until a vehicle finishes serving it. If the demand equals 0, then the task is not needed to be served in the current environment, and can be removed from the current task set. Second, the deadheading cost of each edge is unknown until an vehicle travelled through the edge. The actual deadheading cost can be infinite, indicating that the edge becomes unaccessible (e.g. the street is under maintenance) and can not be traversed temporarily. In this situation, detour is needed. The accessibility of an edge can only be obtained when the vehicle arrives the head of the edge. When the deadheading cost of an edge is infinite, its demand must be zero, as it cannot be traversed or served.

The above two uncertain factors may lead to two kinds of uncontrollable failures in uncertain environments, called as *route failures* and *edge failures*. They are defined as follows:

- *Route failure*: the actual demand of the current task exceeds the remaining capacity of the vehicle. In this case, the vehicle has to go back to the depot to refill in the middle of the service, and then come back to complete the remaining service.
- *Edge failure*: the current edge to go is unaccessible, and the vehicle has to find a detour.

The goal of UCARP is to find a feasible heuristic (i.e. routing policy) that can generate solutions to satisfy all tasks at the lowest possible costs under all possible environments. A solution is represented by a set of routes. Under the vertex representation scheme [16], a solution is represented as $S = (X, Y)$, where $X = \{X^{(1)}, X^{(2)}, ..., X^{(m)}\}$ is route set in which each element $X^{(k)} = (x_1^{(k)}, ..., x_{L_k}^{(k)})$ represents the vertex sequence that a vehicle travels in one route, where L_k represents the number of vertices in the route k. $Y = \{Y^{(1)}, Y^{(2)}, ..., Y^{(m)}\}$ is a set of vectors, each for a route. $Y^{(k)} = (y_1^{(k)}, ..., y_{L_k-1}^{(k)})$, in which $y_i^{(k)} = 1$ means that the edge $(x_i^{(k)}, x_{i+1}^{(k)})$ is a task and is being served. On the contrary, $y_i^{(k)} = 0$ means that the vehicle travels through the edge $(x_i^{(k)}, x_{i+1}^{(k)})$ without serving it. Finally, due to the existence of the route failures, we allow a continuous value $y_i^{(k)} \in (0, 1)$ in the solution, indicating that the edge $(x_i^{(k)}, x_{i+1}^{(k)})$ is a task and being served partially in the current position. In this case, $y_i^{(k)}$ is the fraction of demand served before the vehicle goes back to the depot.

The problem formulation of UCARP is given in Eqs. (1)–(7). Eq. (1) is the objective function, which can be either the average or worst-case performance over all the possible realised instances. There are three constraints in UCARP. First, a vehicle must start from and end at the depot in each route, as shown in Eq. (2). Second, all tasks should be served exactly once by one vehicle, as shown in Eqs. (3)-(6), where $T(\xi)$ and $|T(\xi)|$ are the task set and the number of tasks under environment ξ, and $k_1 \neq k_2$ or $i_1 \neq i_2$ in Eqs. (5) and (6). Third, the total demand of each route must not exceed the vehicle's capacity, as shown in Eq. (7).

$$\min \ F(S), \tag{1}$$

$$s.t. \ \ x_1^{(k)} = x_{L_k}^{(k)} = v_0, \ \forall k = 1, 2, ..., m, \tag{2}$$

$$\sum_{k=1}^{m} \sum_{i=1}^{L_k-1} y_i^{(k)} = |T(\xi)|, \tag{3}$$

$$(x_i^{(k)}, x_{i+1}^{(k)}) \in T(\xi), \ \forall y_i^{(k)} \geq 0, \tag{4}$$

$$(x_{i_1}^{(k_1)}, x_{i_1+1}^{(k_1)}) \neq (x_{i_2}^{(k_2)}, x_{i_2+1}^{(k_2)}), \ \forall y_{i_1}^{(k_1)} = y_{i_2}^{(k_2)} = 1, \tag{5}$$

$$(x_{i_1}^{(k_1)}, x_{i_1+1}^{(k_1)}) \neq (x_{i_2+1}^{(k_2)}, x_{i_2}^{(k_2)}), \ \forall y_{i_1}^{(k_1)} = y_{i_2}^{(k_2)} = 1, \tag{6}$$

$$\sum_{i=1}^{L_k-1} d(x_i^{(k)}, x_{i+1}^{(k)}) \cdot y_i^{(k)} \leq Q, \ \forall k = 1, 2, ..., m, \tag{7}$$

As per the objective function, the average and worst-case performance measures are defined in Eq. (8) and Eq. (9) respectively, where $C(S, \xi)$ is the cost of solution S in the environment ξ, and $|\Xi|$ is the number of uncertain environments for a static CARP.

$$F_1 = \frac{\sum_{\xi \in \Xi|} C(S, \xi)}{|\Xi|} \tag{8}$$

$$F_2 = \max_{\xi \in \Xi} C(S, \xi) \tag{9}$$

2.2 Related Work

Motivated by real scenarios such as waste collection in which demands vary over time, Fleury et al. [8] first extended the static CARP to SCARP. In SCARP, the deterministic demands in static CARP are replaced by normally distributed random variables. They investigated the robustness of a static CARP solution under the stochastic demands. The experiments were conducted by calculating the cost enlargement rate between the stochastic instance and corresponding static instance. Experimental results show that using reduced vehicle capacity in algorithm calculation (called as slack approach) can generate more robust solutions than using full vehicle capacity directly (called as tight approach). This finding is useful for designing robust strategies for stochastic problems.

In order to solve SCARP, Fleury et al. [7] proposed a Stochastic Memetic Algorithm (SMA). SMA modifies the objective function and local search strategy on the basis of MA for static CARP by [13]. Experimental results show that SMA can generate higher quality and more robust solutions than the slack approach. To our knowledge, SMA is the first evolutionary algorithm proposed for solving SCARP, and their work opens this research area. Afterwards, lots of new algorithms have been proposed for solving SCARP. For example, in [6], SCARP was formulated as a set partitioning problem and an exact method based on branch-and-price algorithm was developed.

A closer model to reality than SCARP is the UCARP proposed by Mei et al. [16]. In UCARP, both paths and tasks being presence stochastically and both deadheading costs of paths and demands of tasks are stochastic variables. Along with the detailed definition of UCARP, they analyzed several robustness measures and proposed a performance measure for solutions to UCARP based on the average performance over all the possible environments (Eq. (8)).

To deal with UCARP, Wang et al. [23, 24] proposed two different evolutionary algorithms. In [24], Handa's MA [12] and the memetic algorithm with extended neighborhood search (MAENS) [22] were combined to form a new MA for UCARP. Experimental results in terms of the expected costs were given to show the high quality of the newly proposed MA. In [23], Wang et al. proposed an Estimation of Distribution Algorithm with Stochastic Local Search (EDASLS) for UCARP. Experimental results in terms of the worst-case solution costs (see Eq. (9)) showed that EDASLS outperforms the other existing algorithms. Their analysis showed that the excellent performance of EDASLS was mainly caused by putting the adjacent tasks closer to each other in the solutions.

Although Wang's algorithms take the stochastic factors of both demand and deadheading cost into consideration, they assume that the practical demand of a task can be obtained as soon as the vehicle finishes serving the previous task. Such an assumption prevents the route failures from occurring, since the vehicle can go back to the depot immediately after serving the previous task without visiting the next infeasible task. As a result, they obtained very promising results in terms of the worst-case performance over all the possible environments. However, the assumption of knowing the actual demand of tasks in advance is usually not true in reality. In this paper, we move a step forward towards the reality, and assume that *the actual demand of a task is not known until the vehicle finishes serving the task.*

Weise et al. [25] proposed a GPHH for evolving heuristics for static CARP and stochastic CARP with variable task sets. In their approach, GP was used to evolve a *heuristic function* that will be used to rank the tasks in the waiting list of a vehicle while deciding which task the vehicle should serve next. To this end, a meta-algorithm was designed, and a set of state attributes were carefully chosen as the terminal set of the GPHH. Compared to other existing methods such as the robustness optimisation methods in [23, 24], the GPHH has three advantages. First, the evolved heuristics make step-by-step decisions only using the current local information, and thus are very computationally efficient and scalable. Second, the evolved heuristics can be used to generate high-quality initial solutions in unseen future instances, to adapt to the environment change quickly. Third, by observing the evolved heuristic functions, we are given more chance to discover hidden knowledge on how to effective generate high-quality CARP solutions in different environments, which can be used to guide the method development for similar problems. Weise et al. also tested their algorithm on stochastic CARP with random task set, and the GP-evolved heuristics showed much better test performance than the manually designed heuristics. However, the GPHH only considered the random presence of tasks, but ignored the stochastic demand. As a result, it cannot cope with the route failures well. Furthermore, it does not consider the stochastic deadheading cost, and cannot deal with edge failure either.

3 GENETIC PROGRAMMING HYPER-HEURISTIC FOR UCARP

A GPHH consists of two key components [25]: (1) a *training set* and (2) a *meta-algorithm*. We develop a new framework of GPHH, as described as follows:

Step 1: (*Initialisation*) randomly generate a GP population with n heuristic functions.

Step 2: (*Evaluation*) For each heuristic function, evaluate the fitness by applying the meta-algorithm with the heuristic function to the training set, e.g. as the average objective values of the generated solutions over the training set.

Step 3: If the stopping criteria are met, then return the heuristic function that has the best fitness in this process. Otherwise go to Step 4 to start a new generation.

Step 4: Generate a new population of offsprings using GP search operators. Then go to Step 2.

In our algorithm, we select the typical tree-based representation, and the traditional GP crossover and mutation operators during the evolution process.

3.1 Meta-algorithm

Meta-algorithm is the key component of the GPHH, as it is required for evaluating the GP individuals, which are heuristic functions. A meta-algorithm is expected to take any heuristic function and any UCARP instance, and return a feasible solution. Here, we employ the constructive heuristic-like meta-algorithm, which starts from empty routes, and adds one task to the end of a route at each step. The meta-algorithm constructs the routes sequentially. That is, it will not open a new route unless the current route is closed. This corresponds to the situation in which there is only one vehicle. For serving all the tasks, the vehicle goes back to the depot to refill its capacity when fully-loaded (closing the current route).

The meta-algorithm is described in Algorithm 1. During the initialisation stage (lines 1–3), the solution is initialised to be empty, and the first route $(X^{(1)}, Y^{(1)})$ is initialised to start from the depot. The current location *loc* of the vehicle is v_0, and the initial remaining capacity $\bar{Q}$ is Q. All the tasks are unserved initially. Then, the meta-algorithm uses the heuristic function $h(\cdot)$ to decide the next task that the vehicle should go (or go back to depot to refill) until all the tasks are served. For deciding the next destination of the vehicle, a subset of candidate tasks is firstly selected from the pool by the function `Filter()` (line 5). If no candidate task can be selected, then the vehicle goes back to the depot to refill, closes the current route and opens a new one (lines 7–9). Otherwise, the task u^* with the minimal heuristic value is selected to be served next, and the vehicles goes to its head node (lines 11–13). While serving the task, if the actual demand exceeds the remaining capacity (i.e. route failure occurs), the route is then repaired by the function `Refill()`. Otherwise, the task is served as normal (lines 17–18). Finally, the served task u^* and its opposite task $\hat{u}^*$ (i.e. $head(\hat{u}^*) = tail(u^*)$ and $tail(\hat{u}^*) = head(u^*)$) are removed from the set of unserved tasks, since an edge with positive demand is to be served only once in either direction.

The proposed meta-algorithm has three advantages:

1) It takes advantage of a *filter* function `Filter()` to select a subset of candidate tasks out of the entire pool. This way, one can remove the tasks that are obviously unpromising from consideration, and reduce the noise of the heuristic learning by GPHH;

2) It handles the *edge failure* by frequently detecting the change of the network connectivity (as will be shown in the description of the `GoTo`() function), and adjusting the subsequent decisions accordingly in an efficient way;

3) It deals with the *route failure* in two aspects. First, it adopts the straightforward return-and-refill operation (line 15) when a route failure occurs. Second, by carefully designing the `Filter()` function, one can potentially remove the tasks that are more likely to cause route failure in advance, and thus reduce the probability of having route failures.

Algorithm 1 The meta-algorithm of the GPHH for UCARP.

```
Input: Graph $G = (V, E)$, environment $\xi$, task set $T$, heuristic
    function $h(\cdot)$
Output: A feasible solution $S = (X, Y)$
 1: $X \leftarrow \emptyset$, $Y \leftarrow \emptyset$, $k = 1$;
 2: $X^{(k)} \leftarrow (v_0)$, $Y^{(k)} \leftarrow ()$, $loc \leftarrow v_0$, $\bar{Q} \leftarrow Q$;
 3: $U \leftarrow T$;                          ▷ $U$ is the pool of unserved tasks
 4: while $U \neq \emptyset$ do
 5:     $U' \leftarrow$ Filter($U$);
 6:     if $U' = \emptyset$ then                  ▷ Go back to depot and refill
 7:         GoTo($X^{(k)}, Y^{(k)}, v_0$);
 8:         $X \leftarrow X \cup X^{(k)}$, $Y \leftarrow Y \cup Y^{(k)}$, $k \leftarrow k + 1$;
 9:         $X^{(k)} \leftarrow (v_0)$, $Y^{(k)} \leftarrow ()$, $loc \leftarrow v_0$, $\bar{Q} \leftarrow Q$;
10:     else
11:         Calculate the heuristic value $h(u)$ for each task $u \in U'$;
12:         $u^* = \arg\min_{u \in U'} h(u)$;
13:         GoTo($X^{(k)}, Y^{(k)}, head(u^*)$);
14:         if $\bar{Q} < d_\xi(u^*)$ then                      ▷ Route failure
15:             Refill($X, Y, k, u^*$);
16:         else                                       ▷ Serve the task
17:             $X^{(k)} \leftarrow (X^{(k)}, tail(u^*))$, $Y^{(k)} \leftarrow (Y^{(k)}, 1)$;
18:             $loc \leftarrow tail(u^*)$, $\bar{Q} \leftarrow \bar{Q} - d_\xi(u^*)$;
19:         end if
20:         $U \leftarrow U \setminus u^*$, $U \leftarrow U \setminus \hat{u}^*$;
21:     end if
22: end while
23: return $S = (X, Y)$;
```

3.1.1 *The Filter Function.*

The purpose of the filter function is to select a subset of promising candidate tasks from the pool to reduce the noise to the GPHH-based heuristic learning. Since the problem aims to minimise the total cost without violating the capacity constraint, the *demand* and *deadheading cost to the current location* are clearly the two important attributes of the tasks. First, to prevent route failures, it is reasonable to eliminate the *infeasible tasks* whose demands are greater than the remaining capacity. Second, to minimise the total cost, the tasks that are closer to the current location (with smaller deadheading cost) should be more favourable than the distant tasks. Based on these two factors, we develop the filter function as described in Algorithm 2. It includes two stages. First, it filters the tasks based on demand (lines 2–6). Specifically, it selects the tasks satisfying the condition $\alpha \cdot \tilde{d}(u) \leq \bar{Q}$, where $\tilde{d}(u)$ is the expected demand of the task u, and α is a parameter to control the slack. Then, in the second stage (lines 7–13), it further filters the tasks by the deadheading cost from the current location, where $\tilde{c}(v_1, v_2)$ indicates the expected deadheading cost from v_1 to v_2, and $0 \leq \beta \leq 1$ is another parameter to specify the threshold. If $\beta = 0$, the only the nearest tasks are selected. If $\beta = 1$, then all the tasks are selected regardless of the distance.

3.1.2 *The* `GoTo` *function.*

The `GoTo()` function handles the potential edge failures while traveling to the next task. It is described in Algorithm 3. It updates the real-time deadheading cost matrix $\tilde{C}_{|V| \times |V|}$ when reaching each vertex along the shortest path (line 5). If the edge to the next vertex becomes unaccessible, it adjusts the subsequent path by finding the shortest path using the latest information (line 3).

3.1.3 *The* `Refill` *function.*

The `Refill()` function aims to address the route failure issue. Here, we adopt the repair operator proposed by [6]. The function is described in Algorithm 4. At first, the fraction of demand θ that can be served by the current route is calculated. It is assumed that the vehicle can immediately go back to the depot at the location where the route failure occurs, and the

Algorithm 2 $U' \leftarrow$ `Filter`(U)

```
Input: The unserved task set $U$, parameters $\alpha$ and $\beta$
Output: The filtered task set $U' \subseteq U$
1: $U' \leftarrow \emptyset, U'' \leftarrow \emptyset$;
2: for $u \in U$ do                                  ▷ Filter by demand
3:     if $\alpha \cdot \tilde{d}(u) \leq \bar{Q}$ then
4:         $U'' \leftarrow U'' \cup u$;
5:     end if
6: end for
7: $c_{\max} \leftarrow \max_{u \in U''} \tilde{c}(loc, head(u))$;
8: $c_{\min} \leftarrow \min_{u \in U''} \tilde{c}(loc, head(u))$;
9: for $u \in U''$ do                                ▷ Filter by deadheading cost
10:    if $\tilde{c}(loc, head(u)) \leq c_{\min} + \beta \cdot (c_{\max} - c_{\min})$ then
11:        $U' \leftarrow U' \cup u$;
12:    end if
13: end for
14: return $U'$;
```

Algorithm 3 `GoTo`$(X^{(k)}, Y^{(k)}, des)$

```
1: $loc \leftarrow X^{(k)}[end]$;                     ▷ Current location
2: while $loc \neq des$ do
3:     $(loc, p_1, \ldots, p_l, des) \leftarrow$ Dijkstra$(loc, des, \tilde{C}_{|V|\times|V|})$;
4:     for $i = 1 \to l$ do
5:         Set $\tilde{C}(loc, p_i) \leftarrow c_\xi(loc, p_i)$;   ▷ Update real-time info
6:         if $c_\xi(loc, p_i) = \infty$ then     ▷ Edge actually unaccessible
7:             break;
8:         end if
9:         $X^{(k)} \leftarrow (X^{(k)}, p_i), Y^{(k)} \leftarrow (Y^{(k)}, 0), loc \leftarrow p_i$;
10:    end for
11: end while
```

demand is uniformly distributed along the edge. Therefore, from the route failure location, the cost to *head*(u) is $\theta \cdot c_\xi(u)$, and the cost to *tail*(u) is $(1-\theta)\cdot c_\xi(u)$. Then, the algorithm compares the cost of the two refill directions via the head node and via the tail node, and selects the better direction (lines 2–6). Then, the vehicle goes back to the depot, closes the current route and opens a new one (lines 7–9). After that, the vehicle selects the better direction to go back to serve the remaining demand of the task again (lines 10–17). Note that in line 15, there is a subsequence $X^{(k)} = (\ldots, v_t, v_h, v_t)$ and $Y^{(k)} = (\ldots, 0, 1-\theta)$. This means that the vehicle traverses from v_t towards v_h, but stops at the fraction of $(1-\theta)$, i.e. the route failure location. Then it turns around and serves the remaining demand from the route failure location to v_t.

3.2 Terminal and Function Sets

Similar to the terminal set used in [25], the terminal set in this work consists of the following attributes.

(1) *demand*: the expected demand of the task, normalised by the capacity.
(2) *load*: the remaining capacity, normalised by the capacity.
(3) *cost*: the cost of serving the task from the current location. It is normalised into $[0, 1]$ by maximal *cost* value among all the candidate tasks.

Algorithm 4 `Refill`(X, Y, k, u)

```
1: $\theta \leftarrow \bar{Q}/d_\xi(u), v_h \leftarrow head(u), v_t \leftarrow tail(u)$;
2: if $\tilde{\delta}(v_h, v_0) + \theta \cdot c_\xi(u) < \tilde{\delta}(v_t, v_0) + (1-\theta)\cdot c_\xi(u)$ then
3:     $X^{(k)} \leftarrow (X^{(k)}, v_t, v_h), Y^{(k)} \leftarrow (Y^{(k)}, \theta, 0)$;
4: else
5:     $X^{(k)} \leftarrow (X^{(k)}, v_t), Y^{(k)} \leftarrow (Y^{(k)}, \theta)$;
6: end if
7: $(X^{(k)}, Y^{(k)}) \leftarrow$ GoTo$(X^{(k)}, Y^{(k)}, v_0)$;
8: $X \leftarrow X \cup X^{(k)}, Y \leftarrow Y \cup Y^{(k)}, k \leftarrow k+1$;
9: $X^{(k)} \leftarrow (v_0), Y^{(k)} \leftarrow (), loc \leftarrow v_0, \bar{Q} \leftarrow Q$;
10: if $\tilde{\delta}(v_0, v_h) + \theta \cdot c_\xi(u) < \tilde{\delta}(v_0, v_t) + (1-\theta)\cdot c_\xi(u)$ then
11:     $(X^{(k)}, Y^{(k)}) \leftarrow$ GoTo$(X^{(k)}, Y^{(k)}, v_h)$;
12:     $X^{(k)} \leftarrow (X^{(k)}, v_t, ), Y^{(k)} \leftarrow (Y^{(k)}, 1-\theta)$;
13: else
14:     $(X^{(k)}, Y^{(k)}) \leftarrow$ GoTo$(X^{(k)}, Y^{(k)}, v_t)$;
15:     $X^{(k)} \leftarrow (X^{(k)}, v_h, v_t), Y^{(k)} \leftarrow (Y^{(k)}, 0, 1-\theta)$;
16: end if
17: $loc \leftarrow v_h, \bar{Q} \leftarrow \bar{Q} - (1-\theta)\cdot d_\xi(u)$;
```

(4) *depotCost*: the cost from the tail of the task to the depot. It is normalised into $[0, 1]$ by the maximal *depotCost* value among all the candidate tasks.
(5) *satisfied*: the fraction of tasks that have already been satisfied (served).
(6) *heuristicValue*: the heuristic value calculated in the previous round.
(7) *constant*: a random constant terminal.

The function set is set to $\{+, -, *, /, max, min\}$. All functions are protected, they return 1 instead of $+\infty$, -1 instead of $-\infty$, and 0 instead of NaN.

4 EXPERIMENTAL STUDY

4.1 Experiment Settings

We used the UCARP instance generator proposed in [16] to randomly generate the training and test instances based on the static *gdb* and *val* datasets. For each static instance, we generated 90 UCARP training instances and 30 test instances. The stochastic demands and deadheading costs follow the Gamma distribution with the shape parameter $k = 20$ and the scale parameter $\theta = \tilde{f}/k$, where $\tilde{f}$ is the static demand or deadheading cost. Details of the generator can be found in [16].

During the GP training process, we used a mini-batch learning process. We splitted the 90 training instances into 18 mini-batches, each with 5 instances. Then we rotated the mini-batches generation by generation. This can significantly improve the training efficiency as well as the generalisation.

For the parameter settings of GPHH, we followed the standard GP settings and Weise's suggestions in [25]. The population size was set to 1024. The number of generations was 50. The minimum and maximum tree depths were set to 2 and 6, respectively. The crossover, mutation and reproduction rates were set to 2:5:2.

4.2 Parameter Sensitivity Analysis

The proposed GPHH has two important parameters, i.e. α and β in the filter function (Algorithm 2). α controls the slack of the stochastic task demand when checking the feasibility. β sets the threshold on the deadheading cost.

To investigate the effect of the two parameters on the performance of GPHH, we first carried out some pilot experiments on the static CARP instances. In this case, there is no need to estimate the expected demand, and we can simply fix $\alpha = 1$ to make sure all the feasible tasks are included and all the infeasible tasks are excluded. For the GPHH with each compared β value, 30 independent runs were conducted. For each instance, we made a pairwise comparison using the Wilcoxon rank sum test with significance level of 0.05.

Table 1 shows the average total cost obtained by the proposed GPHH with different β values over the *gdb* and *val* static instances. Due to the space limit, we only show the summary of the results over each dataset. From the table, it can be seen that as the β value increases, the average total cost tends to increase as well. $\beta = 0$ achieves the best average total cost for both the *gdb* and *val* instances. When β is too large ($\beta \geq 0.10$ for *gdb* and $\beta \geq 0.05$ for *val* instances), the total cost of the obtained solutions increases dramatically.

Table 1: The average total cost of the proposed GPHH with different β values over the *gdb* and *val* static instances. The best results are marked in bold.

Data	$\beta = 0$	$\beta = 0.02$	$\beta = 0.05$	$\beta = 0.10$
gdb	**257.31(0.15)**	257.49(0.07)	257.60(0.23)	258.13(0.15)
val	**361.82(0.59)**	361.86(0.53)	363.80(0.58)	369.37(0.77)

Table 2 shows the corresponding Win-Draw-Lose results. It is clear that for both datasets, $\beta = 0$ and $\beta = 0.02$ perform significantly better than $\beta = 0.05$ and $\beta = 0.10$. $\beta = 0$ performed significantly better than $\beta = 0.02$ on only one *gdb* instance.

Table 2: The Win-Draw-Lose results between the total cost obtained by the GPHH with different β values using Wilcoxon rank sum test with the significance level of 0.05.

		$\beta = 0$	$\beta = 0.02$	$\beta = 0.05$	$\beta = 0.10$
gdb	$\beta = 0$	–	1-22-0	2-20-1	7-15-1
	$\beta = 0.02$	–	–	0-23-0	6-16-1
	$\beta = 0.05$	–	–	–	5-17-1
val	$\beta = 0$	–	0-34-0	10-24-0	27-6-1
	$\beta = 0.02$	–	–	13-21-0	28-6-0
	$\beta = 0.05$	–	–	–	25-9-0

Note that when $\beta = 0$, the meta-algorithm is reduced to a nearest-neighbor heuristic. Although it cannot guarantee optimality, the results in Table 1 show that there is a tradeoff between the comprehensiveness of the candidate task set and the noise brought by including more distant tasks. In this case, it seems that the noise plays a more important role, and thus $\beta = 0$ reached the best results by minimising the noise. Based on the above observations, we set $\beta = 0$ for the GPHH in the subsequent experiments.

The sensitivity analysis for α was conducted on the UCARP instances. Here, we assume that the random demands follow the Gamma distribution (as mentioned in [16]), and their mean and standard deviation can be estimated accurately. For a random demand d, its mean value $\tilde{d}$ is given by the corresponding static value, and the standard deviation is $\sigma = \sqrt{k} \cdot \theta = \tilde{d}/\sqrt{20}$ ($\theta = \tilde{d}/k$, and k was set to 20). In the experiment, we set $\alpha = 1 + \lambda\sigma/\tilde{d} = 1 + \lambda/\sqrt{20}$, where $\lambda = 0$, 0.25, 0.50 and 0.75. For each compared α value and each UCARP instance, we used the GPHH to train a heuristic function, and record its test performance on the test set. Again, we conducted pairwise Wilcoxon rank sum test with significance level of 0.05 between the results of different α values.

Table 3 shows the average test performance of the GPHH with different α values over the *Ugdb* and *Uval* instances. Table 4 shows the corresponding Win-Draw-Lose results on the *Ugdb* instances. Similar patterns were found on *Uval* instances. From the tables, on can see that the α values obtained similar results. $\alpha = 1$ ($\lambda = 0$) and $\alpha = 1 + 0.25/\sqrt{20} \approx 1.06$ tend to produce slightly better results than the greater α values (e.g. *Ugdb* instances with objective F_1). From Table 3, it seems like that $\lambda = 0.75$ achieve the best test performance on the *Ugdb* instances with objective F_2. However, as shown in Table 4, such advantage is not significant. In terms of the Win-Draw-Lose result, $\lambda = 0.75$ is even slightly worse than $\lambda = 0.25$ (2-20-1).

Table 3: The average test performance of the GPHH with different α (i.e. λ) values over the *Ugdb* and *Uval* instances. The best results are marked in bold.

	Data	$\lambda = 0$	$\lambda = 0.25$	$\lambda = 0.50$	$\lambda = 0.75$
F_1	*Ugdb*	**285.87(0.44)**	286.86(0.52)	286.86(0.73)	287.77(0.59)
	Uval	**394.85(0.71)**	395.67(0.66)	396.11(0.76)	396.97(0.51)
F_2	*Ugdb*	**327.23(1.69)**	325.79(2.20)	328.28(1.90)	327.92(1.73)
	Uval	**445.28(2.21)**	445.68(1.61)	446.33(1.68)	447.24(1.75)

Table 4: The Win-Draw-Lose results between different α (i.e. λ) values on the *Ugdb* instances using Wilcoxon rank sum test with the significance level of 0.05.

		$\lambda = 0$	$\lambda = 0.25$	$\lambda = 0.50$	$\lambda = 0.75$
F_1	$\lambda = 0$	–	2-19-2	4-18-1	4-16-3
	$\lambda = 0.25$	–	–	2-20-1	3-18-2
	$\lambda = 0.50$	–	–	–	0-21-2
F_2	$\lambda = 0$	–	2-20-1	2-18-3	3-17-3
	$\lambda = 0.25$	–	–	0-23-0	2-20-1
	$\lambda = 0.50$	–	–	–	2-19-2

In summary, we have the following observations from the sensitivity analysis.

- $\beta = 0$ is the best choice for the static instances.
- When $\beta = 0$, $\alpha = 1$ or $\alpha = 1.06$ are reasonably good choice. If α is too large, then the test performance will drop.

Therefore, we select $\alpha = 1$ and $\beta = 0$ for our GPHH. For the sake of convenience, it will be denoted as GPHH(1,0).

4.3 Comparison with Existing Methods

To the best of our knowledge, the GPHH proposed by Weise et al. [25] is the only existing GPHH for UCARP. It was selected as the baseline GPHH (denoted as *W-GPHH*) for comparison. In addition, we compared with five manually designed heuristic functions [13]. They are denoted as $H_1, \ldots, H_5$, and described as follows:

H_1 selects the task with the *maximal* distance to the depot;
H_2 selects the task with the *minimal* distance to the depot;
H_3 selects the task with the *maximal* *demand*/*serveCost* value;
H_4 selects the task with the *minimal* *demand*/*serveCost* value;
H_5 uses H_1 if *load* ≥ 0.5, and H_2 otherwise.

Tables 5 and 6 show the average test performance and corresponding Win-Draw-Lose results of the compared methods against the proposed GPHH(1,0). From the tables, it can be seen that W-GPHH performed much worse than GPHH(1,0) and the manually designed heuristics. This demonstrates the importance of the carefully designed meta-algorithm that filters the tasks based on feasibility and distance. Without such mechanism, W-GPHH failed to learn effective heuristics. Furthermore, GPHH(1,0) significantly outperformed H_1 to H_4. It was defeated by only H_5 on the small and simple *Ugdb* instances. This is consistent with our expectation, since H_5 is a more sophisticated heuristic than the other four. In addition, the advantage of GPHH becomes more obvious on the larger *Uval* instances, which tend to have more complicated patterns for making good decisions.

Table 5: The F_1 test performance and Win-Draw-Lose results of the compared methods against GPHH(1,0).

		H_1	H_2	H_3	H_4	H_5	W-GPHH	GPHH(1,0)
Ugdb	Mean	299.59	308.30	297.68	305.49	**285.85**	316.43	285.87
	Std	(0.00)	(0.00)	(0.00)	(0.00)	**(0.00)**	(0.59)	(0.44)
	W-D-L	1-0-22	0-0-23	0-3-20	1-0-22	**8-6-9**		–
Uval	Mean	409.79	448.51	424.28	431.72	407.69	427.25	**394.85**
	Std	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)	(1.13)	**(0.71)**
	W-D-L	0-3-31	0-0-34	0-0-34	0-0-34	3-1-30		–

Table 6: The F_2 test performance and Win-Draw-Lose results of the compared methods against GPHH(1,0).

		H_1	H_2	H_3	H_4	H_5	W-GPHH	GPHH(1,0)
Ugdb	Mean	345.25	352.92	344.28	347.68	**330.48**	366.58	327.23
	Std	(0.00)	(0.00)	(0.00)	(0.00)	**(0.00)**	(1.83)	(1.69)
	W-D-L	2-2-19	1-2-20	0-3-20	1-3-19	**6-8-9**		–
Uval	Mean	456.16	497.49	477.75	485.51	456.40	482.23	**445.28**
	Std	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)	(2.43)	**(2.21)**
	W-D-L	5-7-22	1-0-33	1-4-29	0-1-33	6-7-21		–

4.4 Further Analysis

4.4.1 The evolved heuristic function. Eq. (10) shows an arbitrarily selected evolved heuristic function. This heuristic is an aggregation of two components (lines 1–2 and 3–4, respectively). The first component contains *demand*, implying that the heuristic tends to favour the tasks with smaller demands. Furthermore, the highlighted fragment in the first component shows that when *load* > 0.51, i.e. the vehicle is less than half-full, *demand*/*depotCost* has a positive coefficient, and thus the heuristic tends to select the tasks with larger *depotCost*. Otherwise, the coefficient of *demand*/*depotCost* is negative, and the *depotCost* tends to be minimised. This observation is consistent with H_5, which means that GPHH can identify the sophisticated relationship between the attributes automatically.

$$\begin{aligned}&(+\ (\times\ (\max\ (\max\ (/\ \mathit{heuristicValue}\ \mathit{cost})\ \mathit{load})\ \mathit{demand})\\&\quad (/\ \boldsymbol{(/\ \mathit{demand}\ \mathit{depotCost})\ (-\ \mathit{load}\ 0.51)}))\\&\ (\times\ (+\ \mathit{load}\ (\times\ \mathit{cost}\ \mathit{load}))\\&\quad (+\ (-\ \mathit{load}\ \mathit{satisfied})\ (/\ \mathit{depotCost}\ \mathit{satisfied}))))\end{aligned} \tag{10}$$

4.4.2 Effect of α. The sensitivity analysis on α (Tables 3 and 4) showed that there is no obvious relationship between the α value and the test performance. Then we conducted further analysis on how different α value affect the decision making during the solution generation. To this end, we randomly select a GP-evolved heuristic obtained from the *Ugdb*15 training instances, and test it on the 30 *Ugdb*15 test instances with different α values. Here the small and simple *Ugdb*15 instances are selected to facilitate investigation. First, we plot the total cost obtained by different α values for each of the 30 test instances, as shown in Fig. 1. From the figure, we further pick two scenarios as follows:

- *Instance 3*, where larger α values performed better;
- *Instance 5*, where smaller α values performed better.

Then, for each scenario, we compared the routes generated by different α values.

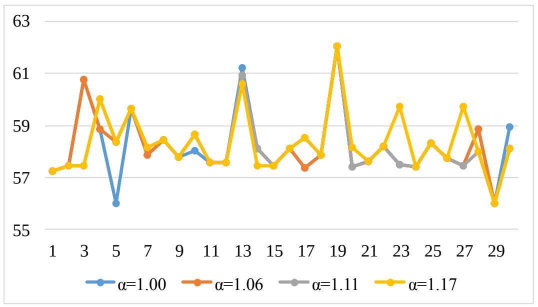

Figure 1: The total cost obtained by the randomly selected GP-evolved heuristic with different α values on the 30 *Ugdb*15 test instances.

(Scenario 1). Fig. 2 shows the scenario of test instance 3. It can be seen that when α is smaller, there is a route failure occurring at the end of $X^{(2)}$. As a result, the total cost of the routes (a) is 60.76, while the routes (b) without any route failure has a total cost of 57.44. Taking a closer look at the decision situation where the task $(2, 6)$ was selected by routes (a), we found that the remaining capacity was 6.56 and the expected demand of $(2, 6)$ was 6. However, the actual demand of the task was $6.81 > 6.56$. For a Gamma distribution with $k = 20$ and $\theta = 6/20$, it is very likely to have a sample of 6.81. In this case, adding a slack $\alpha > 1.06$ to the expected task demand prevents a route failure.

(Scenario 2). Fig. 3 shows the scenario of test instance 5. In this case, the difference occurred when deciding whether to serve the

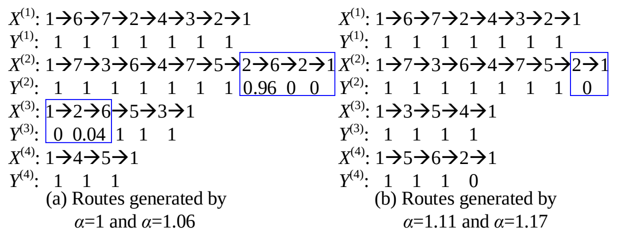

Figure 2: The solutions generated by different α values on the *Ugdb15* test instance 3.

task (4, 1) or not at the end of route $X^{(2)}$. At this decision point, the remaining capacity was 4.03, and the expected demand of the task (4, 1) was 4. The actual demand is 2.57, which is much smaller than the expected demand. When $\alpha \geq 1.06$, the task was eliminated from the candidate task set, and the vehicle skipped it while going back to the depot. As a result, the greater α values led to one more route, and thus a larger total cost (56.00 versus 58.37).

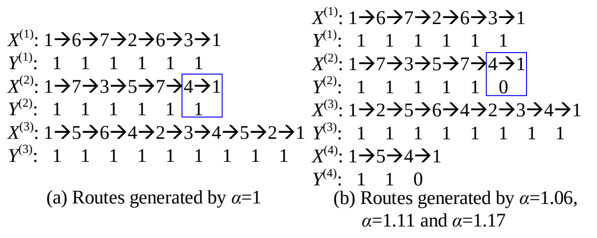

Figure 3: The solutions generated by different α values on the *Ugdb15* test instance 5.

Overall, whether $\alpha > 1$ can improve the performance of the GPHH highly depends on the scenario. Basically, if the actual demand is no larger than the expected demand, then a smaller α tends to be better. If the actual demand is greater, then a larger α is a better choice.

5 CONCLUSION

In this paper, a new GPHH method for evolving routing heuristics for UCARP was proposed. We developed a new meta-algorithm that coped well with the route and edge failures in UCARP. Furthermore, it filtered the infeasible distant candidate tasks to reduce the noise of the GP-based heuristic learning. The experimental results showed that the proposed GPHH method significantly outperformed the existing GPHH for UCARP. Furthermore, it was found that the filter function played an important role in improving the performance of the GPHH. Specifically, it was important to eliminate the distant tasks from the candidate task set. In addition, when the task demands were stochastic, simply adding a slack to the expected demands could not necessarily address the issue. Although it reduced the probability of route failures, it might skip some tasks that can actually be served on the way. It seemed that directly using the expected demand was the best choice on average. In future, we will investigate more intelligent ways of adding slacks, such as scenario-adaptive α values.

REFERENCES

[1] S.K. Amponsah and S. Salhi. 2004. The Investigation of a Class of Capacitated Arc Routing Problems: The Collection of Garbage in Developing Countries. *Waste Management* 24, 7 (2004), 711–721.

[2] R. Baldacci and V. Maniezzo. 2006. Exact Methods Based on Node-Routing Formulations for Undirected Arc-Routing Problems. *Networks* 47, 1 (2006), 52–60.

[3] E. Bartolini, J.-F. Cordeau, and G. Laporte. 2013. An Exact Algorithm for the Capacitated Arc Routing Problem with Deadheading Demand. *Operations Research* 61, 2 (2013), 315–327.

[4] J.F. Campbell and A. Langevin. 2000. *Roadway Snow and Ice Control.* Springer US, Boston, MA, 389–418.

[5] Y. Chen, J.K. Hao, and F. Glover. 2016. A Hybrid Metaheuristic Approach for the Capacitated Arc Routing Problem. *European Journal of Operational Research* 253, 1 (2016), 25–39.

[6] C.H. Christiansen, J. Lysgaard, and S. Wøhlk. 2009. A Branch-and-Price Algorithm for the Capacitated Arc Routing Problem with Stochastic Demands. *Operations Research Letters* 37, 6 (2009), 392–398.

[7] G. Fleury, P. Lacomme, and C. Prins. 2004. *Evolutionary Algorithms for Stochastic Arc Routing Problems.* Springer Berlin Heidelberg, 501–512.

[8] G. Fleury, P. Lacomme, C. Prins, and W. Ramdane-Chérif. 2005. Improving Robustness of Solutions to Arc Routing Problems. *Journal of the Operational Research Society* 56, 5 (2005), 526–538.

[9] B.L. Golden, J.S. Dearmon, and E.K. Baker. 1983. Computational Experiments with Algorithms for a Class of Routing Problems. *Computers & Operations Research* 10, 1 (1983), 47–59.

[10] B.L. Golden and R.T. Wong. 1981. Capacitated Arc Routing Problems. *Networks* 11, 3 (1981), 305–315.

[11] C.E. Gounaris, W. Wiesemann, and C.A. Floudas. 2013. The Robust Capacitated Vehicle Routing Problem Under Demand Uncertainty. *Operations Research* 61, 3 (2013), 677–693.

[12] H. Handa, D. Lin, L. Chapman, and X. Yao. 2006. Robust Solution of Salting Route Optimisation Using Evolutionary Algorithms. In *2006 IEEE International Conference on Evolutionary Computation.* 3098–3105.

[13] P. Lacomme, C. Prins, and W. Ramdane-Cherif. 2004. Competitive Memetic Algorithms for Arc Routing Problems. *Annals of Operations Research* 131, 1 (2004), 159–185.

[14] Y. Mei, X. Li, and X. Yao. 2014. Cooperative Coevolution With Route Distance Grouping for Large-Scale Capacitated Arc Routing Problems. *IEEE Transactions on Evolutionary Computation* 18, 3 (2014), 435–449.

[15] Y. Mei, K. Tang, and X. Yao. 2009. A Global Repair Operator for Capacitated Arc Routing Problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39, 3 (2009), 723–734.

[16] Y. Mei, K. Tang, and X. Yao. 2010. Capacitated Arc Routing Problem in Uncertain Environments. In *IEEE Congress on Evolutionary Computation.* 1–8.

[17] V. Pillac, M. Gendreau, C. Guéret, and A.L. Medaglia. 2013. A Review of Dynamic Vehicle Routing Problems. *European Journal of Operational Research* 225, 1 (2013), 1–11.

[18] M. Polacek, K.F. Doerner, R.F. Hartl, and V. Maniezzo. 2008. A Variable Neighborhood Search for the Capacitated Arc Routing Problem with Intermediate Facilities. *Journal of Heuristics* 14, 5 (2008), 405–423.

[19] U. Ritzinger, J. Puchinger, and R.F. Hartl. 2016. A Survey on Dynamic and Stochastic Vehicle Routing Problems. *International Journal of Production Research* 54, 1 (2016), 215–231.

[20] L. Santos, J. Coutinho-Rodrigues, and J.R. Current. 2009. An Improved Heuristic for the Capacitated Arc Routing Problem. *Computers & Operations Research* 36, 9 (2009), 2632–2637.

[21] L. Santos, J. Coutinho-Rodrigues, and J.R. Current. 2010. An Improved Ant Colony Optimization Based Algorithm for the Capacitated Arc Routing Problem. *Transportation Research Part B: Methodological* 44, 2 (2010), 246–266.

[22] K. Tang, Y. Mei, and X. Yao. 2009. Memetic Algorithm With Extended Neighborhood Search for Capacitated Arc Routing Problems. *IEEE Transactions on Evolutionary Computation* 13, 5 (2009), 1151–1166.

[23] J. Wang, K. Tang, J. A. Lozano, and X. Yao. 2016. Estimation of the Distribution Algorithm With a Stochastic Local Search for Uncertain Capacitated Arc Routing Problems. *IEEE Transactions on Evolutionary Computation* 20, 1 (2016), 96–109.

[24] J. Wang, K. Tang, and X. Yao. 2013. A Memetic Algorithm for Uncertain Capacitated Arc Routing Problems. In *2013 IEEE Workshop on Memetic Computing.* 72–79.

[25] T. Weise, A. Devert, and K. Tang. 2012. A Developmental Solution to (Dynamic) Capacitated Arc Routing Problems Using Genetic Programming. In *Proceedings of GECCO.* ACM, 831–838.

[26] E.J. Willemse and J.W. Joubert. 2016. Constructive Heuristics for the Mixed Capacity Arc Routing Problem under Time Restrictions with Intermediate Facilities. *Computers & Operations Research* 68 (2016), 30–62.