A Predictive-Reactive Approach with Genetic Programming and Cooperative Co-evolution for Uncertain Capacitated Arc Routing Problem

Yuxin Liu liuyx@shmtu.edu.cn
College of Information Engineering, Shanghai Maritime University, Shanghai 201306, China; College of Computer & Information Science, Southwest University, Chongqing 400715, China; School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand

Yi Mei yi.mei@ecs.vuw.ac.nz
School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand

Mengjie Zhang mengjie.zhang@ecs.vuw.ac.nz
School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand

Zili Zhang * zhangzl@swu.edu.cn
College of Computer & Information Science, Southwest University, Chongqing 400715, China; School of Information Technology, Deakin University, Locked Bag 20000, Geelong VIC 3220, Australia

Abstract
The uncertain capacitated arc routing problem is of great significance for its wide applications in the real world. In uncertain capacitated arc routing problem, variables such as task demands and travel costs are realised in real time. This may cause the predefined solution to become ineffective and/or infeasible. There are two main challenges in solving this problem. One is to obtain a high-quality and robust *baseline task sequence*, and the other is to design an effective *recourse policy* to adjust the baseline task sequence when it becomes infeasible and/or ineffective during the execution. Existing studies typically only tackle one challenge (the other being addressed using a naive strategy). No existing work optimises the baseline task sequence and recourse policy simultaneously. To fill this gap, we propose a novel proactive-reactive approach, which represents a solution as a baseline task sequence and a recourse policy. The two components are optimised under a cooperative co-evolution framework, in which the baseline task sequence is evolved by an estimation of distribution algorithm, and the recourse policy is evolved by genetic programming. The experimental results show that the proposed algorithm, called Solution-Policy Co-evolver, significantly outperforms the state-of-the-art algorithms to uncertain capacitated arc routing problem for the *ugdb* and *uval* benchmark instances. Through further analysis, we discovered that route failure is not always detrimental. Instead, in certain cases (e.g. when the vehicle is on the way back to the depot) allowing route failure can lead to better solutions.

Keywords
Capacitated arc routing problem, cooperative co-evolution, hyper-heuristics, genetic programming.

*Corresponding author: Zili Zhang.

1 Introduction

The capacitated arc routing problem (CARP) (Golden and Wong, 1981) is a classic NP-hard combinatorial optimization problem, which is the prototype of many real-world applications when tasks are along roads. CARP can be stated as scheduling optimal routes for vehicles to serve a set of edge tasks with minimal costs subject to the predefined constraints such as the capacity constraint (Dror, 2000). It has wide applications including mail delivery (Aráoz et al., 2006), waste collection (Amponsah and Salhi, 2004) and winter gritting (Handa et al., 2006).

Lots of studies have been done in the area of static and deterministic CARP (Tang et al., 2009; Xing et al., 2010; Mei et al., 2014; Chen et al., 2016), in which all the information about the problem is exactly known beforehand. However, it is often not the case in reality, where some information about the tasks and roads is unknown in advance, and is revealed dynamically while the services are being executed (Fleury et al., 2005; Tagmouti et al., 2011; Chen et al., 2014). For example, in winter gritting, the amount of grit that should be spread on the roads is influenced by the temperature of the road surface, which cannot be known exactly in advance. In addition, the traversing costs depend on the traffic situation, and temporary roadworks may even make the road inaccessible. To meet the reality better, a variety of Uncertain CARP (UCARP) models with stochastic task demands and/or edge costs have been developed (Fleury et al., 2005; Tagmouti et al., 2011; Mei et al., 2010). In this paper, we focus on one of the most general UCARP model (Mei et al., 2010; Wang et al., 2016), which means that other existing models can be seen as special cases of it.

It has been shown that one cannot easily cope with the uncertain information in UCARP by directly applying the effective approaches for static and deterministic CARP (Mei et al., 2010; Ouelhadj and Petrovic, 2009). Specifically, the solutions obtained by static methods may become much less effective or even infeasible when the exact values of the variables (e.g. task demand and presence of edges) are different from expectation (so-called *route/edge failures*). In fact, since the exact values of variables could not be obtained beforehand, the route/edge failures are unavoidable. In this case, a clever *recourse policy* is needed to adjust the predetermined solutions in real time. However, most of existing studies in UCARP typically fix a naive recourse policy (e.g. a greedy recourse operator or a preventive restocking operator) and focus on optimising baseline solutions (Fleury et al., 2004; Wang et al., 2013, 2016). A naive recourse policy can restrict the potential of evolving better baseline solutions. Meanwhile, a few studies in the field of vehicle routing problems with stochastic demands devote to design more intelligent recourse policies (Salavati-Khoshghalb et al., 2017a,b), such as a hybrid recourse policy that balance the risk of failures at the next customer to be visited with the length of replenishment trips. Although the baseline solution and the recourse policy are not independent and the combination of them determines the quality of a total cost in UCARP (Gendreau et al., 2016), none of research optimises these two components simultaneously until now.

From a wider perspective of combinatorial optimisation problems in uncertain (dynamic and/or stochastic) environments, the existing approaches can be divided into three main categories (Ouelhadj and Petrovic, 2009; Shen and Yao, 2015; Nguyen et al., 2016): robust *pro-active*, completely *reactive* and *predictive-reactive*. Among them, predictive-reactive is the most commonly used (Ouelhadj and Petrovic, 2009). It has a reoptimisation process that is triggered by the real-time events. Predictive-reactive approaches have shown excellent performance for solving many uncertain and dynamic optimisation problems such as dynamic vehicle routing (Hanshar and Ombuki-

Berman, 2007) and dynamic job shop scheduling (Chryssolouris and Subramaniam, 2001; Shen and Yao, 2015). However, the operators in predictive-reactive approaches are problem specific, and they could not be used for solving UCARP directly. To the best of our knowledge, no predictive-reactive approach has been designed for solving UCARP so far.

The goal of this paper is to propose a novel predictive-reactive approach that overcomes the drawbacks of the existing approaches to UCARP. The new approach utilizes a new solution representation, in which each individual in the population is composed of two components: (1) a *baseline task sequence* and (2) a *recourse policy*. These two components are optimised simultaneously by a cooperative co-evolution framework. In this way, we expect to obtain a high-quality and robust baseline task sequence along with its corresponding recourse policy. In particular, we focus on the single-vehicle scenario, in which all routes are executed by a single vehicle in a sequential manner. Although it is a simplified version of UCARP, it is a good starting point for investigating the problem in a predictive-reactive way. The paper has the following research objectives:

- Design a new solution representation for UCARP. With the new representation, an individual consists of a baseline task sequence and a recourse policy to adjust the baseline task sequence in real time.
- Develop a cooperative co-evolution algorithm to evolve the baseline task sequence and recourse policy simultaneously. The algorithm is named Solution-Policy Co-evolver (SoPoC).
- Evaluate the effectiveness of SoPoC by comparing with existing state-of-the-art algorithms on benchmark datasets.
- Analyse the structure of the solutions obtained by the proposed algorithm to gain insights of useful patterns in the promising UCARP solutions.

The rest of this paper is structured as follows. Section 2 presents the background including problem description and related work. Section 3 describes the proposed new solution representation for UCARP. Then, Section 4 presents the developed SoPoC, including the fitness evaluation and subcomponent optimisers. Section 5 shows the experimental design, including dataset and parameters. Section 6 shows the experimental comparison between SoPoC and the state-of-the-art algorithms to UCARP. Finally, Section 7 gives the conclusion and future work.

2 Background

In this section, we first introduce the definition of UCARP and then give the literature review on the related work.

2.1 Uncertain Capacitated Arc Routing Problem

We focus on the general UCARP model (Mei et al., 2010; Wang et al., 2016) that covers all the stochastic factors considered in the literature. In this model, an undirected graph $G = \{V, E\}$ is given, where V is the set of vertices and E is the set of edges. Each edge $e \in E$ is associated with three features: a demand $d(e) \geq 0$, a serving cost $sc(e) \geq 0$ and a deadheading cost $dc(e) \geq 0$. The deadheading cost indicates the cost of travelling along the edge without serving it. Edges with positive demands are called *tasks*. The set of all tasks is denoted as $T \subseteq E$. A fleet of vehicles with a limited capacity Q are located at a special depot node $v_0 \in V$ to serve all the tasks. Then, the goal of the problem is to find a least-cost routing plan for the vehicles to serve all the tasks subject to the following constraints.

1. Each vehicle departs from the depot, and returns to the depot after serving all the tasks allocated to it.
2. Each task is served exactly once in either direction.
3. The total demand served by each vehicle *in a single trip* cannot exceed its capacity.

The above UCARP model is defined under multiple vehicles. For the single vehicle scenario considered in this paper, we allow the single vehicle to have multiple trips. Each trip starts and ends at the depot, and the total demand of each trip does not exceed the capacity.

In UCARP, the task demands and deadheading costs are stochastic, whose values are not known exactly in advance (i.e. during the optimisation process). More specifically, UCARP has the following assumptions on the stochastic factors.

- The distribution (e.g. mean and standard deviation) of a task demand is known in advance (estimated from historical data). The actual demand value is realised after the service of the task is completed (e.g. in waste collection, the actual amount of waste on the street is unknown until the street is served completely).
- Each edge has a low probability to be absent from the graph (e.g. caused by temporary road work or traffic accident). The actual presence of an edge is unknown until the vehicle reaches its head node (e.g. the "closed road" sign can only be seen next to the closed road).
- The distribution of the deadheading cost of an edge is known in advance (based on historical data). The actual deadheading cost is realised after the edge is traversed (e.g. the traveling time of a street vary with many factors such as the real-time traffic situations, road conditions and driving skills).

Due to the above stochastic factors, two uncontrollable failures may be encountered during the execution process, which are called the *route failure* and *edge failure*. Route failure occurs when the expected demand of the next task is smaller than the remaining capacity of the vehicle, while the actual demand of a task is larger than the remaining capacity. Edge failure occurs when the vehicle arrives at the head of the edge and finds that the current edge is absent (temporarily inaccessible).

To deal with these failures, recourse policies (Mei et al., 2010; Christiansen et al., 2009) have been designed to adjust the solution in real time when it becomes ineffective and/or infeasible. Under the stochastic environment mentioned above, a commonly used naive recourse strategy (Fleury et al., 2004; Mei et al., 2010; Weise et al., 2012; Wang et al., 2013) works as follows:

- For *route failure*, as soon as the capacity of the vehicle is exceeded, the vehicle goes back to the depot via the tail of the task to replenish its capacity, and then comes back to the interrupted place via the head of the task to continue the service.
- For *edge failure*, if the next edge is inaccessible, the deadheading cost of the edge is replaced by an infinite number. The vehicle then re-calculates the shortest path from the current location to the next destination online (e.g. using Dijkstra's algorithm) based on the updated graph.

A feasible solution to a sampled UCARP instance (e.g. a specified random seed to generate the random variable values) can be represented as a combination of two components (X, Y). $X{=}\{X^{(1)}, X^{(2)}, \ldots, X^{(m)}\}$ is a set of routes, where each route $X^{(k)}{=}(x_1^{(k)}, \ldots, x_{L_k}^{(k)})$ is a sequence of vertices starting and ending at the depot vertex (i.e. $x_1^{(k)}{=}x_{L_k}^{(k)}{=}v_0$). $Y{=}\{Y^{(1)}, Y^{(2)}, ..., Y^{(m)}\}$ is a set of real-value vectors indicating the fractions of services of each edge along the routes. Specifically, $Y^{(k)}{=}(y_1^{(k)}, ..., y_{L_k-1}^{(k)})$ corresponds to $X^{(k)}$, where $0 \leq y_i^{(k)} \leq 1$. $y_i^{(k)} > 0$ means that the edge $(x_i^{(k)}, x_{i+1}^{(k)})$ is a task and is being served. On the other hand, $y_i^{(k)} = 0$ means that the vehicle travels through the edge $(x_i^{(k)}, x_{i+1}^{(k)})$ without serving it. Note that if $0 < y_i^{(k)} < 1$, the task $(x_i^{(k)}, x_{i+1}^{(k)})$ is partially served (when the route failure occurs and the vehicle has to go back to the depot in the middle of the service).

The total cost of a solution (X, Y) is calculated as (1).

$$\begin{aligned} C(X,Y) = \sum_{k=1}^{m} \sum_{j=1}^{L_k-1} (sc(x_j^{(k)}, x_{j+1}^{(k)}) * y_j^{(k)} + \\ dc(x_j^{(k)}, x_{j+1}^{(k)}) * (1 - y_j^{(k)})) \end{aligned} \tag{1}$$

A solution $S_{\mathcal{I}}$ to a UCARP instance $\mathcal{I}$ is represented as either a robust set of routes (practical feasibility is guaranteed by the naive recourse operator) or a routing heuristic (in complete reactive way). Based on the evaluation of a solution to a sampled UCARP instance, a UCARP solution $S_{\mathcal{I}}$ is evaluated in terms of the *average performance* (Mei et al., 2010) or *worst-case performance* (Wang et al., 2016). Let $\Xi(\mathcal{I})$ be the set of all the possible sampled instances of $\mathcal{I}$, the average performance $F_{avg}(S_{\mathcal{I}})$ and worst-case performance $F_{wst}(S_{\mathcal{I}})$ are defined as follows:

$$F_{avg}(S_{\mathcal{I}}) = E[C(\texttt{sol}(S_{\mathcal{I}}, \xi)) \mid \xi \in \Xi(\mathcal{I})], \tag{2}$$

$$F_{wst}(S_{\mathcal{I}}) = \max_{\xi \in \Xi(\mathcal{I})} C(\texttt{sol}(S_{\mathcal{I}}, \xi)), \tag{3}$$

where $C(\texttt{sol}(S_{\mathcal{I}}, \xi))$ is the total cost of the solution generated from $S_{\mathcal{I}}$ in the sampled instance ξ.

2.2 Related Work

Although there are not many studies for UCARP, researchers have investigated a variety of dynamic/stochastic combinatorial optimisation problems that are related to UCARP. As a node-routing counterpart of CARP, the dynamic/stochastic Vehicle Routing Problem (VRP) has received much more research interests so far. Pillac et al. (2013) provided an overview of the approaches for solving Dynamic VRP (DVRP), which were categorised into periodic reoptimisation and continuous reoptimisation. Ritzinger et al. (2016) provided another review of approaches for DVRP. Based on when the decisions were made, the approaches were categorised into preprocessed decision approaches and online decision approaches.

The Dynamic Job Shop Scheduling (DJSS) is another widely studied dynamic combinatorial optimisation problem. There have been various studies in evolving scheduling policies for DJSS with Genetic Programming Hyper-Heuristic (GPHH), which are very useful for designing routing heuristics in UCARP. Ouelhadj and Petrovic (2009) and Nguyen et al. (2016) categorised the dynamic scheduling approaches into three categories: *robust pro-active*, *completely reactive*, and *predictive-reactive*. It can be seen that other categorisations can be easily adapted to this categorisation (e.g. the reoptimisation strategies in Pillac et al. (2013) belong to the predictive-reactive approaches, and the preprocessing-based approaches in Ritzinger et al. (2016) can be seen as the pro-active approaches). In the following, we give the literature review following the above categorisation. The discussion mainly focus on UCARP. Representative works in DVRP and DJSS are also included for reference.

Robust pro-active approaches develop predictive solutions to satisfy performance requirements in a dynamic environment in the first stage. Along with the execution of the first-stage solutions, recourse actions are taken to deal with failures in the second stage, as in two-stage stochastic programming with recourse (Kall and Wallace, 1994; Gendreau et al., 2016). The predictive solutions are mainly based on off-line optimisation strategies. Since the solutions will keep unchanged in the execution process, improving the predictability is the key issue in these approaches.

Fleury et al. (2005) proposed a slack-based approach by reducing the vehicle capacity during the predictive solution generation process for solving CARP with stochastic demands. They also proposed a memetic algorithm by incorporating trip interruption probability in the objective function (i.e. expected cost and expected number of trips) to provide a robust solution for CARP with stochastic demands (Fleury et al., 2004). Wang et al. (2013) proposed a memetic algorithm where the fitness function is defined based on a number of static instances to improve the robustness of solutions for U-CARP. Wang et al. (2016) further proposed the Estimation of Distribution Algorithm with Stochastic Local Search (EDASLS) with the fitness function of the worst-case performance on all the instances. Christiansen et al. (2009) and Christiansen and Lysgaard (2007) formulated the CARP and VRP with stochastic demands as the two-stage s-tochastic programming with classical recourse (Gendreau et al., 2016), and proposed the branch-and-price algorithm to solve them, respectively. Salavati-Khoshghalb et al. (2017b) also formulated the VRP with stochastic demands as the two-stage stochastic model but focused on designing a rule-based recourse policy to achieve better solutions than the classical recourse one. Jensen (2003) proposed a genetic algorithm by defining a robustness measures based on continuous function optimisation for JSS with random machine breakdowns.

Pro-active approaches can provide robust and predictable solutions when applying to new environments with stochastic factors. However, they are non-flexible and cannot cope with real-time adjustment. Besides, these approaches are not applicable for dynamic environment where new customers or jobs may occur, and the solutions must be reconfigured to deal with new events (Hanshar and Ombuki-Berman, 2007). For example, in routing problem, new customers need to be incorporated into the existing vehicle routes or new routes need to be created to deal with them.

Completely reactive approaches do not generate solutions in advance. Instead, they make decisions dynamically and locally in real-time. Dispatching rules for JSS is a typical example (Ouelhadj and Petrovic, 2009). More specifically, whenever a machine becomes idle, the dispatching rule calculates the priority value for the jobs in its queue, and selects the one with the highest priority to be processed next.

Weise et al. (2012) proposed to apply GPHH for automated design of heuristic function (i.e. routing heuristic) for solving static CARP, and tested the performance of the evolved heuristics for dealing with random disappearance of tasks. Liu et al. (2017) extended the automated heuristic function design using GPHH for solving UCARP. They developed a new framework for mapping heuristics to feasible solutions by taking advantage of domain knowledge, and showed that the framework can cope well with the route and edge failures. Secomandi (2001) proposed a rollout algorithm, which is inspired by reinforcement learning methodology, for solving the VRP with stochastic demands by computing a reoptimisation-type rollout policy. Khaligh and MirHassani (2016) proposed a multi-stage stochastic programming model for solving the VRP with stochastic demands, in which the decision of which customer to be visited next is made in each stage. Nguyen et al. (Nguyen et al., 2013, 2017) conducted a wide research on using GPHH for automatic design of scheduling policies (e.g. dispatching rules and due-date assignment rules) for JSS.

Completely reactive approaches have two main advantages (Nguyen et al., 2013). First, they search in the heuristic space rather than the solution space. Therefore, they have a good scalability, since the heuristic space is independent of the problem size. Second, the solutions are generated online or in real time, so the approaches are very efficient in handling dynamic environments. However, these approaches also have some disadvantages. First, the heuristics are usually myopic (only consider local information), and cannot compete with global optimisation approaches (Ouelhadj and Petrovic, 2009). Second, no baseline solution is generated since decisions are purely made in real time. This causes difficulty for planning and measuring in advance (N-guyen et al., 2016).

Predictive-reactive approaches are the most common approaches for dynamic optimisation problems (Ouelhadj and Petrovic, 2009; Shen and Yao, 2015). They first obtain a predictive baseline solution (e.g. by pro-active approaches), and then reoptimise the solution in response to real-time events. These approaches generally consider both the quality of the predictive baseline solution (*efficiency*) and the degree of change to be made on the baseline solution to adapt to the new environment (*stability*).

Montemanni et al. (2002) decomposed a DVRP as a sequence of static VRPs by dividing a working day into a number of discrete time periods, and proposed an ant colony system to reschedule the previous routes at the beginning of each time period to deal with new incoming customers. The DVRP model proposed by Montemanni et al. was further used by Hanshar and Ombuki-Berman (2007), in which a genetic algorithm with a new representation and crossover operator was proposed. Chryssolouris and Subramaniam (2001) proposed to regenerate a new schedule as soon as a dynamic event (e.g. new job arrivals and machine breakdowns) occurs in DJSS. However, stability is not considered in the above works. Lots of works modelled the problem as a multiple-objective optimisation considering both efficiency and stability (Shen and Yao, 2015; Leon et al., 1994; Fattahi and Fallahi, 2010). For example, Shen and Yao (2015) proposed a four-objective JSS model.

Predictive-reactive approaches can be seen as a hybridisation of the pro-active approaches and the complete reactive approaches. They include a baseline solution obtained by the pro-active part, and a reoptimisation strategy in charge of real-time reaction. For reoptimisation, most existing studies mainly focus on searching in the solution space around the current solution (Shen and Yao, 2015; Hanshar and Ombuki-Berman, 2007). For large and complex problem instances, it is still hard to obtain an effective reoptimised solution immediately. To address this issue, in this paper, we propose to implement the reoptimisation process by a *recourse policy*, which is a heuristic that takes the latest information and makes decisions in real time. In this way, the re-optimisation part is closer to the complete reactive approaches, which aim to search for heuristics that can make real-time decisions effectively. In our approach, we evolve the recourse policy in an off-line fashion by GPHH.

3 The New Solution-Policy Representation

In this section, we first present the newly proposed solution representation, which is a hybridisation of a baseline task sequence and a recourse policy. Then, we describe how a solution under the new representation is executed, i.e. how to construct a feasible solution given a UCARP instance when the uncertain information is realised over time.

3.1 Representation

The solution representation is composed of two components: a *baseline task sequence* and a *recourse policy*. The task sequence representation is similar to that used in previous works (Tang et al., 2009; Mei et al., 2014, 2011). That is, each edge task t is associated with two positive integers (IDs), represented as $\texttt{ID}_1(t)$ and $\texttt{ID}_2(t)$. They represent two inverse directions, which means that the head node of $\texttt{ID}_1(t)$ is the tail node of $\texttt{ID}_2(t)$, and the tail node of $\texttt{ID}_1(t)$ is the head node of $\texttt{ID}_2(t)$. Then, the baseline task sequence is a sequence of such task IDs, where for each task $t \in T$, either $\texttt{ID}_1(t)$ or $\texttt{ID}_2(t)$ appears exactly once. When traversing the task sequence, the vehicle follows the shortest path from the tail node of the former task ID to the head node of the latter task ID. The shortest paths are not shown in the task sequence, which can be efficiently computed by Dijkstra's algorithm. The recourse policy has a heuristic function of the current routing state such as the current location and remaining capacity of the vehicle. Whenever the vehicle completes the current service, the recourse policy calculates the function value for the next task in the task sequence based on the current state. If the value is smaller than zero, then the vehicle returns to the depot to replenish its capacity, and then goes to the next task from the depot. Otherwise, it goes straight to serve the next task. Since the recourse policy makes decisions on when to return to the depot, there is no delimiter in the task sequence.

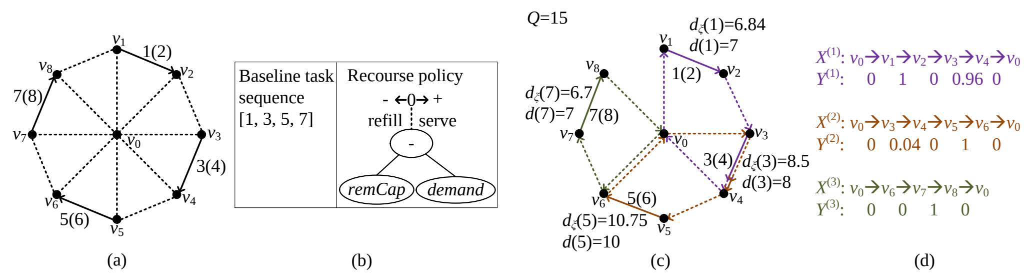


Figure 1: (a) An example of a UCARP instance and (b) an individual under the proposed solution-policy representation for it. (c) and (d) show the feasible routes for the UCARP instance in (a) based on the individual in (b). d_ξ and d represent the actual and expected demands of tasks, respectively.

Figure 1 shows an example of a UCARP instance (Figure 1(a)) and a solution under the newly proposed representation (Figure 1(b)). Figure 1(a) shows a graph with four edge tasks (represented as solid lines), corresponding to eight task IDs (indexed from 1 to 8). Each task is associated with two IDs next to it. The one outside the parenthesis indicates the current direction, and the one in the parenthesis indicates the opposite direction. As shown in Figure 1(b), the baseline task sequence is [1, 3, 5, 7], and the recourse policy has a heuristic function of (*remCap* − *demand*), where *remCap* means the remaining capacity of the vehicle, and *demand* stands for the demand of the next task. In other words, the vehicle will return to the depot to replenish if its remaining capacity is smaller than the demand of the next task.

3.2 Solution Execution

Given a sampled UCARP instance ξ, executing an individual (the baseline task sequence seq plus recourse policy $h(\cdot)$) can construct a feasible solution, i.e. a set of feasible routes. The execution process is described in Algorithm 1. Initially, a vehicle with an empty load is located at the depot and ready to serve. Then, each time when the vehicle becomes ready (initially and upon each service completion), it calculates the heuristic value $h(seq_i)$ of the next task ID in the baseline task sequence (line 4). If its value is smaller than zero, then it returns to the depot and replenishes (lines 6–8). Here, the function $\texttt{GoTo}(X^{(k)}, Y^{(k)}, v)$ updates the current route $(X^{(k)}, Y^{(k)})$ by traversing through the current location (the last vertex of $X^{(k)}$) to the vertex v via the shortest path taking the possible edge failure into account. That is, whenever the vehicle reaches the head of an edge along the shortest path, it checks the actual presence of the edge. If the edge becomes absent (e.g. temporary roadwork), it recalculates the shortest path based on the latest information and follows the new shortest path. Details of the $\texttt{GoTo}(\cdot)$ function can be found in (Liu et al., 2017).

Algorithm 1 $\texttt{Execution}(seq, h(\cdot), \xi)$

```
Input: An individual $(seq, h(\cdot))$, an uncertain instance $\xi$
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: for $i = 0 \rightarrow (|seq| - 1)$ do
4:     Calculate the heuristic value $h(seq_i)$;
5:     if $i \neq 0$ and $h(seq_i) < 0$ then                ▷ return to depot and replenish
6:         $\texttt{GoTo}(X^{(k)}, Y^{(k)}, v_0)$;
7:         $X \leftarrow X \cup X^{(k)}, Y \leftarrow Y \cup Y^{(k)}, k \leftarrow k + 1$;
8:         $X^{(k)} \leftarrow (v_0), Y^{(k)} \leftarrow (), loc \leftarrow v_0, \bar{Q} \leftarrow Q$;
9:     end if
10:    $\texttt{GoTo}(X^{(k)}, Y^{(k)}, head(seq_i))$;
11:    if $\bar{Q} < d_\xi(seq_i)$ then                          ▷ route failure
12:        $\theta \leftarrow \bar{Q}/d_\xi(seq_i)$;                      ▷ served fraction
13:        $X^{(k)} \leftarrow (X^{(k)}, tail(seq_i)), Y^{(k)} \leftarrow (Y^{(k)}, \theta)$;
14:        $\texttt{GoTo}(X^{(k)}, Y^{(k)}, v_0)$;
15:        $X \leftarrow X \cup X^{(k)}, Y \leftarrow Y \cup Y^{(k)}, k \leftarrow k + 1$;
16:        $X^{(k)} \leftarrow (v_0), Y^{(k)} \leftarrow (), loc \leftarrow v_0, \bar{Q} \leftarrow Q$;
17:        $\texttt{GoTo}(X^{(k)}, Y^{(k)}, head(seq_i))$;
18:        $X^{(k)} \leftarrow (X^{(k)}, tail(seq_i)), Y^{(k)} \leftarrow (Y^{(k)}, 1 - \theta)$;
19:        $loc \leftarrow tail(seq_i), \bar{Q} \leftarrow \bar{Q} - (1 - \theta) \cdot d_\xi(seq_i)$;
20:    else                                                       ▷ service successful
21:        $X^{(k)} \leftarrow (X^{(k)}, tail(seq_i)), Y^{(k)} \leftarrow (Y^{(k)}, 1)$;
22:        $loc \leftarrow tail(seq_i), \bar{Q} \leftarrow \bar{Q} - d_\xi(seq_i)$;
23:    end if
24: end for
25: $\texttt{GoTo}(X^{(k)}, Y^{(k)}, v_0)$;                             ▷ return to depot
26: return $S = (X, Y)$;
```

Then, the vehicle goes to the head of the next task seq_i, either from the depot if after replenish, or from the current location if going straight (line 10). After that, it starts to serve the task seq_i. If the actual demand $d_\xi(seq_i)$ is greater than the remaining capacity (line 11), a route failure occurs and the vehicle returns to the depot in the middle of the service to replenish (lines 12–19). Concretely, the vehicle neglects the unserved fraction of demand, moves to the tail of seq_i, then goes back to the depot by $\texttt{GoTo}(X^{(k)}, Y^{(k)}, v_0)$. After that, it creates a new empty route, and goes from the depot to the head of seq_i, and then finishes the remaining demand of seq_i. Otherwise, the service is successful (lines 21–22).

Figures 1(c) and (d) show the execution process of the individual shown in Figure 1(b) ($seq = [1, 3, 5, 7]$, $h(\cdot) = remCap - demand$) given a sample of the UCARP instance shown in Figure 1(a). The actual routes of the sampled instance are shown in Figure 1(c), where all the present edges have a deadheading cost of 1. The actual demand (d_ξ) and the expected demand (d) of each task are shown next to it. In this example, the vehicle starts from the depot v_0, traverses to v_1 and completes the task ID 1 successfully. After serving task ID 1, its remaining capacity becomes $15 - 6.84 = 8.16$. Then, it calculates the heuristic value of the next task ID 3, and finds that $h(3) = remCap - d(3) = 8.16 - 8 > 0$. Therefore, it decides to go straight to serve the task ID 3. However, the actual demand of task ID 3 ($d_\xi(3) = 8.5$) is greater than the expected demand and the remaining capacity. In this case, a route failure occurs, and the vehicle has to go back to the depot after serving $8.16/8.5 = 96\%$ of the demand of task ID 3. The remaining 4% of the task demand is then served by the new route after the replenish.

After completing the service of the task ID 3 in route 2, the remaining capacity becomes $15 - (8.5 - 8.16) = 14.66$. Then, the heuristic value of the task ID 5 is $14.66 - 10 > 0$, and the vehicle goes straight to serve it. In practice, the actual demand of the task ID 5 is also smaller than the remaining capacity ($10.75 < 14.66$). Therefore, the service of the task ID 5 is successful. After serving the task ID 5, the remaining capacity becomes $14.66 - 10.75 = 3.89$. This is smaller than the expected demand of the task ID 7, i.e. $h(7) = 3.89 - 7 < 0$. Therefore, the vehicle decides to return to the depot to replenish, and creates a new route to serve the task ID 7. The resultant feasible routes are shown in Figure 1(d).

4 Solution-Policy Co-evolver

A CC algorithm (Potter and De Jong, 2000) (as shown in Figure 2) is designed to evolve the individuals under the proposed solution-policy representation. The framework of the algorithm, namely Solution-Policy Co-evolver (SoPoC) consists of two subpopulations, one for the baseline task sequence, and the other for the recourse policy, i.e. the heuristic function. During the evolution of one population, each of its individual is evaluated by collaborating with the current best individual (so-called *representative*) from the other subpopulation. In this way, the subpopulations interact with one another through fitness evaluation. It can be seen that SoPoC is based on the asynchronised version of CC, in which the representatives are updated as soon as a subpopulation has been evaluated.

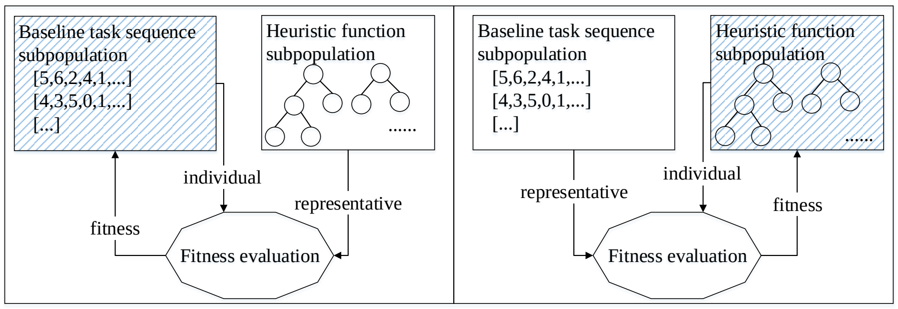


Figure 2: The framework of the Solution-Policy Co-evolver. The figure shows the fitness evaluation phase of SoPoC from the perspective of each of the two subpopulations.

The proposed SoPoC is described in Algorithm 2. It contains two subpopulations: the baseline task sequence subpopulation P_B and the heuristic function subpopulation

P_H. At first, P_H is randomly initialised, P_B is initialised using the same way as Wang et al. (Wang et al., 2016) (i.e. either by heuristics or at random), and the representatives seq^* and $h^*(\cdot)$ are randomly selected from the initial P_B and P_H respectively. Then, the heuristic function subpopulation and the baseline task sequence subpopulation are evolved one by one in an on-line fashion until the stop condition is satisfied. The fitness of each individual in one subpopulation is calculated by collaborating with the best-so-far individual from the other subpopulation (lines 6 and 13). To evaluate the fitness value, a set of sampled training instances are generated. The training instances are sampled from the same UCARP instance, so that they follow the same random distribution. At each generation, the representatives are updated by the new individuals with better fitness values (smaller fitness values in this minimisation case). Finally, the representatives seq^* and $h^*(\cdot)$ are combined together to form the final best-so-far solution.

Algorithm 2 The Solution-Policy Co-evolver

```
Input: A UCARP instance $\mathcal{I}$, a set of sampled training instances $\Xi_{\text{train}}(\mathcal{I})$ for $\mathcal{I}$
Output: An individual $(seq^*, h^*(\cdot))$
1: Initialise the subpopulations $P_B$ and $P_H$;
2: Randomly select $seq^* \in P_B$ and $h^*(\cdot) \in P_H$;
3: while stopping condition is not satisfied do
4:     //The heuristic function subpopulation evolution
5:     for each $h(\cdot) \in P_H$ do
6:         $fit(h(\cdot)) \leftarrow$ Evaluate$(seq^*, h(\cdot), \Xi_{\text{train}}(\mathcal{I}))$;
7:     end for
8:     $h'(\cdot) \leftarrow \arg\min_{h(\cdot)\in P_H} fit(h(\cdot))$;
9:     if $fit(h'(\cdot)) < fit(h^*(\cdot))$ then $h^*(\cdot) \leftarrow h'(\cdot)$; end if
10:    Generate the new heuristic function subpopulation $P_H$;
11:    //The baseline task sequence subpopulation evolution
12:    for each $seq \in P_B$ do
13:        $fit(seq) \leftarrow$ Evaluate$(seq, h^*(\cdot), \Xi_{\text{train}}(\mathcal{I}))$);
14:    end for
15:    $seq' \leftarrow \arg\min_{seq\in P_B} fit(seq)$;
16:    if $fit(seq') < fit(seq^*)$ then $seq^* \leftarrow seq'$; end if
17:    Generate the new baseline task sequence subpopulation $P_B$;
18: end while
19: return $(seq^*, h^*(\cdot))$;
```

As can be seen from Algorithm 2, SoPoC has three important ingredients: (1) the fitness evaluation `Evaluate`$(\cdot)$, (2) the baseline task sequence optimiser and (3) the heuristic function optimiser. The fitness evaluation is based on the execution process described in Section 3.2. Specifically, for each sampled training instance, a feasible solution is constructed by executing the individual $(seq, h(\cdot))$ on it. Then, the fitness value is calculated based on the total costs of all the constructed feasible solutions.

For the subcomponent optimisers, one can theoretically choose any optimisation algorithms that are effective in optimising the corresponding subcomponents. In this paper, we select EDASLS (Wang et al., 2016) as the baseline task sequence optimiser, and GPHH (Liu et al., 2017) as the heuristic function optimiser. EDASLS and GPHH are the current state-of-the-art algorithms for their corresponding subcomponents, and we expect them to show consistently competitive performance when used in SoPoC.

Next, we will describe the three ingredients in more detail.

4.1 Fitness Evaluation

The pseudocode of the fitness evaluation of an individual $(seq, h(\cdot))$ on a set of training instances $\Xi_{\text{train}}(\mathcal{I})$ is described in Algorithm 3. It records a set of total costs $\mathcal{C}$ by executing the individual for each training instance $\xi \in \Xi_{\text{train}}(\mathcal{I})$ (line 3). Then, the fitness value is calculated based on $\mathcal{C}$. If it is the average performance, then the average value of the total costs in $\mathcal{C}$ is returned. Otherwise, if the worst-case performance is desired, then the worst (i.e. maximal) total cost from $\mathcal{C}$ is returned.

Algorithm 3 `Evaluate`$(seq, h(\cdot), \Xi_{\text{train}}(\mathcal{I}))$

```
Input: an individual $(seq, h(\cdot))$, a set of sampled training instances $\Xi_{\text{train}}(\mathcal{I})$
Output: the fitness value
 1: $\mathcal{C} \leftarrow \emptyset$;
 2: for $\xi \in \Xi_{\text{train}}(\mathcal{I})$ do
 3:     $(X_\xi, Y_\xi) \leftarrow$ Execution$(seq, h(\cdot), \xi)$;
 4:     $\mathcal{C} \leftarrow \mathcal{C} \cup C(X_\xi, Y_\xi)$;
 5: end for
 6: if fitness is average performance then
 7:     return $\frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} C$;
 8: else                                        ▷ fitness is worst-case performance
 9:     return $\max_{C \in \mathcal{C}} C$;
10: end if
```

4.2 Subcomponent Optimisers

We directly adopted EDASLS (Wang et al., 2016) for optimising the baseline task sequence. Details of EDASLS can be found in the original literature (Wang et al., 2016) and the supplementary file.

In the GPHH, we represent a heuristic function as a Lisp tree, which is the most commonly used representation for GP. The heuristic function takes the routing state attributes, which act as terminals in the tree-based representation. Here, we consider the following attributes as the terminals: (1)*demand*: the expected demand of the task ID; (2)*remCap*: the remaining capacity of the vehicle; (3) *rCost*: the cost of traversing from the current location to the head of the task ID; (4)*sCost*: the cost of serving the task ID; (5)*dCost*: the cost from the tail of the task ID to the depot; and (6)*constant*: a random constant. The function set is set to $\{+, -, \times, /, \max, \sin\}$. The function / is protected. That is, it returns 1 instead of $+\infty$, -1 instead of $-\infty$, and 0 instead of `NaN`.

At each generation of SoPoC, *one generation* of EDASLS and GPHH are applied in turn to evolve the two components. The detailed description of the subcomponent optimisers is given in the supplementary file.

4.3 Summary

Finally, we conceptually compare SoPoC with the state-of-the-art algorithms (i.e. EDASLS (Wang et al., 2016) and GPHH (Liu et al., 2017)) for UCARP. The differences between them are as follows:

- Compared with EDASLS (Wang et al., 2016), SoPoC is more flexible. EDASLS only optimises the task sequence, and simply uses a greedy recourse policy (i.e. keeps serving the next task until the route failure occurs). On the other hand, SoPoC optimises the recourse policy along with the task sequence, so that different task sequences can have different recourse policies. Due to this higher flexibility, SoPoC is expected to have a higher chance of finding better solutions.

- Compared with GPHH (Liu et al., 2017), SoPoC is more robust. GPHH does not have any baseline task sequence, but decides the next task in real time by prioritising all the unserved feasible tasks. In this way, even for the same UCARP instance (i.e. the same road network and distributions of the random variables), GPHH can generate dramatically different routes for different sampled instances, leading to unstable solution quality especially for the large and complex instances. SoPoC contains a baseline task sequence, and the actual task sequences are expected to be similar to the baseline for any sampled instance. Based on the robust and predictive baseline, the solution quality is expected to be more stable in SoPoC.

5 Experiment Design

In this section, we first present how to generate dataset and split them into training and test groups for experiments. Then, we give parameter settings. Specially, we analysis the sensitivity of the key parameter ρ, which represents the ratio of the numbers of fitness evaluations between EDASLS and GPHH per generation.

5.1 Dataset

Following the idea in Mei et al. (2010), we transformed the benchmark *gdb* and *val* static CARP datasets to UCARP datasets. The *gdb* set contains 23 instances (denoted by 1, 2, ... in Table 2) with 7 to 27 vertices and 11 to 55 edges. The *val* set contains 34 instances with 24 to 50 vertices and 34 to 97 edges. The *val* set comes from 10 different networks. The instances with the same network *I* (denoted by *IA*, *IB*, ..., as shown in Table 3) have different vehicle capacities. It can be seen that the lower the vehicle capacity is, the more cycles (routes) we need. Tables 2 and 3 show more detailed information of the *gdb* and *val* sets, where the column $(|V|, |E|)$ represents the numbers of vertices and edges in the graph and Q stands for the capacity. All the edges are tasks in *gdb* and *val* instances.

For each deterministic CARP instance, we generate a UCARP instance by transforming each task demand and each deadheading cost to a random variable following a Gamma distribution with the following properties:

1. The mean (expected) value of the distribution equals the static value given in the deterministic CARP data file;
2. The shape parameter is $k = 20$ so that the distribution approximates the normal distribution.

Then, for each CARP instance, we generate 120 sampled instances by randomly sampling the random variables independently using different random seeds. After that, we randomly split the 120 sampled instances into 90 training instances and 30 test instances (Liu et al., 2017). For each UCARP instance, we train the baseline task sequence plus the heuristic function using the 90 training instances, and then test their performance on the 30 unseen test instances.

For each algorithm, 30 independent runs were conducted. For each run, the training and tested fitnesses are defined as follows:

$$f_{TR}(seq, h(\cdot)) = \texttt{Evaluate}(seq, h(\cdot), \Xi_{\text{train}}), \tag{4}$$

$$f_{TE}(seq, h(\cdot)) = \texttt{Evaluate}(seq, h(\cdot), \Xi_{\text{test}}), \tag{5}$$

where Ξ_{train} and Ξ_{test} include the 90 training instances and the 30 test instances respectively, and $\texttt{Evaluate}(\cdot)$ is shown in Algorithm 3.

In order to improve the training efficiency and the generalisation, the mini-batch learning process used in (Liu et al., 2017) is adopted, i.e. we split the 90 training instances into 18 mini-batches, each consisting of 5 instances. Then, during the training process, we rotated the mini-batches generation by generation (e.g. mini-batch 1 in generations 1, 19, . . . ; mini-batch 2 in generations 2, 20, . . .). The best individual obtained by the last generation is returned as the output of the algorithm.

In addition, although drawn from the same UCARP instance, different sample instances can still have much different solution quality. To minimise the bias caused during the training process, we normalise the total cost obtained for each training instance $\xi \in \Xi_{\text{train}}$ as follows:

$$\tilde{C}(seq, h(\cdot), \xi) = \frac{C(seq, h(\cdot), \xi)}{C_{\mathit{ref}}}, \tag{6}$$

where C_{ref} is obtained by applying a reference heuristic on the instance. The reference heuristic is the heuristic 5 used in the Path Scanning heuristic (Lacomme et al., 2004), which is an effective manually designed heuristic for CARP. Heuristic 5 works as follows: the task with the maximal distance to the depot is selected next if the vehicle is less than half-full; Otherwise, the task with the minimal distance to the depot is selected next.

5.2 Parameter Settings

There are three categories of parameters in SoPoC: (1) the *overall* parameters such as number of fitness evaluation, (2) the *EDASLS* parameters for evolving the baseline task sequence and (3) the *GPHH* parameters for evolving the heuristic function. For the sake of fair comparison, we set the EDASLS and GPHH parameters to the same values as in the original literatures (Wang et al., 2016; Liu et al., 2017), and set the maximal number of fitness evaluation per generation to 1024. In SoPoC, the balance between the evolution of the two subpopulations is important. To control the balance, we introduce a new parameter ρ, which stands for the ratio between the fitness evaluations of EDASLS and GPHH per generation. Its value will be determined by the sensitivity analysis conducted in Section 5.3. In addition, we set the number of fitness evaluations to 512000 so that all the compared algorithms can converge in most cases. The detailed parameter settings are shown in Table 1.

Table 1: The parameter settings of the compared algorithms.

Category	Parameter	Value
Overall	Total No. of Fitness Evaluation (FE)	512000
	Maximal No. of FE per generation	1024
	The ratio between No. of FE of EDASLS and GPHH per generation	ρ
EDASLS	Population size	120
	Maximal No. of samples per generation	$\frac{\rho}{1+\rho} * 1024 - 120$
	No. of sub segments	2
	Local search probability	0.1
	Bias ratio constant	0.005
GPHH	Population size	$\frac{1}{1+\rho} * 1024$
	Maximal tree depth	8
	Selection	Size-7 tournament
	Crossover/Mutation/Reproduction rates	0.8 / 0.15 / 0.05
	Elitism	Top 10 individuals

5.3 Sensitivity Analysis (ρ)

We investigated a wide range of ρ values, i.e. $1, 3, 5, 7, 9, 11, 13$ and 15. Three representative instances, i.e. *ugdb*21, *uval*8*C* and *uval*10*A*, are chosen based on the problem size (the number of tasks ranges from 33 to 97) and tightness of capacity constraint (the ratio between the total demands of all tasks and the capacity of vehicle ranges from 2.81 to 8.71). Figure 3 plots the box charts of the test results of SoPoC in terms of F_{avg}

((2)) with different values of ρ. The general pattern is that with the increment of ρ, the performance of SoPoC first decreases and then increases. Three instances all achieve the best mean value when $\rho = 9$. Then we use Wilcoxon rank sum test with the significance level of 0.05 to compare $\rho = 9$ and other values of ρ. In each subfigure of Figure 3, the red box charts filled with lines show that the corresponding results are significantly worse than that of $\rho = 9$. For example, on the instance of $uval10A$, $\rho = 1, 3, 5$ or 15 are all significantly worse than $\rho = 9$, and there are no statistical differences between $\rho = 7, 11$ or 13 and $\rho = 9$. From this analysis, we recommend that $\rho \in [7, 13]$ for SoPoC. Specifically, we set $\rho = 9$ in the subsequent experiments.

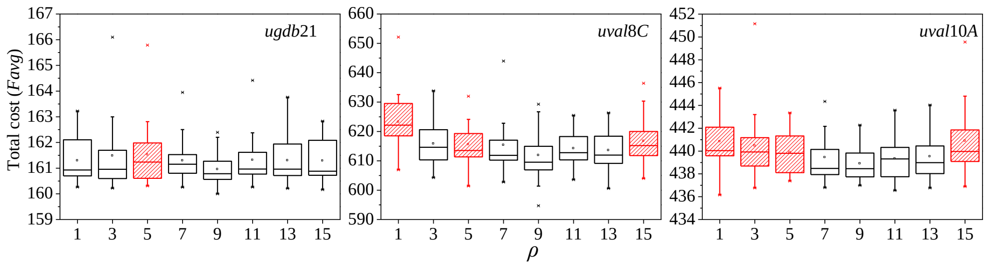


Figure 3: The box charts of the test performance of SoPoC in terms of F_{avg} with different values of ρ. The red boxes filled with lines mean that the results are significantly worse than the result with the minimal mean value (i.e. when $\rho = 9$ on all three instances).

6 Results and Discussions

In this section, we empirically evaluate the effectiveness of the proposed SoPoC by comparing it with EDASLS (Wang et al., 2016) and GPHH (Liu et al., 2017), which are the two state-of-the-art approaches for solving UCARP. We further conduct analysis to understand how the baseline task sequence and heuristic function work together to achieve potentially better performance.

6.1 Results

Tables 2 and 3 show the test performances of the compared algorithms on the *ugdb* and *uval* UCARP instances in terms of F_{avg} ((2)) and F_{wst} ((3)). We conducted Wilcoxon rank sum test with the significance level of 0.05 to compare SoPoC with EDASLS and GPHH. Specifically, for each UCARP instance, if the results of EDASLS or GPHH are significantly better than that of SoPoC, then the corresponding "Mean" entry is marked with "(+)". Otherwise, if the results are significantly worse than that of SoPoC, then the corresponding "Mean" entry is marked with "(-)".

6.1.1 Comparison with EDASLS

From Tables 2 and 3, it can be seen that SoPoC performed significantly better than EDASLS on 14 instances, and no worse on any instance in terms of F_{avg}. In terms of F_{wst}, SoPoC performed significantly better than EDASLS on 4 instances, but was defeated by EDASLS on only 2 instances (i.e. *uval7B* and *uval*10C). More specifically, it is found that SoPoC tended to outperform EDASLS on the instances with more routes (e.g. *uval*2C, *uval*3C, *uval9D* and *uval*10D in terms of F_{avg}). To illustrate this pattern more clearly, Figure 4 shows the relationship between the tightness of the capacity constraint and the test performance of SoPoC relative to that of EDASLS. In the figure, each point represents one UCARP instance. The x-axis is the $\sum_i d(i)/Q$ value of the instances, reflecting the minimum number of required routes, where $\sum_i d(i)$ is the total

Table 2: The test performances over the 30 independent runs of the compared algorithms on the *ugdb* instances.

	$(\|V\|, \|E\|)$	Q	F_{avg}						F_{wst}					
			EDASLS		GPHH		SoPoC		EDASLS		GPHH		SoPoC	
			Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.
1	(12,22)	5	347.73	4.66	354.07(-)	7.43	350.85	6.12	414.71	30.84	412.08	17.73	406.96	23.96
2	(12,26)	5	376.27	2.97	376.47	4.26	375.31	4.05	445.28	18.29	443.77	22.02	441.22	18.23
3	(12,22)	5	308.60	2.43	313.00(-)	4.78	309.25	1.83	356.71	10.40	365.85(-)	18.74	356.59	10.78
4	(11,19)	5	336.31(-)	5.38	335.89(-)	7.08	331.04	4.60	396.34	22.82	390.06(+)	24.78	402.24	17.97
5	(13,26)	5	424.94	4.94	427.60(-)	5.53	424.57	4.88	528.21	28.66	507.57(+)	17.11	523.79	25.65
6	(12,22)	5	331.25	2.32	341.17(-)	6.28	330.18	1.76	372.03	10.82	389.53(-)	21.27	371.01	13.02
7	(12,22)	5	367.30	3.53	367.01	6.66	368.44	4.95	435.85	20.63	431.89	15.36	436.88	15.39
8	(27,46)	27	410.00(-)	3.69	421.01(-)	4.53	404.60	4.52	476.95(-)	14.56	480.29(-)	14.52	466.92	14.96
9	(27,51)	27	368.90(-)	3.41	383.19(-)	5.90	365.68	4.35	421.47	12.98	436.09(-)	12.56	416.64	12.05
10	(12,25)	10	292.02	4.26	299.20(-)	3.90	292.57	3.59	320.03	9.07	339.42(-)	13.82	323.31	11.61
11	(22,45)	50	418.48	4.06	439.98(-)	5.12	419.03	3.89	450.60	14.96	490.62(-)	13.60	452.65	11.38
12	(13,23)	35	554.29	5.78	607.00(-)	12.59	554.95	10.71	682.92	28.19	759.10(-)	44.32	684.78	31.12
13	(10,28)	41	572.75	3.62	584.81(-)	7.94	572.99	3.46	623.87	16.66	651.68(-)	39.87	631.40	15.50
14	(7,21)	21	105.92	0.82	110.36(-)	1.41	105.87	0.67	120.05	4.08	128.30(-)	4.15	119.00	2.65
15	(7,21)	37	58.65	0.33	58.62	0.38	58.57	0.28	63.05	1.89	63.52	1.36	62.71	1.70
16	(8,28)	24	132.80	0.86	134.55(-)	1.11	132.63	1.02	144.02	3.84	149.64(-)	3.86	143.34	3.80
17	(8,28)	41	91.21	0.50	91.57(-)	0.59	91.18	0.33	97.59	2.43	98.62	2.22	97.65	2.42
18	(9,36)	37	170.61	1.32	168.02(+)	1.47	170.79	0.99	182.44	5.70	183.15	4.72	184.52	4.79
19	(8,11)	27	63.70	0.42	64.65(-)	0.83	63.70	1.22	82.90	3.28	83.84	2.59	82.07	5.12
20	(11,22)	27	126.86	1.30	130.07(-)	2.18	127.21	0.91	139.96	6.43	147.74(-)	7.16	138.80	7.12
21	(11,33)	27	161.18	0.71	165.94(-)	1.90	160.95	0.59	176.83	4.32	182.87(-)	5.24	175.21	5.11
22	(11,44)	27	207.49	1.05	209.13(-)	1.12	207.38	0.94	222.75(-)	4.37	224.05(-)	4.78	219.40	4.96
23	(11,55)	27	245.99	1.23	248.54(-)	1.46	245.63	1.57	266.92	5.52	266.56	5.97	264.88	5.46

expected demand of all the tasks in one instance. The y-axis indicates the test performance of EDASLS over the test performance of SoPoC. A larger y value indicates that the advantage of SoPoC over EDASLS is more obvious. The points are marked with three different styles. The marker "+"("-") means that EDASLS performed significantly better (worse) than SoPoC. The marker "$\circ$" means that there is no statistical difference between SoPoC and EDASLS.

From Figure 4, we have the following observations:

- The x and y values are positively correlated. For F_{avg}, the correlation coefficient is 0.5184. For F_{wst}, the correlation coefficient is 0.6175.

- Out of the 43 instances with $\sum_i d(i)/Q < 6.5$, SoPoC significantly outperformed EDASLS on only 5 (12%) instances in terms of F_{avg} and no instances in F_{wst}. In addition, SoPoC performed significantly worse than EDASLS on 2 instances.

- In contrast, out of the 14 instances with $\sum_i d(i)/Q > 6.5$, SoPoC significantly outperformed EDASLS on 9 (64%) instances in F_{avg} and 4 (29%) instances in F_{wst}.

The reasons of the above observations can be explained as follows. If $\sum_i d(i)/Q$ is larger, then more routes are required to serve all the tasks, which leads to a higher probability of route failures. In this case, it is more important to design more effective recourse policies to handle the route failures rather than simply using the greedy recourse policy. The above observations show that for the instances with larger $\sum_i d(i)/Q$ values, SoPoC managed to find more effective recourse policies than the greedy recourse policy in more cases.

Table 3: The test performances over the 30 independent runs of the compared algorithms on the *uval* instances.

	$(\lvert V\rvert,\lvert E\rvert)$	Q	F_{avg}						F_{wst}					
			EDASLS		GPHH		SoPoC		EDASLS		GPHH		SoPoC	
			Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.
1*A*	(24,39)	200	173.48	1.42	174.78(-)	1.57	174.04	2.39	188.64	3.92	192.44(-)	5.54	188.73	2.97
1*B*	(24,39)	120	182.04	0.48	190.32(-)	4.92	182.34	1.29	197.21	3.32	218.90(-)	7.59	198.10	5.70
1*C*	(24,39)	45	280.89	3.53	301.14(-)	4.44	279.62	3.77	324.79	15.66	353.41(-)	9.58	320.67	9.49
2*A*	(24,34)	180	232.44	1.38	249.21(-)	6.17	233.01	1.78	261.98	4.13	310.35(-)	18.35	264.45	6.22
2*B*	(24,34)	120	275.47(-)	5.11	292.30(-)	5.15	272.46	3.02	322.78	13.55	382.05(-)	19.87	326.41	17.31
2*C*	(24,34)	40	549.62(-)	5.27	558.54(-)	6.67	534.95	8.62	640.05(-)	21.26	656.81(-)	16.19	624.79	26.20
3*A*	(24,35)	80	83.75	0.40	85.60(-)	1.20	83.92	0.54	98.16	1.64	101.43	8.16	98.59	2.35
3*B*	(24,35)	50	97.32	2.07	103.79(-)	2.83	97.09	2.80	114.43	4.21	132.83(-)	7.97	115.40	6.29
3*C*	(24,35)	20	168.68(-)	3.07	170.63(-)	1.41	163.94	2.17	192.84	6.30	200.87(-)	4.90	189.97	5.88
4*A*	(41,69)	225	424.97	5.42	438.62(-)	4.56	425.05	5.45	462.73	12.70	482.03(-)	14.12	463.56	13.92
4*B*	(41,69)	170	458.98	7.07	470.92(-)	8.26	459.45	10.59	505.42	11.95	526.51(-)	17.65	512.89	20.11
4*C*	(41,69)	130	505.57(-)	8.31	516.09(-)	7.20	497.05	8.61	572.37	19.36	599.44(-)	22.68	564.49	24.47
4*D*	(41,69)	75	650.57(-)	11.56	664.48(-)	6.09	627.68	9.21	738.42(-)	18.21	754.73(-)	18.49	719.26	26.91
5*A*	(34,65)	220	447.09	5.79	469.86(-)	6.97	446.10	4.86	496.82	14.85	530.80(-)	14.53	500.90	17.79
5*B*	(34,65)	165	480.16	5.35	501.31(-)	7.43	480.10	4.76	534.64	14.70	565.40(-)	14.13	536.38	15.57
5*C*	(34,65)	130	531.57	6.83	543.29(-)	5.12	529.68	5.65	588.02	16.84	616.14(-)	15.26	598.15	25.34
5*D*	(34,65)	75	691.11	8.93	705.64(-)	5.22	687.44	6.89	802.26	32.58	797.99	23.05	799.92	30.26
6*A*	(31,50)	170	231.57	1.44	235.13(-)	5.11	231.41	1.42	252.74	4.33	268.45(-)	10.55	253.73	3.75
6*B*	(31,50)	120	256.04	2.90	267.83(-)	2.56	255.63	4.87	289.20	13.92	301.04(-)	9.50	288.99	7.71
6*C*	(31,50)	50	373.96(-)	4.37	393.07(-)	5.94	371.43	4.37	435.66	16.68	450.24(-)	9.36	430.28	14.95
7*A*	(40,66)	200	284.43	0.01	284.57	2.61	284.42	0.02	300.42	1.02	324.69(-)	15.54	300.68	0.81
7*B*	(40,66)	150	285.35(-)	0.30	294.86(-)	4.05	285.16	0.17	301.07(+)	0.14	333.72(-)	15.18	301.60	1.37
7*C*	(40,66)	65	374.04	2.62	395.15(-)	3.74	372.41	3.35	418.01	6.76	450.26(-)	11.01	416.63	10.94
8*A*	(30,63)	200	399.02	4.07	414.56(-)	4.28	398.61	3.75	417.45	9.29	464.51(-)	18.78	417.57	12.50
8*B*	(30,63)	150	424.64	4.56	445.88(-)	4.90	425.65	5.29	464.14	15.12	511.49(-)	16.18	464.97	13.01
8*C*	(30,63)	65	615.69(-)	5.63	633.53(-)	8.20	611.89	8.22	693.37	18.89	710.43(-)	18.43	692.69	19.02
9*A*	(50,92)	235	328.80	1.16	341.93(-)	3.25	328.77	1.13	337.92	2.54	372.04(-)	10.24	339.13	3.30
9*B*	(50,92)	175	335.32	2.35	356.32(-)	2.49	335.51	2.00	349.79	5.63	387.61(-)	8.06	352.70	8.04
9*C*	(50,92)	140	349.69	3.37	371.28(-)	2.10	350.02	3.10	376.67	7.90	409.08(-)	10.86	376.82	8.93
9*D*	(50,92)	70	465.25(-)	7.86	473.67(-)	4.98	458.95	6.31	537.30	21.48	529.46	11.60	527.25	15.72
10*A*	(50,97)	250	439.77(-)	1.63	454.68(-)	2.41	438.90	1.51	467.43	12.52	494.67(-)	9.33	466.09	7.61
10*B*	(50,97)	190	460.34	3.04	470.84(-)	3.19	461.26	3.34	495.78	10.48	509.95(-)	12.65	498.52	11.96
10*C*	(50,97)	150	493.36	5.29	492.68	3.77	492.67	4.66	534.01(+)	12.14	543.53	16.13	544.34	12.48
10*D*	(50,97)	75	642.82(-)	10.88	613.14(+)	3.87	632.76	8.19	730.73	22.54	673.85(+)	15.54	721.15	22.88

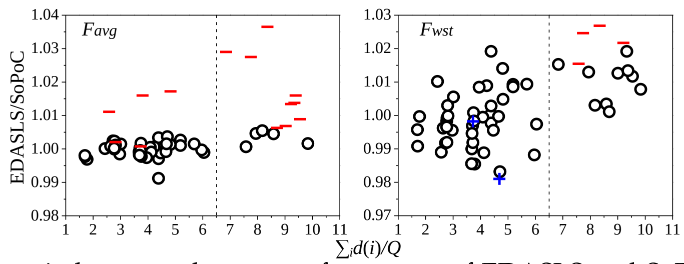


Figure 4: The ratio between the test performances of EDASLS and SoPoC versus the $\sum_i d(i)/Q$ value over all *ugdb* and *uval* instances in terms of F_{avg} and F_{wst}. "+"("-") means that EDASLS performed significantly better (worse) than SoPoC, and ∘ indicates that there is no statistical difference between them under Wilcoxon rank sum test with $\alpha = 0.05$.

6.1.2 Comparison with GPHH

The advantage of SoPoC over GPHH was much more obvious than over EDASLS. In terms of F_{avg}, SoPoC significantly outperformed GPHH on 19 out of the 23 *ugdb* instances, and 31 out of the 34 *uval* instances. SoPoC was significantly beaten by GPHH on only 1 *ugdb* instance and 1 *uval* instance. In terms of F_{wst}, SoPoC performed significantly better than GPHH on 13 *ugdb* instances and 29 *uval* instances, while significantly worse on only 2 *ugdb* instances and 1 *uval* instances. There is no obvious relationship between the advantage of SoPoC over GPHH and the $\sum_i d(i)/Q$ value. When looking into the instances where SoPoC performed significantly worse than GPHH, we found that the $\sum_i d(i)/Q$ values ranges from 4.13 (*ugdb*18) and 9.39 (*uval*10*D*).

6.1.3 Generalisation

To investigate the generalisation behaviour, we compared the training and test performances of the tested algorithms. Specifically, for each *ugdb* and *uval* instance, we conducted pairwise tests between the compared algorithms over the 30 independent runs using Wilcoxon rank sum test with the significance level of 0.05 in terms of both the training fitness and test fitness. Then, for the *ugdb* and *uval* datasets and each pair of algorithms (A, B), we count the number of instances where algorithm A performed significantly better than B ("Win"), where A performed significantly worse than B ("Lose") and where there is no statistical difference between A and B ("Draw").

Table 4 shows the Win-Draw-Lose results of the compared algorithms on the *ugdb* and *uval* datasets in terms of training and test fitnesses. Each entry stands for the comparison between the algorithm in the row and the algorithm in the column. From the table, we have the following observations:

- For the *ugdb* dataset, SoPoC performed similarly with EDASLS in terms of training fitness, and tended to be significantly worse than GPHH (e.g. 4 wins and 14 losses in F_{wst}).
- For the *uval* dataset, the training performance of SoPoC performed significantly better than EDASLS (e.g. 9 wins and no loss when compared with EDASLS in F_{avg}, and draw with GPHH (e.g. both of SoPoC and GPHH win on 13 instances in F_{avg}).
- The test performance of SoPoC was better than that of EDASLS. For example, in terms of F_{avg}, SoPoC significantly outperformed EDASLS on 3 *ugdb* instances and 11 *uval* instances, and never performed significantly worse than EDASLS.
- The test performance of SoPoC was much better than that of GPHH. For example, in terms of F_{avg}, SoPoC significantly outperformed GPHH on 19 *ugdb* instances and 31 *uval* instances, while was defeated by GPHH on only 1 *ugdb* instance and 1 *uval* instance.
- Overall, the relative performance of SoPoC to EDASLS and GPHH was much better on the test set than on the training set. The pattern is especially obvious when compared SoPoC with GPHH. This is because GPHH is quite sensitive to the problem instances. When applying the heuristics evolved by GPHH to unknown test instances, the heuristics would generate dramatically different routes, leading to the solution quality fluctuate greatly. While SoPoC contains a baseline task sequence. It can make sure that the actual routes are similar for any training or test instances, which indicates that SoPoC is expected to have more stable solution quality.
- For both the *ugdb* and *uval* datasets, EDASLS showed much better generalisation than GPHH. In terms of both F_{avg} and F_{wst}, the Win-Draw-Lose results between

EDASLS and GPHH are much better in test fitness than in training fitness. This is consistent with our hypothesis, since EDASLS is expected to be more robust than GPHH.

Table 4: The training and test Win-Draw-Lose results using Wilcoxon rank sum test with the significance level of 0.05 in F_{avg} and F_{wst}.

			F_{avg}		F_{wst}	
			GPHH	SoPoC	GPHH	SoPoC
ugdb	train	EDASLS	9-5-9	2-20-1	3-8-12	2-21-0
		GPHH	–	10-5-8	–	14-5-4
	test	EDASLS	17-5-1	0-20-3	11-11-1	0-21-2
		GPHH	–	1-3-19	–	2-8-13
uval	train	EDASLS	11-7-16	0-25-9	8-12-14	2-30-2
		GPHH	–	13-8-13	–	15-12-7
	test	EDASLS	31-2-1	0-23-11	30-3-1	2-30-2
		GPHH	–	1-2-31	–	1-4-29

6.1.4 Computational Efficiency

In order to make sure that the comparisons are conducted in a fair environment, we compare the computational time of the compared algorithms in the training process on the *ugdb* and *uval* instances. As shown in Figure 5, the results are obtained by averaging the time in terms of F_{avg}. The F_{wst} measure shows similar patterns. From Figure 5, it can be seen that the computational time of the three algorithms are statistically the same, which is consistent with our expectation, since they were given the same number of fitness evaluations.

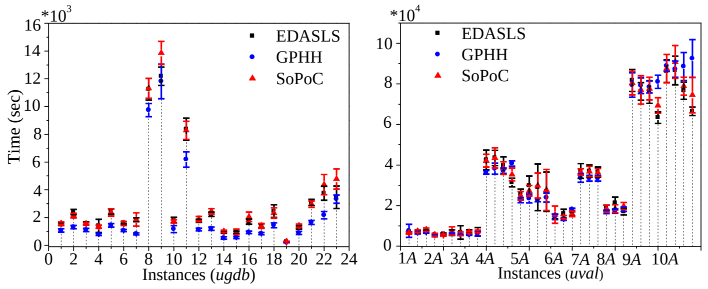


Figure 5: The computational time of the training process of compared algorithms on the *ugdb* and *uval* instances.

6.2 Further Analysis

To gain a better understanding, we conduct the following experiments to have a deep analysis on the results.

6.2.1 Convergence Curve

Figure 6 shows the convergence process of the compared algorithms in the test performance for three representative instances (i.e. *ugdb*21, *uval*8C and *uval*10*A*) in terms of F_{avg}. The F_{wst} measure shows similar patterns. In the subfigures, the x-axis represents the number of fitness evaluations, and the y-axis stands for the average total costs of the best-so-far solution over 30 runs in each generation.

From Figure 6, we have the following observations:

- GPHH converged much faster than EDASLS and SoPoC, but easier to get stuck in poor local optima (e.g. $ugdb21$ and $uval10A$). Additionally, GPHH obtained much better results than that of EDASLS and SoPoC at the beginning of the search process. This is because GPHH uses a greedy meta-algorithm, which always chooses the next task from the *nearest feasible* tasks (Liu et al., 2017). This also explains why GPHH is easier to get stuck in poor local optima than EDASLS and SoPoC.

- SoPoC and EDASLS had similar convergence patterns, but SoPoC tended to find better final results than EDASLS.

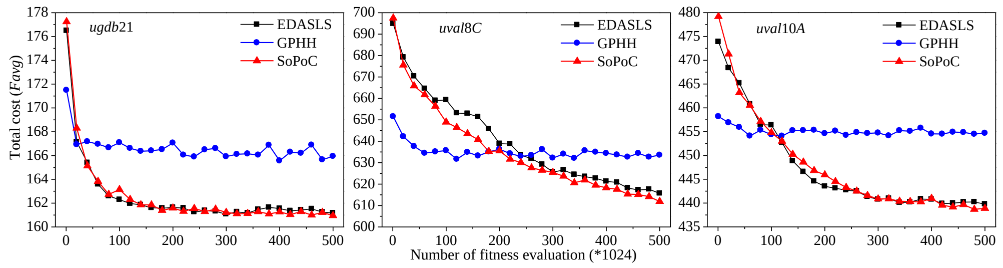


Figure 6: The convergence curves of the test performances of compared algorithms on the instances $ugdb21$, $uval8C$ and $uval10A$ in terms of F_{avg} along with the number of fitness evaluations.

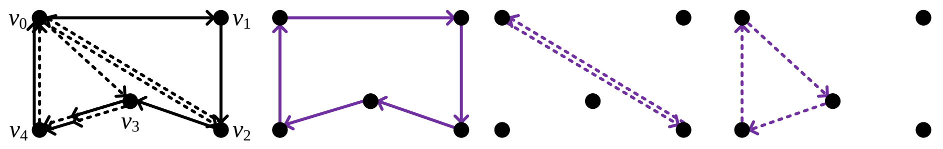
6.2.2 Cost Components

SoPoC is more similar to EDASLS than GPHH in that both SoPoC and EDASLS represent a solution as a baseline sequence and a recourse policy. The only difference is that EDASLS fixes the recourse policy as a greedy one (keep serving the next task until a route failure occurs), while SoPoC evolves a potentially more effective recourse policy.

To understand why SoPoC can generate better solutions than EDASLS, we analyse the structures of routes generated by them. The total cost is composed of two parts: (1) the preplanned cost caused by directly following the task sequence and traversing from one task to another through the shortest path (determined by the quality of the task sequence), and (2) the refill cost caused by returning to the depot and coming back to the interrupted location (determined by the recourse policy). The refill cost can be further divided into two parts: (1) expected refill cost and (2) unexpected (route failure) refill cost. The expected refill cost is induced when the recourse policy decides to go back to the depot to refill before going to the next task, i.e. before the potential route failure occurs. The unexpected refill cost, on the other hand, is induced when the recourse policy decides to continue with the service, but then a route failure occurs and the vehicle has to go back to the depot to refill in the middle of the service.

An illustration is given in Figure 7. In Figure 7(a), the vehicle starts from the depot v_0, and completes the services of the two tasks (v_0, v_1) and (v_1, v_2) (the solid lines) successfully. Then, after calculating the heuristic value of (v_2, v_3), the vehicle decides to return to v_0 to refill. It then comes back to v_2 and completes serving the task (v_2, v_3). Afterwards, the vehicle calculates the heuristic value of (v_3, v_4), and decides to go straight to serve it. However, the actual demand of (v_3, v_4) is greater than the remaining capacity, and a route failure occurs. The vehicle has to go back to the depot in the middle of the service to refill its capacity. It then returns to the interrupted place and finishes the remaining demand of (v_3, v_4). Finally, the vehicle completes the service of (v_4, v_0)

successfully. In this example, Figs. 7(b), 7(c) and 7(d) show the paths that induce the preplanned cost, expected refill cost and unexpected refill cost, respectively.



(a) paths for the total cost (b) paths for the preplanned cost (c) paths for the expected refill cost (d) paths for the unexpected refill cost

Figure 7: An example of how the total cost is divided into the preplanned cost, expected refill cost and unexpected refill cost.

Table 5 shows the total cost and the above components obtained by EDASLS and SoPoC on the instances in terms of F_{avg} that SoPoC significantly outperforms EDASLS. These instances are chosen as representatives to highlight the difference. The F_{wst} measure shows similar patterns. For each instance and each cost, the better algorithm is highlighted in bold. From the table, a general pattern is that SoPoC obtained smaller preplanned cost and expected refill cost, but a larger unexpected refill cost. In other words, SoPoC tends to have more route failures. However, SoPoC obtained significantly better total cost than EDASLS. To explain this, we investigated the actual routes more deeply, and observed that the route failures in SoPoC actually had a positive effect on the total cost. That is, the route failure often occurs along the shortest path to the depot, so that refilling in the middle of the service is better than before the service since the vehicle can serve a part of the demand with no extra cost, leaving less remaining demand for future routes. This can potentially reduce the number of refills in the future, and thus reduce the total cost. Figure 8 gives an example of our observation. In the example, the route failure occurs when serving (v_{20}, v_{10}) which is on the shortest path $(v_{20}, v_{10}, v_9, v_3, v_0)$ from v_{20} to the depot v_0. In this case, the expected and unexpected refill costs are equivalent, while the vehicle can serve 55% of the demand of (v_{20}, v_{10}) before the refill in the route failure case.

Table 5: Comparison between the cost components of SoPoC and EDASLS in terms of F_{avg}. For each instance and cost, the better algorithm is highlighted in bold.

Name	total		preplanned		expected		unexpected	
	EDASLS	SoPoC	EDASLS	SoPoC	EDASLS	SoPoC	EDASLS	SoPoC
*ugdb*4	336.31	**331.04**	**290.25**	298.27	27.47	**13.88**	**18.59**	18.89
*ugdb*8	410.00	**404.60**	276.33	**265.25**	109.07	**53.92**	**24.60**	85.43
*ugdb*9	368.90	**365.68**	284.47	**260.30**	65.17	**64.26**	**19.26**	41.12
*uval*2*B*	275.47	**272.46**	**260.30**	261.88	12.04	**3.26**	**3.13**	7.32
*uval*2*C*	549.63	**534.95**	259.43	**250.87**	241.81	**126.05**	**48.39**	158.02
*uval*3*C*	168.68	**163.94**	89.48	**83.63**	64.63	**41.82**	**14.57**	38.49
*uval*4*C*	505.57	**497.05**	428.01	**405.69**	**64.00**	78.11	13.56	**13.25**
*uval*4*D*	650.57	**627.68**	432.73	**408.24**	183.00	**168.01**	**34.84**	51.44
*uval*6*C*	373.96	**371.43**	263.37	**235.50**	95.89	**88.00**	**14.70**	47.94
*uval*7*B*	285.35	**285.16**	**284.31**	284.39	0.83	**0.00**	**0.20**	0.77
*uval*8*C*	615.69	**611.89**	429.29	**405.47**	153.87	**142.62**	**32.53**	63.80
*uval*9*D*	465.25	**458.95**	377.85	**343.22**	**72.72**	96.03	**14.67**	19.69
*uval*10*A*	439.77	**438.90**	436.84	**436.14**	2.30	**0.52**	**0.63**	2.24
*uval*10*D*	642.82	**632.76**	467.81	**451.08**	144.45	**136.18**	**30.56**	45.49

The above observation suggests that route failure is not always detrimental to the total cost. On the contrary, allowing route failure when it does not cause extra refill cost (e.g. the task is on the shortest path to the depot) can potentially lead to a better result.

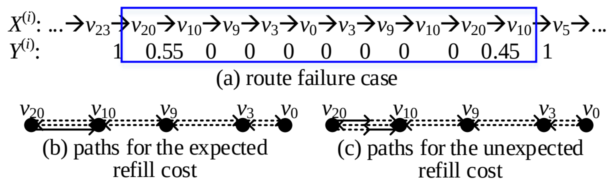

$X^{(i)}$: ...→v_{23}→v_{20}→v_{10}→v_9→v_3→v_0→v_3→v_9→v_{10}→v_{20}→v_{10}→v_5→...
$Y^{(i)}$: 1 0.55 0 0 0 0 0 0 0 0.45 1
(a) route failure case
v_{20} v_{10} v_9 v_3 v_0
(b) paths for the expected refill cost
v_{20} v_{10} v_9 v_3 v_0
(c) paths for the unexpected refill cost

Figure 8: (a) shows an example of a common route failure case in routes generated by SoPoC. The dotted lines in (b) and (c) illustrate the paths for the expected and unexpected refill costs for serving the task (v_{20}, v_{10}), respectively.

6.2.3 Structure of the Evolved Heuristics

We first analyse the frequency of the six terminals in the best heuristics evolved by SoPoC in the 30 runs. The results on the instances *uval*2*C* and *uval*9*B* in terms of F_{avg} are shown in Figure 9. It can be seen that among these six terminals, *remCap* and one of the *costs* (*rCost* in *uval*2*C* and *dCost* in *uval*9*B*) are the two most frequently used. Other instances show the same pattern.

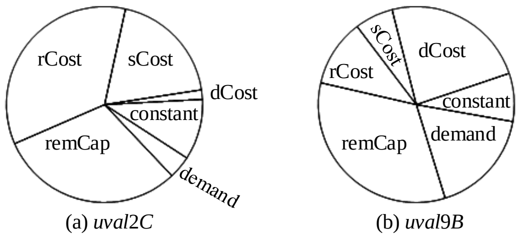


Figure 9: The pie plots of the frequency of terminals on the instances *uval*2*C* and *uval*9*B*.

Second, we find that the subtree of *remCap* minus one of the *costs* frequently appeared. Among all the best GP trees obtained by SoPoC over the 30 runs on *uval*2*C* and *uval*9*B*, we calculated the ratio of the occurrences of the subtrees ($-$ *remCap rCost*), ($-$ *remCap sCost*) or ($-$ *remCap dCost*) over the occurrences of all the depth-2 subtrees. The ratio is 47.02% for *uval*2*C* and 20% for *uval*9*B*. Equation (7) shows an arbitrarily selected heuristic function of SoPoC. It can be seen that among the depth-2 subtrees, ($-$ *remCap rCost*) appears 5 out of 6 times (highlighted). This observation can be interpreted as follows. If the remaining capacity is smaller, then the vehicle will be less likely to continue the next service (the value of the heuristic function is less likely to be positive). The vehicle will only continue the service if the costs related to the next task are small, indicating that even if route failure occurs, the extra refill cost is minimised. This is consistent with our intuition and our analysis about the cost components.

$$\begin{aligned}&(+\ (+\ (-\ (-\ \boldsymbol{(-\ remCap\ rCost)}\ rCost)\ sCost)\\&(-\ (+\ \boldsymbol{(-\ remCap\ rCost)}\ \boldsymbol{(-\ remCap\ rCost)})\\&(+\ sCost\ (max\ sCost\ \boldsymbol{(/\ remCap\ sCost)}))))\\&(+\ \boldsymbol{(-\ remCap\ rCost)}\ (-\ \boldsymbol{(-\ remCap\ rCost)}\ sCost)))\end{aligned} \tag{7}$$

7 Conclusions

In this paper, a new predictive-reactive approach to UCARP is proposed. In the new approach, a new solution representation is designed. The new representation contains a baseline solution and a recourse policy, and thus inherits the robustness of the baseline solution and the flexibility of the recourse policy. Then, a cooperative co-evolution algorithm named SoPoC is proposed to evolve these two components simultaneously. The balance between the evolution of these two components is controlled by the ratio between the number of fitness evaluation for evolving them per generation. Experimental results show that SoPoC significantly outperformed the existing state-of-the-art algorithms (i.e. EDASLS (Wang et al., 2016) and GPHH (Liu et al., 2017)) to UCARP. Moreover, SoPoC showed obvious advantages over EDASLS on complex instances with more routes. Since the difference between SoPoC and EDASLS is the recourse policy, the advantage of SoPoC over EDASLS demonstrates that SoPoC is capable of evolving much more effective recourse policies than the greedy (intuitive) recourse policy that have been commonly used in EDASLS and many existing studies.

We have also conducted deep analysis on the route structures obtained by SoPoC, and discovered that the better routes found by SoPoC actually contain more route failures that do not incur extra refill cost (the next task is on the way back to the depot). This discovery is interesting and somehow contradict with the intuition that route failure is detrimental and should always be avoided. Our finding further demonstrates the ability of SoPoC to identify new useful knowledge.

Due to the complicatedness of UCARP, we focus on the simplified single-vehicle scenario in this paper, which is a typical scenario that have been considered in existing studies (Rei et al., 2010; Balaprakash et al., 2015). The newly proposed algorithm shows promise in the simplified version, which is an encouraging result, and suggests the potential of solving UCARP in a predictive-reactive way. For extending our approach to more general multi-vehicle problems, we can replace the baseline task sequence with multiple task sequences (each for one vehicle), and consider the collaborations between different vehicles. Moreover, we will study on UCARP with other constraints (e.g. time) that can meet more complex requirements in the real world.

Acknowledgment

The work is supported by the National Natural Science Foundation of China (Nos. 61732019 and 61673194), the Marsden Fund of New Zealand (Nos. VUW1614 and VUW1209) and the Chongqing Graduate Student Research Innovation Project (No. CY-B16064).

References

Amponsah, S. and Salhi, S. (2004). The investigation of a class of capacitated arc routing problems: The collection of garbage in developing countries. *Waste Management*, 24(7):711–721.

Aráoz, J., Fernández, E., and Zoltan, C. (2006). Privatized rural postman problems. *Computers & Operations Research*, 33(12):3432–3449.

Balaprakash, P., Birattari, M., Stützle, T., and Dorigo, M. (2015). Estimation-based metaheuristics for the single vehicle routing problem with stochastic demands and customers. *Computational Optimization and Applications*, 61(2):463–487.

Chen, L., H, M. H., Langevin, A., and Gendreau, M. (2014). Optimizing road network daily maintenance operations with stochastic service and travel times. *Transportation Research Part E: Logistics and Transportation Review*, 64:88–102.

Chen, Y., Hao, J., and Glover, F. (2016). A hybrid metaheuristic approach for the capacitated arc routing problem. *European Journal of Operational Research*, 253(1):25–39.

Christiansen, C., Lysgaard, J., and Wøhlk, S. (2009). A branch-and-price algorithm for the capacitated arc routing problem with stochastic demands. *Operations Research Letters*, 37(6):392–398.

Christiansen, C. H. and Lysgaard, J. (2007). A branch-and-price algorithm for the capacitated vehicle routing problem with stochastic demands. *Operations Research Letters*, 35(6):773–781.

Chryssolouris, G. and Subramaniam, V. (2001). Dynamic scheduling of manufacturing job shops using genetic algorithms. *Journal of Intelligent Manufacturing*, 12(3):281–293.

Dror, M. (2000). *Arc Routing: Theory, Solutions and Applications*. Kluwer Academic Publishers, Norwell, MA, USA.

Fattahi, P. and Fallahi, A. (2010). Dynamic scheduling in flexible job shop systems by considering simultaneously efficiency and stability. *CIRP Journal of Manufacturing Science and Technology*, 2(2):114–123.

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

Fleury, G., Lacomme, P., Prins, C., and Ramdane-Chérif, W. (2005). Improving robustness of solutions to arc routing problems. *Journal of the Operational Research Society*, 56(5):526–538.

Gendreau, M., Jabali, O., and Rei, W. (2016). 50th anniversary invited article future research directions in stochastic vehicle routing. *Transportation Science*, 50(4):1163–1173.

Golden, B. and Wong, R. (1981). Capacitated arc routing problems. *Networks*, 11(3):305–315.

Handa, H., Chapman, L., and Yao, X. (2006). Robust route optimization for gritting/salting trucks: a cercia experience. *IEEE Computational Intelligence Magazine*, 1(1):6–9.

Hanshar, F. T. and Ombuki-Berman, B. M. (2007). Dynamic vehicle routing using genetic algorithms. *Applied Intelligence*, 27(1):89–99.

Jensen, M. T. (2003). Generating robust and flexible job shop schedules using genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 7(3):275–288.

Kall, P. and Wallace, S. W. (1994). *Stochastic Programming*. John Wiley & Sons, Chichester.

Khaligh, F. H. and MirHassani, S. (2016). A mathematical model for vehicle routing problem under endogenous uncertainty. *International Journal of Production Research*, 54(2):579–590.

Lacomme, P., Prins, C., and Ramdane-Cherif, W. (2004). Competitive memetic algorithms for arc routing problems. *Annals of Operations Research*, 131(1):159–185.

Leon, V. J., Wu, S. D., and Storer, R. H. (1994). Robustness measures and robust scheduling for job shops. *IIE Transactions*, 26(5):32–43.

Liu, Y., Mei, Y., Zhang, M., and Zhang, Z. (2017). Automated heuristic design using genetic programming hyper-heuristic for uncertain capacitated arc routing problem. In *Proceedings of GECCO*, pages 290–297. ACM.

Mei, Y., Li, X., and Yao, X. (2014). Cooperative coevolution with route distance grouping for large-scale capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 18(3):435–449.

Mei, Y., Tang, K., and Yao, X. (2010). Capacitated arc routing problem in uncertain environments. In *IEEE Congress on Evolutionary Computation*, pages 1–8.

Mei, Y., Tang, K., and Yao, X. (2011). Decomposition-based memetic algorithm for multiobjective capacitated arc routing problem. *IEEE Transactions on Evolutionary Computation*, 15(2):151–165.

Montemanni, R., Gambardella, L. M., Rizzoli, A. E., and Donati, A. V. (2002). A new algorithm for a dynamic vehicle routing problem based on ant colony system. Technical report, Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale (IDSIA).

Nguyen, S., Mei, Y., Ma, H., Chen, A., and Zhang, M. (2016). Evolutionary scheduling and combinatorial optimisation: Applications, challenges, and future directions. In *IEEE Congress on Evolutionary Computation*, pages 3053–3060.

Nguyen, S., Mei, Y., and Zhang, M. (2017). Genetic programming for production scheduling: a survey with a unified framework. *Complex & Intelligent Systems*, 3(1):41–66.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2013). A computational study of representations in genetic programming to evolve dispatching rules for the job shop scheduling problem. *IEEE Transactions on Evolutionary Computation*, 17(5):621–639.

Ouelhadj, D. and Petrovic, S. (2009). A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*, 12(4):417–431.

Pillac, V., Gendreau, M., Guéret, C., and Medaglia, A. L. (2013). A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11.

Potter, M. A. and De Jong, K. A. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29.

Rei, W., Gendreau, M., and Soriano, P. (2010). A hybrid monte carlo local branching algorithm for the single vehicle routing problem with stochastic demands. *Transportation Science*, 44(1):136–146.

Ritzinger, U., Puchinger, J., and Hartl, R. F. (2016). A survey on dynamic and stochastic vehicle routing problems. *International Journal of Production Research*, 54(1):215–231.

Salavati-Khoshghalb, M., Gendreau, M., Jabali, O., and Rei, W. (2017a). A hybrid recourse policy for the vehicle routing problem with stochastic demands. Technical report, CIRRELT-2017-42.

Salavati-Khoshghalb, M., Gendreau, M., Jabali, O., and Rei, W. (2017b). A rule-based recourse for the vehicle routing problem with stochastic demands. Technical report, CIRRELT-2017-36.

Secomandi, N. (2001). A rollout policy for the vehicle routing problem with stochastic demands. *Operations Research*, 49(5):796–802.

Shen, X.-N. and Yao, X. (2015). Mathematical modeling and multi-objective evolutionary algorithms applied to dynamic flexible job shop scheduling problems. *Information Sciences*, 298:198–224.

Tagmouti, M., Gendreau, M., and Potvin, J.-Y. (2011). A dynamic capacitated arc routing problem with time-dependent service costs. *Transportation Research Part C: Emerging Technologies*, 19(1):20–28.

Tang, K., Mei, Y., and Yao, X. (2009). Memetic algorithm with extended neighborhood search for capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 13(5):1151–1166.

Wang, J., Tang, K., Lozano, J. A., and Yao, X. (2016). Estimation of the distribution algorithm with a stochastic local search for uncertain capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 20(1):96–109.

Wang, J., Tang, K., and Yao, X. (2013). A memetic algorithm for uncertain capacitated arc routing problems. In *2013 IEEE Workshop on Memetic Computing*, pages 72–79.

Weise, T., Devert, A., and Tang, K. (2012). A developmental solution to (dynamic) capacitated arc routing problems using genetic programming. In *Proceedings of GECCO*, pages 831–838. ACM.

Xing, L., Rohlfshagen, P., Chen, Y., and Yao, X. (2010). An evolutionary approach to the multidepot capacitated arc routing problem. *IEEE Transactions on Evolutionary Computation*, 14(3):356–374.