EVOLVING ENSEMBLES OF ROUTING POLICIES USING GENETIC PROGRAMMING FOR UNCERTAIN CAPACITATED ARC ROUTING PROBLEM

A PREPRINT

Shaolin Wang
Victoria University of Wellington
Wellington, New Zealand
wangshao3@ecs.vuw.ac.nz

Yi Mei
Victoria University of Wellington
Wellington, New Zealand
yi.mei@ecs.vuw.ac.nz

John Park
Victoria University of Wellington
Wellington, New Zealand
john.park@ecs.vuw.ac.nz

Mengjie Zhang
Victoria University of Wellington
Wellington, New Zealand
mengjie.zhang@ecs.vuw.ac.nz

June 25, 2019

ABSTRACT

The Uncertain Capacitated Arc Routing Problem (UCARP) has a wide range of real-world applications. There have been a number of studies for solving UCARP. Genetic Programming Hyper-heuristic (GPHH) approaches have shown success by evolving routing policies rather than solutions. However, existing GPHH approaches evolve large and complex policies that are too difficult to interpret. On the other hand, small and easily interpretable policies generally are not as effective as the larger (longer) policies evolved by GP. In this paper, we adopt three ensemble methods, BaggingGP, BoostingGP and Cooperative Co-evolution GP (CCGP) for the first time in UCARP to evolve a group of interpretable routing policies. Experiment studies show that CCGP significantly outperformed BaggingGP and BoostingGP. CCGP also has comparable test performance to the GPHH that evolves single routing policy (called SimpleGP) while having significantly smaller tree depth and rule complexity, and CCGP significantly outperform the SimpleGP with comparative tree sizes. Although the analysis of the best-evolved ensemble shows that ensembles can contain redundant and uninterpretable members, the most ensemble members evolved by CCGP show clear rule behaviors that explain their effectiveness on UCARP.

***K*eywords** GPHH · UCARP

1 Introduction

The Capacitated Arc Routing Problem (CARP), which was first proposed by Golden and Wong in 1981 [1], is an important optimisation problem of serving a set of streets in a road network using a fleet of vehicles with minimum cost. CARP has a wide range of real-world applications such as waste collection and winter gritting.

Theoretically, CARP is an arc routing counterpart to the well-known Vehicle Routing Problem (VRP), and has been proven to be NP-hard. It has received much research interest, and there have been extensive studies for solving it [2]. However, so far most studies have focused on static environment, where all the problem parameters (e.g. task demand, travel cost) are fixed and known in advance. This assumption is usually not true in reality, where the problem parameters are often stochastic.

For example, the amount of waste to be collected on a street varies from one day to another and can be quite different from expected. To better reflect the reality, the Uncertain Capacitated Arc Routing Problem (UCARP) was proposed.

UCARP contains four basic stochastic parameters, which are the presence of tasks, demands of tasks, the presence of paths and traversal costs of paths.

Traditional optimisation approaches generally show poor performance in UCARP. The stochastic nature of UCARP can result in situations where the actual demand of the tasks can be greater than expected, leading to infeasible routes which violate the capacity constraint. Traditional optimisation generally requires a very long computation time to adjust the preplanned solution to account for the failure. Instead, heuristic approaches such as *routing policies* [3] are better able to handle UCARP than traditional optimisation techniques. A routing policy [3] helps the vehicle to determine which task to do once it becomes idle without any preplanned solutions. It can respond to the uncertain environment immediately. However, the effectiveness of a routing policy largely depends on the scenario, the objective(s), and even the graph topology. It is very time-consuming to design effective routing policy for a given problem scenario manually. To overcome this drawback, Genetic Programming Hyper-heuristic approaches have been applied to UCARP to to evolve routing policies automatically [4].

However, a major limitation of existing GPHH approaches is that they use large tree depth (e.g. depth of 8 [5]), which generally results in complex routing policies. These large and complicated rules are often too difficult for a human to interpret effectively. In a real-world scenario, the users need to be able to understand the evolved routing policies to feel confident to use them. In addition, if we can interpret the evolved routing policies, we can determine what contribution each feature makes in the decision-making process, and understand the inner mechanism of each routing policy so that we can reuse the knowledge to other cases. Evolving effective, small and interpretable heuristics has not yet been investigated for UCARP.

However, there are challenges to evolving small interpretable heuristic for UCARP. Without sufficient complexity, evolved rules cannot accurately capture the properties of a problem [6], and existing GPHH approaches in other combinatorial optimisation problem show that best-evolved rules generally tend to be quite complex [7]. This has been addressed in the literature using various parsimony techniques (e.g. [6]) that balance the performance of a GP tree with its growth. Other methods have also addressed the issue of improving the performance of GP rules using *ensemble learning* [8]. A key concept in ensemble learning is that a group of "weak" rules can be combined to form a much stronger rule [8]. By incorporating ensemble learning, it may be possible to evolve small interpretable routing policies that have a competitive performance to complex routing policies.

The overall research goal of this paper is to evolve effective and interpretable ensemble of routing policies for UCARP. To do this, we carry out the first investigation into the effectiveness of ensemble GPHH approaches in UCARP. Three existing ensemble GPHH approaches that have been applied to other combinatorial optimisation problems are adapted to handle the UCARP [9], [10]. By identifying high performing ensembles with small sizes, it may be possible to identify the behaviors. The research goal is broken down into the following objectives:

1. Investigate the effectiveness of three ensemble GPHHs for UCARP: BaggingGP [9], BoostingGP [9], and Cooperative Co-evolution GP (CCGP) [10].
2. Evaluate the evolved ensembles against a standard GPHH that evolves single rules, testing multiple levels of depth (and complexity) of evolved rules.
3. Analyse the structures and the behaviors of the evolved ensemble members.

2 Background

This section briefly describes some background of the UCARP, previous approaches and the hyper-heuristic approaches that have been applied to UCARP.

2.1 Uncertain Capacitated Arc Routing Problem

A UCARP instance can be described as follows: consider a graph $G(V, E)$, where V is the set of vertices, E is the set of edges. Each edge $e \in E$ has a positive random deadheading cost $dc(e)$, indicating the cost of traversing the edge. A set of edge *tasks* $E_R \subseteq E$ are required to be served by the vehicles. Each task $e_R \in E_R$ has positive random demand $d(e_R)$ which represents the demand to serve, and a positive serving cost $sc(e_R)$. A set of vehicles with capacity Q are located at the depot $v_0 \in V$. The goal is to serve all the tasks in R with the least total cost. A vehicle must start at v_0 and finish at v_0, and the total served demand for each route cannot exceed the capacity of the vehicle.

A UCARP instance *sample* is a realisation of the corresponding UCARP instance, in which each random variable (i.e. task demand and deadheading cost) has a realised value. However, the actual demand of a task is unknown until the vehicle finishes serving it, and the actual deadheading cost of an edge is unknown until the vehicle finishes traversing over the edge.

Due to the uncertain environment, routes can have failures during execution. Specifically, the actual demand of a task can be greater than expected, and the capacity of the vehicle expires in the middle of serving the task. In this case, a *route failure* occurs, and the route has to be repaired. A typical recourse operator is that the vehicle goes back to the depot to refill, and return to the failed task to complete the remaining service.

2.2 Related Work

The static CARP has received extensive research interest. Golden and Tong [1] developed an integer linear programming model and solve it using branch-and-cut. However, this approach can only solve small instances. Tabu search [11]–[13] approach was proposed for static CARP for improvement. After that, Genetic Algorithm [14] and Memetic Algorithms [15]–[17] were proposed to solve the problem better. Lacomme et al. [18] proposed an ant colony schema, and Doerner et al. [19] developed an ant colony optimization. However, these studies are not directly applicable to UCARP, since they are not able to deal with the route failure effectively.

To solve UCARP, both proactive and reactive approaches have been proposed. Proactive approaches [20], [21] aim to find robust solutions that are expected to handle all the possible realisations of the random variables. On the other hand, reactive approaches mainly use GPHH to evolve routing policies [4], [5], [22] to generate solution on-the-fly. In this paper, we focus on the GPHH approaches.

Effective GPHH approaches have been applied to UCARPs in the literature. Liu et al. [4] proposed a GPHH approach to UCARP by designing a novel and effective meta-algorithm that incorporate stochastic information from the problem. The proposed meta-algorithm improves the learning efficiency of GPHH by employing domain knowledge and filtering redundant candidate tasks, and lead to improved performance than an existing GPHH [23]. Mei et al. [5] extended the decision making process from single-vehicle to multiple-vehicle version, so that the solution can be generated with multiple vehicles on the road simultaneously. MacLachlan et al. [22] further improved the GPHH by proposing a novel task filtering method and an effective look-ahead terminal.

Ensemble learning is widely used in classification problems [24]–[26] and has been proven to improve performance. Ensemble learning has been combined with various algorithms, such as decision tree [27], active example selection algorithm [24], neural network [25], SVM and Naive Bayes algorithm [26], and GPHH [10]. The nature of Ensemble approach, using a group of weak learners to form up a stronger learner [28], has the potential to solve the drawback of GPHH we proposed in Section 1.

3 Ensemble GP Hyper-Heuristics for UCARP

In this section, we describe the three Ensemble GP (EGP) methods for UCARP, namely BaggingGP, BoostingGP and Cooperative Co-evolution GP (CCGP). BaggingGP and BoostingGP employ the idea of bagging and boosting in traditional ensemble learning [8], and evolves the routing policies in the ensemble in a sequential manner. CCGP, on the other hand, co-evolves the routing policies in the ensemble simultaneously. These ensemble approaches were proposed to create ensembles for job shop scheduling problems [9], [10], and showed good performance. In this paper, the idea of BaggingGP and BoostingGP is similar with Durasevic and Jakobovic's [9] idea, which were applied to a problem outside of CARP. The idea of CCGP is similar with the EGP-JSS [10].

3.1 GP Representation and Evaluation

During the GP process, a GP tree represents a routing policy, which is essentially an arithmetic priority function. The routing policy is applied in a solution generation process, which is modelled as a decision making process. During the process, the vehicles serve a task at a time. Once a vehicle completes serving the current task and becomes idle, the routing policy is called to calculate the priority of the unserved candidate tasks. Then, the candidate task with the best priority is selected to be served next. The process completes when all the tasks have been served, and the routes of the vehicles are returned as the solution generated by the routing policy.

To evolve the routing policies, terminal set commonly used by GPHH approaches to UCARP [5] is shown in Table 1. The function set consists of the operators $+, -, \times$, protected division $/$ which returns 1 if divided by 0, $\max$, and $\min$. The $\max$ and $\min$ operators take two child nodes, and return the maximum and minimum value between them, respectively. As an example, a routing policy "CFH $-$ CTD" tends to select the candidate task that is closest to the current location and farthest to the depot to serve next whenever a vehicle becomes idle.To evolve the routing policies, terminal set commonly used by GPHH approaches to UCARP [5] is shown in Table 1. The function set consists of the operators $+, -, \times$, protected division $/$ which returns 1 if divided by 0, $\max$, and $\min$. The $\max$ and $\min$ operators take two child nodes, and return the maximum and minimum value between them, respectively. As an example, a

routing policy "CFH − CTD" tends to select the candidate task that is closest to the current location and farthest to the depot to serve next whenever a vehicle becomes idle.

Table 1: A commonly used terminal set.

Terminal	Description
SC	serving cost of the candidate task
CFH	cost from the current location to the candidate task
CTD	cost from the candidate task to the depot
CR	cost from the current location to the depot
DEM	expected demand of the candidate task
DEM1	demand of closest unserved task to the candidate task
RQ	remaining capacity of the vehicle
FULL	fullness (served demand over capacity) of the vehicle
FRT	fraction of unserved tasks
FUT	fraction of unassigned tasks
CFR1	cost from the closest other route to the candidate task
RQ1	remaining capacity of the closest other route to the candidate task
CTT1	the cost from the candidate to its closest remaining task

To evaluate the fitness of a routing policy or an ensemble x, a set of training UCARP instance samples T_{train} is used. Specifically, x is applied to each of the training instance samples and generate a corresponding solution. For each training instance sample $Sample_t \in T_{train}$, the total cost of the solution obtained by x is $tc(x, Sample_t)$. Then, the fitness of x is defined as the average total cost of the solution obtained by x over the set of training instance samples, i.e.

$$fit(x) = \frac{1}{|T_{train}|} \sum_{Sample_t \in T_{train}} tc(x, Sample_t), \tag{1}$$

where $|T_{train}|$ is the number of training instance samples.

While generating solutions, the routing policy or ensemble is applied to select the next task for vehicles to serve next. A single policy simply selects the one with the best priority. An ensemble, on the other hand, needs a combination scheme. Here, we use the simple aggregation combination. That is, the final priority of a task is calculated as the sum of the priority given by the routing policies in the ensemble.

3.2 BaggingGP

The basic idea of BaggingGP is to evolve a set of routing policies with different biases using different training subsets [9]. Then, the evolved routing polices are combined together to form an ensemble. Durasevic and Jakobovic's [9] bagging approach evolves the ensemble from multiple separate GP runs, where each run outputs a single GP rule. In this paper, the ensemble is formed in a single GP run, which consists of a number of *cycles*. Each routing policy is evolved in a cycle. The pseudocode of the Bagging GP approach is shown in Algorithm 1. Given the number of generations G and the cycle length $cycleLength$, the algorithm will evolve $G/cycleLength$ routing policies to be included in the ensemble. Each routing policy is evolved using a different training subset randomly sampled from the training set T_{train}.

When applying the trained ensemble $\mathcal{RP}$ to an unseen test instance sample, the ensemble makes decisions by the aggregating priorities from each routing policies and select the task with the best total priority.

3.3 BoostingGP

In BoostingGP, the GP run is split into a number of cycles, each evolves a routing policy. Each training instance is given a weight, which is initially set equally to one divide the number of training instances. After each cycle, the weights of the instances that are poorly solved are increased, and the weights of instances that the evolved policies show good performance are decreased. This means that the poorly solved instances are more important in the subsequent cycle. To reflect the importance of different training instances, BoostingGP modifies the fitness function of a routing policy or ensemble x as the weighted sum of the total cost of the solutions obtained by it over the training set, i.e.

$$fit(x) = \sum_{t=1}^{N_{train}} w(Sample_t) \cdot tc(x, Sample_t). \tag{2}$$

Algorithm 1: The BaggingGP approach

Input: Training set T_{train}, number of generations G, cycle length $cycleLength$, evolutionary method $evolve$
Output: An ensemble of routing polices $\mathcal{RP}$

initialize the population pop;
initialize an empty ensemble $\mathcal{RP} = \emptyset$;
randomly sample a training subset $T' \subseteq T_{train}$;
$g = 0$;
while $g < G$ **do**
  if $g \mod cycleLength = 0$ **then**
    add the best individual in pop to $\mathcal{RP}$;
    reintialize the population pop;
    randomly re-sample a training subset $T' \subseteq T_{train}$;
  end
  for *each routing policy* $rp \in pop$ **do**
    for *each training sample* $S \in T'$ **do**
      while *tasks in queue is not empty* **do**
        if *vehicle is idle* **then**
          select which task to serve using rp;
        end
      end
      get the total cost of S using rp;
    end
    calculate $fit(rp)$ using Eq. (1);
  end
  $evolve(pop)$;
  $g = g + 1$;
end
return the ensemble $\mathcal{RP}$;

where $w(Sample_t)$ is the weight of the training instance sample t, and N_{train} is the number of training samples.

The pseudocode of Boosting GP approach is shown in Algorithm 2. Given the number of generations G and cycle length $cycleLength$, BoostingGP also generates $G/cycleLength$ routing policies to form the ensemble $\mathcal{RP}$.

After obtaining each routing policy, the weights of the training samples are updated by Algorithm 3 and the best routing policy rp^* that is just added into the ensemble, where the AdaBoost algorithm is employed for the weight update.

3.4 Cooperative Co-evolution GP

The basic idea of the Cooperative Coevolution (CC) approach [29] is to divide the original problem into several sub-problems. Each sub-problem is solved by a sub-population of the GP population. Finally, the best individuals from each sub-population are combined to form a complete solution to solve original problem. The CCGP has been successfully used for evolving an ensemble of dispatching rules for dynamic job shop scheduling [10], and different types of rules for dynamic flexible job shop scheduling [30] (i.e. routing and sequencing rules). In this paper, the CCGP approach partitions the population into several smaller sub-populations. Each sub-population has the same number of individuals. Representative of each sub-population will be formed as an ensemble. To evaluate the fitness of an individual in a sub-population, CCGP first replaces the corresponding representatives in the ensemble with the individual being evaluated. The fitness of the individual is the fitness of the resultant ensemble.

Initially, the representative of each sub-population is randomly selected. Then, in each generation, each representative is replaced by the best individual in the corresponding sub-population if the fitness of the ensemble is improved.

The pseudocode of the CCGP approach is described in Algorithm 4.

4 Experimental Studies

To evaluate the performance of the proposed BaggingGP, BoostingGP and CCGP, we conduct experiments on a number of UCARP instances and compare with the SimpleGP [5], which evolves a single complex routing policy.

Algorithm 2: The BoostingGP approach

Input: Training set T_{train}, number of generations G, cycle length $cycleLength$, evolutionary method $evolve$
Output: An ensemble of routing polices $\mathcal{RP}$

initialize the population *pop*;
for *each training sample* $Sample_t \in T_{train}$ **do**
  $w(Sample_t) = 1/|T_{train}|$;
end
initialize an empty ensemble $\mathcal{RP} = \emptyset$;
$g = 0$;
while $g < G$ **do**
  if $g \mod cycleLength = 0$ **then**
    get the best individual $rp^* \in pop$;
    add rp^* to $\mathcal{RP}$;
    update the weights of the training samples based on rp^* by Algorithm 3;
    reintialize the population *pop*;
  end
  for *each routing policy* $rp \in pop$ **do**
    for *each training sample* $S \in T_{train}$ **do**
      while *tasks in queue is not empty* **do**
        if *vehicle is ready* **then**
          select which task to serve using rp;
        end
      end
      get the total cost of S using rp;
    end
    calculate $fit(rp)$ using Eq. (2);
  end
  $evolve(pop)$;
  $g = g + 1$;
end
return the ensemble $\mathcal{RP}$;

Algorithm 3: Boosting GP weights update approach

Input: Training set T_{train}, weights for the training samples $w(Sample_t)$, routing policy rp^*
Output: new weights for the training samples $w(Sample_t)$

for *each training sample* $Sample_t \in T_{train}$ **do**
  calculate the total cost of solution obtained by rp^* on $Sample_t$, i.e. $tc(rp^*, Sample_t)$;
end
for *each training sample* $Sample_t \in T_{train}$ **do**
  compute the loss of rp^* on $Sample_t$: $Loss_t = \frac{tc(rp^*, Sample_t)}{\max\limits_{Sample_i \in T_{train}} \{tc(rp^*, Sample_i)\}}$;
end
compute the average Loss: $Loss_{avg} = \sum\limits_{Sample_t \in T_{train}} Loss_t * w(Sample_t)$;
compute the confidence for rp^*: $\beta = \frac{Loss_{avg}}{1 - Loss_{avg}}$;
calculate the constant for normalization: $constant = \sum\limits_{Sample_t \in T_{train}} w(Sample_t) \cdot \beta^{1 - Loss_t}$;
for *each training sample* $Sample_t \in T_{train}$ **do**
  update the weight of $Sample_t$: $w(Sample_t) = \frac{w(Sample_t) \cdot \beta^{1 - Loss_t}}{constant}$;
end
return $w(Sample_t)$;

Algorithm 4: The CCGP approach.

Input: number of generation G, evolutionary method $evolve$, ensemble size n
Output: An ensemble of routing polices $\mathcal{RP}$

initialize n sub-populations $pop_1, \ldots, pop_n$;
for $i = 1 \rightarrow n$ **do**
  randomly select a representative $rp_i^* \in pop_i$;
end
$\mathcal{RP} = \{rp_1^*, rp_2^*, \ldots, rp_n^*\}$;
$g = 0$;
while $g < G$ **do**
  for $i = 1 \rightarrow n$ **do**
    for *each routing policy* $rp_i \in pop_i$ **do**
      Form an ensemble $\mathcal{RP}' = \{rp_1^*, \ldots, rp_{i-1}^*, rp_i, rp_{i+1}^*, \ldots, rp_n^*\}$;
      for *each training sample* $Sample_t \in T_{train}$ **do**
        while *tasks in queue is not empty* **do**
          if *vehicle is ready* **then**
            select which task to serve using $\mathcal{RP}'$;
          end
        end
        get the total cost of S using $\mathcal{RP}'$;
      end
      calculate $fit(rp_i) = fit(\mathcal{RP}')$ using Eq. (2);
    end
    update rp_i^* in $\mathcal{RP}$ if $fit(rp_i)$ is better than $fit(rp_i^*)$;
  end
  $evolve(pop)$;
  $g = g + 1$;
end
return the ensemble $\mathcal{RP}$;

4.1 Experiment Setup

We selected 8 representative UCARP instances from the widely used Ugdb and Uval datasets [4], [5], [20]–[22], where the problem size ranges from small (22 tasks and 5 vehicles) to large (97 tasks and 10 vehicles). This way, we can examine the effectiveness of the proposed methods in different problem scenarios.

In the experiments, for each UCARP instance and each algorithm, a routing policy or an ensemble is first trained in the training phase. Then, the trained policy or ensemble is tested on a test set, which contains 500 unseen test instance samples.

Due to the different nature of the algorithms, we use different settings for the training set, so that all the algorithms have the same number of sample evaluations during the training phase. The setting of the training set is given in Table 2.

Table 2: The setting of the training set of the compared algorithms.

Algorithm	Training set setting
SimpleGP	5 training samples, re-sampled at each generation
BaggingGP	5 training samples, re-sampled at each cycle
BoostingGP	5 training samples, reweight at each cycle
CCGP	5 training samples, re-sampled at each generation

In the experiments, the terminal set of all the compared algorithms is shown in Table 1, and The population size of SimpleGP, BaggingGP and BoostingGP are set to 1000. For CCGP, there are 5 sub-populations, each with size of 200. The number of generations is set to 100. For BaggingGP and BoostingGP, the cycle length is set to 20, so that all the EGP methods have ensemble size of 5. All the methods use ramp-half-and-half initialisation and size-7 tournament selection. The crossover, mutation and reproduction rates are 80%, 15% and 5%, respectively. The maximal depth of

SimpleGP is 8. However, to evolve simple and interpretable routing policies in the ensemble, we intentionally set the maximal depth for the EGP methods to 4.

The algorithms were implemented based on the Evolutionary Computation Java (ECJ) package [31] and run 30 times independently for each UCARP instance.

4.2 Results and Discussions

Table 3 shows the mean and standard deviation of test performance of the compared algorithms. We also conduct Wilcoxon rank sum test with significance level of 0.05 to compare each EGP method with SimpleGP. For each EGP method, if it is significantly lower (better), higher (worse) and comparable with SimpleGP, then it is marked with (+), (-), or (=), respectively.

Table 3: The mean and standard deviation of the test performance of the compared algorithms. For each EGP method, (+), (-) and (=) indicates it is significantly lower (better), higher (worse) and comparable with SimpleGP.

Instance	SimpleGP	BaggingGP	BoostingGP	CCGP
Ugdb1	354.8(15.0)	397.1(62.9)(-)	399.4(56.5)(-)	364.9(13.9)(-)
Ugdb2	377.1(26.6)	424.9(34.8)(-)	433.0(32.0)(-)	372.3(9.7)(=)
Ugdb8	499.8(35.5)	548.7(26.0)(-)	568.9(39.4)(-)	467.7(33.7)(+)
Ugdb23	252.1(3.3)	266.4(5.1)(-)	268.8(7.5)(-)	256.0(4.2)(-)
Uval9A	340.3(13.3)	374.0(18.3)(-)	375.9(16.5)(-)	341.3(13.6)(=)
Uval9D	478.3(18.7)	561.4(33.1)(-)	542.2(22.7)(-)	490.4(30.5)(-)
Uval10A	440.6(5.6)	475.2(24.8)(-)	471.9(13.2)(-)	445.2(9.7)(-)
Uval10D	630.2(62.7)	693.9(29.6)(-)	685.2(26.0)(-)	649.1(16.3)(-)

From Table 3, BaggingGP and BoostingGP performed significantly worse than SimpleGP for all the test instances. This indicates that the ensembles evolved by BaggingGP and BoostingGP were still much weaker than the single routing policy evolved by SimpleGP. CCGP, on the other hand, performed much better than BaggingGP and BoostingGP. It performed significantly better than SimpleGP on Ugdb8, and statistically comparable with SimpleGP on Ugdb2 and Uval9A. For the other instances, although CCGP performed statistically significantly worse, the results of CCGP were much closer to that of SimpleGP than BaggingGP and BoostingGP.

Table 4 shows the mean and standard deviation of the number of nodes in the routing policies or routing policies in the ensembles evolved by the compared algorithms. From the table, it is obvious that the EGP approaches obtained much smaller routing policies. This is mainly because the maximal tree depth of the EGP approaches was set to 4, while that of SimpleGP was set to 8. Therefore, the EGP approaches generally evolve smaller and and simpler routing policies than SimpleGP. Although they may be weaker, we aim to form a much stronger ensemble by combining these weak routing policies together. Furthermore, even with the same maximal depth, the sizes of the routing policies evolved by CCGP are significantly smaller than those evolved by BaggingGP and BoostingGP. This shows that evolving the ensemble as a whole tends to lead to smaller and simpler routing policies in the ensemble than bagging and boosting.

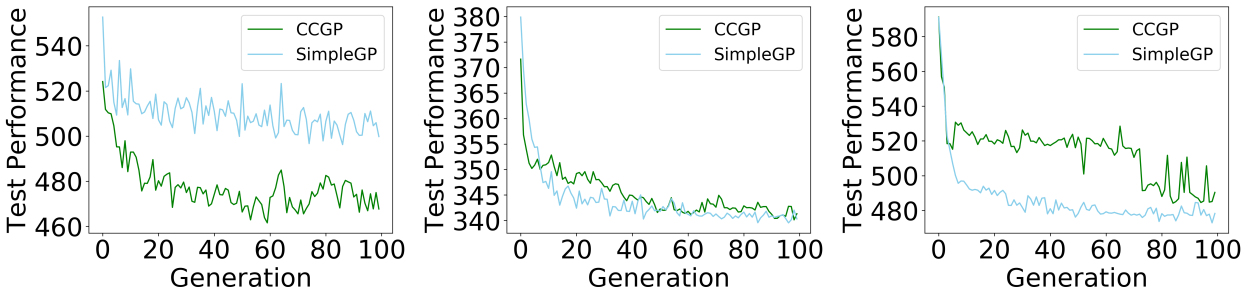
Table 4: The mean and standard deviation of the number of nodes in the routing policies or routing polices in the ensembles evolved by the compared algorithms.

Instance	SimpleGP	BaggingGP	BoostingGP	CCGP
Ugdb1	75.7(19.3)	12.2(2.3)	12.9(2.0)	5.2(3.6)
Ugdb2	68.3(16.3)	12.8(2.1)	12.2(2.9)	6.7(3.7)
Ugdb8	75.7(19.3)	12.2(2.3)	13.1(1.9)	6.9(3.7)
Ugdb23	68.3(16.9)	12.7(2.0)	12.7(2.1)	5.6(3.5)
Uval9A	65.3(21.6)	12.2(2.3)	12.9(2.0)	7.6(3.6)
Uval9D	58.1(17.7)	11.8(2.4)	12.1(2.3)	7.4(4.0)
Uval10A	58.1(17.7)	11.8(2.4)	12.1(2.3)	6.9(3.7)
Uval10D	65.7(13.3)	12.9(1.9)	12.6(2.1)	6.7(3.2)

Fig. 1 shows the **test** performance curves of the routing policies evolved by SimpleGP and the ensembles evolved by CCGP on three representative instances. Since BaggingGP and BoostingGP evolve the routing policies sequentially, they obtain the final ensemble only in the end of the training process, and do not have a curve for the test performance of the ensemble. Therefore, BaggingGP and BoostingGP are not included in the figure.

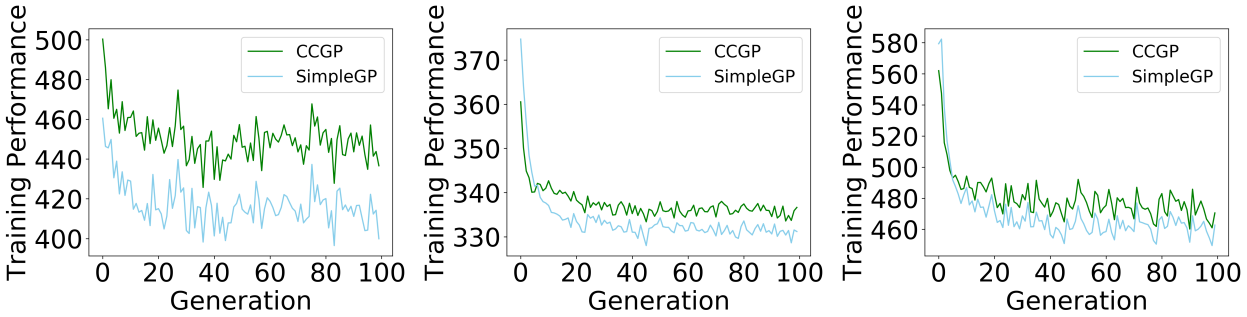
From Fig. 1, one can see that for Ugdb8, the ensemble evolved by CCGP always performed better than the routing policy evolved by SimpleGP. For val9D, on the other hand, the curve of CCGP is always above that of SimpleGP. For val9A, the two curves are almost the same, especially in the final stage. This shows the relative performance of the algorithms are consistent throughout the GP process.

Fig. 2 shows the **training** performance curve of SimpleGP and CCGP on the same instances. Again, BaggingGP and BoostingGP were excluded since they do not store intermediate ensembles. From the figure, it is obvious that the training performance of CCGP is worse than that of SimpleGP on all the instances. This shows that the learning ability of CCGP is still weaker than SimpleGP. Note that the routing policies in CCGP are naturally weaker than those in SimpleGP due to the smaller tree depth. To from a much stronger ensemble from the weak routing policies, an important requirement is the diversity of the routing policies, i.e. they compliment each other. The training performance of CCGP implies that the routing policies in the ensemble may not be complimentary to each other sufficiently, either due to the small ensemble size, or the way to co-evolve the routing policies in the ensemble. Nevertheless, Figs. 1a and 2a shows that CCGP leads to a better generalisation than than SimpleGP on the Ugdb8 instance.



(a) Ugdb8 (b) Uval9A (c) Uval9D

Figure 1: The **test** performance curves of SimpleGP and CCGP on the representative instances.



(a) Ugdb8 (b) Uval9A (c) Uval9D

Figure 2: The **training** performance curves of SimpleGP and CCGP on the representative instances.

Table 5 shows the average training time of the compared algorithms on the tested instances. From the table, we can see that the training time of all the EGP methods are shorter than that of SimpleGP. A main reason is that the routing policies evolved by the EGP methods (max depth of 4) are much smaller than the routing policies in SimpleGP (max depth of 8), and it is much less time consuming to calculate the priority values for the routing policies in the EGP methods. CCGP is more time consuming than BaggingGP and BoostingGP, since it requires to calculate 5 priority values in each evaluation, while BaggingGP and BoostingGP only need to calculate once.

4.3 Further Analysis

To investigate the effectiveness of CCGP in forming routing policies into ensembles, we compare it with the SimpleGP with the maximal depth of 4 (the same as the routing policies in CCGP). The results are shown in Table 6. It can be seen that CCGP clearly outperformed the SimpleGP with maximal depth of 4. This demonstrates the effectiveness of

Table 5: The training time of the compared algorithms.

Instance	SimpleGP	BaggingGP	BoostingGP	CCGP
Ugdb1	239.3(31.3)	133.2(4.8)	128.3(7.0)	128.2(30.1)
Ugdb2	319.5(37.9)	199.5(8.5)	193.7(12.9)	240.8(39.0)
Ugdb8	1024.2(107.4)	769.6(48.4)	758.8(47.8)	1041.9(84.1)
Ugdb23	1389.9(138.6)	1033.6(72.3)	1028.0(74.1)	1418.8(67.9)
Uval9A	3777.2(474.7)	2554.5(178.3)	2537.1(174.1)	4023.3(276.9)
Uval9D	5088.9(506.9)	3598.9(245.2))	3638.9(265.7)	4951.5(327.3)
Uval10A	4499.9(443.1)	2983.0(203.7)	3003.8(210.4)	4398.1(365.7)
Uval10D	5798.6(604.7)	4192.1(279.1)	4198.9(257.9)	5342.1(349.9)

combining weak policies into stronger ensembles. This demonstrates that EGP is more effective than SimpleGP when both of them produce interpretable routing polices.

Table 6: The mean and standard deviation of the test performance of CCGP and SimpleGP with maximal depth of 4.

Instance	SimpleGP (depth ≤ 4)	CCGP
Ugdb1	367.8(17.6)	364.9(13.9)(=)
Ugdb2	386.1(10.4)	372.3(9.7)(+)
Ugdb8	485.0(38.7)	467.7(33.7)(=)
Ugdb23	255.5(3.1)	256.0(4.2)(=)
Uval9A	348.0(8.6)	341.3(13.6)(+)
Uval9D	501.1(27.3)	490.4(30.5)(+)
Uval10A	444.4(6.3)	445.2(9.7)(=)
Uval10D	641.7(24.6)	649.1(16.3)(=)

To gain further understanding of the behaviours of the routing policies in the ensemble, we selected an effective ensemble for Ugdb8, and show its routing policies as follows:

$$
\begin{aligned}
rp_1 &= (\text{DEM1} - \text{FULL}) \times \text{CTT1} \times \text{CTT1}, \\
rp_2 &= \text{SC} \times (\text{CR} \times \text{CFH} + \text{SC} \times \text{RQ} \times \text{FRT}, \\
rp_3 &= \min\{\text{CFR1}, \text{DEM}\}, \\
rp_4 &= \min\{\text{CFR1}, \text{DEM1}\}, \\
rp_5 &= \text{CTD} + \text{CFH} + \text{DEM} + \text{DEM}.
\end{aligned}
$$

The above ensemble is applied to a Ugdb8 sample, and Table 7 shows the behaviour of the routing policies in some decision situations, which start from 92 candidate tasks to 56 candidate tasks. In the table, each column indicates a decision situation. The first row is the number of candidate tasks ("pool size") of the decision situation, and each row below is the rank of the task selected by the ensemble calculated by each routing policy (the lower the better). For example, in the first decision situation, there are 92 candidate tasks, and the ranks of the selected task is ranked 0th by both rp_1 and rp_2, 7th by rp_3 and rp_4, and 9th by rp_5. From the table, one can see that rp_1 always makes the same decision as the ensemble, followed by rp_2. When looking into rp_1, we see that the best priority is usually 0. This is because many tasks are directly connected with an unserved task, especially at the beginning of the decision process, and thus have CTT1 $= 0$. This indicates rp_1 prefers the tasks which are next to at least one unserved task. rp_2 has a very important feature CFH with a large coefficient SC $\times$ CR. In other words, rp_2 tends to select the tasks with small distance from the current location (CFH) and small serving cost (SC). rp_3 and rp_4 disagree with the ensemble decisions most of the time, which is consistent with their uninterpretable structures. rp_5 also prefers nearest neighbours (CFH) with less demand (DEM), as well as the ones close to the depot (CTD). This is consistent with Table 7, which shows that the task selected by the ensemble has a better rank in rp_5 in the later decisions situations (pool size ≤ 70) where the routes are more full and tend to return to the depot.

Overall, we can see CCGP can evolve routing policies which play different roles and at different stages of the decision process. However, there are sufficient redundancy (e.g. rp_3 and rp_4 in the example) which limits the effectiveness of the ensemble. There is potential to further improve the performance by further enhancing the collaborations between the routing policies.

Table 7: Behaviour of the routing policies in a decision making process.

Pool size	92	90	88	86	84	82	80	78	76	74	72	70	68	66	64	62	60	58	56
Rank in rp1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rank in rp2	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Rank in rp3	7	24	30	61	0	77	41	31	0	31	30	0	0	41	39	52	0	13	14
Rank in rp4	7	16	34	22	0	56	39	27	0	17	24	0	0	40	53	53	0	18	14
Rank in rp5	9	17	42	42	1	77	53	54	66	67	66	1	2	5	7	6	7	0	9

5 conclusions

This paper aims to improve the interpretability of the routing policies evolved by GPHH for UCARP. To this end, we employed ensemble methods with a smaller tree depth to evolve a group of smaller and more interpretable routing policies. We examined three widely used ensemble methods, namely BaggingGP, BoostingGP and Cooperative Co-evolution GP (CCGP). The experimental results show that CCGP performed the best, and can evolve both effective and interpretable routing policies. For future work, we will identify the reason why Bagging GP and Boosting GP achieve bad performance. Apart from that, we will try to develop more effective ensemble approaches for UCARP.

References

[1] B. L. Golden and R. T. Wong, "Capacitated arc routing problems," *Networks*, vol. 11, no. 3, pp. 305–315, 1981.

[2] S. Wøhlk, "A decade of capacitated arc routing," in *The vehicle routing problem: Latest advances and new challenges*, Springer, 2008, pp. 29–48.

[3] U. Ritzinger, J. Puchinger, and R. F. Hartl, "A survey on dynamic and stochastic vehicle routing problems," *International Journal of Production Research*, vol. 54, no. 1, pp. 215–231, 2016.

[4] Y. Liu, Y. Mei, M. Zhang, and Z. Zhang, "Automated heuristic design using genetic programming hyper-heuristic for uncertain capacitated arc routing problem," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ACM, 2017, pp. 290–297.

[5] Y. Mei and M. Zhang, "Genetic programming hyper-heuristic for multi-vehicle uncertain capacitated arc routing problem," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '18, Kyoto, Japan: ACM, 2018, pp. 141–142, ISBN: 978-1-4503-5764-7. DOI: 10.1145/3205651.3205661. [Online]. Available: http://doi.acm.org/10.1145/3205651.3205661.

[6] B.-T. Zhang and H. Mühlenbein, "Balancing accuracy and parsimony in genetic programming," *Evolutionary Computation*, vol. 3, no. 1, pp. 17–38, 1995.

[7] T. Hildebrandt, J. Heger, and B. Scholz-Reiter, "Towards improved dispatching rules for complex shop floor scenarios: A genetic programming approach," in *Proceedings of the 12th annual conference on Genetic and Evolutionary Computation*, ACM, 2010, pp. 257–264.

[8] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and systems magazine*, vol. 6, no. 3, pp. 21–45, 2006.

[9] M. Durasević and D. Jakobović, "Comparison of ensemble learning methods for creating ensembles of dispatching rules for the unrelated machines environment," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 53–92, 2018.

[10] J. Park, S. Nguyen, M. Zhang, and M. Johnston, "Evolving ensembles of dispatching rules using genetic programming for job shop scheduling," in *European Conference on Genetic Programming*, Springer, 2015, pp. 92–104.

[11] R. W. Eglese and L. Y. Li, "A tabu search based heuristic for arc routing with a capacity constraint and time deadline," in *Meta-Heuristics*, Springer, 1996, pp. 633–649.

[12] A. Hertz, G. Laporte, and M. Mittaz, "A tabu search heuristic for the capacitated arc routing problem," *Operations research*, vol. 48, no. 1, pp. 129–135, 2000.

[13] J. Brandão and R. Eglese, "A deterministic tabu search algorithm for the capacitated arc routing problem," *Computers & Operations Research*, vol. 35, no. 4, pp. 1112–1126, 2008.

[14] P. Lacomme, C. Prins, and W. Ramdane-Chérif, "A genetic algorithm for the capacitated arc routing problem and its extensions," in *Workshops on Applications of Evolutionary Computation*, Springer, 2001, pp. 473–483.

[15] P. Lacomme, C. Prins, and W. Ramdane-Cherif, "Competitive memetic algorithms for arc routing problems," *Annals of Operations Research*, vol. 131, no. 1-4, pp. 159–185, 2004.

[16] K. Tang, Y. Mei, and X. Yao, "Memetic algorithm with extended neighborhood search for capacitated arc routing problems," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 5, pp. 1151–1166, 2009.

[17] Y. Mei, K. Tang, and X. Yao, "Decomposition-based memetic algorithm for multiobjective capacitated arc routing problem," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 2, pp. 151–165, 2011.

[18] P. Lacomme, C. Prins, and A. Tanguy, "First competitive ant colony scheme for the carp," in *International Workshop on Ant Colony Optimization and Swarm Intelligence*, Springer, 2004, pp. 426–427.

[19] K. F. Doerner, R. F. Hartl, V. Maniezzo, and M. Reimann, "Applying ant colony optimization to the capacitated arc routing problem," in *International Workshop on Ant Colony Optimization and Swarm Intelligence*, Springer, 2004, pp. 420–421.

[20] J. Wang, K. Tang, and X. Yao, "A memetic algorithm for uncertain capacitated arc routing problems," in *Memetic Computing (MC), 2013 IEEE Workshop on*, IEEE, 2013, pp. 72–79.

[21] J. Wang, K. Tang, J. A. Lozano, and X. Yao, "Estimation of the distribution algorithm with a stochastic local search for uncertain capacitated arc routing problems," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 96–109, 2016.

[22] J. MacLachlan, Y. Mei, J. Branke, and M. Zhang, "An improved genetic programming hyper-heuristic for the uncertain capacitated arc routing problem," in *Australasian Joint Conference on Artificial Intelligence*, Springer, 2018, pp. 432–444.

[23] T. Weise, A. Devert, and K. Tang, "A developmental solution to (dynamic) capacitated arc routing problems using genetic programming," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, ACM, 2012, pp. 831–838.

[24] S. Oh, M. S. Lee, and B.-T. Zhang, "Ensemble learning with active example selection for imbalanced biomedical data classification," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 8, no. 2, pp. 316–325, 2011.

[25] L. Yu, S. Wang, and K. K. Lai, "Credit risk assessment with a multistage neural network ensemble learning approach," *Expert systems with applications*, vol. 34, no. 2, pp. 1434–1444, 2008.

[26] L. Shi, X. Ma, L. Xi, Q. Duan, and J. Zhao, "Rough set and ensemble learning based semi-supervised algorithm for text classification," *Expert Systems with Applications*, vol. 38, no. 5, pp. 6300–6306, 2011.

[27] M. Pal, "Random forest classifier for remote sensing classification," *International Journal of Remote Sensing*, vol. 26, no. 1, pp. 217–222, 2005.

[28] Z.-H. Zhou, *Ensemble methods: Foundations and algorithms*. Chapman and Hall/CRC, 2012.

[29] M. A. Potter and K. A. D. Jong, "Cooperative coevolution: An architecture for evolving coadapted subcomponents," *Evolutionary computation*, vol. 8, no. 1, pp. 1–29, 2000.

[30] D. Yska, Y. Mei, and M. Zhang, "Genetic programming hyper-heuristic with cooperative coevolution for dynamic flexible job shop scheduling," in *European Conference on Genetic Programming*, Springer, 2018, pp. 306–321.

[31] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, and A Chircop, "Ecj: A java-based evolutionary computation research system," *Downloadable versions and documentation can be found at the following url: Http://cs. gmu. edu/eclab/projects/ecj*, 2006.