Evolving Large Reusable Multi-pass Heuristics for Resource Constrained Job Scheduling

Su Nguyen*, Dhananjay Thiruvady†

*Centre for Data Analytics and Cognition, La Trobe University, Australia
*School of Information Technology, Deakin University, Australia
Email: *p.nguyen4@latrobe.edu.au, †dhananjay.thiruvady@deakin.edu.au

***Abstract*—Resource constrained job scheduling is a challenging combinatorial optimisation with many real-world applications. A number of exact methods and meta-heuristics have been proposed in the literature to solve this problem, but often encounter scalability issues. This paper investigates an automated heuristic design approach to deal with this problem. The aim of this approach is to generate heuristics that can quickly construct good solutions, which can be applied directly or used to initialise other meta-heuristics. A new adaptive genetic programming algorithm is proposed to coevolve a large set of reusable heuristics to solve the resource constrained job scheduling problem. There are three different aspects to the novelty behind our proposed algorithm: (a) a new phenotypic representation of heuristics, (b) an efficient mapping technique to monitor the evolutionary process, and (c) an adaptive fitness function to guide the search towards a diverse and competitive population. The experimental results show that evolved heuristics show promise and are able to outperform some existing meta-heuristics for large-scale instances. Analyses also show that the algorithm can be further improved if appropriate parameters are selected.**

***Index Terms*—genetic programming; scheduling, learning**

I. Introduction

Several scheduling problems can be modelled as a variant of job scheduling. Of particular interest is resource constrained job scheduling with shared resources (RCJS), which can be mapped to many real-world applications. A typical example, derived from the mining supply chain, is the transport of ore from mines to ports. The movement of one unit of ore can be considered as a job. The transporting of ore takes place with either trucks or trains. Since there are a limited number of them, transporting ore at the same time is in effect limited capacity or a resource limit on concurrently executing jobs. Furthermore, there may be an ordering imposed on the arrival of ore at ports, which is the same as precedences between jobs. The objective is to ensure that the ore arrives at the ports in a timely manner, thus avoiding demurrage costs associated with making ships wait unnecessarily. This is effectively a total weighted tardiness (TWT) of all jobs to be minimised.

Exact methods for solving the RCJS problem have not proved effective. As a result there have been several meta-heuristic and hybrid methods proposed [1], [2], [3], [4], [5], [6], [7]. The studies by [1], [2], [3] show that combining Lagrangian relaxation with simulated annealing, particle swarm optimisation and ant colony optimisation (ACO), respectively, can provide very good solutions in reasonable run-times. However, scalability still proves to be a issue and for this reason parallel implementations of ACO and hybrid methods were attempted in [4], [6]. To the authors; knowledge, the latest study [7] applied a biased random key genetic algorithm and showed that very good results can be obtained, even in the non-parallel setting.

In the scheduling literature, simple scheduling heuristics such as priority rules [8] were usually developed to provide quick and acceptable solutions for large-scale instances. However, designing effective scheduling heuristics is a complex task and requires a lot of trial-and-error and evaluations. Automated heuristic design (AHD) has been recently proposed to automatically discover effective heuristics for a wide range of scheduling and combinatorial optimisation problems such as job shop scheduling [9], bin packing [10], dynamic pickup and delivery [11] and time-tabling [12]. AHD is considered a machine learning problem [13] in which optimisation search algorithms, e.g. genetic programming [9] and evolutionary strategy [14] are developed to explore the space of heuristics to determine the most effective among these based on a set of training instances. Different from conventional optimisation algorithms, the heuristics found are *reusuable*, i.e. able to generate solutions for new instances without rerunning the costly optimisation process. Previous studies have shown that AHD methods can evolve powerful heuristics for dynamic scheduling problems [15], [16] or effective reusable heuristics to deal with large static scheduling problems [10], [17].

Nguyen et al. [18] presented a preliminary study on genetic programming (GP) algorithm for RCJS. They proposed a multiple tree representation and a simple fitness approximation technique to help GP evolve multi-pass scheduling heuristics. Their proposed GP is trained with a small number RCJS instances and the experiment results showed that the proposed algorithm outperformed the simple GP algorithm and evolved heuristics can produce competitive solutions compared to specialised hybrid optimisation algorithms, especially for large problem instances. They also showed that better results can be achieved when a larger number of priority rules are included within the multi-pass heuristics. One of the key limitations of their method is that the proposed representation can only accommodate a restricted number of rules. When the number of rules increases, the search space of GP will significantly increase which makes it harder to find good programs.

The goal of this paper is to develop an adaptive GP (AGP) algorithm that can coevolve a large diverse set of priority rules, which can be combined at the end of the evolutionary process to become a powerful multi-pass heuristic. The aim is not to beat the state-of-the-art but rather to demonstrate that this algorithm can quickly construct high solutions, which can be applied directly or used to initialise other meta-heuristics. Different from Nguyen et al. [18], the number of candidate rules in evolved multi-pass heuristics does not need to be predefined as it will be the same as the population size. The key challenge here is to ensure that the population diversity is preserved during the evolutionary process while general and effective rules can still be discovered efficiently. To address this challenge, we focus on the following three objectives:

1) Developing a new phenotypic representation that captures the behavioural characteristics of evolved priority rules for RCJS,
2) Developing a mapping technique based on the proposed phenotype that can monitor the evolutionary process,
3) Developing a fitness function to evolve a diverse and competitive population.

As it is very computationally expensive to compare evolved rules using their genotypic representation (e.g. tree structure), a phenotypic representation of evolved rules is proposed to capture how a priority rule constructs a scheduling solution, which decides whether two rules are similar. Then a mapping technique based on growing neural gas (GNG) and principal component analysis (PCA) is applied to capture the topological relationships of evolved rules. The outcome of the mapping technique is a network that allows us to conveniently and efficiently analyse evolutionary patterns and diversity of the population in an incremental manner. The network will store the information of evolved rules such as their program sizes, phenotypes and fitnesses. From the knowledge accumulated by the network over time, the AGP is informed of which areas of the search space it has been explored and which of those are more promising. To reduce the computational costs for fitness evaluation and maintain diversity in the population, we only apply a small subset of training instances in each generation while supporting the selection process with an approximate fitness function based on the GNG network.

The rest of this paper is organized as follows. Section II-A describes the RCJS problem investigated in this paper and discusses related work. Section III presents the AGP algorithm and its key components. Section IV presents the experiment settings and the results of AGP compared to other GP and a state-of-the-art optimisation algorithm. Finally, conclusions and future studies are presented in Section V.

II. Problem and Related Work

In this section we provide details of the RCJS problem including its integer programming formulation. We also review different solution methods that have been proposed for solving it. Finally, we also provide an overview of automated heuristic design with GP.

A. The Resource Constrained Job Scheduling Problem

We provide a formal description of the RCJS problem. We are given a set of jobs $\mathcal{J} = \{1, \ldots, n\}$ and each job i has a release time r_i, processing time p_i, due time d_i, weight w_i, resource consumption g_i and a machine m_i. The jobs need to be need to be executed on a set of machines $\mathcal{M} = \{1, \ldots, l\}$ and a job must execute on the machine that it is assigned to. Additionally, a machine can execute only one job at one time. We consider a horizon $\mathcal{T} = \{0, \ldots, D\}$, which consists of discrete time points. On any machine, there may be precedences between jobs that have to execute on it. Formally, we denote precedences between two jobs $i, j \in \mathcal{J}$, $i \rightarrow j$ which states that job i must complete executing before job j can start executing. For all jobs executing at any single point in time, the cumulative resources consumed by the jobs must be at most $\mathcal{G}$.

There have been different integer programming formulations proposed for the RCJS problem. The most efficient one to date is that of [1], which we also use in this study. This model uses binary variables z_{jt}, which is 1 if job j completes by time t or earlier. The formulation is:

$$\min \quad \sum_{j\in\mathcal{J}} \sum_{t\in\mathcal{T}} c_{jt}\left(z_{jt} - z_{j,t-1}\right) \tag{1}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{J}^m} z_{j,t+p_j} - z_{jt} \leq 1 \qquad \forall\, t \in \mathcal{T},\ \forall\, m \in M \tag{2}$$

$$z_{jt} \geq z_{j,t-1} \qquad \forall\, j \in J,\ \forall\, t > 0 \tag{3}$$

$$z_{jT} = 1 \qquad \forall\, j \in J \tag{4}$$

$$z_{j,t-p_k} \geq z_{kt} \qquad \forall\, j \rightarrow k,\ \forall\, t \in \mathcal{T} \tag{5}$$

$$z_{j,t} = 0 \qquad \forall\, t < r_j + p_j,\ \forall\, j \in J \tag{6}$$

$$\sum_{j\in J} g_j\left(z_{j,t+p_j} - z_{jt}\right) \leq \mathcal{G} \qquad \forall\, t \in \mathcal{T} \tag{7}$$

$$z_{jt} \in \{0,1\} \qquad \forall\, t \in \mathcal{T}$$

where $c_{jt} = w_j \times \max\{0, t - d_j\}$ is a penalty imposed on job j when it completes later than its due time. The objective is represented by Equation (1), which is the sum of the tardiness of all jobs or the total weighted tardiness (TWT). Only one job may execute at any one time on a machine, which is specified by Equation (2). In Equation (3), once a jobs completes it stays completed and Equation (4) requires that all jobs complete. The precedences between jobs are imposed by Equation (5) and the constraint on the release times are defined by Equations (6). Finally, the resource constraints are satisfied via Equation (7).

B. Previous Work Related to RCJS

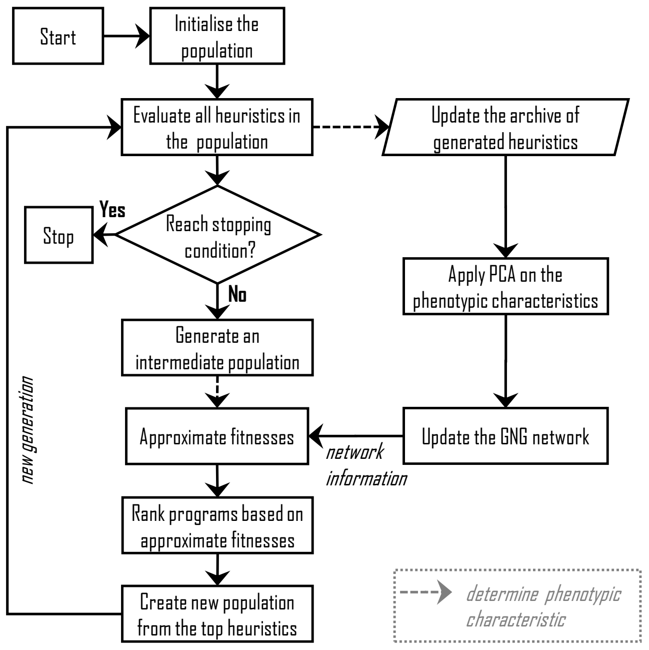
A closely related problem to RCJS is Project scheduling [19]. This problem is very popular and has thus been studies extensively. The studies by [20], [19], [21], [22], [23], [24], [25] examine various algorithms for the problem including local search, branch & bound, matheuristics and parallel implementations. The study by [21] explore simulated annealing, genetic algorithms and exact approaches for different variants of the project scheduling problem. [22] consider a variant which incorporates time windows and they propose solution methods based on heuristics and exact approaches. A similar project scheduling variant to RCJS is one that is considered by [20], where the objective is to minimise the cumulative deviation from the desired task completion times. Their main result is that an iterated local search proves very effective. The studies by [23], [24], [25] consider the problem variant where the aim is to maximise the net present value. They show that matheuristics and their parallel implementations can be very effective for this problem.

The studies [26], [5] also consider an extension to the RCJS, with hard deadlines. [26] explore a hybrid of constraint programming, beam search and ant colony optimisation, which prove to be effective at solving the problem. A limitation of this study was large run-time requirements and so [5] develop a parallel implementation to find good solutions more quickly.

Other studies on similar problems to RCJS have also considered the TWT objective [27], [28]. Both studies, [27] and [28] solve a similar problem to the RCJS and explore an agent based approach coupled with as little information sharing as possible. An assumption of this study is to assume decentralised data.

III. Adaptive Genetic Programming

Fig. 1 shows how the proposed AGP works. The algorithm starts with a random population $\mathbb{H} = \{\mathcal{H}_1, \mathcal{H}_2, \cdots, \mathcal{H}_N\}$ of scheduling heuristics, i.e. priority rules used to construct RCJS solutions. In each generation, all heuristics in the population will be evaluated by using a set $\mathcal{T}$ of RCJS instances. The phenotypic characteristics $phenotype(\mathcal{H})$, the corresponding fitnesses $fitness(\mathcal{H})$ and the sizes of evaluated heuristics are recorded in archive $\mathcal{A}$. A dimensional reduction technique is applied to transform the dataset X of phenotypic characteristics (with the dimension of D stored in $\mathcal{A}$) to X' with a lower dimensionality (by using the first K principal components where $K << D$). Following this, a modified version of growing neural gas (GNG) [29] is used to learn topological relations of generated heuristics from the transformed dataset X'. The network $\mathcal{N} = (V, E)$ obtained with the modified GNG (mGNG) contains a set of nodes V and a set of undirected edges E that represent the distributions of generated heuristics on the search space. Genetic operators are then applied to generate an intermediate (or trial) population. The fitnesses of these newly generated heuristics are estimated by using the inputs from the phenotypic characteristics and sizes of those heuristics, archive $\mathcal{A}$ and the network $\mathcal{N}$. The heuristics in the intermediate population are ranked based on the approximate fitnesses and only the top heuristics are used to build the population for the next generations. In the rest of this section, we will provide the details of each key component in this algorithms including the GP representation of scheduling heuristics, genetic operations, the mGNG algorithm, and



Start
Initialise the population
Evaluate all heuristics in the population
Update the archive of generated heuristics
Yes
Stop
Reach stopping condition?
No
Apply PCA on the phenotypic characteristics
Generate an intermediate population
new generation
Approximate fitnesses
Update the GNG network
network information
Rank programs based on approximate fitnesses
Create new population from the top heuristics
determine phenotypic characteristic

Fig. 1. Adaptive Genetic Programming.

approximate fitnesses. Different from most GP algorithms, the output of AGP is the whole population, i.e. $\mathbb{H}$, instead of the best priority function generated during the evolutionary process. All priority functions in the final population are candidate rules for the multi-pass heuristic. To handle a unseen test instance, the multi-pass heuristic will execute all $\mathcal{H}_k \in \mathbb{H}$ to construct N schedules one the best schedule in terms of TWT will be the final schedule.

A. Representation

Each individual in the AGP population is a priority function $\mathcal{H}$ which is used to rank jobs during the scheduling construction procedure. These priority functions will be represented as expression trees composed of arithmetic operators (i.e. function set) and relevant attributes (i.e. terminal set or attribute set) [30]. This representation has been widely adopted by most previous studies with GP for automated heuristic design and has shown very promising results. Table I shows the attributes and functions used to design priority functions. The first part of the table shows five static attributes which will not be changed during the schedule construction process. The second part shows two dynamic attributes to reflect the urgency and the resource tightness as more jobs are added into the schedule. To construct a schedule using $\mathcal{H}$, we apply the schedule construction procedure in Algorithm 1. The output of this algorithm is the starting times of all jobs $j \in \mathcal{J}$, which can be used to calculate the schedule performance, i.e. TWT. To better compare or aggregate the performance of evolved heuristic across different instances, we calculate the objective value $obj(\mathcal{H})$ divided by the TWT obtained with $\mathcal{H}$ by the TWT obtained by a reference heuristic, i.e. weighted shortest processing time in this paper.

TABLE I
TERMINAL SET AND FUNCTION SET

Notation	Description	Value
PR	processing time	p_j
W	weight	w_j
R	required resource	g_j
TPS	total load of successors	$\sum_{l\in\{i\in\mathcal{J}\mid j\to i\}} p_l$
TWS	total weight of successors	$\sum_{l\in\{i\in\mathcal{J}\mid j\to i\}} w_l$
SL	slack	$d_j - (t^{min} + p_j)$
RT	resource tightness	$\sum_{t=t^{min}}^{t^{min}+p_j} (R_t - g_{j^*})/p_j$
$Functions$	$+, -, *, \%, \max, \min$	

j is the job to be prioritised; $t^{\min}$ can be calculated as in step 7 of Algorithm 1
$\tilde{s}_j$ is the start time from previous constructed schedule; $\%$ is protected division

Algorithm 1 Schedule construction procedure

```
    Input: A RCJS instance $\mathcal{I}$, a priority function $\mathcal{H}$
    Output: A resource feasible schedule $\mathcal{S}(\mathcal{H})$ defined by job start times $s_j$ for $\forall\ j \in \mathcal{J}$
 1: $\Omega$ is the set of ready jobs $j \in \mathcal{J}$ ($\nexists i \to j$)
 2: $\Omega' = \emptyset$ is the set of scheduled jobs
 3: $R_t := \mathcal{G}\ \forall\ t \in \mathcal{T}$
 4: while $\Omega$ is not empty do
 5:     determine the priorities of jobs $j \in \Omega$ with $\mathcal{H}$
 6:     select job $j^*$ with the highest priority
 7:     $t^{\min} := \max\{r_{j^*}, \max\{s_i + p_i \mid i \in \Omega' \wedge m_i = m_{j^*}\}\}$
 8:     $s_{j^*} := \min\{t \geq t^{\min} \mid R_{t+k} \geq g_{j^*}\ \forall\ 0 \leq k < p_{j^*}\}$
 9:     $R_t := R_t - g_{j^*} \quad \forall\ t = s_{j^*}, \ldots, s_{j^*} + p_{j^*} - 1$
10:     add $j^*$ to $\Omega'$
11:     remove $j^*$ from $\Omega$ and add new ready jobs into $\Omega$
12: return $\mathcal{S}(\mathcal{I}, \mathcal{H}) = \{s_1, \ldots, s_n\}$
```

1) Phenotype: Since the goal of AGP is to evolve a set of diverse heuristics which can cope with a wide range of situations, it is important to capture how evolved heuristics can construct a complete schedule to measure the similarity/distance of two priority functions. In this case, the starting time vector $\mathcal{S}$ obtained from Algorithm 1 is a good way to capture the behavioural characteristics of evolved priority functions as it provides the information of how priorities are assigned from an empty schedule to a complete schedule. In AGP, a random instance $\mathcal{I}_p$ is generated and fixed during the evolutionary process and each newly generated function $\mathcal{H}$ will be applied to $\mathcal{I}_p$ to determine $phenotype(\mathcal{H}) = \mathcal{S}(\mathcal{I}_p, \mathcal{H})$. Although it is possible to use a large instance as $\mathcal{I}_p$ or a subset of RCJS instances, we prefer a single and small instance to minimise the computational cost as it is sufficient to represent different job permutations or schedules. For example, if the instance has no precedence constraints, a single machine, and unlimited resources, the number of jobs permutation is $n!$, which is more than three million for a small instance with $n = 10$ jobs.

B. Genetic operations

In the proposed algorithm, subtree crossover and subtree mutation [30] are employed to generate new scheduling heuristics $\mathcal{H}$. A heuristic will be randomly picked and a random genetic operation is applied. The subtree crossover creates new heuristics for the next generation by randomly recombining subtrees from two selected parent heuristics. The subtree mutation is performed by selecting a node of a chosen functions and replacing the subtree rooted by that node with a newly randomly-generated subtree. To improve the diversity of the population, all the same duplicated heuristics (based on phenotypic similarity) will be eliminated [31].

C. Mapping evolutionary process

To efficiently explore competitive scheduling heuristics, it is crucial for AGP to capture the topological relations and distributions of previous generated heuristics. Since the heuristics generated by genetic operations are complicated and usually contain redundant components, phenotypic characteristics are more useful than genotypic characteristics to determine the similarity between generated heuristics. Also, phenotypic characteristics are a good predictor of program fitnesses [31], [32] and the phenotypic diversity has good correlation with fitness improvements [33]. Therefore, we employ the phenotypic characteristics to map the evolutionary process of AGP. In this paper we apply the mapping technique proposed in [34] which includes two steps: (1) dimensionality reduction with PCA, and (2) topological learning with mGNG.

1) Dimensionality reduction: High-dimensional data X can slow down the mapping algorithm. In this paper, we use principal component analysis (PCA) for dimensionality reduction to transform X into a lower dimensional space. PCA is chosen for this task because it is an efficient and well-established approach and it copes very well with the multicollinearity issues in the phenotypes. Since there is a good chance that jobs to be prioritised will have similar attributes (as shown in Table I, their phenotypes (start times) can be quite similar.

2) Modified growing neural gas: Algorithm 2 shows how mGNG can adapt the network to the input data. All data points in $X = \{phenotype(\mathcal{H}) | \mathcal{H} \in \mathcal{A}\}$ are transformed into X' by PCA. In each epoch, an input x is then sampled from the transformed data X' and mGNG will determine the nearest unit (s_1) and the second nearest unit (s_2) based on Euclidean distance. An edge connecting these two units are created (if it does not yet exist) and its age is set to zero. After each epoch, a new unit will be inserted near the unit with the maximum error in order to reduce the total error of the network. If an unit has a very low utility as compared to the maximum error, it will be removed from the network. This mechanism is proposed in [29] to deal with non-stationary distributions. This growing and removing mechanism is created to reduce the computational costs of the mapping process and help AGP track the dynamics of the population. The maximum age of edges are set to the size of the dataset to preserve most relations of programs. We also adopt a scheme like *k-means* to adapt the learning rate over time [35], i.e. $e_b = 1/n_{win}$ and

Algorithm 2 Modified growing neural gas (mGNG)

```
    Input: dataset $X$, current network $\mathcal{N}$ (if exist)
    Output: (updated) mGNG network $\mathcal{N}$, PCA model
 1: $X' \leftarrow$ transform dataset $X$ with PCA
 2: $epoch \leftarrow 0$
 3: randomly initialise $\mathcal{N}$ with two nodes if $\mathcal{N}$ is empty
 4: repeat
 5:     randomly shuffle $X'$
 6:     for each data point $x \in X'$ do
 7:         $w_{s_1} \leftarrow$ the nearest (winning) node $s_1$ for input $x$
 8:         update the error of $s_1$: $E_{s_1} = E_{s_1} + ||x - w_{s_1}||^2$
 9:         updating unit $s_1$: $w_{s_1} = w_{s_1} + \epsilon_b(x - w_{s_1})$
10:         updating neighbour $n$ of $s_1$: $w_n = w_n + \epsilon_n(x - w_n)$
11:         create an edge between $s_1$ and $s_2$ (the second-nearest unit to $x$) and set its age to zero if the edge does not exist; otherwise set the edge's age to zero
12:         increment the age of edges connecting $s_1$ and each neighbor $n \in \mathcal{N}$
13:         removing edges with age larger than $a_{max} = |X|$ and nodes with no emanating edges
14:         updating the utility of $s_1$: $U_{s_1} = U_{s_1} + ||x - w_{s_2}||^2 - ||x - w_{s_1}||^2$
15:         let $q$ is the unit with maximum error and $l$ is the unit with minimum utility
16:         remove the unit $l$ if $E_q/U_l > \theta$
17:         for each unit $c \in \mathcal{N}$ do
18:             $E_c = (1 - \beta)E_c$
19:             $U_c = (1 - \beta)U_c$
20:     insert a new unit to $\mathcal{N}$ between $q$ and its neighbor if the number of nodes in $\mathcal{N} < max\_node$
21: until $epoch = maximum\_epoch$ or mGNG $converges$
```

$e_n = 100/n_{win}$ (see steps 9–10 in Algorithm 2) where n_{win} is the number of input signals for which the considered node has been a winner.

It is noted that mGNG is applied at the end of each generation. The obtained network $\mathcal{N}$ can show what heuristics have been generated and evaluated through generations. As compared to existing mapping-based method in the literature such as MAP-Elites [36], mGNG has a number of advantages. First, mGNG does not make any assumption about the phenotype search space, which is important for MAP-Elites to predefine the map size. Second, the output of mGNG is a network that can help AGP perform analyses more efficiently. Finally, mGNG can efficiently control and dynamically updating the map (i.e. network) during the evolutionary process.

D. Adaptive fitness function

When a new heuristic $\mathcal{H}$ is generated by genetic operations, its corresponding phenotypic characteristic $phenotype(\mathcal{H})$ is determined by using the technique discussed in section III-A1. Ideally, to measure the fitness of a heuristic $phenotype(\mathcal{H})$, the heuristic will be applied to a set of training instances. In previous studies [9], a large training set is needed to avoid overfitting issues. However, since GP usually requires a large population with expensive program evaluations, using a large training set can significantly increase the running times of the algorithm. To overcome this computational difficulty, we propose an incremental learning approach to measure the fitnesses of evolved heuristics. In each generation, all heuristics in the population are only applied to a small subset of random training instance to determine $obj(\mathcal{H})$. Then, the fitness function is calculated as follow:

$$fitness(\mathcal{H}) = obj_{avg}(\mathcal{H},\mathcal{N}) \times SP(\mathcal{H},\mathcal{N}) \times BF(\mathcal{H},\mathcal{N}) \quad (8)$$

where $obj_{avg}(\mathcal{H},\mathcal{N})$ is the estimated objective values, $SP(\mathcal{H},\mathcal{N})$ is the selection pressure and $BF(\mathcal{H},\mathcal{N})$ is the bloat control factor. $obj_{avg}(\mathcal{H},\mathcal{N})$ is the average of the objective values from all evolved heuristics, similar to $\mathcal{H}$:

$$obj_{avg}(\mathcal{H},\mathcal{N}) = \frac{\sum\limits_{\mathcal{H}' \in NB(n^{\mathcal{N}*}_{\mathcal{H}})} obj(\mathcal{H}')}{freq(n^{\mathcal{N}*}_{\mathcal{H}})} \quad (9)$$

where $obj(\mathcal{H}')$ is the objective of an evolved heuristic $\mathcal{H}'$ recorded in the archive $\mathcal{A}$, $n^{\mathcal{N}*}_{\mathcal{H}}$ is a node in $\mathcal{N}$ with the minimum distance to the transformed $phenotype(\mathcal{H})$, and $NB(n^{\mathcal{N}*}_{\mathcal{H}})$ is the set of neighbours of $n^{\mathcal{N}*}_{\mathcal{H}}$, i.e. basically any nodes in $\mathcal{N}$ that are directly connected to $n^{\mathcal{N}*}_{\mathcal{H}}$ by an edge. Meanwhile, $freq(n)$ is the matching frequency of node $n \in \mathcal{N}$ determined by the number of entries in $\mathcal{A}$ that best match node n (noted that $|\mathcal{A}| = \sum_{n' \in \mathcal{N}} freq(n')$). Because $obj_{avg}(\mathcal{H},\mathcal{N})$ are based on the historical information recorded in $\mathcal{A}$, its values are different in each generation (as $\mathcal{A}$ is updated) and this estimate will be more accurate in the later generations.

The selection pressure can be calculated as follow:

$$SP(\mathcal{H},\mathcal{N}) = 1 + nd(\mathcal{H},\mathcal{N})^{\alpha} \quad (10)$$

where

$$nd(\mathcal{H},\mathcal{N}) = \frac{1}{|\mathcal{A}|}\left(freq(n^{\mathcal{N}*}_{\mathcal{H}}) + \sum_{n \in NB(n^{\mathcal{N}*}_{\mathcal{H}})} freq(n)\right) \quad (11)$$

$SP(\mathcal{H},\mathcal{N})$ allows AGP to reduce the chance of selecting and reproducing heuristics located in a well-explored area of the search space. Since the distribution of evolved heuristics has been captured by mGNG, we can easily determine the density of a certain area in the search space by the neighbourhood density $nd(\mathcal{H},\mathcal{N})$. The coefficient $\alpha \geq 1$ is used to control the impact of diversity on the fitness function. When α is low, AGP will evolve a more diverse population.

The bloat factor can be calculated as follow:

$$BF(\mathcal{H},\mathcal{N}) = \begin{cases} 1 + \left(\frac{size(\mathcal{H})}{ms(\mathcal{H},\mathcal{N})} - 1\right)^{B} & \frac{size(\mathcal{H})}{ms(\mathcal{H},\mathcal{N})} > 1 \\ 1 & otherwise \end{cases} \quad (12)$$

where $size(\mathcal{H})$ is the size of heuristic $\mathcal{H}$ (total number of terminals and functions) and $ms(\mathcal{H},\mathcal{N})$ is the maximum size of heuristics $\mathcal{H}' \in \mathcal{A}$ that best matched node $n^{\mathcal{N}*}_{\mathcal{H}}$ (defined in the previous section). If $size(\mathcal{H}) \leq ms(\mathcal{H},\mathcal{N})$, the bloat factor will have no impact on the estimated fitness of $\mathcal{H}$. Otherwise, the parameter $B \geq 1$ governs the tradeoffs between the sizes of the evolved heuristics and their quality.

IV. Experiment results

To evaluate the effectiveness of the new algorithm, we have two datasets for RCJS. The first data set includes a wide range of randomly generated instances based on the instance generator proposed in [6]. The second data set includes a set of benchmark instances widely used in the literature [1], [2], [3], [4]. The first data set containing 108 instances (from 3 to 6 machines and 30 to 64 jobs) is used for training with AGP. The second data set containing 36 benchmark instances is used for testing the performance of evolved heuristics at the end of AGP runs.

For the experiments in this paper, we set the default AGP's parameters as follows. The population size of AGP is 40000, the size of the intermediate population is 5 times the population size, the number of random training instances used in each generation is 5, the diversity parameter α is 4 and the maximum depth of the program trees is 5. For genetic operators, we randomly use subtree crossover and subtree mutation as described in the previous section. In this paper, we use a very large population to ensure that we have enough genetic materials to maintain a diverse set of scheduling heuristics. The value $\alpha = 4$ is selected based on our pilot experiments, which help to maintain both the exploration and exploitation of AGP.

A. Testing performance

The test performance based on 30 runs of AGP are shown in Table II. In this table, we also show the best solution (BS), the CGACO algorithm [3], and the ACO algorithm [4]. The results of CGACO and ACO are obtained from 1-hour run. The first number in the instance name is the number of machines, hence the results are arranged by the increasing order of complexity. In this table, Obj and $\%dev$ is the average objective values and the average deviation from BSs obtained by the algorithms from all random independent runs. The AGP's results are marked as bold if AGP outperforms CGACO and marked as bold-italic if AGP outperforms both ACO and CGACO.

From the results, it is easy to see that AGP does not perform as well as CGACO and ACO for small instances (with fewer than 6 machines). This is expected because these meta-heuristics can usually find very good, sometimes optimal, solutions for small instances. However, when we move to medium-size instances (from 6 to 9 machines), AGP becomes more competitive and even outperforms CGACO in some instances. AGP shows its real advantage when dealing with large instances. Although CGACO seems to perform better than ACO in large instances, it is clear that AGP outperforms CGACO in most instances with 10 machines or more. The gaps (for both objective values and $\%deviation$) between AGP and CGACO become more significant as the instance size increases. For instances with more 12 machines, the $\%dev$ of AGP is kept below 20% in most cases while the those values of CGACO and ACO become very high. Given that AGP use reusable evolved heuristics to generate solutions for these instances, the time it needs to solve an instance is significantly lower than those of ACO (with 1-hour) and

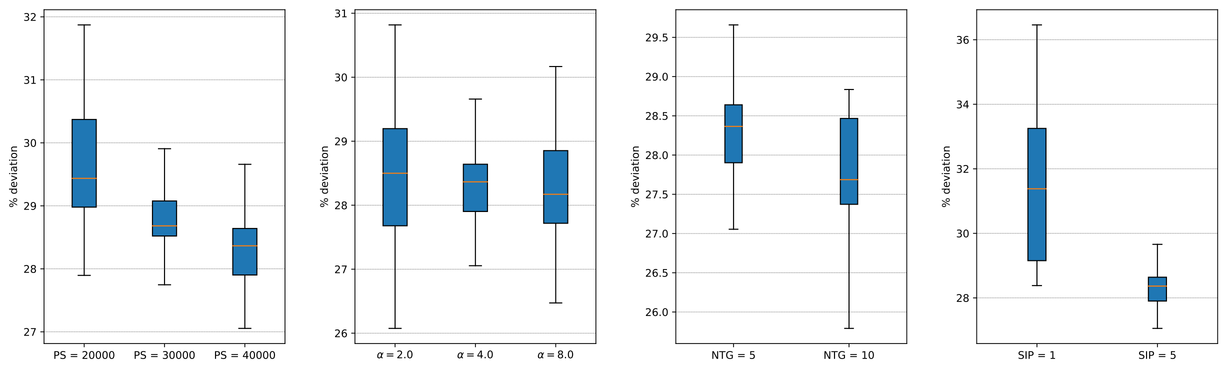
TABLE II
Test performance of AGP compared to existing meta-heuristics

Instance	BS	AGP		CGACO [3]		ACO [4]	
		Obj	$\%dev$	Obj	$\%dev$	Obj	$\%dev$
3–5	505.00	594.96	17.81	559.67	10.83	553.60	9.62
3–23	149.07	164.88	10.61	162.83	9.23	151.79	1.82
3–53	69.36	73.47	5.93	70.02	0.95	70.11	1.08
4–28	23.81	35.40	48.66	28.73	20.66	23.96	0.63
4–42	66.73	111.91	67.70	70.38	5.47	68.55	2.73
4–61	45.96	86.03	87.18	48.58	5.70	45.96	0.00
5–7	252.90	332.56	31.50	308.33	21.92	256.39	1.38
5–21	168.63	213.24	26.45	177.11	5.03	177.84	5.46
5–62	250.67	342.12	36.48	300.70	19.96	279.22	11.39
6–10	861.35	**1001.75**	**16.30**	1031.51	19.76	996.01	15.63
6–28	228.46	311.18	36.21	294.64	28.97	240.07	5.08
6–58	243.20	360.64	48.29	300.89	23.72	265.91	9.34
7–5	438.71	553.51	26.17	537.17	22.44	477.51	8.84
7–23	562.82	**681.96**	**21.17**	764.25	35.79	585.43	4.02
7–47	439.41	635.94	44.73	614.46	39.84	547.12	24.51
8–3	631.81	**956.53**	**51.39**	967.58	53.14	873.44	38.24
8–53	449.22	603.06	34.25	575.94	28.21	480.57	6.98
8–77	1237.21	1597.48	29.12	1544.54	24.84	1498.52	21.12
9–20	930.18	1154.98	24.17	1129.66	21.45	1033.48	11.11
9–47	1233.13	1655.84	34.28	1638.35	32.86	1365.43	10.73
9–62	1460.72	**1767.43**	**21.00**	1807.34	23.73	1630.97	11.66
10–7	2538.17	**2999.12**	**18.16**	3304.73	30.20	2942.94	15.95
10–13	2191.75	**2764.06**	**26.11**	2834.93	29.35	2695.90	23.00
10–31	614.57	**769.24**	**25.17**	815.69	32.73	677.61	10.26
11–21	1017.13	1281.54	26.00	1247.38	22.64	1114.53	9.58
11–56	1790.09	**2234.09**	**24.80**	2337.83	30.60	2184.68	22.04
11–63	2021.89	**2396.62**	**18.53**	2451.35	21.24	2329.96	15.24
12–14	1766.43	***2125.57***	***20.33***	2342.19	32.59	2295.43	29.95
12–36	2968.87	***3683.40***	***24.07***	4136.53	39.33	3756.95	26.54
12–80	2457.55	***2753.37***	***12.04***	3215.07	30.82	3061.05	24.56
15–2	3927.99	***4258.26***	***8.41***	5529.95	40.78	6036.81	53.69
15–3	4327.26	***5108.68***	***18.06***	6346.85	46.67	5764.59	33.22
15–5	3490.20	***4256.99***	***21.97***	5497.48	57.51	4629.45	32.64
20–2	8344.87	***9449.50***	***13.24***	10421.02	24.88	11497.30	37.78
20–5	14533.30	***15520.40***	***6.79***	18238.09	25.49	20678.80	42.29
20–6	7438.49	***8327.00***	***11.94***	9740.80	30.95	9850.56	32.43
# wins	–	9	–	2	–	25	–

CGACO (with 1-hour and 8 CPU cores). These results show that AGP is a very efficient approach to dealing with large RCJS instances. In additional, the result here also confirm that the reusable heuristics have good generalisation and scalability. Although trained with small and unrelated (i.e. different sources) instances, the reusable heuristics show very good performance on the test instances. In the rest of this section, further analyses are presented to show the influences of parameters on AGP's performance and behaviours.

B. Influence of population size

The test performance $\%dev$ in Fig. 2(a) shows increasing the population size can significantly improve the performance of AGP. However, it should be noted that the improvement is not exceptional as the $\%dev$ only reduce 1% as the population increases from $20,000$ to $40,000$. This is further confirmed when we observe the training progress of AGP in Fig. 3(a) (noted that the test performance is only sampled every 20 generation to save computational costs) and Fig. 4(a). AGP behaves very similarly as the population size increases. These



(a) Population size (b) Diversity control (c) Training instances (d) Size of intermediate population

Fig. 2. Parameter sensitivity analyses.

analyses suggest that population size alone cannot efficiently and significantly improve the performance AGP.

C. Influence of diversity control

Fig. 2(b) and Fig. 3(b) show that $\alpha \geq 2$ do not show a significant impact on the test performance and the program size although a slight decrease in test performance is observed as the $alpha$ values increase. AGP with a high values of α (i.e. 4,8) seems to produce less variance. In the future study, a more detailed analysis is need to carefully examine the role of diversity control in AGP.

D. Influence of training instances

Fig. 2(c) and Fig. 3(c) show that increasing the number of training instances per evaluation can improve the test performance of AGP. One explanation is that more training set will allow AGP to more accurately identify good heuristics. However, as shown in Fig. 3(c), the number of training instances show a greater impact in the early stage of the evolution. As the evolution progresses, the impact of a larger training set is not very clear. This is expected as more data is available in $\mathcal{A}$ and $\mathcal{N}$, which makes the adaptive fitness function predict the heuristic performance more accurately.

E. Influence of intermediate population

The size of the intermediate population is the most important parameter as shown in Fig. 2(d) and Fig. 3(d). A larger intermediate population not only significantly improves the test performance but also reduces its variance. It is also clear that the advantage of a large intermediate population is sustained as the evolution progresses, which is different from what we have observed with the number of training instances. Given that the size of intermediate population does not significantly increase the computational times (as compared to the increase in the number of training instances) of AGP, a reasonably large intermediate population is desirable. One disadvantage of a large intermediate population size, as shown in Fig. 4, is that it may slightly increase the average size of evolved heuristics and slow down the evaluation process. However, due to the bloat control trick embedding directly into the adaptive fitness function, the sizes of evolved heuristics are systematically controlled.

The results here are also compared to the GP methods proposed in [18] for evolving multi-pass iterative heuristics. It is clear that the proposed AGP outperforms GP [18] for all test instances (the results are not shown here due to space limitations). Nonetheless, in future studies, additional experiments need to be conducted to fully understand the contribution of each of the evolved heuristics.

V. Conclusions

In this paper, we develop a new adaptive genetic programming (AGP) algorithm to coevolve a diverse set of scheduling heuristics for RCJS. Contributions of this paper include a new phenotypic representation of scheduling heuristic, a mapping technique that monitor the evolutionary process, and an adaptive fitness function that balance both the quality and size of evolved heuristics. The experimental results show that the reusable heuristics evolved by AGP are very effective for medium and large RCJS instances. For large instances, evolved heuristics can easily outperform customised meta-heuristics such as ACO and CGACO.

In future studies, we plan to further examine diversity control strategy in AGP and look at the solution to dynamically adjust the population size of AGP to make the algorithm more efficient. As the gaps between solutions found by evolved heuristics and the best known solutions are still significant, there is a lot room for further improvements.

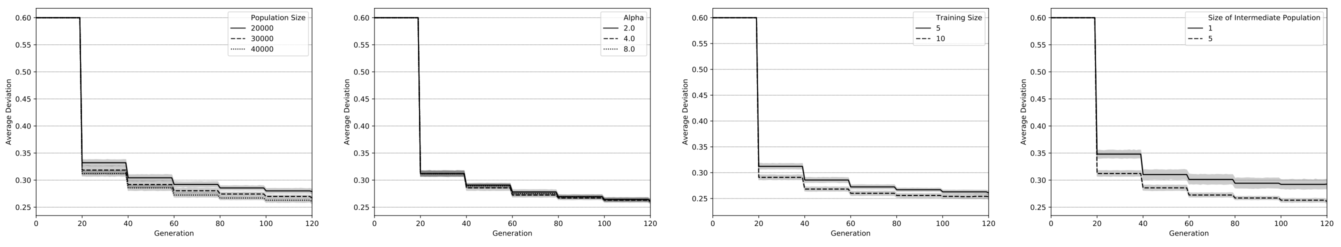
References

[1] G. Singh and A. T. Ernst, “Resource Constraint Scheduling with a Fractional Shared Resource,” *Operations Research Letters*, vol. 39, no. 5, pp. 363 – 368, 2011.

[2] A. T. Ernst and G. Singh, “Lagrangian Particle Swarm Optimization for a resource constrained machine scheduling problem,” in *2012 IEEE Congress on Evolutionary Computation*, June 2012, pp. 1–8.

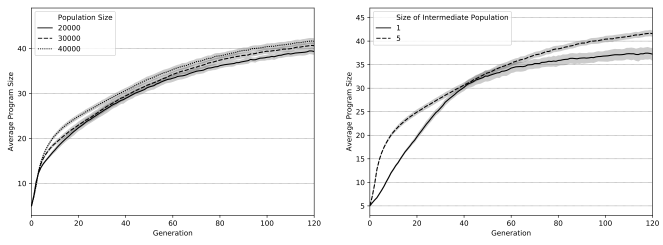
[3] D. Thiruvady, G. Singh, and A. T. Ernst, “Hybrids of Integer Programming and ACO for Resource Constrained Job Scheduling,” in *Hybrid Metaheuristics*, vol. 8457. LNCS, 2014, pp. 130–144.

[4] D. Thiruvady, A. T. Ernst, and G. Singh, “Parallel Ant Colony Optimization for Resource Constrained Job Scheduling,” *Annals of Operations Research*, vol. 242, no. 2, pp. 355–372, July 2016.



(a) Population size (b) Diversity control (c) Training instances (d) Size of intermediate population

Fig. 3. Influence of parameters on training progress.



(a) Population size (b) Size of intermediate population

Fig. 4. Influence of parameters on program sizes.

[5] D. Cohen, A. Gómez-Iglesias, D. Thiruvady, and A. T. Ernst, *Resource Constrained Job Scheduling with Parallel Constraint-Based ACO*. Cham: Springer International Publishing, 2017, pp. 266–278.

[6] S. Nguyen, D. Thiruvady, A. T. Ernst, and D. Alahakoon, "A Hybrid Differential Evolution Algorithm with Column Generation for Resource Constrained Job Scheduling," *Computers & Operations Research*, vol. 109, pp. 273–287, 2019.

[7] C. Blum, D. Thiruvady, A. T. Ernst, M. Horn, and G. Raidl, "A Biased Random Key Genetic Algorithm with Rollout Evaluations for the Resource Constraint Job Scheduling Problem," in *AI 2019: Advances in Artificial Intelligence*, 2019, pp. 549–560.

[8] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 3rd ed. New York: Springer, 2008.

[9] S. Nguyen, Y. Mei, and M. Zhang, "Genetic programming for production scheduling: A survey with a unified framework," *Complex & Intelligent Systems*, vol. 3, no. 1, pp. 41–66, Mar. 2017.

[10] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward, "Automating the Packing Heuristic Design Process with Genetic Programming," *Evolutionary Computation*, vol. 20, no. 1, pp. 63–89, 2012.

[11] R. van Lon, J. Branke, and T. Holvoet, "Optimizing Agents with Genetic Programming: An Evaluation of Hyper-heuristics in Dynamic Real-time Logistics," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1, pp. 93–120, Jun 2018.

[12] N. Pillay, "Evolving hyper-heuristics for the uncapacitated examination timetabling problem," *Journal of the Operational Research Society*, vol. 63, no. 1, pp. 47–58, 2012.

[13] J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 110–124, 2016.

[14] J. Branke, T. Hildebrandt, and B. Scholz-Reiter, "Hyper-heuristic Evolution of Dispatching Rules: A Comparison of Rule Representations," *Evolutionary Computation*, vol. 23, no. 2, pp. 249–277, Jun. 2015.

[15] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Automatic Design of Scheduling Policies for Dynamic Multi-objective Job Shop Scheduling Via Cooperative Coevolution Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 2, pp. 193–208, 2014.

[16] R. R. S. van Lon, J. Branke, and T. Holvoet, "Optimizing Agents with Genetic Programming: An Evaluation of Hyper-heuristics in Dynamic Real-time Logistics," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1, pp. 93–120, 2017.

[17] E. Hart and K. Sim, "A Hyper-heuristic Ensemble Method for Static Job-shop Scheduling," *Evolutionary Computation*, vol. 24, no. 4, pp. 609–635, Dec. 2016.

[18] S. Nguyen, D. Thiruvady, A. Ernst, and D. Alahakoon, "Genetic Programming Approach to Learning Multi-pass Heuristics for Resource Constrained Job Scheduling," in *Proceedings of the Genetic and Evolutionary Computation Conference - GECCO 2018*, 2018, pp. 1167–1174.

[19] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch, "Resource-constrained Project Scheduling: Notation, Classification, Models, and Methods," *European Journal of Operational Research*, vol. 112, pp. 3–41, 1999.

[20] F. Ballestin and N. Trautmann, "An Iterated-local-search Heuristic for the Resource-constrained Weighted Earliness-tardiness Project Scheduling Problem," *International Journal of Production Research*, vol. 46, pp. 6231–6249, 2008.

[21] E. Demeulemeester and W. Herroelen, *Project Scheduling: A Research Handbook*. Boston, MA, USA: Kluwer, 2002.

[22] K. Neumann, C. Schwindt, and J. Zimmermann, *Project Scheduling with Time Windows and Scarce Resources*. Berlin, Germany: Springer, 2003.

[23] D. Thiruvady, M. Wallace, H. Gu, and A. Schutt, "A Lagrangian Relaxation and ACO Hybrid for Resource Constrained Project Scheduling with Discounted Cash Flows," *Journal of Heuristics*, vol. 20, no. 6, pp. 643–676, 2014.

[24] O. Brent, D. Thiruvady, A. Gmez-Iglesias, and R. Garcia-Flores, "A Parallel Lagrangian-ACO Heuristic for Project Scheduling," in *2014 IEEE Congress on Evolutionary Computation*, 2014, pp. 2985–2991.

[25] D. Thiruvady, C. Blum, and A. T. Ernst, "Maximising the Net Present Value of Project Schedules Using CMSA and Parallel ACO," in *Hybrid Metaheuristics*, 2019, pp. 16–30.

[26] D. Thiruvady, G. Singh, A. T. Ernst, and B. Meyer, "Constraint-based ACO for a Shared Resource Constrained Scheduling Problem," *International Journal of Production Economics*, vol. 141, no. 1, pp. 230–242, 2012.

[27] G. Singh and R. Weiskircher, "Collaborative Resource Constraint Scheduling with a Fractional Shared Resource," in *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, vol. 2. IEEE, 2008, pp. 359–365.

[28] ——, "A Multi-Agent System for Decentralised Fractional Shared Resource Constraint Scheduling," *Web Intelligence and Agent Systems*, vol. 9, no. 2, pp. 99–108, 2011.

[29] B. Fritzke, "A Self-organizing Network That Can Follow Non-stationary Distributions," in *Proceeedings of International Conference on Artificial Neural Networks*, 1997, pp. 613–618.

[30] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[31] T. Hildebrandt and J. Branke, "On Using Surrogates with Genetic Programming," *Evolutionary Computation*, vol. 23, no. 3, pp. 343–367, 2014.

[32] S. Nguyen, M. Zhang, and K. C. Tan, "Adaptive Charting Genetic Programming for Dynamic Flexible Job Shop Scheduling," in *GECCO '18*, 2018, pp. 1159–1166.

[33] E. K. Burke, S. Gustafson, and G. Kendall, "Diversity in Genetic Programming: An Analysis of Measures and Correlation with Fitness," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 1, pp. 47–62, Feb. 2004.

[34] S. Nguyen, M. Zhang, D. Alahakoon, and K. C. Tan, "People-Centric Evolutionary System for Dynamic Production Scheduling," *IEEE Transactions on Cybernetics*, 2019.

[35] S. Furao and O. Hasegawa, "An Incremental Network for On-line Unsupervised Classification and Topology Learning," *Neural Networks*, vol. 19, no. 1, pp. 90–106, 2006.

[36] J. Mouret and J. Clune, "Illuminating Search Spaces by Mapping Elites," *CoRR*, vol. abs/1504.04909, 2015.