An empirical evaluation of permutation-based policies for stochastic RCPSP

Olivier Regnier-Coudert
AIRBUS AI Research
Toulouse, France
olivier.regnier-coudert@airbus.com

Guillaume Povéda
AIRBUS AI Research
Toulouse, France
guillaume.poveda@airbus.com

ABSTRACT

While optimization methods used for deterministic scheduling such as Linear Programming, Constraint Programming or Evolutionary Algorithms can be very successful at optimizing scheduling problems, the resulting schedules may not always be feasible and applicable at execution on domains with uncertainty. In this paper, we focus on the Stochastic Resource Constrained Project Scheduling Problem (SRCPSP) and propose several adaptive scheduling policies that use task priorities as input to a schedule generation scheme (SGS) at execution. In particular, we focus on the use of Genetic Programming (GP) to evolve robust heuristics that can assign priority levels to scheduling tasks online. The benefit of this approach is two-fold. First, it enables the adaptation of the SGS permutation input during the execution. Second, because the evolved heuristic uses task features rather than domain features, it offers the advantage to be applicable on completely unseen domains, potentially of higher dimension and complexity. Experiments on domains with stochastic durations show that using GP-evolved heuristics yield better makespan than using fixed permutations derived from optimal schedules. They also demonstrate that the update of the permutation is key to getting full benefit from SGS policies.

CCS CONCEPTS

• **Theory of computation** → **Evolutionary algorithms**; **Scheduling algorithms**; **Genetic programming**;

ACM Reference Format:
Olivier Regnier-Coudert and Guillaume Povéda. 2021. An empirical evaluation of permutation-based policies for stochastic RCPSP. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3449726.3463154

1 INTRODUCTION

Scheduling is a vast topic concerned with the optimized allocation of resources to tasks under a set of constraints. There exist many scheduling problems which are broadly studied from different perspectives. Operational research methods [12] can produce optimal solutions but the use of Evolutionary Algorithms (EAs) [10, 11, 19] or hybrid approaches such as Large Neighborhood Search (LNS) [21] is often required to scale up to large dimensions or specific variants of scheduling with large search spaces. Regardless of the method used, a schedule produced by any optimization method may face challenges when executed in an environment that presents uncertainty, for example an environment where the duration of tasks is not precisely known, or where resource availability may vary unexpectedly in time or where additional tasks may need to be inserted at execution in the schedule. On such domains, an optimized schedule may quickly become unfeasible requiring repair or re-computation. Both approaches have pros and cons. Where repairing a schedule by applying local changes to it will ensure that it remains close to the initial schedule, it may also not be possible to do so easily, especially in the face of highly disruptive events. On the other hand, re-optimization taking into account the up-to-date problem data will likely produce a solution of high quality but it will also be time-consuming.

In order to address this issue, there are alternatives to fixed schedules. First, efforts have been put in producing flexible schedules where options are pre-computed and optimized. For example, such schedules can allow the possible permutations of groups of tasks at execution without impact on the final objectives [2]. Flexible scheduling solutions reduce the re-computation needed at execution although some may still be required if events lead to a state where the pre-computed options are no longer applicable.

In this paper, we focus on another type of scheduling output: scheduling policies. The term policies is often used in reinforcement learning and planning to define a procedure that recommends an action given a state of the system. Such policies are typically optimized offline using training data and can then be used in most states of a system without the need for online computation. As a concrete example, in a simple robot control policy, the state is defined by the measurement of a robot' sensors (e.g. camera input) and the actions are linked to the robot's actuators (e.g. joints' movements). In the scheduling context, we propose policies where a state is defined using what is known about the scheduling domain at execution (i.e. tasks already started, tasks completed, current resource allocations...) and an action can either be to start a task or to do nothing at the current time.

More precisely, we build on the work by [6] and define permutation based scheduling policies where task priority levels are modelled into a permutation used as input to a schedule generation scheme (SGS). This approach offers the advantage to generate a schedule at minimal computational cost whenever the execution domain changes and to do so based on the results of an offline optimization. The main algorithm presented in this study uses Genetic Programming to evolve hyper-heuristics (GPHH) to specify task

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463154

priority levels. The optimization is done in a robust way, using several training instances. It is compared on the Stochastic Resource Constrained Project Scheduling Problem (SRCPSP) against permutation-based policies where the permutation is optimized once and fixed.

In order to obtain feasible schedules, the heuristics produced by GP generate permutations that are used as inputs to a decoding procedure (i.e. SGS). This characteristics makes GPHH a case of indirect search. While we can argue that indirect search prevents the full search space from being explored, there is also a large body of literature on the topic, revealing that it has been successfully applied to many areas of computational intelligence including set covering problems [1], constraint satisfaction and graph colouring problems [13], routing and sequencing [18], project scheduling [4] or earth observation satellite mission planning [22]. It has also been shown that indirect search is a good compromise between exploration of the search space and integration of domain knowledge [20].

The paper is organised as follows. First, in section 2, the SRCPSP variant used in this study is detailed. In section 3, we describe the SGS procedure and permutation-based policies. In section 4, details on GPHH are given. Finally, sections 5 and 6 present the experimental setup and the analysis of the results.

2 STOCHASTIC RESOURCE CONSTRAINED PROJECT SCHEDULING

2.1 RCPSP

A resource-constrained project scheduling problem (RCPSP) considers resources of limited availability and activities of known durations and known resource need, linked by precedence relations. The problem consists of finding a schedule of minimal duration by assigning a start time to each activity such that the precedence relations and the resource availabilities are respected.

Formally a RCPSP is defined by a set of tasks T, a set of resource types R and a set of precedence relation pairs P of dimension $|T|\times|T|$. $P_{ij} = 1$ if task i needs to be completed before task j can be started, 0 otherwise. The quantity of resource of type k required to perform task i is given by $r_{i,k}$. We denote by R_k the total number of resources of type k available at any time. In RCPSP, this level of resource does not vary. Finally the duration of a task i is given as d_i.

Solving RCPSP consists in assigning a start time x_i to all tasks in T such that the makespan expressed in (1) is minimized.

$$f = \max_{i\in[1,|T|]} (x_i + d_i) \tag{1}$$

It is subject to the following constraints: the precedence constraints (2), the number of resources used at any time should not exceed the resources available for any resource type k (3), where h is the time horizon (a reasonably large number fitting most feasible schedules) and $c_{k,t}$ the amount of resource of type k used by active tasks at time t as expressed in (4).

$$x_i + d_i \leq x_j \ \forall i, j \in T \text{ if } P_{i,j} = 1 \tag{2}$$

$$c_{k,t} \leq R_k \ \forall t \in [0, h] \ \forall k \in R \tag{3}$$

$$c_{k,t} = \sum_{i\in T,\ x_i \leq t < x_i + d_i} r_{i,k} \ \forall k \in R \tag{4}$$

For more information on RCPSP, we refer the reader to [3].

2.2 SRCPSP and MDP formulation

In this paper, we focus on a stochastic version of the RCPSP where the duration d_i of a task i is unknown until the task is started. Thus, the RCPSP previously described is made stochastic by sampling d_i in the uniform distribution $X \sim U(\max(1, d_i^o - \delta),\ d_i^o + \delta)$ where d_i^o is the duration in the original RCPSP instance and δ a noise level. Because setting δ to 10 yields a variety of distinct instances while remaining close enough to the original durations, this settings was used for this work.

Sequentially searching for a strategy that optimises long-term rewards (or minimises costs) under probabilistic uncertain events is a long-standing research problem dating back to the mid-twentieth century and Bellman's work on stochastic dynamic programming [5]. To handle such dynamic environments and define strategies, the problem can be expressed as a Markov Decision Process (MDP) [23]. A MDP is defined by sets of states and actions, a probability transition function describing noised transitions between states depending on the chosen action, and a reward (or cost) function labelling the transitions. The aim of typical MDP methods such as Policy Iteration [23] or Q-Learning [25] is to find a policy (i.e. a function mapping states to actions) which maximises the average discounted sum of cumulated rewards when executing this policy from a given state. Despite their efficiency, MDP solvers rely on an efficient modelling of the problem with search and action spaces of reasonable size, which is not the case with scheduling problems where the state space dimension exponentially increases with the number of tasks [8], illustrating the famous curse of dimensionality. Dealing with SRCPSP's uncertainty is nevertheless possible using MDP approaches. In [24], the authors formulated a MDP of a SRCPSP with uncertain task arrival during execution and used an approximate policy iteration to deal with the high dimension of the state space. Similar MDP formulations for RCPSP had been used in [8] and [7] for SRCPSP with stochastic task duration. In those studies, the authors reduced the state space by using heuristics before optimizing the policy on this subspace.

The algorithms presented in this paper circumvent the high dimension issue of the SRCPSP by not using the standard MDP solvers but by still using the MDP formulation as a backbone to provide executable policies.

In this section, we provide the definition of the *states* and *actions* as modelled in the stateful MDP formulation of the SRCPSP used in this paper.

2.2.1 State space. The scheduling state s is defined by the attributes described below.

- $s.time$, $t \in [0, h]$: the time associated with the state
- $s.status_i \in \{REMAINING, ONGOING, ENDED\}$, $\forall i \in T$: the status of any task i.
- $s.start_i \in [0, h]$, $\forall i \in T$: the start time of any task i. We assume $s.start_i = \varnothing$ if i has not started.
- $s.end_i \in [0, h]$, $\forall i \in T$: the end time of any task i. We assume $s.end_i = \varnothing$ if i has not been completed.

- $s.duration_i, \forall i \in T$: the sampled duration of any task i. In a deterministic setting, $s.duration_i = d_i$
- $s.progress_i \in [0, 1], \forall i \in T$: the progress of any task expressed as a percentage. The progress is calculated as a linear function of the active time given a task' sampled duration.
- $s.res_k^{used}, \forall k \in R$: the amount of resources used across all active tasks.
- $s.res_k^{avail}, \forall k \in R$: the amount of resources available.

2.2.2 Actions. An action a is defined by :

- $a.type \in \{START, PROGRESS\}$: the type of action to perform. An action can either lead to starting a task or progressing in time.
- $a.task \in T$: the task on which to apply the action. This attribute is only considered when $a.type = START$ (i.e. progressing in time is not an action associated with any task in particular).

We show in Figure 1 an example of how actions and states interact and in particular how many actions can be used in sequence to perform changes on several tasks at the same time step before eventually progressing in time.

3 SCHEDULE GENERATION SCHEME AND FIXED PERMUTATION-BASED POLICIES

SGS procedures are commonly used as a component of the optimization pipelines for the different variants of RCPSP. SGS uses as input a permutation of tasks in order to create a feasible schedule taking into consideration the problem data. SGS has been widely used to enable the use of EAs on RCPSP problems, allowing them to search in the space of permutations rather than in the space of schedules [4, 26]. There are two main SGS procedures. In the *serial SGS*, tasks are considered in the order of the input permutation and inserted in the schedule at their earliest possible slot. On the other hand, the *parallel SGS* inserts all the tasks that can be started at each time step, incrementing the time until all tasks are scheduled. Like for its serial counterpart, the *parallel SGS* also inserts tasks considering the order of the input permutation. Both approaches are suitable for RCPSP although each exhibits its own strengths and weaknesses. In this paper, only the *serial SGS* is used. We refer the reader to [15] for the detailed description of SGS and a comparison of the different versions of SGS. Note that to be able to run SGS at execution, SGS can be initialized with a partial schedule set with tasks already ongoing and resource levels that account for the resources already in use.

Using SGS, it is possible to define policies for the MDP defined in the previous section. These policies take a current state s as input and output a recommended action a. To do so, a schedule is computed using SGS and a fixed permutation π. If a task can be started at the current time $s.time$, the recommended action a will be to start the first task from all the tasks that can be started at $s.time$. If no task can be started, the recommended action will be to do nothing and progress to the next time step. The algorithm 1 gives the details of the permutation-based policy using SGS.

The problem under study being of stochastic nature, the duration of a task is not known until the task is started. In order to run SGS that requires a task duration for each task, estimated durations are used. These are taken from the original RCPSP instance, i.e. the instance that is noised when generating the stochastic domain. Because of the model used to generate the SRCPSP from it, the estimates are the mean expected durations.

It is important to note that the permutation π has a strong impact on SGS and therefore on the quality of the actions that can be returned by the policy. With fixed permutation-based policies, π remains unchanged at execution. However, π can be optimized prior to the execution. In this paper, we use a Constraint Programming (CP) solver to optimally solve a deterministic RCPSP instance that represents an average instance of the SRCPSP. More precisely, the CP solver is run on the original RCPSP instance.

Algorithm 1: Permutation policy procedure

Input: state s
$x = SGS(s, \pi)$
$T^{selected} =$
$\{i, \forall i \in [1, |T|],\ x_i = s.time,\ s.status_i = REMAINING\}$
`// tasks whose start time equals current time`
if $|T^{selected}| > 0$ **then**
  $a.type = START$
  $task = T^{selected}[0]$
  $a.task = task$
else
  $a.type = PROGRESS$
end
Output: action a

4 GENETIC PROGRAMMING AND DYNAMIC PERMUTATION-BASED POLICIES

Although the permutation-based policies described in the previous section enables adaptation of the schedule to stochastic events, the permutation at the core of the policy remains unchanged during the execution of the schedule even if the sampled durations have significantly derived from the expected durations used for the offline optimization of the fixed permutation.

In this section we suggest an approach to re-generate the permutations used by SGS within the scheduling policy. To do so, we define heuristics that perform a series of operations on a selection of task features to compute a priority level for any given task. A simple example of such a heuristic is the *sum* of the *number of successors* of a task and its *earliest possible start*. By computing the result of this heuristic for all the tasks to schedule, a priority level α_i is generated for each. From all α_i, a permutation can be generated by simply ordering tasks by priority levels, the tasks with lowest α_i being in the last positions of the permutations and inversely the tasks with the highest priority levels being in the first positions.

To define the heuristics, the following 7 operators are allowed: *addition*, *subtraction*, *multiplication*, *division*, *maximum*, *minimum* and *negative*. Among all operators, only *negative* takes a single input argument, the others taking exactly 2. Also note that division by zero is prevented, the *division* operator returning 1 in this case.

In terms of task features, 11 were defined to be used by the heuristics. We give here their description for a task i. Features fall

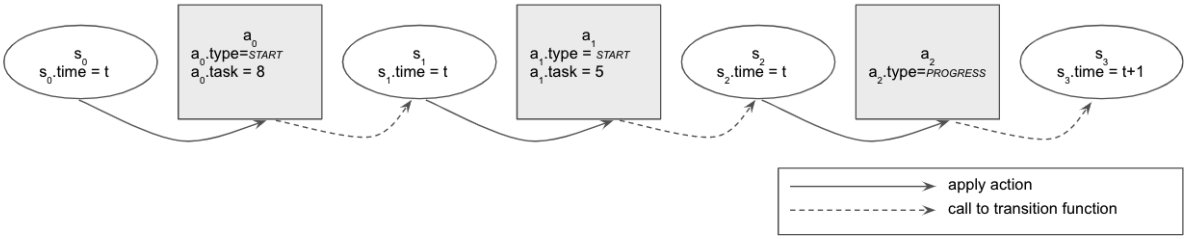


Figure 1: Example of a sequence of actions applied at the same time step t. It shows that only an action of type *PROGRESS* can lead to a change in time. The example also shows that actions are related to at most 1 task. Starting from a state s_0, an action a_0 that starts the task 8 is applied and leads to state s_1 with similar time t. Another action a_1 where another task (task 5) is started is then applied and leads to state s_2. The timestamp of s_2 is still unchanged. The third action applied if of type *PROGRESS* and leads to state s_3 whose timestamp is later than s_2.

in 3 categories. First, there are 3 features related to the precedence graph of the problem: *number of predecessors*, *number of successors* and *number of descendants*. The difference between *number of successors* and *number of descendants* is that the former counts the number of tasks that are directly paired with i in P, while the latter counts all the tasks that depend directly or indirectly on the completion of i to be started. The second category of features is related to resource needs and concerns the following 4 features: *average resource*, *maximum resource*, *minimum resource* and *distinct resource types*, respectively described in (5), (6), (7) and (8). Finally, there are 4 additional features that describe information about expected scheduling time bounds. These are *earliest start time*, *latest start time*, *earliest end time* and *latest end time*. They are calculated from the critical path schedule [14] obtained using known durations for tasks that are started and durations from the original problem for pending tasks. To be generic across problems with different task durations, these features are normalized through division with the earliest start time of the latest completed task in the critical path schedule.

$$avgr = \frac{\sum_{k \in R_k} \frac{r_{i,k}}{R_k}}{|R|} \tag{5}$$

$$maxr = \max_{k \in R_k} \frac{r_{i,k}}{R_k} \tag{6}$$

$$minr = \min_{k \in R_k} y, y = \begin{cases} \frac{r_{i,k}}{R_k}, \text{if } r_{i,k} > 0 \\ 1 \text{ otherwise} \end{cases} \tag{7}$$

$$sumr = \frac{\sum_{k \in R_k} y}{|R|}, y = \begin{cases} 1, \text{if } r_{i,k} > 0 \\ 0 \text{ otherwise} \end{cases} \tag{8}$$

Although it is possible to generate heuristics by hand, it is difficult to do so and despite offering perspective for interpretability, it does not offer any guarantee in terms of quality. Thus, we turned towards GP to generate and optimize those heuristics.

We provide the pseudo-code of GPHH in Algorithm 2. The main steps of GPHH are those of a generic GP, where a population of heuristics *pop* is evolved for *max_gen* generations through selection, crossover and mutation of candidate heuristics c. Of particular importance is the evaluation of the fitness f_c that is the average makespan obtained by running SGS on each training instance L_m from the initial state s_0 and using the permutation π computed from the priority levels α obtained when applying c on each task. The instances used for training are deterministic and generated by sampling the SRCPSP.

5 EXPERIMENTS

5.1 Baseline approaches

To assess and compare the different permutation-based scheduling policies presented in this paper, we also include results from baseline methods. We briefly describe the two baselines considered: a simple but efficient priority rule and a method to compute a lower bound on SRCPSP.

5.1.1 Max Descendant Priority rule (MDPR). Priority rules for RCPSP have been widely studied by Kolisch [16] and are simple and explainable ways of solving RCPSP. They can also be used as online policies in order to deal with uncertainty. The MDPR priority rule policy is derived from the *most total successors* priority rule also described in [16]. We define for all $i \in T, Q_i$ as the number of activity nodes in the RCPSP (directed) precedence graph that are reachable from i, which corresponds in graph theory terminology to descendants nodes of i. The MDPR policy is described in Algorithm 3. In the first 3 steps, unscheduled tasks whose predecessors are complete and whose resource needs can be handled are selected as $T^{selected}$. If there is at least one task in this selection, the policy will start the task with the highest number of descendants. Note that MDPR does not rely on SGS.

5.1.2 Lower bound. We use a CP approach to solve RCPSP and get a lower bound on the makespan for each of the test instances used for the execution of the policies. Thus, to compute the lower bound, the test durations were provided to the CP solver. Our model is similar to the *minizinc-benchmark* codebase[1] developed by University of Melbourne and NICTA. Using the lazy clause generation CP solver Chuffed [2] [9], with a 60 second timeout, final results on

[1] https://github.com/MiniZinc/minizinc-benchmarks/blob/master/rcpsp/rcpsp.mzn
[2] https://github.com/chuffed/chuffed

Algorithm 2: GPHH

Generate initial population of heuristics pop
$gen = 1$
for $c \in pop$ **do**
  $f_c = evaluate(c)$
end
while $gen \leq max\_gen$ **do**
  $pop^c = \{\}$
  while $|pop^c| < |pop|$ **do**
    $c1, c2 = \text{select}(pop)$
    $c1, c2 = \text{crossover}(c1, c2)$, with probability c_rate
    $c1 = \text{mutate}(c1)$, with probability m_rate
    $c2 = \text{mutate}(c2)$, with probability m_rate
    for $c \in \{c1, c2\}$ **do**
      $f_c = evaluate(c)$
      Add c to pop_c
    end
  end
  $pop = pop_c$
  $gen += 1$
end
function `evaluate(c)`:
  $f = 0$
  for $m \in [1, |L|]$ **do**
    Get tasks T and task durations D from training domain L_m
    for $i \in T$ **do**
      $\alpha_i = c(i, L_m)$
    end
    $\pi = perm(\alpha)$
    $x = SGS(s_0, \pi)$
    $f += \max_{i \in [1,|T|]}(x_i + d_i)$
  end
  $f = f/|L|$
return

Algorithm 3: MDPR

Input: state s
$T^{selected} = \{i \; \forall i \in [1, |T|], \text{if } s.status_i = REMAINING\}$
$T^{selected} = \left\{i \; \forall i \in T^{selected}, \text{if } \forall j \in T, P_{j,i} = 1 \rightarrow s.status_j = ENDED\right\}$
$T^{selected} = \left\{i \; \forall i \in T^{selected}, \text{if } \forall k \in R, r_{i,k} + s.res_k^{used} \leqslant s.res_k^{avail}\right\}$
if $|T^{selected}| > 0$ **then**
  $a.type = START$
  $task = \arg\max_{i \in T^{selected}} Q_i$
  $a.task = task$
else
  $a.type = PROGRESS$
end
Output: action a

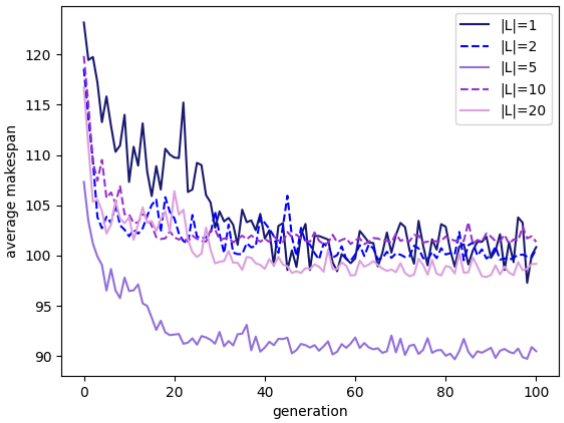


Figure 2: Average population fitness during 5 runs of GPHH, using different numbers of training instances.

the instances are obtained from 0.5 seconds (j30) to an order of 1 minute for j120. Note that 3 out of 1500 noised instances have not been solved optimally but are kept as lower bounds nevertheless.

5.2 Experimental setup

Following preliminary experiments, GPHH was configured as follows: population size of 40, 100 generations, tournament selection with tournament size of 10% of the population size, crossover rate of 0.7, mutation rate of 0.3 and initial tree depth between 1 and 4. In Figure 2, the evolution of the population fitness in GPHH is shown for 5 runs, each using a different number of training instances. It shows that the above settings allow enough evaluations for GPHH to converge, regardless of the number of training samples. For the main set of experiments, we use a total of 10 deterministic training instances sampled from the SRCPSP instance.

To evaluate our approaches, we build SRCPSP instances starting from well studied PSPLIB instances [17] and using the process described in Section 2. We consider the 3 classes of problems: j30, j60, and j120 each having respectively 30, 60 and 120 tasks. We build a SRCPSP for each of the 10 instances of RCPSP of each class (30 SRCPSP in total). To evaluate the different policies, we execute them 50 times per SRCPSP, each execution sampling different task durations. Therefore, in total, each method is evaluated on 1500 distinct executions. We emphasise the fact that none of the execution instances were seen by any of the approaches during training to the exception of the CP solver used to compute the lower bound baseline. The comparison is based on the makespan achieved after execution of the policies on each test run. We also compute after each run the relative deviation (RD) to the lower bound LB as $RD(algo) = (makespan(algo) - makespan(LB))/makespan(LB)$. Statistical significance is tested using unpaired student t-tests using the relative deviation to the lower bound. We also consider the ranks of the algorithms and perform a Wilcoxon rank-sum test. We consider statistical significance if the p-value is lower than 0.01.

All experiments described in this paper were conducted using the SRCPSP MDP implementation of the scikit-decide [3] library. The scheduling algorithms GPHH, CP-SGS and MDPR are also available in the same library.

6 RESULTS AND DISCUSSION

Tables 1, 2 and 3 present the results obtained on the 30 instances of SRCPSP. The makespan and RD are averaged over the 50 executions performed. The rank shown in the tables is the rank of each algorithm calculated based on the mean makespan. Focusing on the rank, GPHH ranks first in 63% of the test runs and so clearly outperforms MDPR (23%) and CP-SGS (17%) as best approach. In particular, the difference in ranks is high on the j30 and j60 instances. However, it seems harder to distinguish a best method on j120 (40% for MDPR, 50% for GPHH). The average ranking obtained over all the test problems confirms this trend with GPHH exhibiting a rank of 1.56, MDPR a rank of 2.03 and CP-SGS a rank of 2.37. Of particular interest is the difference in performance between the two SGS-based approaches CP-SGS and GPHH, demonstrating the benefit of using dynamic permutations rather than fixed ones. It also highlights the fact that the heuristics built from the selected features and operators can discriminate between tasks and have a real influence on SGS and on the quality of the resulting schedules. The observations done by studying the ranks are also confirmed when comparing makespans and RDs. In Figure 3, the distribution of the distance to the lower bound obtained over the 1500 test executions is shown for both CP-SGS and GPHH. The distribution associated with both methods is different. GPHH produces many more schedules with makespan close to the lower bound than CP-SGS does and more generally speaking, the distribution of the results of GPHH is more skewed towards low values than the one of CP-SGS. We provide in Table 5 the p-values from student t-tests run to compare each pair of algorithms in terms of their relative deviation from the lower bound. The student t-tests show that the difference between GPHH and CP-SGS is statistically significant (p-values < 0.01) regardless of the problem dimension. When comparing GPHH with MDPR, we can consider GPHH as statistically better than MDPR but we observe that this difference is highly linked with the problem family as significance is not reached on the instances j60 and j120. The final message from this table is that we cannot significantly differentiate CP-SGS and MDPR based on the results presented in this paper. Note that performing Wilcoxon rank-sum test did not yield statistical differences.

Another performance indicator we use to compare the approaches is the number of test instances on which the lower bound was reached. These numbers are given in Table 4 and shows that GPHH reaches the optimum makespan at execution on 13% of the runs, against 11% and 10% for CP-SGS and MDPR.

When focusing on the RD presented in Tables 1, 2 and 3, some large differences are obtained between GPHH and the other algorithms. On some runs, GPHH outperforms CP-SGS by more than 30%. It is however interesting to note that the same findings occur the other way round with CP-SGS being much closer to the lower bound than GPHH. Thus and despite very good average results, GPHH suffers on some occasions of a lack of consistency. Because the evolution of the heuristics rely on stochastic processes, the quality of the learnt heuristic can vary a lot between successive runs of GPHH and we suggest that the poor performance seen on some instances is linked to this and should be targeted in the future. In order to confirm this intuition, we selected the execution instance where the RD of GPHH was the worst in comparison to the other approaches. We re-run 50 times the GPHH evolution, resulting in 50 different heuristics and executed these heuristics on the selected execution instance of *j301-7* (random seeds were set to ensure the sampled durations were similar across the 50 executions). This side experiment showed that out of 50 heuristics, 48 produced a similar makespan at execution (69), while 2 of them produced considerably worst schedules of makespan 87 and 98 respectively. This strengthens the idea that the poor performance on some instances may be linked to occasional poor evolved heuristics. Although this is a reliability issue that will be the focus of future work, we note that all algorithms suffer from similar issue.

Despite being effective at producing quality schedules, the heuristics produced by GPHH are highly complex with typically 80+ operations required to compute the priority level for a task. Such observation is not surprising as rule complexity is not modelled nor optimized by GPHH. However, making the heuristics more efficient as done in [6] would improve their explanability.

7 CONCLUSION

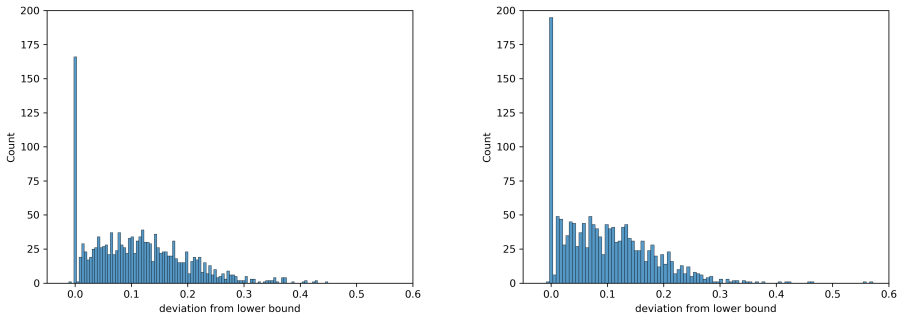
In this paper, we have conducted an empirical analysis of three priority-based policies for stochastic RCPSP. The experiments have revealed that there is value in updating the permutation used by SGS policies at execution, leading to improved makespans. Moreover, where CP-SGS performed at a similar level than the baseline MDPR, GPHH outperforms this latter, suggesting that the use of dynamic permutations is needed to unlock the full benefits of SGS policies.

Despite encouraging results, the comparison carried out suggests several areas for future research on GPHH. First, issues of performance and reliability should be investigated. Anticipated solutions to this problem include the inclusion of more task features (e.g. slack time in the critical path schedule), the move from deterministic to stochastic training data and the use of ensembles of heuristics to generate priority levels rather than relying on a single evolved heuristic. The high complexity of the evolved rules suggests that multi-objective optimization could be used with the consideration of complexity-related objectives. In order to better understand the suitability of GPHH on stochastic scheduling problems, future work should consider comparison with MDP-oriented methods such as Monte-Carlo Tree Search and evaluation on a wider range of RCPSP problems (e.g. multi-skill RCPSP, uncertain resource availability...). Finally, a key advantage of GPHH over alternative algorithms lies in the fact that evolved heuristics are generic and therefore, we believe that an evaluation of their ability to generalize to completely unseen problems of possibly higher dimensions than the training instances should be carried out.

[3] scikit-decide is an open source python and C++ library for reinforcement learning, planning and scheduling. It is developed by Airbus and available at https://github.com/airbus/scikit-decide

REFERENCES

[1] Uwe Aickelin. 2002. An indirect genetic algorithm for set covering problems. *Journal of the Operational Research Society* 53, 10 (2002), 1118–1126.



(a) CP-SGS (b) GPHH

Figure 3: Distribution of the deviation from lower bound for CP-SGS and GPHH over the 1500 test instances

Table 1: Results obtained by GPHH, MDPR and CP-SGS on j30 instances

Instance	Algorithm	Avg. RD	Avg. makespan	Rank
	CP-SGS	0.06	68.94 (10.75)	2
j301-1	MDPR	0.07	69.31 (10.74)	3
	GPHH	0.06	**68.76 (10.97)**	1
	CP-SGS	0.06	**68.76 (8.94)**	1
j301-2	MDPR	0.08	70.61 (11.16)	3
	GPHH	0.07	69.63 (9.79)	2
	CP-SGS	0.09	71.45 (11.37)	3
j301-3	MDPR	0.08	70.57 (10.01)	2
	GPHH	0.04	**68.37 (11.87)**	1
	CP-SGS	0.09	80.57 (12.48)	3
j301-4	MDPR	0.07	79.35 (11.88)	2
	GPHH	0.06	**78.47 (12.08)**	1
	CP-SGS	0.08	**68.02 (11.06)**	1
j301-5	MDPR	0.08	68.12 (10.14)	2
	GPHH	0.10	69.69 (12.21)	3
	CP-SGS	0.12	76.49 (11.37)	3
j301-6	MDPR	0.07	73.45 (12.91)	2
	GPHH	0.06	**72.22 (11.87)**	1
	CP-SGS	0.11	80.88 (13.92)	3
j301-7	MDPR	0.09	79.65 (14.4)	2
	GPHH	0.07	**78.59 (15.61)**	1
	CP-SGS	0.07	76.43 (11.75)	2
j301-8	MDPR	0.09	78.04 (12.35)	3
	GPHH	0.05	**74.98 (12.07)**	1
	CP-SGS	0.06	**73.51 (8.77)**	1
j301-9	MDPR	0.07	74.41 (9.85)	2
	GPHH	0.08	74.92 (8.39)	3
	CP-SGS	0.04	68.8 (11.91)	2
j301-10	MDPR	0.07	70.67 (12.64)	3
	GPHH	0.04	**68.76 (11.31)**	1

Table 2: Results obtained by GPHH, MDPR and CP-SGS on j60 instances

Instance	Algorithm	Avg. RD	Avg. makespan	Rank
	CP-SGS	0.08	103.51 (12.41)	2
j601-1	MDPR	0.09	103.98 (12.51)	3
	GPHH	0.07	**102.22 (13.15)**	1
	CP-SGS	0.08	103.86 (11.39)	2
j601-2	MDPR	0.10	105.1 (12.87)	3
	GPHH	0.06	**102.02 (10.67)**	1
	CP-SGS	0.11	98.63 (11.41)	3
j601-3	MDPR	0.08	**96.41 (13.27)**	1
	GPHH	0.09	96.8 (12.95)	2
	CP-SGS	0.09	118.08 (15.48)	2
j601-4	MDPR	0.10	119.31 (14.78)	3
	GPHH	0.09	**117.84 (15.97)**	1
	CP-SGS	0.10	107.96 (13.88)	3
j601-5	MDPR	0.10	107.88 (14.13)	2
	GPHH	0.09	**106.73 (13.13)**	1
	CP-SGS	0.08	94.92 (12.16)	3
j601-6	MDPR	0.06	93.43 (12.78)	2
	GPHH	0.06	**93.08 (13.31)**	1
	CP-SGS	0.16	109.37 (10.49)	3
j601-7	MDPR	0.13	107.45 (14.07)	2
	GPHH	0.12	**105.86 (11.65)**	1
	CP-SGS	0.11	112.0 (11.77)	3
j601-8	MDPR	0.09	**108.92 (10.67)**	1
	GPHH	0.11	111.14 (10.23)	2
	CP-SGS	0.11	116.57 (12.29)	2
j601-9	MDPR	0.10	**115.49 (12.93)**	1
	GPHH	0.13	119.02 (14.59)	3
	CP-SGS	0.09	108.8 (14.13)	3
j601-10	MDPR	0.06	106.53 (14.36)	2
	GPHH	0.04	**104.82 (14.54)**	1

Table 3: Results obtained by GPHH, MDPR and CP-SGS on j120 instances

Instance	Algorithm	Avg. RD	Avg. makespan	Rank
j1201-1	CP-SGS	0.14	158.86 (13.35)	3
	MDPR	0.14	158.57 (16.28)	2
	GPHH	0.12	**156.47 (15.38)**	1
j1201-2	CP-SGS	0.16	167.39 (18.73)	2
	MDPR	0.16	**166.88 (17.32)**	1
	GPHH	0.17	168.18 (16.56)	3
j1201-3	CP-SGS	0.14	181.53 (20.54)	3
	MDPR	0.13	**180.12 (22.8)**	1
	GPHH	0.13	180.69 (22.46)	2
j1201-4	CP-SGS	0.16	**150.16 (12.38)**	1
	MDPR	0.15	**150.16 (15.69)**	1
	GPHH	0.16	150.33 (13.92)	3
j1201-5	CP-SGS	0.15	165.35 (15.4)	3
	MDPR	0.14	**163.1 (15.88)**	1
	GPHH	0.14	163.53 (14.2)	2
j1201-6	CP-SGS	0.21	138.0 (11.88)	3
	MDPR	0.18	134.51 (10.65)	2
	GPHH	0.18	**134.43 (10.61)**	1
j1201-7	CP-SGS	0.19	171.63 (14.84)	3
	MDPR	0.17	169.37 (16.82)	2
	GPHH	0.13	**163.37 (13.14)**	1
j1201-8	CP-SGS	0.17	164.55 (14.91)	3
	MDPR	0.14	159.63 (16.77)	2
	GPHH	0.13	**159.27 (15.91)**	1
j1201-9	CP-SGS	0.15	**166.29 (13.04)**	1
	MDPR	0.18	171.12 (18.99)	2
	GPHH	0.19	171.18 (14.28)	3
j1201-10	CP-SGS	0.14	164.84 (13.91)	2
	MDPR	0.19	172.02 (15.68)	3
	GPHH	0.13	**163.67 (14.89)**	1

Table 4: Number of test instances where CP-SGS, MDPR and GPHH have reached the lower bound makespan

CP-SGS	MDPR	GPHH
167	151	196

Table 5: p-values from student t-tests performed between algorithms on the different problem classes and based on the relative deviation from the lower bound

	GPHH / CP-SGS	GPHH / MDPR	CP-SGS / MDPR
All	$\mathbf{1.39.10^{-6}}$	**0.0011**	0.1068
j30	**0.0038**	**0.0029**	0.9745
j60	**0.0008**	0.2845	0.0166
j120	**0.0021**	0.0321	0.3994

[2] Christian Artigues, Jean-Charles Billaut, Azzedine Cheref, Nasser Mebarki, and Zakaria Yahouni. 2016. Robust machine scheduling based on group of permutable jobs. In *Robustness Analysis in Decision Aiding, Optimization, and Analytics*. Springer, 191–220.

[3] Christian Artigues, Sophie Demassey, and Emmanuel Neron. 2008. *Resource-constrained project scheduling*. Wiley Online Library.

[4] Mayowa Ayodele, John McCall, and Olivier Regnier-Coudert. 2017. Estimation of distribution algorithms for the multi-mode resource constrained project scheduling problem.. In *Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE.

[5] Richard Bellman. 1957. *Dynamic Programming* (1 ed.). Princeton University Press, Princeton, NJ, USA.

[6] Shelvin Chand, Hemant Singh, and Tapabrata Ray. 2019. Evolving heuristics for the resource constrained project scheduling problem with dynamic resource disruptions. *Swarm and evolutionary computation* 44 (2019), 897–912.

[7] Jaein Choi, Matthew Realff, and Jay Lee. 2007. A Q-Learning-based method applied to stochastic resource constrained project scheduling with new project arrivals. *International Journal of Robust and Nonlinear Control* 17 (09 2007), 1214 – 1231. https://doi.org/10.1002/rnc.1164

[8] Jaein Choi, Matthew J. Realff, and Jay H. Lee. 2004. Dynamic programming in a heuristically confined state space: a stochastic resource-constrained project scheduling application. *Computers Chemical Engineering* 28, 6 (2004), 1039 – 1058. FOCAPO 2003 Special issue.

[9] Geoffrey Chu. 2011. *Improving combinatorial optimization*. Ph.D. Dissertation. University of Melbourne, Australia.

[10] Kaizhou Gao, Zhiguang Cao, Le Zhang, Zhenghua Chen, Yuyan Han, and Quanke Pan. 2019. A review on swarm intelligence and evolutionary algorithms for solving flexible job shop scheduling problems. *IEEE/CAA Journal of Automatica Sinica* 6, 4 (2019), 904–916.

[11] Mitsuo Gen, Wenqiang Zhang, Lin Lin, and YoungSu Yun. 2017. Recent advances in hybrid evolutionary algorithms for multiobjective manufacturing scheduling. *Computers & Industrial Engineering* 112 (2017), 616–633.

[12] Farhad Habibi, Farnaz Barzinpour, and Seyed Sadjadi. 2018. Resource-constrained project scheduling problem: review of past and recent developments. *Journal of project management* 3, 2 (2018), 55–88.

[13] Xingxing Hao and Jing Liu. 2017. A multiagent evolutionary algorithm with direct and indirect combined representation for constraint satisfaction problems. *Soft Computing* 21, 3 (2017), 781–793.

[14] James E Kelley. 1963. The critical-path method: resource planning and scheduling. *Industrial scheduling* (1963).

[15] Jin-Lee Kim and Ralph D Ellis Jr. 2010. Comparing schedule generation schemes in resource-constrained project scheduling using elitist genetic algorithm. *Journal of construction engineering and management* 136, 2 (2010), 160–169.

[16] Rainer Kolisch. 1996. Efficient priority rules for the resource-constrained project scheduling problem. *Journal of Operations Management* 14, 3 (1996), 179 – 192.

[17] Rainer Kolisch and Arno Sprecher. 1997. PSPLIB - A project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program. *European Journal of Operational Research* 96, 1 (1997), 205 – 216.

[18] Jiří Kubalík and Michal Snížek. 2014. A novel evolutionary algorithm with indirect representation and extended nearest neighbor constructive procedure for solving routing problems. In *Intelligent Systems Design and Applications (ISDA), 2014 14th International Conference on*. IEEE, 156–161.

[19] Lin Lin and Mitsuo Gen. 2018. Hybrid evolutionary optimisation with learning for production scheduling: state-of-the-art survey on algorithms and applications. *International Journal of Production Research* 56, 1-2 (2018), 193–223.

[20] Charles Neau, Olivier Regnier-Coudert, and John McCall. 2018. An Analysis of Indirect Optimisation Strategies for Scheduling. In *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1–8.

[21] Mireille Palpant, Christian Artigues, and Philippe Michelon. 2004. LSSPER: Solving the resource-constrained project scheduling problem with large neighbourhood search. *Annals of Operations Research* 131, 1 (2004), 237–257.

[22] Guillaume Povéda, Olivier Regnier-Coudert, Florent Teichteil-Königsbuch, Gérard Dupont, Alexandre Arnold, Jonathan Guerra, and Mathieu Picard. 2019. Evolutionary approaches to dynamic earth observation satellites mission planning under uncertainty. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1302–1310.

[23] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley Sons, Inc.

[24] Mahshid Salemi Parizi, Yasin Gocgun, and Archis Ghate. 2017. Approximate policy iteration for dynamic resource-constrained project scheduling. *Operations Research Letters* 45, 5 (2017), 442 – 447.

[25] Richard S Sutton, Andrew G Barto, et al. 1998. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge.

[26] Vincent Van Peteghem and Mario Vanhoucke. 2014. An experimental investigation of metaheuristics for the multi-mode resource-constrained project scheduling problem on new dataset instances. *European Journal of Operational Research* 235, 1 (2014), 62–72.