Evolving priority rules for resource constrained project scheduling problem with genetic programming

Mateja Đumić[a,*], Dominik Šišejković[b], Rebeka Čorić[a], Domagoj Jakobović[c]

[a]Department of Mathematics, University of Osijek, Trg Ljudevita Gaja 6, Osijek, Croatia
[b]Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
[c]Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

Abstract

The main task of scheduling is the allocation of limited resources to activities over time periods to optimize one or several criteria. The scheduling algorithms are devised mainly by the experts in the appropriate fields and evaluated over synthetic benchmarks or real-life problem instances. Since many variants of the same scheduling problem may appear in practice, and there are many scheduling algorithms to choose from, the task of designing or selecting an appropriate scheduling algorithm is far from trivial. Recently, hyper-heuristic approaches have been proven useful in many scheduling domains, where machine learning is applied to develop a customized scheduling method. This paper is concerned with the resource constrained project scheduling problem (RCPSP) and the development of scheduling heuristics based on Genetic programming (GP). The results show that this approach is a viable option when there is a need for a customized scheduling method in a dynamic environment, allowing the automated development of a suitable scheduling heuristic.

Keywords: genetic programming, resource constrained scheduling, hyper-heuristics
2010 MSC: 00-01, 99-00

1. Introduction

Scheduling is a process that deals with the allocation of resources to tasks over given time periods. It is used on everyday basis in many manufacturing and services industries. The task of scheduling is to optimize one or more objectives. Often, the process includes some limitations like the capacity of vehicles, working time of employees, limited funds etc.

*Corresponding author
Email addresses: `mdjumic@mathos.hr` (Mateja Đumić), `sisejkovic@ice.rwth-aachen.de` (Dominik Šišejković), `rcoric@mathos.hr` (Rebeka Čorić), `domagoj.jakobovic@fer.hr` (Domagoj Jakobović)

The focus of this work is on one of the scheduling problems with limitations - the resource constrained project scheduling problem (RCPSP). RCPSP is a problem with two kinds of constraints - precedence and resource. One job can require other activities to be completed before it starts with processing. All jobs need resources in various amounts, and resources are limited. The goal of solving RCPSP is to schedule all jobs in a given project in way that all constraints are met, and one or more criteria are optimized.

Many real life problems can be formulated as RCPSP, and due to the fact that RCPSP is very hard to solve, many solving methods were developed. The decision of which solving method to use must be made depending on the problem characteristics. Usually, solving methods can be divided into two groups - exact methods and heuristics. Exact methods are impractical on problems which have a large number of jobs to schedule, so the heuristic methods are mostly used. Some heuristic methods show better results on specific problem instances, but no heuristics that are good for all problems exist (proven by the No free lunch theorem [1]). Because of that there are some authors that try to combine different heuristic approaches. A common example of this approach are genetic algorithms combined with local search methods.

Since heuristic methods give good results only when applied to problems containing specific characteristics, thereby being application specific, there is a need to rise to a higher level of solving - searching the space of solving methods, and not searching the solution space. This is why hyper-heuristics are being developed. Genetic programming (GP) is one of algorithms that can be used as a hyper-heuristic which produces new heuristics. This paper demonstrates the use of GP as a hyper-heuristic to evolve appropriate scheduling heuristics for the RCPSP.

The paper is organized as follows: in section 2 a definition of RCPSP is given, while in section 3 a brief overview of methods for solving RCPSP is given. Section 4 describes how GP can be used in evolving new scheduling heuristics in form of priority rules. The results are presented in section 5, and short conclusion and future research directions are given in section 6.

2. RCPSP

In general, the resource constrained project scheduling problem considers activities and resources. Activities have known durations and resource demands, while resources are of limited availability. Additionally, activities are linked by precedence relations. The problem consists of finding a feasible schedule with minimal total duration by assigning start times to each activity such that the precedence and resource constraints are satisfied.

All *activities* constituting the project are defined by the set $A = \{A_0, \dots, A_{n+1}\}$. By convention the activities A_0 and A_{n+1} represent the start and the end of a schedule. These activities are usually referred to as dummy activities. Duration of activity A_i is p_i, and duration of both dummy activities is 0. An activity can not be interrupted once it is started. This property is referred to as not allowing *preemption*.

The *precedence relation* is given by the set E which defines pairs such that $(A_i, A_j) \in E$ means that activity A_i precedes activity A_j. Additionally, we assume that A_0 precedes all other activities and that A_{n+1} succeeds all other activities.

Generally, resources can be categorized as *renewable*, *non-renewable* and *doubly-constrained* [2]. In this article we shall work exclusively with renewable resources which are available at any given time with full replenishment capacity. Each activity has demands on resources and in order for an activity to execute, all demands have to be satisfied.

In the most of cases, goal of the problem is to minimize makespan, i.e. total project duration time, but objective function can be anything else like profit maximization, the uniform use of resource etc.

To make problem easier to solve for each activity in the problem instance a variety of properties can be calculated to define the *time-frame* in which a specific activity can be scheduled. Let the set of all activities that can be scheduled at a given time t be defined as $E(t)$ - the set of eligible activities at time t. Then we can define $E(t)$ as follows:

$$E(t) = \{A_j : A_j \in A, ES_j + 1 \leq t \leq LF_j\}, \tag{1}$$

where ES_j stays for the earliest start time of activity A_j, and LF_j for the latest finish time of activity A_j. i.e. each activity A_j can be scheduled and executed in the given time frame between its earlier start and latest finish time, where we assume integer time values. More about how to calculate this time-frame for each activity can be found in [3] with remark that the aforementioned calculation procedures assume unlimited resources and rely only on the precedence relations.

The concrete difficulty of a RCPSP instance depends upon many different parameters of which the following are mentioned as most important in literature [4]: network complexity (NC) and the effect of resources: resource factor (RF) and resource strength (RS). Experiments conducted by *Kolisch et al.* [5] show that a negative but very weak correlation exists between the network complexity and the project execution time, while a great magnitude of a positive correlation between the resource factor and execution time exists as for a negative correlation between the resource strength and the execution time.

3. Previous work

RCPSP was introduced in 1963. by Kelley. Even though it is a more than 50 years old problem, it is still interesting and new solving methods are still being developed. According to the computational complexity theory, the RCPSP is one of the most intractable combinatorial optimization problems and belongs to the class of *NP-hard* problems [6]. RCPSP solving methods can generally be divided into exact and heuristic approaches [4].

Exact solving approaches search the complete space of feasible solutions and therefore guarantee optimality [7]. But the search space is often of impractical size which makes such approaches almost useless for a very large number of problem instances [8]. However, in literature there are few examples of exact algorithms that produce good results on small problem instances like mathematical planning [9] and numerical methods like dynamical programming [10]. Exact algorithms can be divided into the following four main categories: Integer Programming ([3, 11]), Implicit Enumeration ([12, 13]), Branch-and-bound ([14, 15]), Dynamic Programming ([16]).

As exact methods are generally not applicable for larger problem instances, many different heuristic approaches have been developed. Heuristics differ from exact methods by searching only a part of the solution space which offers possible better performance in a given time frame but not optimality. Nevertheless, in most cases generating a feasible and good enough solution is far more important than optimality. Therefore heuristic approaches are a popular and useful option for solving the RCPSP.

Known scheduling heuristics for solving the resource constrained project scheduling problems can further be classified into two categories: *priority rule-based methods* (or constructive heuristics) and *metaheuristic-based approaches* (or improvement heuristics) [17].

The first class of methods always starts with no scheduled activity. Construction of a complete schedule is controlled by combination of schedule generation scheme ([18]) and *priority rules* ([17]). The general idea is that the SGS builds a schedule from scratch taking resource and precedence constraints into account. During the building process, the SGS upgrades *partial schedules* until all activities are scheduled and a complete feasible schedule is generated. Which activity SGS is going to schedule next depends on priority rule. In the literature, two different schedule generation schemes are available: the *serial* SGS and the *parallel* SGS. Both schemes generate feasible schedules but differ in the way activities and time is handled throughout the procedures. More about SGS can be found in [18].

The second class of methods is applied to initial complete solutions with the goal of achieving improvement in terms of a selected criterion. The main representatives of this category are genetic algorithms [19, 20], tabu search, simulated annealing [21], ant colony optimization [22] and others.

One of the heuristic methods, that is not mentioned yet, is Genetic programming (GP). In the last 15 years GP was used in scheduling and achieved good results. GP has been used for wide variety of environments like single machine scheduling [23, 24], multiple machine scheduling [25], job shop scheduling [26, 27, 28], unrelated machine environment [29]. In scheduling GP is mostly used for evolving dispatching rules (priority functions) and recently Branke et al. [30] brought the survey of GP approaches used in scheduling. Also in literature we can find comparison of different rule representation [27, 31], and comparision with human-produced results [32]. GP is also interesting because it offers a bridge between the mentioned two heuristics classes: GP is a metaheuristic-based approach capable of evolving priority rules used in incremental schedule generation. Rather than searching the space of schedules, the GP may be used to search for an appropriate priority rule to generate schedules. This approach, also known as hyper-heuristic [33, 34], may be efficient in cases where different conditions and user-defined criteria make it nontrivial to choose an adequate priority rule from existing ones. It is important to notice that GP is mostly used in dynamic environment, because once evolved rule can be used for different problem instances, and give result in short time that is not case with other above mentioned heuristics.

Despite of good results in scheduling, to our best knowledge, only one article [35] where GP is used for RCPSP exists in literature. In this article GP was shown as good approach for designing rules which are fast and can be used in dynamical environment which is important because lot of real-life situations demand quick reaction and quick decision. Our approach is different from approach in this article in way of producing more exhaustive terminal and function set which results with improvement of fitness. Also, we made comparison with best known priority rules in literature, not with other metaheuristic methods, because priority rules are mostly intended for usage in dynamic rules and their advantage is not in small deviation from best known result, but in fast and *good enough* result, so comparison with other schedule improvement metaheuristics would not be fair.

4. Priority Resource Constrained Scheduling With Genetic Programming

Functionally, Genetic Programming (GP) is an evolutionary algorithm inspired by biological evolution which aims to find *computer programs* that perform a user-defined computational task. It is therefore a machine learning technique that searches the solution space of programs instead of actual solutions to problem instances. GP is able to evolve *higher-level* solutions (programs) that can be applied to generate solutions to a variety of problem instances.

In our approach, GP is used to generate appropriate *priority functions* that govern the selection of activities to be scheduled at the given moment in time. This allows the customization of the scheduling algorithm, because different tailor made priority functions can be evolved for any concievable scheduling criteria. Unlike the priority rule, the schedule generation scheme is defined manually; note that different priority functions may be used within a single SGS.

As GP is a machine learning technique, it is necessary to provide a learning set upon which GP can evolve priority functions that can be used in combination with a schedule generation scheme to form a list of activities and provide a feasible complete schedule of good quality. The priority function is used to calculate activities' priorities which are used to prioritize certain activities in the selection process of a given SGS.

The main goal is to evolve appropriate priority functions on a selected learning set and successfully apply them to any new project instance, even outside the learning set. Even though the process of evolving a quality priority function takes longer time than actually finding a solution to a given project instance, the resulting priority function can afterwards be used for generating schedules for any instance in a matter of milliseconds.

In order to create a flexible and customizable development-experimental environment a *hyper-heuristic* model was devised and implemented to evolve appropriate scheduling heuristics for the RCPSP using genetic programming. To ease the usage of available GP implementations, the concrete implemented model is coupled with ECF (Evolutionary Computation Framework) [36] and written using the *C++* programming language.

4.1. RCPSP Terminal and Function Set

In order for GP to be expressive and have the tools to evolve intelligent solutions, it is necessary to define an appropriate and domain-specific terminal and function set. The defined function set comprises of basic arithmetic and logic operators as well as some custom ones as shown in Table 1. It is up to GP to use these operators to build more complex and meaningful operations.

Table 1: GP Function Set

Function name	**Definition**
+, -, *	addition, subtraction and multiplication
/	Protected division: $DIV(a,b) = \begin{cases} 1, & \lvert b \rvert < 0.000000001 \\ \frac{a}{b}, & otherwise. \end{cases}$
MAX	$MAX(a) = \begin{cases} a, & a > 0 \\ 0, & otherwise. \end{cases}$
POS	$POS(a) = \begin{cases} a, & a > 0 \\ -a, & otherwise. \end{cases}$
NEG	$NEG(a) = -a$
IF	$IF(a,b,c) = \begin{cases} b, & a > 0 \\ c, & otherwise. \end{cases}$

Choosing an appropriate terminal set is a far more comprehensive task, especially in terms of RCPSP. In general, domain-specific terminals can be divided into two categories: static and dynamic terminals. Each of these categories can further be divided into project-specific and activity-specific terminals. Project-specific terminals include terminals which are specific for a project instance and depend only on the general project properties and not on specific activities. Activity-specific terminals are terminals which reflect properties specific to a concrete activity.

Static terminals do not change during execution, which means it is possible to calculate their values only once before the scheduling begins. The complete set of static terminals is presented in table 2.

Table 2: Static Terminal Set

Category	**Terminal**	**Description**
Project-specific	RF	resource factor
	RS	resource strength
	TNA	total number of activities (not including dummies)
	TD	total project duration (horizon)
Activity-specific	D	activity duration
	RR	number of required resources
	RRT	RR times quantity required for each resource
	ARU	average resource usage
	DPC	number of direct predecessors
	DSC	number of direct successors
	TPC	total number of predecessors
	TSC	total number of successors
	SPC	number of stages (levels) in predecessors' tree
	SSC	number of stages (levels) in successors' tree
	GRPW*	greatest rank positional weight all
	ES	earliest activity start
	EF	earliest activity finish
	LS	latest activity start
	LF	latest activity finish

Table 3: Dynamic Terminal Set

Category	Terminal	Description
Project-specific	NUA	number of unprocessed activities
	NAA	number of active activities
	NPA	number of processed activities
	SUD	sum of durations of unprocessed activities
	SAD	sum of durations of active activities
	SPD	sum of durations of processed activities
Activity-specific	NSP	number of scheduled predecessors
	SL	slack: $max(EF - D - time_, 0)$

Dynamic terminals are more flexible and their value can change according to the current state of activities and resources. Therefore, if dynamic terminals are used, it is necessary to recalculate priorities of certain activities during execution. The set of used dynamic terminals is given in table 3.

4.2. Custom SGS

While the priority functions are evolved with GP, the SGS in this work is defined manually, based on existing RCPSP schemes. In this work three custom SGS versions are devised and implemented. Two versions are based on the parallel SGS while the third is an implementation of the serial SGS. Complexity of SGS does not change by this customization and for both, serial and parallel SGS, is $\mathcal{O}(n^2 \cdot |\mathcal{R}|)$, where $|\mathcal{R}|$ stands for number of resources [37].

The first implementation features a parallel SGS with inserted idleness. This version selects the highest priority activity and schedules other activities only if they do not delay the start of the selected activity. This is true only if the highest-priority activity can be scheduled at a given time as the eligible set only takes the precedence constraint into account while the resource constraint is resolved dynamically during the scheduling process. The property that allows a certain activity to be delayed and scheduled at a later point is called *inserted idleness*. This approach is presented as Algorithm 1 .

Algorithm 1 Custom Parallel SGS with inserted idleness

Input: *prFunc* - priority function, A - set of activities, p - vector of activities' duration, B - vector of resources' availabilities, D - activity demands on resource, G - precedence relation

```
if $prFunc$ has no dynamic terminals then
  calculate priorities for $A$
end if
while some activity from $A$ is not proccessed do
  find set of eligible activities E
  if $prFunc$ has no dynamic terminals then
    calculate priorities for $A$
  end if
  Sort $E$ by priority
  find next possible scheduling time $t_{next}$ for first activity in $E$
  for $i$ to $|E|$ do
    $A_i = E[i]$
    if $i > 0$ $A_i$ create delay then
      continue
    end if
    if resource are available for $A_i$ then
      Schedule $A_i$
    end if
  end for
  Skip to next time
end while
```

Output: S - vector of starting time for each activity

The second version of a parallel SGS is very similar but does not include idleness. Therefore, activities are scheduled (when possible) by priority without special focus on the activity with the highest priority. This version is shown as Algorithm 2.

Algorithm 2 Custom Parallel SGS with no inserted idleness

Input: *prFunc* - priority function, A - set of activities, p - vector of activities' duration, B - vector of resources' availabilities, D - activity demands on resource, G - precedence relation

```
if $prFunc$ has no dynamic terminals then
  calculate priorities for $A$
end if
while some activity from $A$ is not proccessed do
  find set of eligible activities E
  if $prFunc$ has no dynamic terminals then
    calculate priorities for $A$
  end if
  Sort $E$ by priority
  find next possible scheduling time $t_{next}$ for first activity in $E$
  for $i$ to $|E|$ do
    $A_i = E[i]$
    if resource are available for $A_i$ then
      Schedule $A_i$
    end if
  end for
  Skip to next time
end while
```

Output: S - vector of starting time for each activity

The parallel SGS versions are of particular importance as they are built upon a time-incrementation approach which is easily applicable in dynamic conditions.

Another approach concerns the serial SGS. Here no special customizations of the usual serial SGS are made, with the only difference that the implemented serial SGS supports dynamic activity calculation. Here it is important to note that the implementation performance of the parallel SGS in our experiments outperforms the serial SGS by far. This is due to the need of the serial SGS to check resource and precedence constraints for each time moment of an activity's duration, which is a very demanding operation.

Also, since the serial SGS is based on activity-incrementation it is therefore not suitable for use in dynamic conditions, at least not in its primal form.

4.3. The Learning and Test Set

As a machine learning technique, GP needs a selected learning set to learn and evolve results of expected quality. In this work existing problem sets were

Table 4: Data set values range.

Property name	**Values**
Number of activities	{30, 60, 90, 120}
Network complexity	{1.5, 1.8, 2.1}
Resource factor	{0.25, 0.5, 0.75, 1.0}
Resource strength	{0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 1.0}

used from the *project scheduling problem library* (PSPLIB[1]) [38] which contains various types of synthetic resource constrained project scheduling problems as well as optimal and heuristic solutions.

The existing project instances are of different properties differing in the number of activities, network complexity, resource factors, resource strengths and project horizon. The range and various values of the used instances is presented in Table 4.

Here the first step is to decide upon a large enough learning set that will cover a variety of instances with different properties. While the total number of available project instances is slightly greater than 2000, a learning set of 56 representative instances has been established that covers variants of property values in Table 4.

The second step includes establishing a large enough test set containing instances the algorithm has not seen before during the learning phase. As the test set all available instances without the selected learning set are used, comprising a test set of 1984 instances.

4.4. Optimization Criterion

A crucial step in using evolutionary computation techniques is modelling the fitness function. There are many different criteria that can be analysed in the RCPSP as TWT (total weighted tardiness), ET_w (weighted earliness and tardiness), total weighted completion time and others. The initial experiments focus on the *project makespan*.

As project instances differ in the number of activities (30 to 120) and their hardness (in terms of finding good solutions) it is advisable to define a fitness function that is normalized accordingly, meaning that it gives the same range of values for any project instance.

In this approach, we include the number of activities across different instances, so the following fitness function is used:

$$f_i = \frac{C_i}{p_i^{avg} \cdot \sqrt{n_i}}, \tag{2}$$

[1]http://www.om-db.wi.tum.de/psplib/

where f_i is the fitness value of the i_{th} problem instance, C_i the achieved makespan, p_i^{avg} the average activity duration and n_i the number of activities for project instance i. One could argue about the selected division with $\sqrt{n_i}$ as being a subjective measure, but a handful of tests for a variety of project instances and measures proved this decision valid, since normalized values are achieved for the whole range of used instances.

As GP learns on base of a learning set containing more than one project instance, the final fitness value is determined on behalf of the achieved values of all projects as follows:

$$f_k = \frac{\sum_i^N f_i}{N} = \frac{\sum_i^N \frac{C_i}{p_i^{avg} \cdot \sqrt{n_i}}}{N}, \tag{3}$$

where f_k is the fitness value for one individual solution for all projects in a given set with N instances. As can be seen from the selected equation, in each evaluation step only the value C_i is subject to change while all other values are constants. Therefore, the goal to find the minimal f_k reflects in favouring ever smaller values of C_i.

4.5. Variable Optimization Criteria

Another beneficial aspect of the GP approach for solving the RCPSP is the ability to easily target any optimization criterion by simply changing the fitness function.

Even though the makespan criterion is mainly targeted in this work, here we present three other possible optimization criteria which can be simply inserted into the existing framework: the total weighted completion time (4), the uniform use of resource (5) and the net present value (6). In (5) $RF_i(t)$ stands for number of units of resource i used in time period t, and c_i^F in (6) is cash-flow for activity i. Test results for this criteria will also be given in this work. This optimization criterion variability contributes to the benefits of the genetic programming approach.

$$\frac{1}{n}\sum_{i=1}^{n} w_i C_i, \tag{4}$$

$$\begin{gathered} \frac{1}{r}\sum_{i=1}^{r}\sum_{t=1}^{T} \left(RF_i(t) - RF_{i,avg}\right)^2, \\ RF_{i,avg} = \frac{1}{N}\sum_{t=1}^{T} RF_i(t), i = 1, \ldots, r. \end{gathered} \tag{5}$$

$$\frac{1}{n}\sum_{i=1}^{n} c_i^F e^{-\alpha C_i}. \tag{6}$$

5. Experimental Results

5.1. Initialization

As in other EC techniques, the GP working environment is also coupled to the necessity of finding good parameters for the underlying algorithms and domain-specific evaluator where every decision should be supported by an extensive round of experiments.

Initial Parameters. Before any experiments can be conducted it is necessary to define an initial set of parameters to start with. In terms of GP, a set of used terminals and functions must be decided upon and the basic algorithmic parameters must be set. Initially, all available functions are used for the function set, while a subset of activity-specific and project-specific terminals are used to form the initial terminal set (see table 5).

Table 5: Initial function and terminal set.

Functions	+, -, *, /, MAX, POS, NEG, IF
Terminals	GRPW*, DSC, RS, RR, ARU, TPC, NSP

Additionally, the basic evolution parameters have to be set. Here a *Steady State Tournament* algorithm is used with the parameters given in table 6. In this algorithm, three individuals are selected randomly in each iteration and the worse one is eliminated. The remaining two are used as parents to produce a new individual to replace the eliminated one. The new individual is then mutated with given mutation rate. This configuration is used for the first round of experiments in dataset selection and convergence analysis.

Table 6: Initial EC parameters.

Tournament size	3
Tree max depth	7
Mutation probability	0.3
Population size	500
SGS type	PSGS (inserted idleness)

Convergence Analysis. The first step in the GP experimental flow is to determine the appropriate number of evaluations needed for GP to converge. This is necessary to set the maximal number of evaluations to be used in all following experiments. Ideally, this number should be as low as possible to save time and resources. Convergence analysis is done by running a set of N experiments for which the terminal condition is set to be an excessive number of evaluations. After plotting the generated values, one can easily determine the appropriate number of evaluations to be used in further experiments. In this work for the selected learning set of 56 instances two rounds of experiments have been conducted, one for each type of parallel SGS (with and without inserted idleness).

Each round consists of 30 separate runs with the terminal evaluation number set to 10^6.

The convergence results for both algorithms, showing minimum, maximum, average and median population value, are given in figures 1 and 2 using 100 time points.

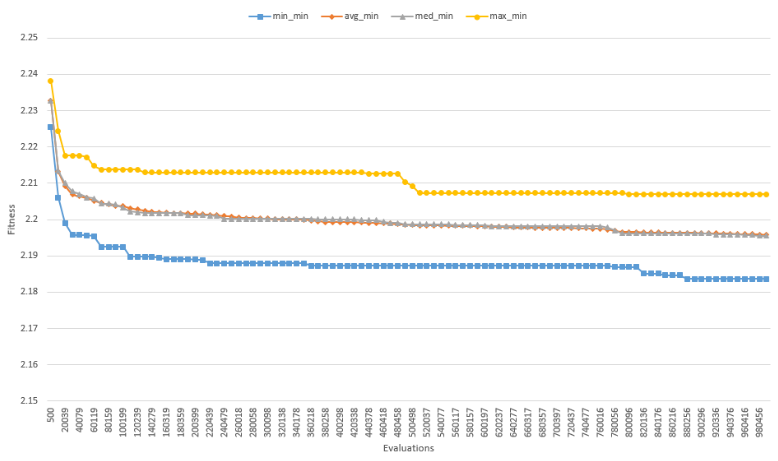

Figure 1: Convergence for the parallel SGS with inserted idleness.

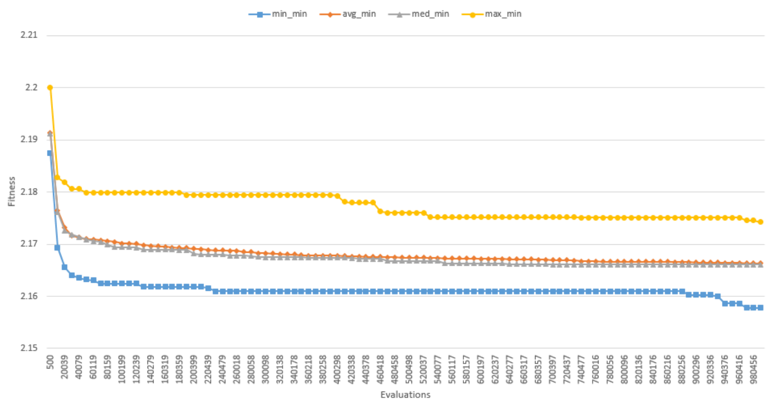

Figure 2: Convergence for the parallel SGS with no inserted idleness.

As can be seen in the figures, the greatest fitness value change happens in the first $2 \cdot 10^5$ to $3 \cdot 10^5$ evaluations. Therefore, the maximum evaluations termination number is set to $2.5 \cdot 10^5$ evaluations. This value is further used in the process of feature selection.

Feature Analysis. The goal of this procedure is to filter functions and terminals which are most important and beneficial for the evolution and to eliminate those which do not present any useful information source. The selection of terminals starts with an empty set of terminals (disregarding the initial set that was used for parameter tuning). The process then iteratively chooses from the unused terminals and puts the selected one in the set of terminals. Afterwards, a complete evolution is executed using the current terminal set. If the achieved fitness is greater after adding the selected operator, the operator is left in the set, otherwise it is removed. The procedure is repeated as long as there is an operator that can improve fitness. After this process, terminals TNA, RRT, DPC, DSC, TPC, TSC, SPC, SSC, GRPW*, EF and NSP have been selected. Notice that the majority of the selected terminals are static activity-specific terminals, while the only dynamic terminal is NSP.

Parameter Optimization. The final set of selected parameters for GP is shown in table 7.

Table 7: Final selected algorithm parameters.

Population size	1000
Mutation probability	0.3
Tree max depth	7

5.2. Test Results - Makespan

After initialization each algorithm was run 50 times. The result of each run is one priority rule. One example of a priority rule is given in Figure 3.

$$\left(\mathrm{RRT} + POS\left(MAX\left(\frac{\mathrm{EF}}{\mathrm{SSC}}\right) - \frac{\mathrm{NSP}}{\mathrm{SPC}}\right) - \left(NEG\left(\mathrm{SSC}\right) + \frac{MAX\left(\frac{\mathrm{EF}}{\mathrm{TSC}}\right)}{POS\left(\mathrm{DPC}\right)\cdot\mathrm{TSC}\cdot\mathrm{DPC}}\right)\right)\cdot\mathrm{TSC}$$

Figure 3: Example of priority rule evolved with GP

The best evolved rules on the learning set are then used to generate schedules for the unseen test set of problem instances, on which the final criteria are recorded. The results achieved by all three versions of SGS can be seen in Table 8 and in the box plot representation given in Figure 4. From the presented results it can be seen that PSGS1 achieves better results than the other two algorithms.

5.3. Comparison With Existing Heuristics

In order to examine the quality of the generated solutions, it is necessary to compare them to the best known heuristics which in this case are human-made priority rules. The reason for comparing GP solutions only with priority rules and not with other meta heuristics is that GP is developing a priority rule which is afterwards used in dynamic conditions, while other meta heuristics deal

Table 8: Fitness value achieved by GP.

Algorithm	**Min**	**Avg**	**Max**	**StDev**
PSGS	2.08186	2.092656	2.11355	0.005545
PSGS1	2.05499	2.063861	2.08976	0.007034
SSGS	2.07644	2.083652	2.10301	0.005432

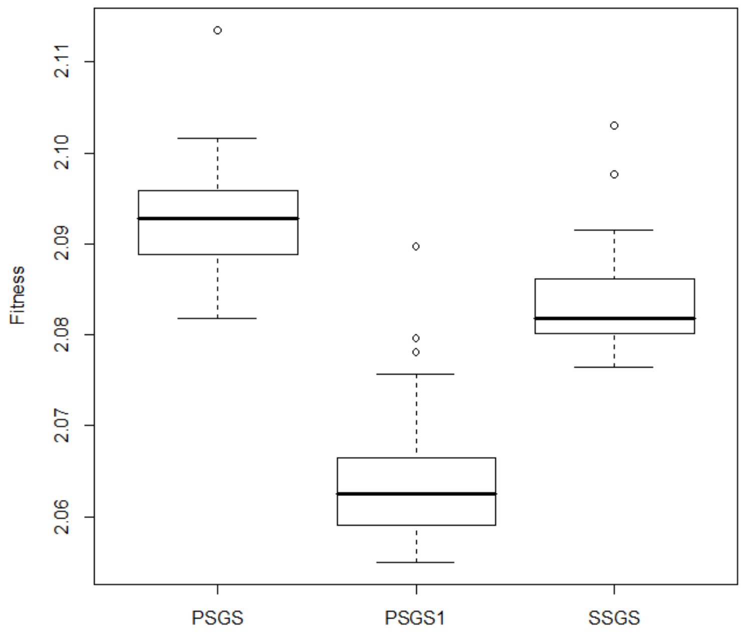

Figure 4: Box plotes for results achieved by GP.

only with static conditions. For this reason, a few well known scheduling rules are selected and applied to the aforementioned test set. The description of all benchmark heuristics can be found in Table 11.

For easier comparison, all results are presented in two tables, where the results in Table 9 show heuristic and GP achievements for the learning set, while the Table 10 presents achievements for the test set. Here the best available GP solution for each SGS version on the test set is used to calculate the given values.

Table 9: Heuristic and GP results for the learning set.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	2.2394	2.2098	2.2184
LST	2.2403	2.2043	2.2212
LFT	2.2324	2.1920	2.2236
GRPW	2.3780	2.3632	2.4339
SPT	2.4764	2.3720	2.5958
MSL	2.3680	2.3297	2.3664
MIS	2.3050	2.2560	2.3505
MTS	2.2382	2.2086	2.2483
GP	**2.1947**	**2.1465**	**2.1818**

Table 10: Heuristic and GP results for the test set.

Heuristics	**PSGS**		**PSGS1**		**SSGS**	
	Fitness value	Domination	Fitness value	Domination	Fitness value	Domination
GRPW*	2.0968*	54.13%	2.0711*	56.20%	2.0915*	59.88%
LST	2.0867	19.46%	2.0634	21.57%	2.0803	18.65%
LFT	2.0927	19.61%	2.0635	21.32%	2.0947*	18.75%
GRPW	2.2085*	29.39%	2.1854*	31.70%	2.2570*	26.76%
SPT	2.2879*	26.31%	2.2209*	27.62%	2.4083*	24.40%
MSL	2.1694*	19.71%	2.1387*	21.77%	2.2035*	18.65%
MIS	2.1752*	30.14%	2.1399*	33.87%	2.2116*	28.23%
MTS	2.1144*	43.95%	2.0815*	49.45%	2.1154*	46.82%
GP	**2.0819**	**67.24%**	**2.055**	**70.06%**	**2.0764**	**76.51%**

The comparison clearly shows that the best solutions evolved by GP achieve better results for all SGS versions than priority rules from literature. For those priority rules that are statistically worse (p-value less than 0.01) than priority rules evolved by GP a * sign is put next to the fitness value. The rules obtained by all versions of SGS have significantly better results than rules GRPW, GRWP*, SPT, MLS, MIS and MTS, while for SSGS GP result is also significantly better than the result obtained by LFT. Table 10 introduces a column named *Domination* which shows the percentage of problem instances in which a given method obtained the best result (which more than one heuristic may obtain).

Table 11: Priority rules definition.

Priority rule	Description	Sort type	Priority value
GRPW*	Greatest rank positional weight all	max	$d_j + \sum_{i \in F_j^*} d_i$
LST	Latest starting time	min	LS_j
LFT	Latest finish time	min	LF_j
GRPW	Greatest rank positional weight	max	$d_j + \sum_{i \in F_j} d_i$
SPT	Shortest processing time	min	d_j
MSL	Minimum slack time	min	$LS_j - ES_j$
MIS	Most immediate successors	max	$\lvert F_j \rvert$
MTS	Most total successors	max	$\lvert F_j^* \rvert$

Additionally it is interesting to see the difference between the achieved results for all project instances in the test set and the best known results in terms of makespans. This comparison is given in figure 5 where the results are achieved by the parallel SGS with no inserted idleness as this SGS proved to generate best results.

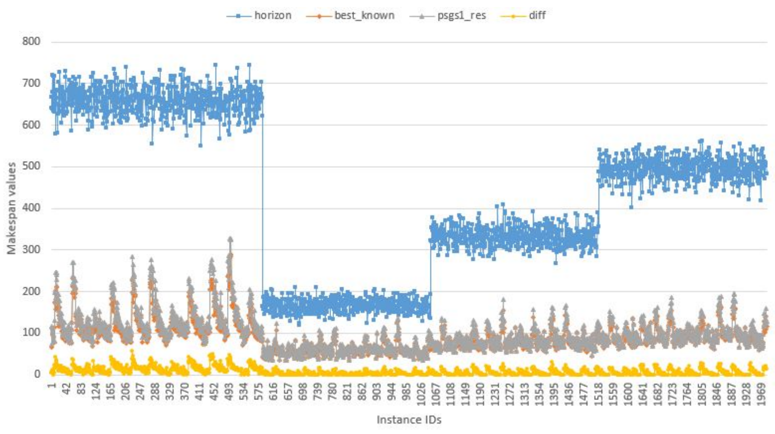

Figure 5: Best known and achieved results differences (PSGS1).

In the given figure, the blue line marked with squares represents the projects' horizon values, the orange line marked with rotated squares represents best-known solutions, the grey line marked with triangles represents the results achieved with the GP solution and finally, the yellow line marked with circles represents the absolute difference between best-known and achieved results.

One can notice that the overall difference between best known and achieved results differentiates only in small amounts, proving that the GP solutions give high-quality results for any instance in the test set. Also it is worth mentioning that the gap between achieved makespans and the maximum total project durations (horizons) is relatively large for all project instances without exception.

This indicates that genetic programming is capable of evolving solutions which will tend to generate high-quality schedules for a variety of different project instances.

5.4. Comparison with GA

Although it is already mentioned that it is only fair to compare GP with existing human-made priority rules (which is given in previous subsection) for the sake of completeness we also give a brief comparison of our results and the results obtained by using genetic algorithm (GA). For the GA, tests were conducted on test set comprising of 20 randomly chosen problems from PSPLIB problem set. Test instances had 30, 60, 90 and 120 activities. Table 12 shows optimal makespan, makespan obtained with GA (the best found solution over all runs) and makespans obtained by using all three versions of SGS for every test case. The last row of the table shows fitness value of the entire subset of problems that was selected for comparison. Genetic algorithm implementation is the same as in [39]. We used permutation representation and as a stopping criterion we used maximum number of generations which was set to 80.

The table shows that GA almost always finds optimal solution, while GP fails to do that in more instances. But, if we want to use GA for solving some problem, we have to know all the activities and their properties in advance and with regard to those properties try to find optimal solutions for every problem separately. We need to select representation for an individual, make initial population and run several generations in order to get the solution, which is much more time consuming than simply applying priority rule on a given problem. Furthermore, the GA cannot be easily applied in dynamic conditions where one may need to take into account variable activity properties while the project is being executed. It can also be noticed that solutions obtained by using GP do not deviate too much from optimal solutions and so they are good enough for application in practice when it is important to get a good enough solution very quickly.

Table 12: Comparison of makespan achieved by GA and GP

InstanceNo	**OptimalSolution**	**GA solution**	**SSGS**	**PSGS**	**PSGS1**
1	39	39	42	41	41
2	54	54	54	54	54
3	42	42	49	43	43
4	36	36	36	36	36
5	46	46	49	50	47
6	69	69	70	75	73
7	71	71	71	72	72
8	103	**105**	118	119	114
9	77	77	77	77	77
10	71	**72**	80	76	80
11	92	92	92	92	92
12	111	**116**	131	133	121
13	85	85	85	85	85
14	126	126	126	126	126
15	160	**165**	189	186	179
16	95	96	111	107	104
17	72	72	72	77	75
18	102	**105**	116	116	113
19	104	104	104	107	109
20	101	**103**	107	109	106
Fitness	1.80107	1.82259	1.95761	1.94865	1.91704

5.5. Test Results for Other Criteria

The optimization criterion variability allows us to easily apply the GP for different optimization criteria. The tests for the additional three optimization criteria were performed: the total weighted completion time (4), the uniform use of resource (5) and the net present value (6). The terminal AW (activity weight) was added into the terminal set since weight information is relevant in the first criteria. This terminal is used for the cash-flow of an activity in the net present value. Results for each of these criteria are presented in two tables, where the results in the first table show heuristic and GP achievements for the learning set, while the second table presents achievements for the test set. Same as for makespan, the best available GP solution for each SGS version is used to calculate the given values. Table 13 and table 14 show results for the total weighted completion time criteria. Results for the uniform use of resource are shown in table 15 and table 16 and results for the net present value are in table 17 and table 18. The comparison clearly shows that the best solutions evolved by GP for this criteria also achieve better results than most of the existing priority values for all SGS versions. The total weighted completion time criterion results for the test set show that the best solutions found by GP are better than solutions found by existing priority functions for all SGS versions, while only in serial SGS version GRPW* and MTS priority rules for the uniform use of resource criterion, and LST, LFT and MSL for the net present value criterion achieve slightly better results.

Table 13: Heuristic and GP results for the learning set for the total weighted completion time criteria.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	24.516	24.045	24.579
LST	27.736	26.746	29.647
LFT	27.605	26.850	29.441
GRPW	25.869	25.502	26.448
SPT	26.598	26.290	26.692
MSL	27.733	26.834	29.644
MIS	24.587	24.013	25.205
MTS	**24.231**	23.756	24.453
GP	24.275	**23.022**	**24.186**

Table 14: Heuristic and GP results for the test set for the total weighted completion time criteria.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	23.134	22.679	23.390
LST	25.469	24.839	27.393
LFT	25.509	24.902	27.312
GRPW	24.237	23.850	24.784
SPT	24.746	24.483	24.844
MSL	25.375	24.707	27.365
MIS	23.401	22.949	23.874
MTS	22.985	22.526	23.225
GP	**22.926**	**22.281**	**23.205**

Table 15: Heuristic and GP results for the learning set for the uniform use of resource criteria.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	36076.43	35292.02	36001.97
LST	41096.56	39552.54	44254.19
LFT	40900.45	39553.89	43879.78
GRPW	38038.51	37490.00	38920.54
SPT	39519.25	38639.39	39203.33
MSL	41000.07	39572.69	44233.48
MIS	35858.62	35122.37	36759.06
MTS	**35547.60**	34825.03	**35883.51**
GP	35595.93	**33452.76**	36399.44

Table 16: Heuristic and GP results for the test set for the uniform use of resource criteria.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	36996.12	36175.40	37462.71
LST	41327.42	40194.47	44929.59
LFT	41414.87	40265.41	44750.33
GRPW	39109.88	38424.47	40075.94
SPT	40113.85	39587.27	40141.50
MSL	41149.22	39921.09	44875.22
MIS	37393.15	36609.35	38274.28
MTS	36711.46	35872.61	**37136.56**
GP	**36575.43**	**35309.24**	37524.581

Table 17: Heuristic and GP results for the learning set for the net present value criteria.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	64.839	65.266	64.755
LST	62.079	62.896	60.346
LFT	62.078	62.800	60.579
GRPW	63.456	63.916	62.921
SPT	62.811	63.119	62.710
MSL	62.070	62.831	**60.331**
MIS	64.760	65.332	64.211
MTS	65.119	65.547	64.955
GP	**61.675**	**61.941**	61.523

Table 18: Heuristic and GP results for the test set for the net present value criteria.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	66.410	66.805	66.172
LST	64.584	65.096	62.939
LFT	64.529	65.016	62.999
GRPW	65.414	65.763	64.907
SPT	64.940	65.189	64.852
MSL	64.639	65.176	**62.955**
MIS	66.216	66.592	65.776
MTS	66.545	66.939	66.329
GP	**64.472**	**64.842**	63.842

6. Conclusions and Further Work

This paper demonstrates the application of genetic programming as a hyper-heuristic to generate suitable scheduling heuristics for the resource constrained scheduling problem. The results are promising, as the GP based priority rules are able to provide the best or second to best results among all the included human-made scheduling heuristics. This is especially evident in cases where custom optimization criteria is used, in which case the existing heuristics exhibit varying efficiency. On the other hand, the presented approach obtains heuristics that perform consistently among the best for different scheduling requirements.

As future research it is planned to use GP in solving the multi-mode RCPSP. In this work we consider only renewable resources; however, GP can also be applied to devise heuristics for other types of resources. Additionally, ensemble methods will be used in order to try to improve the quality of the obtained solution.

References

[1] D. Wolpert, W. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 4 (1997) 6782.

[2] R. Słowinski, Multiobjective network scheduling with efficient use of renewable and nonrenewable resources, European Journal of Operational Research 7 (3) (1981) 265–273.

[3] R. Klein, Scheduling of resource-constrained projects, Springer-Science+ Business Media LLC, New York, 2000.

[4] K. S. Hindi, H. Yang, K. Fleszar, An evolutionary algorithm for resource-constrained project scheduling, Evolutionary Computation, IEEE Transactions on 6 (5) (2002) 512–518.

[5] R. Kolisch, A. Sprecher, A. Drexl, Characterization and generation of a general class of resource-constrained project scheduling problems, Management science 41 (10) (1995) 1693–1703.

[6] J. Blazewicz, J. K. Lenstra, A. R. Kan, Scheduling subject to resource constraints: classification and complexity, Discrete Applied Mathematics 5 (1) (1983) 11–24.

[7] M. Abdolshah, A review of resource-constrained project scheduling problems (rcpsp) approaches and solutions, International Transaction Journal of Engineering, Management, Applied Sciences and Technologies.

[8] P. Brucker, A. Schoo, O. Thiele, A branch and bound algorithm for the resource constrained project scheduling problem, European Journal of Operation Research 17 (1998) 143–158.

[9] R. F. Deckro, E. P. Winkofsky, J. E. Hebert, R. Gagnon, A decomposition approach to multi-project scheduling, European Journal of Operation Research 51 (1991) 110–118.

[10] O. Icmeli, W. O. Rom, Solving the resource-constrained project scheduling problem with optimization subroutine library, Computers and Operation Research 23 (1996) 801–817.

[11] A. A. B. Pritsker, L. Watters, P. Wolfe, Multiproject scheduling with limited resources: A zero-one programming approach, Management Science 16 (1969) 93–108.

[12] J. Patterson, F. Talbot, R. Slowinski, J. Weglarz, Computational experience with a backtracking algorithm for solving a general class of precedence and resource-constrained project scheduling problems, European Journal of Operational Research 49 (1990) 68–79.

[13] L. Schrage, Solving resource-constrained network problems by implicit enumeration - nonpreemptive case, Operations Research 18 (1970) 263–278.

[14] E. Demeulemeester, W. Herroelen, A branch-and-bound procedure for the multiple resource-constrained project scheduling problem, Management Science 38 (1992) 1083–1818.

[15] J. P. Stinson, E. W. Davis, B. M. Khumawala, Multiple resource-constrained scheduling using branch and bound, AIIE Transactions 10 (1978) 252–259.

[16] S. Elmaghraby, Resource allocation via dynamic programming in activity networks, European Journal of Operational Research 64 (1993) 199–215.

[17] R. Klein, Bidirectional planning: improving priority rule-based heuristics for scheduling resource-constrained projects, European Journal of Operational Research 127 (3) (2000) 619–638.

[18] R. Kolisch, Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation, European Journal of Operational Research 90 (2) (1996) 320–333.

[19] S. Kadam, N. Kadam, Solving resource-constrained project scheduling problem by genetic algorithms, Business and Information Management (ICBIM) (2014) 159–164.

[20] H. Ouerfelli, A. Dammak, The genetic algorithm with two point crossover to solve the resource-constrained project scheduling problems, Modeling, Simulation and Applied Optimization (ICMSAO) (2013) 1–4.

[21] V. Valls, F. Ballestn, Population-based approach to the resource-constrained project scheduling problem, Annals of Operations Research 131 (2004) 305–324.

[22] D. Merkle, M. Middendorf, H. Schmeck., Ant colony optimization for resource-constrained project scheduling, IEEE Transactions on Evolutionary Computation 6 (2002) 333346.

[23] T. P. Adams, Creation of simple, deadline, and priority scheduling algorithms using genetic programming, Genetic Algorithms and Genetic Programming at Stanford 2002 (2002).

[24] C. Dimopoulos, A. Zalzala, Investigating the use of genetic programming for a classic one-machine scheduling problem, Advances in Engineering Software 32 (6) (2001) 489 – 498. doi:https://doi.org/10.1016/S0965-9978(00)00109-5.
URL http://www.sciencedirect.com/science/article/pii/S0965997800001095

[25] D. Jakobović, L. Budin, Dynamic Scheduling with Genetic Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 73–84. doi:10.1007/11729976_7.
URL https://doi.org/10.1007/11729976_7

[26] D. Jakobovic, K. Marasovic, Evolving priority scheduling heuristics with genetic programming, Applied Soft Computing 12 (9) (2012) 2781 – 2789. doi:https://doi.org/10.1016/j.asoc.2012.03.065.
URL http://www.sciencedirect.com/science/article/pii/S1568494612001780

[27] S. Nguyen, M. Zhang, M. Johnston, K. C. Tan, A computational study of representations in genetic programming to evolve dispatching rules for the job shop scheduling problem, IEEE Transactions on Evolutionary Computation 17 (5) (2013) 621–639. doi:10.1109/TEVC.2012.2227326.

[28] R. Hunt, M. Johnston, M. Zhang, Evolving "less-myopic" scheduling rules for dynamic job shop scheduling with genetic programming, in: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO '14, ACM, New York, NY, USA, 2014, pp. 927–934. doi:10.1145/2576768.2598224.
URL http://doi.acm.org/10.1145/2576768.2598224

[29] M. Durasevic, D. Jakobovic, K. Knezevic, Adaptive scheduling on unrelated machines with genetic programming, Applied Soft Computing 48 (Supplement C) (2016) 419 – 430. doi:https://doi.org/10.1016/j.asoc.2016.07.025.
URL http://www.sciencedirect.com/science/article/pii/S1568494616303519

[30] J. Branke, S. Nguyen, C. W. Pickardt, M. Zhang, Automated design of production scheduling heuristics: A review, IEEE Transactions on Evolutionary Computation 20 (1) (2016) 110–124. doi:10.1109/TEVC.2015.2429314.

[31] J. Branke, T. Hildebrandt, B. Scholz-Reiter, Hyper-heuristic evolution of dispatching rules: A comparison of rule representations, Evolutionary Computation 23 (2) (2015) 249–277, pMID: 24885679. arXiv:https://doi.org/10.1162/EVCO_a_00131, doi:10.1162/EVCO_a_00131.
URL https://doi.org/10.1162/EVCO_a_00131

[32] J. R. Koza, Human-competitive results produced by genetic programming, Genetic Programming and Evolvable Machines 11 (3) (2010) 251–284. doi:10.1007/s10710-010-9112-3.
URL https://doi.org/10.1007/s10710-010-9112-3

[33] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, J. R. Woodward, Exploring Hyper-heuristic Methodologies with Genetic Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 177–201. doi:10.1007/978-3-642-01799-5_6.
URL https://doi.org/10.1007/978-3-642-01799-5_6

[34] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, J. R. Woodward, A Classification of Hyper-heuristic Approaches, Springer US, Boston, MA, 2010, pp. 449–468. doi:10.1007/978-1-4419-1665-5_15.
URL https://doi.org/10.1007/978-1-4419-1665-5_15

[35] T. Frankola, M. Golub, D. Jakobovic, Evolutionary algorithms for the resource constrained scheduling problem, in: 30th International Conference on Information Technology Interfaces, 2008.

[36] D. Jakobovic, et al., Evolutionary computation framework, http://ecf.zemris.fer.hr/ (Oct. 2015).

[37] R. Kolisch, Shifts, Types, and Generation Schemes for Project Schedules, Springer International Publishing, Cham, 2015, pp. 3–16. doi:10.1007/978-3-319-05443-8_1.
URL https://doi.org/10.1007/978-3-319-05443-8_1

[38] R. Kolisch, C. Schwindt, A. Sprecher, Benchmark instances for project scheduling problems, in: Project Scheduling, Springer, 1999, pp. 197–212.

[39] R. Coric, M. Dumic, D. Jakobovic, Complexity comparison of integer programming and genetic algorithms for resource constrained scheduling problems, in: 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, 2017, pp. 1394–1400.