Constraints (2018) 23:310–334
https://doi.org/10.1007/s10601-018-9287-4

Online over time processing of combinatorial problems

Robinson Duque[1] · Alejandro Arbelaez[2] ·
Juan F. Díaz[1]

Published online: 4 May 2018

Abstract In an online environment, jobs arrive over time and there is no information in advance about how many jobs are going to be processed and what their processing times are going to be. In this paper, we study the online scheduling of Boolean Satisfiability (SAT) and Mixed Integer Programming (MIP) instances that are well-known NP-complete problems. Typical online machine scheduling approaches assume that jobs are completed at some point in order to minimize functions related to completion time (e.g., makespan, minimum lateness, total weighted tardiness, etc). In this work, we formalize and present an online over time problem where arriving instances are subject to waiting time constraints. We propose computational approaches that combine the use of machine learning, MIP, and instance interruption heuristics. Unlike other approaches, we attempt to maximize the number of solved instances using single and multiple machine configurations. Our empirical evaluation with well-known SAT and MIP instances, suggest that our interruption heuristics can improve generic ordering policies to solve up to 21.6x and 12.2x more SAT and MIP instances. Additionally, our hybrid approach observed up to 90% of solved instances with respect to a semi clairvoyant policy (SCP).

This article belongs to the Topical Collection: *Topical Collection on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*
Guest Editor: Willem-Jan van Hoeve

✉ Robinson Duque
robinson.duque@correounivalle.edu.co

Alejandro Arbelaez
alejandro.arbelaez@cit.ie

Juan F. Díaz
juanfco.diaz@correounivalle.edu.co

[1] Universidad del Valle, Cali, Colombia

[2] Cork Institute of Technology, Cork, Ireland

Keywords Online scheduling · Combinatorial problems · Machine learning · Regression models · Classification models · Runtime estimation · Mixed integer programming

1 Introduction

In the last decade, the interest for using cloud resources to solve large combinatorial problems has been growing and has attracted much attention from the Boolean Satisfiability (SAT) community for solving real-life problems with hundreds of thousands of variables and millions of clauses and constraints. Indeed, SAT solvers are nowadays used in multiple domains ranging from software verification [21] to computational biology [19] and automated planning [15]. Mixed Integer Programming (MIP) has also been used to model a wide range of applications and it is now an indispensable tool in business and engineering [29]. In this context, recently [11] used SAT technology to compute *the largest-ever mathematical proof*, a 200-terabyte proof called the Boolean Pythagorean triples problem. To solve this problem the authors required about 30,000 h of CPU time to solve more than a million SAT subproblems.

On-demand services are an increasingly popular pricing model in which users are charged on usage bases without long-term commitment. Users who rent cloud processing resources have to deal with the task of using the rented resources efficiently. For instance, when renting some computational resources for t amount of time to process some jobs (or problem instances), users might have to face situations where an efficient usage of the resources implies to maximize the number of solved jobs. Thus, they have to find a good balance between the number of solved instances and the rented processing time. Indeed, this is usually a non trivial task, specially when there is a lack of knowledge about the jobs that will be released in the future or when information about the jobs being processed is either incomplete, uncertain, or absent as in the case of many online scheduling problems.

In this paper, we use the terms *instance* (or combinatorial problem instance) and *job* as synonyms. Additionally, the term *online* refers to a kind of problem where instances arrive over time. Due to the runtime variability, we present information about maximum waiting times instead of due dates (see Section 3). Additionally, we assume that the actual runtime required to process or solve an instance is unknown in advance. Therefore, we use machine learning to estimate processing times and take this information into account to make decisions in our scheduling policies.

This paper is structured as follows. Section 2 introduces the reader to online machine scheduling and overviews the importance of supervised machine learning to estimate instances runtime for SAT and MIP instances. In Section 3, we propose and introduce a notation to represent our online over time scheduling problem. In Section 4 we present our computational approach for online processing of combinatorial problems under single and multiple machine configurations. In Section 5 we empirically evaluate our models and present results. Finally, Section 6 describes related work and Section 7 presents our conclusions.

2 Preliminaries

2.1 Online machine scheduling

Machine scheduling usually refers to an allocation of jobs within a set of machines given a series of constraints in order to optimize a specific criterion. In offline scheduling, every job j and its respective data are known in advanced (e.g., processing times (p_j), weights (w_j), arrival or release dates (r_j), and due dates (d_j)) [17]. In contrast, in online scheduling problems, the number of jobs to be processed is unknown and no information is given about future jobs [1, 20].

Online machine scheduling has been extensively studied in the last decade from two perspectives. Namely, in a *clairvoyant online over time* problem, a job is released (i.e., arrives to the system) and all the relevant data of the job is presented (i.e., p_j, w_j, d_j). However, in a *non-clairvoyant online over time* problem, the processing time p_j remains unknown until the job is completed [20].

There are many different scheduling configurations that have been considered in the literature, therefore, it is convenient to use the standard $\alpha|\beta|\gamma$ classification scheme as proposed in [10] by Graham et al.:

– α describes the machine environment and contains a single entry. Some values might be: (1) single machine, (Pm) identical machines in parallel, (Qm) machines in parallel with different speeds, (Rm) unrelated machines in parallel, (Fm) flow shop, (Jm) job shop, etc.
– β provides details of the processing constraints and may contain one, multiple or no entries at all. For instance, (p_{mj}) processing times of job j on machine m, (p_j) processing times of job j if the machines are identical, (r_j) release dates, (d_j) due dates, (w_j) weights, ($prmp$) preemption, etc.
– γ describes the objective function and often contains a single entry. The objective function is typically a function related to the completion time of the jobs (C_j). For instance, the lateness ($L_j = C_j - d_j$), the tardiness ($T_j = max(L_j, 0)$), the unit penalty of tardy jobs U_j ($U_j = 1$ if $C_j > d_j$, otherwise $U_j = 0$). Some examples of objective functions to minimize are: makespan or completion time of the last job (C_{max}); maximum lateness(L_{max}) or worst violation of the due dates $max(L_1, \ldots, L_n)$; total weighted tardiness ($\sum w_j T_j$), weighted number of tardy jobs ($\sum w_j U_j$), etc.

To illustrate the notation, $1|p_j, r_j, prmp|\sum C_{max}$ denotes an offline single machine scheduling problem where jobs' release dates (r_j) and processing times (p_j) are given in advance. Additionally, preemption is allowed ($prmp$) and the objective is to minimize the makespan C_{max}. Many generalizations have been suggested for scheduling models that cannot be represented with the original three-field notation. For instance, the *online* term is typically added to denote online problems. Thus, $Pm|online|\sum w_j T_j$ denotes an online problem with identical parallel machines where job arrivals and processing times are unknown. Additionally, weights are presented at each job arrival and the objective is to minimize the total weighted tardiness.

2.2 SAT and MIP problems

Determining whether a given propositional logic formula is satisfiable or not, is one of the fundamental problems in computer science. It was the first NP-Complete problem proven by Cook in 1971 and it is also known as the canonical NP-complete Boolean satisfiability (SAT) problem [14].

A Boolean formula is represented by a pair (X, P), where X is a set of Boolean variables and P is a set of clauses in Conjunctive Normal Form (CNF). The *Boolean variables* $\{x_1, \ldots, x_n\} \in X$ can be assigned truth values (true, false). A *literal* is either a variable x_i or its negation $\neg x_i$. *A clause* ω is a disjunction of literals and a CNF formula δ is a conjunction of clauses. Consequently, a CNF formula is said to be *satisfied* if all its clauses are satisfied (i.e., at least one of its literals assumes the value of 1). Respectively, it is *unsatisfied* if at least one clause is unsatisfied (i.e., all of its literals assume the value of 0).

In the literature, the satisfiability problem is typically tackled by creating *stand-alone SAT-solvers* that run sequentially or in parallel. However, state-of-the-art solvers typically display extreme runtime variations across instances and there is little theoretical understanding of this behavior [13]. On the other hand, throughout more than 50 years of existence, Mixed-integer programming (MIP) has provided a mechanism for optimizing decisions including those encountered in biology and medicine. MIP problems are typically modeled and implemented in different solvers (e.g., CPLEX, Gurobi, LPsolve, etc.) and performance variations can be achieved from one solver to another [29].

2.3 Supervised machine learning for runtime prediction

Supervised machine learning consists of inferring a model (or hypothesis) $f : \Omega \rightarrow Y$ that predicts the value of an output variable $y_i \in Y$, given the values of the example description (e.g., a vector of features, $\Omega = R^d$). The output can be numerical (i.e., regression) or categorical (i.e., classification). The idea of predicting algorithm runtime is not new and has been studied in the past using methods such as: Ridge regression [18], neural networks [23], regression trees [3], etc.

Recently in [13], the authors use machine learning techniques to build models of the algorithm's runtime as a function of SAT, MIP, and traveling salesperson (TSP) problems. Interestingly, random forest was the best performing model across all the datasets and among the presented methods. Mainly due to the heterogeneous datasets and also because tree-based approaches can model different parts of the data separately, in contrast to other methods. For instance, regression trees are typically sensitive to small changes in the data and tend to overfit. However, random forests overcome this problem by combining multiple regression trees into an ensemble. That is, a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest [4].

In this study we train regression models to estimate algorithm runtime and classification models to determine whether a given instance can be solved within an execution time limit t. This process involves collecting a vector of features $x_i \in X$ and the runtime $y_i \in Y$, in order to train the models. The vector of features used for this study was extracted from the literature in machine learning for SAT and MIP runtime prediction. In [13], the authors provide a complete set of 138 and 121 features to describe SAT and MIP problems as well as the runtimes. Interestingly, the authors divide the features into four categories according the overall complexity to collect the descriptors: trivial, cheap, moderate, and expensive. These feature categories play an important role in this study since they allow us to explore different complexity-based approaches.

3 Online over time scheduling notation and problems

Cloud computing offers an interesting opportunity to solve combinatorial problems in different domains. Computational time can be rented by the hour and for a given number of processors. However, as for NP-Complete problems, many instances cannot be solved within reasonable time and typically have a very large runtime variability. Thus, attempting to solve a single job might consume all the rented computational time. To mitigate such situation, in [9] the authors assume scenarios where jobs can be interrupted from running if an execution time limit t is met or if an *inaccurate* scheduling decision is made.

As presented in Section 2, typical online machine scheduling approaches assume that every job j is completed at some point and usually minimize some completion time criterion (e.g., makespan, maximum lateness, worst violation of due dates, total weighted tardiness, etc). However, our approach can not use such functions since instances can be interrupted from running and remain unsolved at the end of the schedule. For this reason, we introduce some constraints and an objective function that maximizes the number of solved instances (or processed jobs). Hereafter we propose a generalization of the three-field notation. This generalization lets us represent online machine scheduling problems, including the two problems considered in this work:

$$1|online, \overrightarrow{wt_j}, f, \overset{\approx}{p_j}, interrupt|\sum S_j$$
$$Pm|online, \overrightarrow{wt_j}, f, \overset{\approx}{p_j}, interrupt|\sum S_j \tag{1}$$

We recall that *clairvoyant approaches* present all the relevant data of a job once it is released. In contrast, *non-clairvoyant* approaches might reveal some relevant data when jobs are released or remain unknown until job completion (see Section 2.1). We propose to use a top arrow to denote that certain information is presented on the arrival of a job (e.g., $\overrightarrow{p_j}$ if the processing time is presented at the arrival of the job in a clairvoyant approach). We also propose to use a top bullet to denote that certain information remains unknown until the job is completed (e.g., $\overset{\bullet}{p_j}$ in a non-clairvoyant approach). Additionally, if a value is somehow estimated at the arrival of a job, we propose to denote it using an approximation symbol (e.g., $\overset{\approx}{p_j}$ for a semi-clairvoyant approach).

In this work we propose to train regression and classification models in order to use them in our scheduling approach. In general, a regression model can be described as a function $f : R \rightarrow \overset{\approx}{p_j}$ that estimates the processing time (i.e., $\overset{\approx}{p_j}$) of an algorithm A on an instance j. Additionally, a classification model can be described as a function $f : C \rightarrow \overset{\approx}{s_j}$ that determines whether a job j is solvable or not (i.e., $\overset{\approx}{s_j}$) by algorithm A within time t. In Sections 4.1 and 5, we further explain how we train/test such models with individual algorithms. The notation from the scheduling problems in (1) is then described as follows:

- 1 and Pm: denote single and parallel identical machine environments.
- *online*: denotes that jobs arrive over time and we do not have any information before a job is released.
- $\overrightarrow{wt_j}$: denotes the *maximum waiting time* that an instance can remain in the system waiting to be attended.
- $f \in \{R, C\}$ denotes a model that can be described as a function where $f : R \rightarrow \overset{\approx}{p_j}$ represents a regression model and $f : C \rightarrow \overset{\approx}{s_j}$ represents a classification model.
- *interrupt*: interruptions reduce the impact of wrong decisions or inaccurate processing time estimations. In our approach, *interrupt* denotes that a running job j may be interrupted, then it loses all the work done on it and become available to be rescheduled (i.e., unattended).
- Our objective function consists in maximizing the number of solved jobs S_j. A job j is considered to be **attended**, if its waiting time in the system is less than $\overrightarrow{wt_j}$ and it is

processed for some time greater than 0. Additionally, a job j is considered to be **solved**, if it is attended and the solver finds an answer in the assigned processing time. In (2), ST_j represents the time when job j starts being processed (resp. ET_j represents the end time). WT_j denotes the time that a job waits in the system (i.e., $WT_j = ST_j - \overrightarrow{r_j}$):

$$S_j = \begin{cases} 1, & \text{if } WT_j \leq \overrightarrow{wt_j} \wedge solve(j, ST_j, ET_j) \\ 0, & otherwise \end{cases} \tag{2}$$

In this work, we consider scenarios where waiting time constraints $\overrightarrow{wt_j}$ must be met and all the problem instances have the same weight. Notice that our problem definitions in (1), did not include $\overrightarrow{p_j}$ since processing time of plenty of computational problems, including NP-Complete problems, cannot be perfectly estimated in practice (neither $\overset{\bullet}{p}_j$ since jobs can remain unsolved at the end of the schedule, thus their processing times might continue to be unknown).

Typical online approaches use due dates $\overrightarrow{d_j}$ instead of waiting time constraints $\overrightarrow{wt_j}$. However, in practice, due dates might be underestimated for NP-Complete problems and can lead to situations where jobs will not have a chance to be solved under any circumstances, even if they start running at the moment they are released (e.g., cases when $\overrightarrow{r_j} + \overset{\bullet}{p}_j > \overrightarrow{d_j}$). For instance, suppose we have a machine that begins to process a job at time 1. Now suppose that the actual runtime of such job is 10s (this information is not available until we actually run, complete, and solve the job). Assume that we underestimate the due date of such job and we expect it to finish at time 5. Then, that job would be attended but will end up unsolved since it will not have a chance to satisfy the due date constraint.

Based on the idea that every job should have a chance to be solved, the usage of $\overrightarrow{wt_j}$ does not prevent instances from being solved since the smallest waiting time possible is 0, thus an instance has a chance to fulfill this constraint if it starts being attended when released. Finally, for the scheduling problems in (1), we will represent data related to an arriving job j with the tuple:

$$j = (\overrightarrow{r_j}, \overset{\approx}{p}_j, \overrightarrow{wt_j}) \tag{3}$$

4 Online computational approach to process combinatorial problems

In this approach we differentiate between three processes: a training/testing phase to create regression and classification models to estimate runtime or determine if a job can be solved within an execution time limit; scheduling policies to tackle the online problems presented in the previous section; instance interruption heuristics to mitigate inaccurate scheduling decisions based on missed predictions.

4.1 Training/Testing phase

Machine learning can be used to estimate algorithms runtime using features extracted from the instance to solve. To train regression models ($f : R \rightarrow \overset{\approx}{p}_j$) that estimate the runtime of algorithm A on an instance j, we used the vector of features and runtime data for SAT and MIP problems introduced in [13]. In general, a problem instance j can be described by a list of m features $d_j = [z_1, \ldots, z_m]$ that can be computable by a piece of problem-specific code. To obtain the necessary training data, algorithm A can be run on a set of n

instances for a time limit t, in order to record their features d_j as well as their processing times p_j. The training data of the n instances is then compound by each vector of features and processing time $\{(d_1, p_1), \ldots, (d_j, p_j)\}$.

We also used the training data to create classification models ($f : C \rightarrow \overset{\approx}{s}_j$). We replaced processing times with a Boolean variable $s_j = (p_i < t)$, thus timeouts can be represented with 0's and solved instances with 1's (i.e., false and true respectively). Then, we use a dataset of the form $\{(d_1, s_1), \ldots, (d_j, s_j)\}$ to train a classifier to estimate whether an algorithm A is able to solve an instance j within time t. We recall that in the experiments section we limited our approach to the usage of a single algorithm A per dataset (i.e., MiniSAT for SAT instances and CPLEX for MIP instances).

Since we want to avoid situations in which calculating the features adds a considerably big overhead to the schedule, we take advantage of the *trivial and cheap* features from [13] to Train-Test our regression/classification models. Additionally, getting training/testing data for runtime estimation of combinatorial problems can be computationally expensive. Thus, we show that our general approach can get significant improvements in the number of solved instances even when using a relative small number of instances in our training sets. Therefore, we used two partitions sizes for the training-testing sets (respectively 20–80% and 30–70%).

In particular in this paper, we train/test our models using Random forest [4], an ensemble of trees that can be used to improve the classification and regression tree methods by combining a large number of trees. Each tree is built by recursively partitioning the sample into more homogeneous groups, and each split within a tree is created based on a random subset of candidate variables. After training, predictions for unseen samples can be made by averaging the predictions from all the individual regression trees or by taking the majority vote in the case of classification trees. In Section 5 we give more details about the configurations to train/test our models.

Finally, to evaluate our models, we considered three measures [30]. Namely, Root Mean Squared Error (RMSE) and Pearson's Correlation Coefficient (CC) for regression models. Also (RMSE) and (Accuracy) for classification models. Root Mean Squared Error is defined as $\sqrt[2]{1/n \sum_{i=1}^{n} (y_i - \mu_i)^2}$ and is used to measure the differences or discrepancies between true values (y_i) and the estimated ones (μ_i), therefore, lower RMSE are better. In our case, we calculated RMSE using true processing times p_j and estimated processing times $\overset{\approx}{p}_j$. We also use such values to calculate Pearson's Correlation Coefficient, which is defined as $(\sum_{i=1}^{n} (y_i \cdot \mu_i) - n \cdot \bar{y} \cdot \bar{\mu})/((n-1) \cdot s_y \cdot s_\mu)$ and is used to indicate the statistical correlation between two variables (e.g., true values y_i and the estimated ones μ_i). Both $\bar{y}$ and $\bar{\mu}$ denote sample mean, while s_y and s_μ denote standard deviations. CC is a value between -1 and 1 where 1 is a perfect correlation, 0 is no correlation, and -1 is an inverse correlation. The Accuracy of a classification model represents the number of instances that are predicted correctly a, expressed as a proportion of all instances b (i.e., a/b).

4.2 Scheduling policies

Scheduling is a common mechanism for improving different metrics such as waiting time, response time, and slowdown without purchasing additional resources. For instance, in First Come First Serve (FCFS), jobs are processed in the same order as they arrive. However, many systems give preference to short jobs applying policies like Shortest Job First (SJF) and to reduce fragmentation backfilling they schedule small jobs into scheduling gaps [24]. However, policies like SJF require to know how long each job will run, which is unknown for many type of jobs, including combinatorial problems that typically have a very large runtime variability.

4.2.1 Generic policies behavior for online scheduling of combinatorial problems

In this paper we focus on the idea of maximizing the number of solved combinatorial problem instances using limited processing resources. It makes sense to use the SJF policy instead of a FCFS policy in order to complete as many short jobs as possible. However, the presented problems in Section 3 include waiting time constraints that become known on job arrivals (i.e., $\overrightarrow{wt_j}$) and in some scenarios a SJF policy leads to solutions where perfectly solvable instances remain unsolved.

Example 1 in Fig. 1: we assume an environment with a single machine that becomes available at time 7 and jobs a and b were released while the system was busy. By the time the system becomes available, a has already waited 2 units of time and b has waited for 1. Therefore, either job can be selected since there are no violations to their waiting time constraints. In the best case scenario for SJF, the processing time estimations are perfect, so it can schedule the shortest job first. Assuming that $\overset{\approx}{p_a}$ and $\overset{\approx}{p_b}$ are perfect estimations, then b is selected first since $(5 < 10)$. b would end at time 12 and a would have waited 7 units of time, thus violating its maximum waiting time which is 4. In this case only b was solved.

It can be observed that if a is selected first, then b will have waited 11 units of time by the time a completes execution. Then b can be scheduled for execution due to its waiting constraint of 15 units of time. In this case both a and b are solved, however, SJF only solves one instance.

Based on the Example 1 scenario, it comes to mind to select instances by *"Shortest Waiting Time First (SWTF)"*. However, some modifications in the previous example can cause that perfectly solvable instances remain unsolved:

Example 2 in Fig. 1: we assume an environment with a single machine that becomes available at time 7 and two jobs a and b were released while the system was busy. It can be observed that if the SWTF policy is used, then job a is selected since $(7 < 8)$, leaving b unsolved due to its waiting time violation. However, both jobs can be solved if b is selected first.

A commonly used policy in offline/online problems is the Shortest Weighted Processing Time (SWPT) where jobs are scheduled by the lowest ratio (p_j/w_j). If we explore a similar approach but using waiting times instead, then we would have a "SWTPT - shortest waiting-processing time ratio" (p_j/wt_j). SWTPT would solve all instances in example 2. However, when applied to the first example, job b would be first selected to execution with a ratio of 0.33. This selection would leave a unsolved with a ratio of 2.5. Additionally, if a Longest

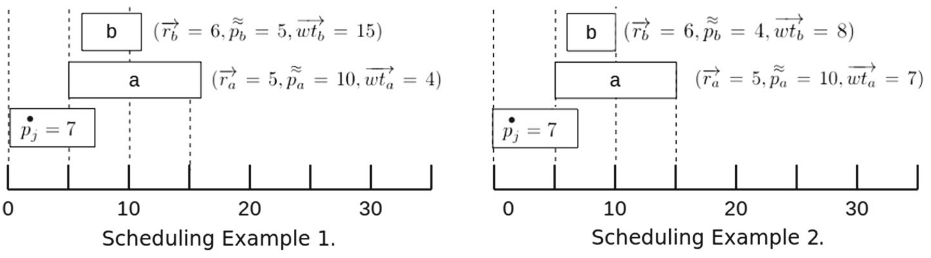
b
$(\overrightarrow{r_b} = 6, \overset{\approx}{p_b} = 5, \overrightarrow{wt_b} = 15)$
a
$(\overrightarrow{r_a} = 5, \overset{\approx}{p_a} = 10, \overrightarrow{wt_a} = 4)$
$\overset{\bullet}{p_j} = 7$
0
10
20
30
Scheduling Example 1.
b
$(\overrightarrow{r_b} = 6, \overset{\approx}{p_b} = 4, \overrightarrow{wt_b} = 8)$
a
$(\overrightarrow{r_a} = 5, \overset{\approx}{p_a} = 10, \overrightarrow{wt_a} = 7)$
$\overset{\bullet}{p_j} = 7$
0
10
20
30
Scheduling Example 2.

Fig. 1 Examples of single machine scheduling environments

Waiting-Processing Time ratio policy was used (i.e., LWTPT), it would also solve a single instance when used in the second example.

4.2.2 SJF-MIP hybrid policy for online scheduling of combinatorial problems

As we showed above, it is clear that selecting the right instance to be executed depends on the overall constraints, including the time that instances have been waiting in the system. Previously in [9], the authors presented a MIP approach to tackle the online problem $1|online, \overrightarrow{wt_j}, f, \overset{\approx}{p}_j, interrupt| \sum S_j$. They found interesting results by running MIP on a queue of released jobs. They used MIP every time the system needed to select a job for execution and assumed that solving the MIP model added no overhead to the system. However, in practice, when the number of released instances in queue becomes large, using MIP has a negative impact on the system due to the added overhead to solve the problem (i.e., the number of solved instances decreases).

Hereafter, we present a hybrid scheduling policy using SJF and MIP. Our approach implements a bounded queue to reduce the time that MIP takes to schedule instances for execution. This policy mitigates the overhead added by using MIP over all the released instances in queue and can be used together with regression/classification for the problems presented in Section 3. Namely, our approach implements 3 priority queues and the *Algorithm SJF-MIP* describes how to select an instance for execution when a machine becomes available:

- Q^{rel} - (*released instances queue*) priority queue that stores online released instances $j = (\overrightarrow{r_j}, \overset{\approx}{p}_j, \overrightarrow{wt_j})$. Instances are added using a SJF priority based on the processing time estimation. $\overset{\approx}{p}_j$ is calculated with a regression model R (see Section 3). In some scenarios we also use a classification model C to schedule instances that are predicted to be solvable by algorithm A within time t.
- Q^{int} - (*interrupted instances queue*) priority queue that stores instances that are interrupted from running and might be scheduled for execution again. Instances are stored using a SJF ordering policy based on the runtime estimations.
- Q^{sch} - (*scheduled instances queue*) queue that has a fixed capacity k to limit the number of instances that our MIP model will schedule for execution.

Algorithm SJF-MIP (j, f, Q^{rel}, Q^{int}, Q^{sch}, MIP-model, k)

```
Step 1.  Delete all jobs that break the waiting time constraint $\overrightarrow{wt_j}$ from $Q^{rel}$, $Q^{sch}$, and $Q^{int}$.
Step 2.  If either queue ($Q^{rel}$ or $Q^{sch}$) have jobs, then go to Step 4. Otherwise, continue to Step 3.
Step 3.  If $Q^{int}$ is not empty, return the interrupted job with the shortest estimated processing time $\overset{\approx}{p}_j$. Otherwise, wait for $Q^{rel}$ to get an online (released) instance from a user and go to Step 4.
Step 4.  If the schedule queue $Q^{sch}$ is empty, move at most the first $k$ instances from $Q^{rel}$ to $Q^{sch}$ and go to Step 6. Otherwise, move to Step 5.
Step 5.  If the first job in $Q^{rel}$ has a smaller processing than the last job in $Q^{sch}$, move all the jobs from $Q^{sch}$ to $Q^{rel}$ and go to Step 4. Otherwise, go to Step 7.
Step 6.  If $Q^{sch}$ has two or more jobs, run the MIP model with the jobs in the queue to compute the execution ordering of the jobs in $Q^{sch}$. Then go to Step 7.
Step 7.  Return the job with the highest priority in $Q^{sch}$.
```

Notice that *Algorithm SJF-MIP* guarantees to run MIP only when Q^{sch} has been updated, that is, when new released instances in Q^{rel} have smaller estimated processing times than the last instance scheduled by MIP. Thus, MIP runs not only on a smaller bounded queue Q^{sch} of size k, but it also might run less times. Next, we extend the MIP model proposed in [9] by modifying the objective function and generalizing it for multiple machines.

4.2.3 MIP-model to schedule combinatorial problems in multiple machines

In this section we introduce a MIP model to maximize the number of solved instances in multiple machines using runtime estimations. This model can be used with either all instances or only with instances classified as solvable with a classification model:

Indices and sets

- J: set of instances (also jobs) in the queue;
- i, j: instances ($i, j \in J$)
- M: set of machines to process the instances;
- m: machine index ($m \in M$)

Parameters

- $\overrightarrow{r_j}$: release time of instance j;
- $\overset{\approx}{p}_j$: runtime of instance j (estimation using a regression model);
- $\overrightarrow{wt_j}$: maximum expected waiting time of instance j.
- ct: cost of solving an instance;
- $time$: time when the system becomes available;
- $nextET_m$: time when machine m is expected to become available;

Decision variables

- ST_j: start time of instance j;
- ET_j: end time of instance j;
- WT_j: waiting time in the system of instance j;
- AT_j: boolean variable used to determine whether an instance j is attended before the maximum waiting time or not;
- X_j^m: boolean variable used to determine whether an instance j is assigned to machine m or not;

Maximize:

$$\sum_{j \in J} (ct - \overset{\approx}{p}_j) * AT_j \tag{4}$$

Subject to:

$$ST_j \geq \overrightarrow{r_j} \wedge ST_j \geq time + 1 \quad \forall j \in J \tag{5}$$

$$WT_j = ST_j - \overrightarrow{r_j} \quad \forall j \in J \tag{6}$$

$$AT_j = \begin{cases} 1, & \text{if } WT_j \leq \overrightarrow{wt_j} \\ 0, & otherwise \end{cases} \quad \forall j \in J \tag{7}$$

$$ET_j = ST_j + (\overset{\approx}{p}_j * AT_j) \quad \forall j \in J \tag{8}$$

$$\sum_{m \in M} X_j^m = 1 \quad \forall j \in J \tag{9}$$

$$ST_j * X_j^m \geq (nextET_m + 1) * X_j^m \quad \forall j \in J, \forall m \in M \tag{10}$$

$$\begin{aligned} & ET_j \leq (ST_i - 1) \vee ST_j \geq (ET_i + 1) \\ & \forall i, j \in J, \forall m \in M \quad | \quad i \neq j \wedge X_i^m = X_j^m = 1 \end{aligned} \tag{11}$$

$$ST_j, ET_j, WT_j \geq 0 \quad \forall j \in J \tag{12}$$

$$AT_j, X_j^m \in \{0, 1\} \quad \forall j \in J \tag{13}$$

The objective function of the *MIP-model* is highly influenced by Constraint (7) which enforces that only instances with a valid waiting time can be marked as attended. Thus, the objective of the model is to compute a schedule that maximizes the amount of attended-solvable instances. Notice that our approach attempts to maximize the number of solved instances based on estimated data, then our MIP model maximizes the number of attended jobs that are more likely to be solved under the given constraints.

We added a cost value ct to the objective function, in order to prioritize instances with smaller processing time estimations $\overset{\approx}{p}_j$ among those schedules that are able to attend the same amount of instances. This cost also determines the "estimated processing time upper bound" of the instances we want to process. e.g., if we set $ct = 1200$, then instances with processing time estimations lower than 1200 will be used in the schedule. In contrast, instances with higher estimations are discarded since they are marked as unattended. Thus, the lower the runtime estimation is, the higher the cost if offers to the objective function if it gets to be attended.

Deciding the value of ct highly depends on the maximum job execution time limit defined for the experiments. In our case we set $ct = 3601s$, a higher value than the job execution time limit t (in our case $t = 3600s$ since this is the time limit for the available data to train/test our models). Thus, our objective function prioritizes jobs with runtime estimations smaller than ct. Despite the fact we did not observe any runtime overestimation in the predictive models, we recall that jobs can not go beyond t in our experiments since the training/testing data is limited to such cutoff time. Therefore, our MIP model would mark instances with runtime overestimations as unattended since $(ct - \overset{\approx}{p}_j)$ would become negative, thus forcing AT_j to take a value of zero since it attempts to maximize each term. Additionally, instances marked as unattended by our MIP model (i.e., $AT_j = 0$) are deleted from the Q^{sch} queue.

Constraint (5) enforces that every instance has to start after its arrival time and after the system becomes available. Constraint (6) calculates the waiting time of each instance. It is also used to determine if an instance is attended or not in constraint (7). Constraint (8) calculates the end time of an instance j. Such end time, depends mainly on its start time and on the runtime estimation to solve such instance. However, it can also assign an end time equals to the start time (i.e., 0 time for processing) when an instance is not marked as attended (i.e., $ET_j = ST_j$, if $AT_j = 0$).

Constraint (9) guarantees that every instance is assigned to a single machine m. Additionally, when an instance is assigned to machine m, Constraint (10) makes sure that the start time of such instance is greater than the estimated time for the machine to become available. The estimation of when a machine m becomes available can be calculated by keeping track of when each machine started processing an instance j plus the runtime estimation. Finally, the disjunctive constraints (11) ensure that every pair of instances i and j, both assigned to machine m (i.e., $X_i^m = X_j^m = 1$) do not overlap.

Notice that some constraints (e.g., Constraint (10)) contain the multiplication of two variables and lead to a non-linear model. However, one of the variables is a decision variable and the remaining one is a Boolean variable, thus we followed a conversion to linearize those constraints using upper (u) and lower (l) bounds. Let $Z_j^m = ST_j * X_j^m$ where X_j^m is a binary variable and ST_j is a decision variable ranging from two bounds known in advanced (i.e., $l \leq ST_j \leq u$). The lower bound can be represented by the job release time $l = \overrightarrow{r_j}$ and the upper bound can be represented by a sufficiently big number. For instance, we used the latest time when machine m is expected to become available multiplied by a θ value (i.e., u = $nextET_m * \theta$). Then, Constraint (10) can be linearized as:

$$l * X_j^m \leq Z_j^m \leq u * X_j^m \tag{14}$$

$$ST_j - u(1 - X_j^m) \leq Z_j^m \leq ST_j - l(1 - X_j^m) \tag{15}$$

Consider first the case $X_j^m = 0$, which means the product $Z_j^m = ST_j * X_j^m$ should be 0. The first pair of inequalities would turn into $0 \leq Z_j^m \leq 0$, forcing $Z_j^m = 0$. The second pair of inequalities would turn into $ST_j - u \leq Z_j^m \leq ST_j - l$, and $Z_j^m = 0$ satisfies those inequalities. Now consider the case $X_j^m = 1$, so that the product should be $Z_j^m = ST_j$. The first pair of inequalities become $l \leq Z_j^m \leq u$, which is satisfied by $Z_j^m = ST_j$. The second pair of constraints would turn into $ST_j \leq Z_j^m \leq ST_j$, forcing $Z_j^m = ST_j$ as desired.

4.3 Instance interruption heuristics

Trying to schedule plenty of computational problems (including NP-Complete problems), using processing time estimations is open to inaccurate predictions that lead to bad scheduling decisions. In this paper we study a series of heuristics to interrupt instances from running in order to overcome the impact of inaccurate predictions:

N - The Naive Interruption: we will denote this heuristic with N to represent a straight forward execution where the selected instance j is executed to completion or until a cap time limit t is met.

$\mathbf{H_n}$ - The Heuristic Based Execution: let Q denote all the un-executed instances in (Q^{rel} and Q^{sch}). Also, let P denote all the currently running jobs. Then, we will use H_n to denote that a running instance $j \in P$ can be interrupted if there are n or more instances in Q that would not be attended if j continues to run. Equation (16) depicts how we make such decision. In general, we estimate the end time of a running instance j by adding the time when the instance started to be processed and the estimated runtime ($ET_j = ST_j + \overset{\approx}{p_j}$). Indeed, a waiting instance $i \in Q$, will not be attended if ($ET_j > \overrightarrow{r_i} + \overrightarrow{wt_i}$). However, processing times might be overestimated, then to avoid early interruptions, we decided to extend the execution of instance j by including $\overset{\approx}{p_i}$ in the constraint:

$$NotExecuted_{ij} = \begin{cases} 1, \text{ if } ET_j > (\overrightarrow{r_i} + \overrightarrow{wt_i} + \overset{\approx}{p_i}) & \forall i \in Q, \forall j \in P \\ 0, \; otherwise \end{cases}$$
$$\sum_{i \in Q} NotExecuted_{ij} \leq n \quad \forall j \in P \tag{16}$$

In general, Constraint (16) must be satisfied during execution of all jobs $j \in P$. For instance, the usage of heuristic H_4 means that any running instance j can be interrupted if 4 or more instances in Q are detected not to be attended if j continues to run. Notice that the heuristic requires to estimate the end time ET_j of a running instance j. Since j is already running, then its start time ST_j is already known but the processing time $\overset{\approx}{p_j}$ is an estimation.

5 Experiments

We recall that all the tests and results from this section were obtained by using the LaScILab computational cluster from the Universidad del Valle in Cali-Colombia (http://lascilab.univalle.edu.co/). It features 320 Cores and 768Gb of RAM memory accessed through HTCondor, a distributed batch computing system that supports High Throughput Computing (HTC) [27]. To evaluate the online approach proposed in Section 4, we use the same set of instances and algorithm runtime data as those used and reported in [13]. We recall that our study is limited to the usage of a single algorithm/solver per dataset type. The SAT instance dataset comprises data about MiniSAT with a time limit t of one hour. The dataset was collected from the international SAT competitions and races from 2002 to 2010. It contains 7012 instances that includes industrial (INDU), hand crafted (HAND), and random problems (RAND). We also used the combination of all SAT instances into a single dataset (INDU-HAND-RAND). On the other hand, the MIP instance dataset contains data for CPLEX running within an hour time limit and attempting to solve (BIGMIX) and (CORLAT) instances with 1510 and 2000 MILP each. Complete details of the instances and runtimes are available at www.cs.ubc.ca/labs/beta/Projects/EPMs/.

5.1 Training/Testing predictive models

To train our regression models we performed a log transformation of the runtimes. To train the classification models we managed to label timeouts as *not solvable* and valid runtimes as *solvable* (i.e., 0 or 1 for instances running for a cap time limit t as explained in Section 4). Moreover, we randomly split the instance sets into 2 partitions (i.e. 20–80%, 30–70%) to study the impact of the training partition size over the testing one. We recall that obtaining training/testing data for runtime estimation of combinatorial problems can be computationally expensive. Thus, we wanted to show that our general approach can get significant improvements in the number of solved instances even when using a relative small number of instances in our training datasets. Furthermore, we anticipate that increasing our training dataset will certainly increase the accuracy of our machine learning models, thus leading to better performance.

We also use the feature categories (trivial and cheap introduced in [13]) to present our results. We use a random forest implementation from Weka [30] (version 3.8) with its default hyperparameters to Train-Test our models. A preliminary evaluation of other techniques such as Gaussian process, neural networks, and linear regression showed lower performance. We recall that this behavior is consistent to the literature and similarly to results presented in [13].

In Table 1, we report evaluation measures of the regression and classification models for SAT instances with two different dataset partitions. In general, it can be observed that the models reduce the Root Mean Squared Error (RMSE) and increase their Correlation Coefficient (CC) as the training partition becomes bigger (e.g., 0.90 and 0.88, 0.78 and 0.91 for the HAND dataset with cheap features across the two partitions). Moreover, models trained with *Trivial* features report a lower CC and a higher RMSE with respect to models trained with *Cheap* features. It is worth noting that a similar behavior was also observed in the MIP datasets.

Interestingly, it can be observed that classification models present lower RMSE than Regression models (i.e., less than 0.40 for classification and over 0.44 for regression), we recall that this behavior is very important since classification has a direct impact in our approach and a similar behavior was reported for MIP models. Additionally, there is a

Table 1 SAT predictive models evaluation with two different testing-training partitions

Data	Domain		Regression models		Classification models	
Part.	DataSet (Solver)	Features	RMSE	CC	RMSE	Acc.
20–80	INDU (MiniSat)	Trivial	1.19	0.74	0.35	83.8%
		Cheap	0.92	0.86	0.34	84.6%
	HAND(MiniSat)	Trivial	1.47	0.65	0.39	78.9%
		Cheap	0.90	0.88	0.32	86.4%
	RAND(MiniSat)	Trivial	1.75	0.82	0.28	89.2%
		Cheap	0.47	0.96	0.21	93.5%
30–70	INDU (MiniSat)	Trivial	1.14	0.78	0.31	88.1%
		Cheap	0.80	0.90	0.30	87.7%
	HAND(MiniSat)	Trivial	1.42	0.68	0.37	79.9%
		Cheap	0.78	0.91	0.28	90.2%
	RAND(MiniSat)	Trivial	0.94	0.85	0.26	89.7%
		Cheap	0.45	0.96	0.20	94.1%

trade-off between the feature family and the quality of the predictions. *Trivial* features are basically free but they have an impact on the quality of the models. On the other hand, the set of *Cheap* features have a lower error and a higher correlation but the computational cost to calculate the features is higher. *Cheap* features, offer an interesting trade-off between prediction quality and computational cost. We would like to remark that we are not considering the *expensive* feature set described in [13]. This feature set introduces an extra overhead (about 75 h of the actual solving time).

5.2 Online scheduling policies and interruption heuristics

Table 2 shows the configuration of our experimental evaluations. In particular, we tested our model for 1, 2, and 4 machines. Predictive models were trained with 30 and 20% of the data. Respectively, we use the remaining 70 and 80% of the instances to test our approach and compare results.

We configured our evaluation by releasing instances (in the test set) one at a time with random inter-arrival interval times ranging 1s to 120s; the waiting time for each instance is randomly distributed within five intervals ranging from 0s to 480s; we ran our experiments with the SAT and MIP instances described above; and tested three scheduling policies

Table 2 Configuration values for test simulations

M	Interr.	Train/Test	r_j	wt_j	Dataset/solver	Tests
1	N	30–70%	1–10 s	0–30 s	BIGMIX/Cplex	FCFS(R)
2	H2	20–80%	1–30 s	0–90 s	CORLAT/Cplex	FCFS(CR)
4	H4		1–60 s	0–180 s	INDU/Minisat	SJF(R)
	H8		1–120 s	0–240 s	HAND/Minisat	SJF(CR)
				240–480 s	RAND/Minisat	SJF-MIP(R)
					INDU HAND RAND/Minisat	SJF-MIP(CR)

(i.e, FCFS, SJF, SJF-MIP) using either Regression (R) or Classification-Regression (CR) models. We recall that we use regression to estimate the runtime of a given instance and classification to estimate whether an instance can be solved or not. The combination of all the possible evaluation configurations in Table 2 resulted in 8640 simulations. Additionally, we ran each simulation 5 times using different instance orderings and calculated the average number of solved instances for each test. We recall that the same data configuration was used to test and compare all the approaches.

5.2.1 Interruption heuristics analysis

In Fig. 2, we reported the average number of solved instances of two datasets under a single machine (i.e., Indu-Hand-Rand and Bigmix). The results are presented with regression and classification models Trained-Tested with 30–70% of the data, using*Trivial* (left) and *Cheap* (right) features. The results were obtained across all the tests using different interruption heuristics. Namely, naive (N) and three configurations for (H_n) as proposed in Section 4.3. It can be observed that the interruption heuristic with the highest number of solved instances is (H_2) for both datasets despite the features used to train the models. We recall that the same behavior was observed across the six datasets studied here.

Notice that there is an increment on the number of solved instances when using models trained with *Cheap* features. For instance, our hybrid SJF-MIP (RC) policy with the interruption heuristic H_2, increments from 1101 to 1286 solved instances when using *Cheap* features instead of *Trivial*. We recall that such behavior is expected since the models trained with *Cheap* features outperform those trained with *Trivial* features (see Table 1).

Interestingly, Fig. 2 shows that (H_n) seems to reduce its effectiveness as the number of observed instances to perform an interruption grows. Additionally, when using *Cheap* features, our interruption heuristic is able to improve the number of solved instances of a generic policy like FCFS-R up to 21.6× and 12.2× for INDU-HAND-RAND and CORLAT instances. Despite FCFS does not need runtime estimations to schedule instances, we use regression in order to estimate runtimes and execute the interruption heuristic (i.e., FCFS-R).

We also observed that our interruption heuristic H_2 increments the number solved instances across every scheduling policy. Furthermore, our hybrid approach SJF-MIP-RC solves more instances than any other policy for INDU-HAND-RAND instances. When using *Cheap* features, it also reported 118 more instances than SJF-RC, as a positive impact of our approach using MIP. Similarly happens for BIGMIX, where the impact of using classification models is less significant and both SJF-MIP-R and SJF-MIP-RC, report nearly the same number of solved instances. Finally, when using *Cheap* features in BIGMIX, our hybrid approach SJF-MIP-R solves 83 and 125 more instances than SJF-R and FCFS-R, which corresponds to an increment of 13.9 and 21% respectively.

5.2.2 Policy evaluation

Semi clairvoyant policy (SCP) The performance of online algorithms is generally measured on the basis of competitive analysis. For instance, if the objective function value of a schedule obtained from the online algorithm is no more than p times the optimal (offline) objective function value for any problem instance, then the algorithm is p-competitive [1]. However, when scheduling combinatorial problems, this comparison against an all knowledgeable adversary seems unrealistic, specially because there are unknown processing times in the testing data (i.e., timeouts). Preliminary experimentations showed that our hybrid policy SJF-MIP(CR) using classification and regression behaves better than any other policy

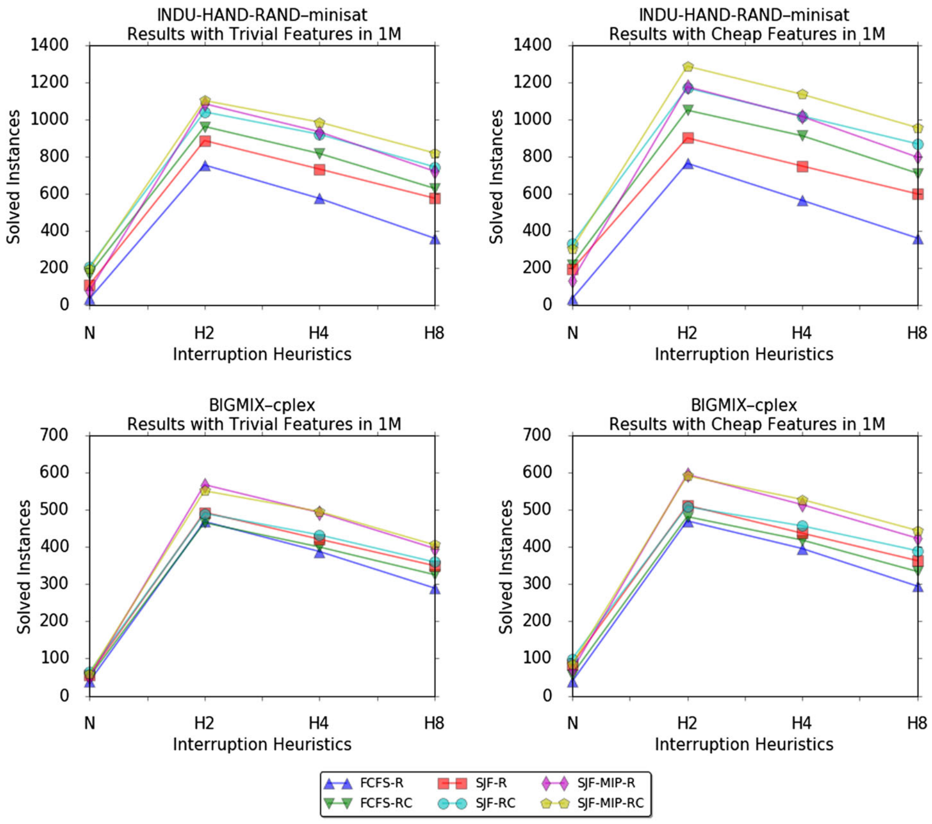

Fig. 2 Graphic reporting the average number of solved instances using four interruption heuristics and featuring a single machine configuration. Predictive models were trained with *Trivial* and *Cheap* features using a 30–70% Test-Train partition

if working with an accurate predictive model. Thus, every policy can be compared with the Semi clairvoyant Policy that is a version of our SJF-MIP(RC) that is non-anticipative but is able to do perfect classifications and runtime estimations based on the known data.

We now move our attention to Fig. 3, where we reported the average number of instances across all the tests grouped by interarrival configurations. The results are presented with a two machine configuration and the best known interruption heuristic (i.e., H_2). Additionally, policies presented in this graphic use regression (R) and classification (C) models trained with a 30–70% partition using *Cheap* features . The blue line represents the semi clairvoyant policy (SCP) and it can be observed that our hybrid approaches are typically closer than other policies, specially when the interarrival frequency is high (i.e., for small interarrivals). We can also observe that as the interarrival range grows, the gap among all policies seems to become smaller. Notice that policies of the same kind (e.g., SJF-R and SJF-RC) tend to be closer one to another in some experiments, specially when the interarrival is incremented. Such behavior indicates that the positive impact of using classification, seems to become smaller as the interarrival is incremented. We recall that across all the experiments with the six datasets, our hybrid approaches SJF-MIP(R) and SJF-MIP(CR) displayed best performance on average with respect to the SCP.

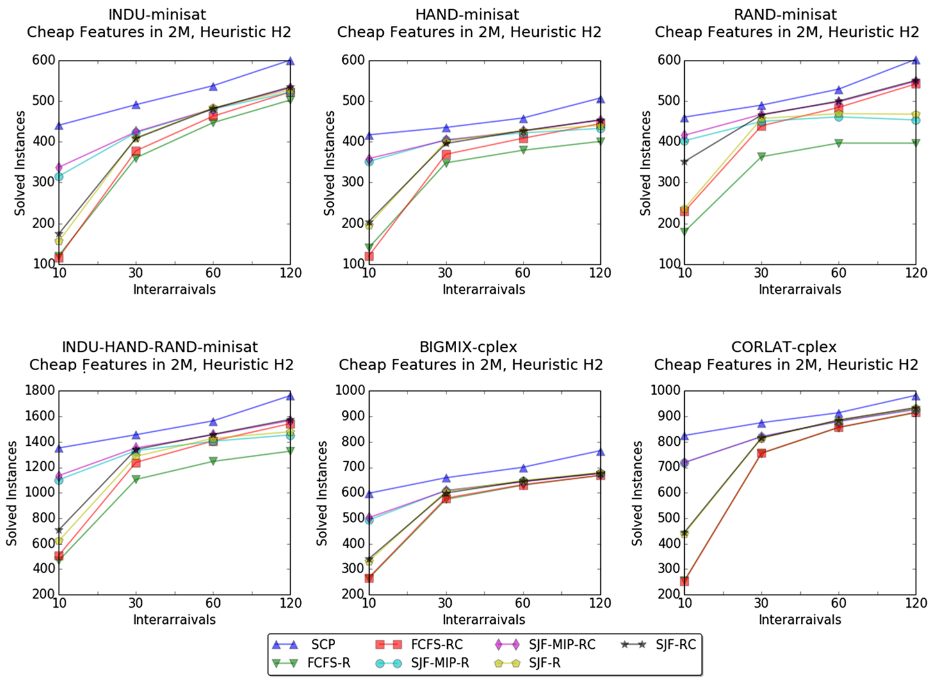

Fig. 3 Experimental results using the interruption heuristic H_2 and varying interarrivals to report the average number of solved instances in comparison to the Semi Clairvoyant Policy (SCP). Predictive models were trained using *Cheap* features and results are reported with a 30–70% Test-Train dataset partition featuring a two machine configuration

We now focus on Table 3 that summarizes the results of our experiments using *Cheap* features. For this table, we reported the average number of solved instance per dataset partition, test, machine, and scheduling policy. It can be observed that our hybrid approach SJF-MIP(RC) is the overall winner and is typically followed by our SJF-MIP(R) approach. We also included two columns to compare the differences in solved instances of SJF(RC) and SJF-MIP(RC) with respect to the SCP. These two columns include percentages that represent how distant both policies are from the semi clairvoyant policy (SCP), therefore, the smaller the percentage is, the closer it is to the baseline reference. At the broadest level, SJF-MIP(RC) reported lower differences than SJF(RC). This lower percentages indicate that our hybrid approach is closer in performance to the SCP than the SJF(RC) policy. For instance, under a 20–80% partition for CORLAT in 1M, our hybrid approach reported 5.4% solved instances below the SCP while SJF-MIP(R) reported 19.7%. Such values represent 94.6 and 80.3% of the solved instances featuring perfect estimations.

We recall that our approach seems to solve more instances using the partition 20–80%, however, the increment in solved instances is due to the testing dataset size. Nevertheless, notice that the SJF-MIP(RC) differences with respect to the SCP tend to be smaller when using the bigger training partitions. For instance, the INDU-HAND-RAND dataset configured with 1 machine reports 12.4 and 14.4% differences using 30–70% and 20–80% partitions respectively. These differences indicate that models Trained-Tested with a 30–70% partition, typically perform better than those Trained-Tested with a 20–80% partition.

Table 3 Experiments reporting the average number of solved instances per dataset, machine, and policy combinations using the H_2 interruption heuristic

Dataset	M	SCP	FCFS (R)	FCFS (RC)	SJF (R)	SJF (RC)	SJF-MIP (R)	SJF-MIP (RC)	SJF-MIP (RC) Diff. SCP	SJF (RC) Diff. SCP
20–80% partition results using *Cheap* features and the interruption heuristic H_2										
INDU	1M	**562**	351	364	381	403	461	**468**	16.7%	28.3%
	2M	**584**	401	406	443	444	**499**	496	15.1%	24.0%
	4M	**667**	406	416	453	451	499	**501**	24.9%	32.4%
HAND	1M	**501**	284	352	327	380	396	**404**	19.4%	24.2%
	2M	**525**	372	374	423	414	**453**	445	15.2%	21.1%
	4M	**588**	372	404	428	445	460	**467**	20.6%	24.3%
RAND	1M	**564**	261	435	333	489	436	**499**	11.5%	13.3%
	2M	**603**	386	473	462	530	515	**548**	9.1%	12.1%
	4M	**694**	390	529	494	584	512	**597**	14.0%	15.9%
INDU-HAND-RAND	1M	**1665**	893	1176	1036	1335	1275	**1425**	14.4%	19.8%
	2M	**1745**	1173	1320	1357	1428	1486	**1542**	11.6%	18.2%
	4M	**1982**	1195	1398	1407	1506	1507	**1612**	18.7%	24.0%
BIGMIX	1M	**743**	533	556	567	579	653	**654**	12.0%	22.1%
	2M	**772**	611	615	643	644	**683**	681	11.8%	16.6%
	4M	**849**	618	624	646	648	682	**685**	19.3%	23.7%
CORLAT	1M	**1002**	752	746	809	805	947	**948**	5.4%	19.7%
	2M	**1023**	794	790	875	874	951	**952**	6.9%	14.6%
	4M	**1117**	821	829	909	912	981	**984**	11.9%	18.4%
30–70% partition results using *Cheap* features and the interruption heuristic H_2										
INDU-HAND-RAND	1M	**1467**	763	1049	900	1168	1175	**1285**	12.4%	20.4%
	2M	**1534**	1035	1174	1203	1270	1322	**1377**	10.2%	17.2%
	4M	**1745**	1051	1254	1242	1358	1331	**1445**	17.2%	22.2%
40–60% partition results using *Cheap* features and the interruption heuristic H_2										
INDU-HAND-RAND	1M	**1255**	651	921	797	1017	**1018**	1105	12.0%	19.0%
	2M	**1315**	880	1005	1025	1105	1140	**1198**	8.9%	16.0%
	4M	**1498**	893	1085	1067	1194	1150	**1272**	15.1%	20.3%
50–50% partition results using *Cheap* features and the interruption heuristic H_2										
INDU-HAND-RAND	1M	**1044**	531	786	662	875	861	**926**	11.3%	16.2%
	2M	**1092**	728	831	865	928	951	**995**	8.9%	15.0%
	4M	**1246**	737	910	897	1003	959	**1062**	14.8%	19.5%

Policies use models trained with *Cheap* features and they are compared against a Semi Clairvoyant Policy (SCP) that features perfect processing time estimations. Bold numbers represent the policy with highest number of solved instances with respect to the SCP. Percentages for two chosen policies indicate how distant they are from the baseline reference SCP

Interestingly, even smaller differences were observed in some extra INDU-HAND-RAND experiments Trained-Tested with 40–60% and 50–50% partitions (12.0 and 11.3% respectively). We attribute such behavior to the fact that predictive models trained with bigger partitions demonstrated to be more accurate as previously shown on Table 1. Similar results were obtained across every dataset (more detailed results are available in Appendix).

We also observed an increment in the number of solved instances as the number of machines grow on Table 3. It also seems that the rate of solved instances tend to grow slower as the number of machines is incremented. Such behavior can be observed in the HAND dataset, where 41 extra instances were solved by moving from one to two machines (from 404 to 445) , then 22 instances were increased by moving from two to four machines.

We now move our attention to Table 4 where we report the increment in the number of solved instances when moving from *Trivial* to *Cheap* features in order to Train-Test predictive models. Notice that we decided to report policies that use Regression and Classification (RC) models, because they typically display the highest number of solved instances. It can be observed that our hybrid SJF-MIP approach featuring regression and classification

Table 4 Experiments comparing the average number of solved instances of policies that use regression and classification models (RC) trained with *Trivial* and *Cheap* features

Dataset	M	Trivial features			Cheap features			SJF-MIP (RC)
		FCFS (RC)	SJF (RC)	SJF-MIP (RC)	FCFS (RC)	SJF (RC)	SJF-MIP (RC)	Increment
20–80% partition results using cheap and trivial features and the interruption heuristic H_2								
INDU	1M	359	453	384	364	403	**468**	17.95%
	2M	397	471	426	406	444	**496**	14.11%
	4M	408	482	438	416	451	**501**	12.57%
HAND	1M	286	335	314	352	380	**404**	22.28%
	2M	311	383	338	374	414	**445**	24.04%
	4M	325	390	349	404	445	**467**	25.27%
RAND	1M	404	444	469	435	489	**499**	6.01%
	2M	455	520	501	473	530	**548**	8.58%
	4M	489	541	526	529	584	**597**	11.89%
INDU-HAND-	1M	1034	1203	1131	1176	1335	**1425**	20.63%
RAND	2M	1178	1382	1243	1320	1428	**1542**	19.39%
	4M	1222	1405	1291	1398	1506	**1612**	19.91%
BIGMIX	1M	529	629	542	556	579	**654**	17.13%
	2M	591	649	608	615	644	**681**	10.72%
	4M	602	651	608	624	648	**685**	11.24%
CORLAT	1M	770	916	823	746	805	**948**	13.19%
	2M	812	927	866	790	874	**952**	9.03%
	4M	841	952	881	829	912	**984**	10.47%
30–70% partition results using cheap and trivial features and the interruption heuristic H_2								
INDU-HAND-	1M	961	1101	1040	1049	1168	**1285**	19.07%
RAND	2M	1062	1263	1146	1174	1270	**1377**	16.78%
	4M	1106	1293	1196	1254	1358	**1445**	17.23%

Bold numbers represent the policy with highest number of solved instances. Percentages indicate the increment of solved instances when using models trained with *Cheap* features instead of *Trivial* for the best performing policy SJF-MIP (RC) models Trained-Tested with *Cheap* features solves more instances than the rest of the policies, including those using *Trivial* features.

Finally, the last column indicates the increment percentage displayed by our SJF-MIP approach when moving from Trivial to Cheap features. Notice that for all the experiments we reported increments in the number of solved instances independently from the partition used to Train-Test the models. For instance, the HAND dataset reported an increment of 25.27%, featuring four machines when models were Trained-Tested with 20–80% of the dataset. On the other hand, the lowest increment was reported for the RAND dataset (i.e., 6.01%) featuring a single machine when models were Trained-Tested with 20–80% of the dataset (more detailed results are available in Appendix).

6 Related work

Online scheduling has been extensively studied in the last decade and most of the approaches attempt to minimize processing time related functions. In [5] the authors study the online scheduling of parallel jobs on two machines where jobs must be assigned to a time slot in the schedule before the next job is presented. The authors show that no online algorithm for this problem can have a competitive ratio less than $1+\sqrt{2/3}$. Interestingly, in [12] the authors consider parallel jobs on single and parallel machines to minimize makespan. They show that online scheduling of parallel jobs on two machines is 2-competitive. In [1] the authors consider a single machine to minimize total weighted completion time and show that the online problem has a competitive ratio of 2. Other approaches include online parallel-batch scheduling as surveyed in [28].

In this work we present a computational approach that benefits from queuing policies and a MIP model to schedule instances to maximize an objective function. Interestingly, in [26] the authors investigate about the integration of long-term stochastic reasoning as studied in queuing theory with short-term combinatorial reasoning as studied in scheduling. The authors show that the long-term, stochastic reasoning of queuing theory can be combined with short-term combinatorial reasoning to produce a hybrid scheduling algorithm that achieves better performance than either approach alone.

As part of our research, we presented a computational approach that uses regression models to estimate runtimes and used interruption heuristics to mitigate missed predictions. However, other interesting approaches typically assume the existence of an accurate runtime predictor. For instance, in [22] the authors assume accurate runtime predictions and propose a family of *Cloud-based, online, Hybrid scheduling policies* (CoH) in order to minimize rental cost by using on-demand and reserved instances for which the authors formulate the *resource provisioning* and *job allocation* as integer programming problems.

Cloud computing provides on-demand resources and services over a network and has brought new challenges to the construction of new scheduling algorithms. In first-come-first-serve [16] jobs ordered by their arrival time. However, many system designs give preference to short jobs applying policies like Shortest-Job-First (SJF) and to reduce fragmentation backfilling they schedule small jobs into scheduling gaps [24]. Another scheduling policy includes Preemptive-Shortest-Job-First (PSJF) where the new jobs entering the system with the smallest expected duration (size) are given preemptive priority [2].

In an approach proposed by Deng et al. [8] the authors explore the usage of a portfolio scheduler to select the most suitable policy for a given workload using an online simulator. However, the proposed simulation becomes infeasible as the number of constituent policies grows. Interestingly, In [6] the authors tackled this problem by using an online simulator with a time limit constraint to avoid an exhaustive policy evaluation. In [7] the authors propose a portfolio approach to identify the most suitable parallel job scheduling policy. The authors show that selecting an appropriate policy can lead to a performance improvement from 7 to 100% when compared to the best individual policy. Contrasting with Deng et al. the authors in [25], use a portfolio methodology to select robust scheduling policies in presence of system variances.

7 Conclusions

In this paper, we have presented an online approach to solve SAT and MIP combinatorial problem instances with release and waiting time constraints. Unlike typical approaches, we attempted to maximize the number of solved instances using single and multiple machine configurations. To formalize our problem, we presented an extension of the $\alpha|\beta|\gamma$ notation that allowed us represent the necessary constraints. We also proposed an approach for online scheduling of combinatorial problems that consisted of three parts. Namely, training/testing models for processing time estimations; implementation of a hybrid scheduling policy using SJF and MIP; usage of instance interruption heuristics to mitigate inaccurate predictions.

We extensively tested our approach and reported considerably big improvements of up to 4.7× and 21.6× when using our interruption heuristic with SAT instances using SJF and FCFS ordering policies. Additionally, our hybrid approach observed results that are close to a semi clairvoyant policy (SCP) featuring perfect estimations. We trained regression and classification models using random forest that were later used in our hybrid SJF-MIP approach. We also studied the impact of the partition size to Train-Test our predictive models and observed that with very limited data to train the models (e.g., 30% of the dataset) our approach reports scenarios with up to 90% of solved instances with respect to the SCP.

We also experimented using models that were trained with different feature families and observed an interesting trade-off between the quality of the predictions and the computational cost to calculate such features. For instance, *Trivial* features are basically free to compute but they have impact on the quality of the models. On the other hand, *Cheap* features offer an interesting trade-off between prediction quality and computational cost. Finally, as part of this research, we also plan to extend this study in order to explore the impact of other constraints such as due dates when trying to maximize the number of solved jobs.

Acknowledgments The authors would like to thank the anonymous reviewers for their comments and suggestions which helped to improve the paper. Robinson Duque is supported by the Universidad del Valle and also by Colciencias, the Colombian Administrative Department of Science, Technology and Innovation under the PhD scholarship program.

Appendix: Experiment results across all datasets

In this appendix we present further results of our online scheduling approach tested with SAT and MIP instances with waiting time constraints. The results of our approach include six different datasets and we managed to extend some of our experiments using 40–60% and 50–50% partitions:

Table 5 Experiments reporting the average number of solved instances per dataset, machine, and policy combinations using the H_2 interruption heuristic

Dataset	M	SCP	FCFS (R)	FCFS (RC)	SJF (R)	SJF (RC)	SJF-MIP (R)	SJF-MIP (RC)	SJF-MIP (RC) Diff. SCP	SJF (RC) Diff. SCP
20–80% partition results using *Cheap* features and the interruption heuristic H_2										
INDU	1M	**562**	351	364	381	403	461	**468**	16.7%	28.3%
	2M	**584**	401	406	443	444	**499**	496	15.1%	24.0%
	4M	**667**	406	416	453	451	499	**501**	24.9%	32.4%
HAND	1M	**501**	284	352	327	380	396	**404**	19.4%	24.2%
	2M	**525**	372	374	423	414	**453**	445	15.2%	21.1%
	4M	**588**	372	404	428	445	460	**467**	20.6%	24.3%
RAND	1M	**564**	261	435	333	489	436	**499**	11.5%	13.3%
	2M	**603**	386	473	462	530	515	**548**	9.1%	12.1%
	4M	**694**	390	529	494	584	512	**597**	14.0%	15.9%
INDU-HAND-	1M	**1665**	893	1176	1036	1335	1275	**1425**	14.4%	19.8%
RAND	2M	**1745**	1173	1320	1357	1428	1486	**1542**	11.6%	18.2%
	4M	**1982**	1195	1398	1407	1506	1507	**1612**	18.7%	24.0%
BIGMIX	1M	**743**	533	556	567	579	653	**654**	12.0%	22.1%
	2M	**772**	611	615	643	644	**683**	681	11.8%	16.6%
	4M	**849**	618	624	646	648	682	**685**	19.3%	23.7%
CORLAT	1M	**1002**	752	746	809	805	947	**948**	5.4%	19.7%
	2M	**1023**	794	790	875	874	951	**952**	6.9%	14.6%
	4M	**1117**	821	829	909	912	981	**984**	11.9%	18.4%
30–70% partition results using *Cheap* features and the interruption heuristic H_2										
INDU	1M	**493**	302	339	343	368	416	**428**	13.2%	25.4%
	2M	**518**	358	370	394	400	435	**444**	14.3%	22.8%
	4M	**591**	360	387	400	415	448	**461**	22.0%	29.8%
HAND	1M	**440**	240	315	296	345	350	**377**	14.3%	21.6%
	2M	**455**	317	336	366	370	403	**411**	9.7%	18.7%
	4M	**511**	323	364	370	396	407	**431**	15.7%	22.5%
RAND	1M	**490**	232	383	294	430	377	**446**	9.0%	12.2%
	2M	**521**	334	424	408	467	442	**483**	7.3%	10.4%
	4M	**600**	341	480	427	519	439	**531**	11.5%	13.5%
INDU-HAND-	1M	**1467**	763	1049	900	1168	1175	**1285**	12.4%	20.4%
RAND	2M	**1534**	1035	1174	1203	1270	1322	**1377**	10.2%	17.2%
	4M	**1745**	1051	1254	1242	1358	1331	**1445**	17.2%	22.2%
BIGMIX	1M	**657**	470	481	512	508	**595**	591	10.0%	22.7%
	2M	**680**	533	536	566	565	606	**607**	10.7%	16.9%
	4M	**746**	540	546	573	571	609	**612**	18.0%	23.5%
CORLAT	1M	**878**	661	664	717	718	**831**	**831**	5.4%	18.2%
	2M	**899**	695	695	768	770	836	**837**	6.9%	14.3%
	4M	**978**	728	731	797	798	863	**865**	11.6%	18.4%

Table 5 (continued)

Dataset	M	SCP	FCFS (R)	FCFS (RC)	SJF (R)	SJF (RC)	SJF-MIP (R)	SJF-MIP (RC)	SJF-MIP (RC) Diff. SCP	SJF (RC) Diff. SCP
40–60% partition results using *Cheap* features and the interruption heuristic H_2										
INDU-HAND-	1M	**1255**	651	921	797	1017	**1018**	1105	12.0%	19.0%
RAND	2M	**1315**	880	1005	1025	1105	1140	**1198**	8.9%	16.0%
	4M	**1498**	893	1085	1067	1194	1150	**1272**	15.1%	20.3%
50–50% partition results using *Cheap* features and the interruption heuristic H_2										
INDU-HAND-	1M	**1044**	531	786	662	875	861	**926**	11.3%	16.2%
RAND	2M	**1092**	728	831	865	928	951	**995**	8.9%	15.0%
	4M	**1246**	737	910	897	1003	959	**1062**	14.8%	19.5%

Policies use models trained with *Cheap* features and they are compared against a Semi Clairvoyant Policy (SCP) that features perfect processing time estimations. Bold numbers represent the policy with highest number of solved instances with respect to the SCP. Percentages for two chosen policies indicate how distant they are from the baseline reference SCP

Table 6 Experiments comparing the average number of solved instances of policies that use regression and classification models (RC) trained with *Trivial* and *Cheap* features

Dataset	M	Trivial features			Cheap features			SJF-MIP (RC)
		FCFS (RC)	SJF (RC)	SJF-MIP (RC)	FCFS (RC)	SJF (RC)	SJF-MIP (RC)	Increment
20–80% partition results using cheap and trivial features and the interruption heuristic H_2								
INDU	1M	359	453	384	364	403	**468**	17.95%
	2M	397	471	426	406	444	**496**	14.11%
	4M	408	482	438	416	451	**501**	12.57%
HAND	1M	286	335	314	352	380	**404**	22.28%
	2M	311	383	338	374	414	**445**	24.04%
	4M	325	390	349	404	445	**467**	25.27%
	1M	404	444	469	435	489	**499**	6.01%
RAND	2M	455	520	501	473	530	**548**	8.58%
	4M	489	541	526	529	584	**597**	11.89%
INDU-HAND-	1M	1034	1203	1131	1176	1335	**1425**	20.63%
RAND	2M	1178	1382	1243	1320	1428	**1542**	19.39%
	4M	1222	1405	1291	1398	1506	**1612**	19.91%
BIGMIX	1M	529	629	542	556	579	**654**	17.13%
	2M	591	649	608	615	644	**681**	10.72%
	4M	602	651	608	624	648	**685**	11.24%
CORLAT	1M	770	916	823	746	805	**948**	13.19%
	2M	812	927	866	790	874	**952**	9.03%
	4M	841	952	881	829	912	**984**	10.47%

Table 6 (continued)

Dataset	M	Trivial features			Cheap features			SJF-MIP (RC)
		FCFS (RC)	SJF (RC)	SJF-MIP (RC)	FCFS (RC)	SJF (RC)	SJF-MIP (RC)	Increment
30–70% partition results using cheap and trivial features and the interruption heuristic H_2								
INDU	1M	339	409	363	339	368	**428**	15.19%
	2M	359	425	384	370	400	**444**	13.51%
	4M	381	446	408	387	415	**461**	11.50%
HAND	1M	250	306	280	315	345	**377**	25.73%
	2M	275	349	298	336	370	**411**	27.49%
	4M	285	348	308	364	396	**431**	28.54%
RAND	1M	360	388	411	383	430	**446**	7.85%
	2M	397	457	443	424	467	**483**	8.28%
	4M	425	478	467	480	519	**531**	12.05%
INDU-HAND-	1M	961	1101	1040	1049	1168	**1285**	19.07%
RAND	2M	1062	1263	1146	1174	1270	**1377**	16.78%
	4M	1106	1293	1196	1254	1358	**1445**	17.23%
BIGMIX	1M	465	550	490	481	508	**591**	17.09%
	2M	523	576	539	536	565	**607**	11.20%
	4M	527	578	549	546	571	**612**	10.29%
CORLAT	1M	674	801	717	664	718	**831**	13.72%
	2M	721	808	747	695	770	**837**	10.75%
	4M	745	837	774	731	798	**865**	10.52%

Bold numbers represent the policy with highest number of solved instances. Percentages indicate the increment of solved instances when using models trained with *Cheap* features instead of *Trivial* for the best performing policy SJF-MIP (RC)

References

1. Anderson, E.J., & Potts, C.N. (2004). Online scheduling of a single machine to minimize total weighted completion time. *Mathematics of Operations Research*, *29*(3), 686–697.
2. Arpaci-Dusseau, R.H., & Arpaci-Dusseau, A.C. (2014). *Operating systems: three easy pieces*, chap. Scheduling: Introduction. Arpaci-Dusseau Books.
3. Bartz-Beielstein, T., & Markon, S. (2004). Tuning search algorithms for real-world applications: a regression tree based approach. In *Congress on evolutionary computation, 2004. CEC2004*, (Vol. 1 pp. 1111–1118). IEEE.
4. Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.
5. Chan, W.T., Chin, F.Y., Ye, D., Zhang, G., Zhang, Y. (2008). On-line scheduling of parallel jobs on two machines. *Journal of Discrete Algorithms*, *6*(1), 3–10.
6. Deng, K., Song, J., Ren, K., Iosup, A. (2013). Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds. In *Proceedings of the international conference on high performance computing, networking, storage and analysis* (p. 55). ACM.
7. Deng, K., Song, J., Ren, K., Iosup, A. (2013). Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds. In *SC*.
8. Deng, K., Verboon, R., Ren, K., Iosup, A. (2013). A periodic portfolio scheduler for scientific computing in the data center. In *Workshop on job scheduling strategies for parallel processing* (pp. 156–176). Springer.

9. Duque, R., Arbelaez, A., Díaz, J.F. (2017). *Off-line and on-line scheduling of SAT instances with time processing constraints*, (pp. 524–539). Cham: Springer International Publishing.
10. Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, *5*, 287–326.
11. Heule, M.J.H., Kullmann, O., Marek, V.W. (2016). Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT*.
12. Hurink, J.L., & Paulus, J.J. (2008). Online scheduling of parallel jobs on two machines is 2-competitive. *Operations Research Letters*, *36*(1), 51–56.
13. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K. (2014). Algorithm runtime prediction: methods & evaluation. *Artificial Intelligence*, *206*, 79–111.
14. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L. (2012). The international sat solver competitions. *AI Magazine*, *33*(1), 89–92.
15. Kautz, H.A. (2006). Deconstructing planning as satisfiability. In *IAAI* (pp. 1524–1526).
16. Krueger, P., Lai, T., Dixit-Radiya, V. (1994). Job scheduling is more important than processor allocation for hypercube computers. *IEEE Transactions on Parallel and Distributed Systems*, *5*(5), 488–497.
17. Lawler, E.L., Lenstra, J.K., Kan, A.H.R., Shmoys, D.B. (1993). Sequencing and scheduling: algorithms and complexity. *Handbooks in Operations Research and Management Science*, *4*, 445–522.
18. Leyton-Brown, K., Nudelman, E., Shoham, Y. (2009). Empirical hardness models: methodology and a case study on combinatorial auctions. *Journal of the ACM (JACM)*, *56*(4), 22.
19. Lynce, I., & Marques-Silva, J. (2006). SAT in bioinformatics: making the case with haplotype inference. In *SAT*.
20. Pinedo, M.L. (2016). *Scheduling: theory, algorithms, and systems*, 5th edn. Cham: Springer International Publishing.
21. Prasad, M.R., Biere, A., Gupta, A. (2005). A survey of recent advances in sat-based formal verification. *STTT*, *7*(2), 156–173.
22. Shen, S., Deng, K., Iosup, A., Epema, D. (2013). Scheduling jobs in the cloud using on-demand and reserved instances. In *European conference on parallel processing* (pp. 242–254). Springer.
23. Smith-Miles, K., & van Hemert, J.I. (2011). Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence*, *61*(2), 87.
24. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P. (2002). Characterization of backfilling strategies for parallel job scheduling. In *ICPP workshops* (pp. 514–522).
25. Sukhija, N., Malone, B., Srivastava, S., Banicescu, I., Ciorba, F.M. (2014). Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning. In *IPDPS workshops*.
26. Terekhov, D., Tran, T.T., Down, D.G., Beck, J.C. (2014). Integrating queueing theory and scheduling for dynamic scheduling problems. *Journal of Artificial Intelligence Research*, *50*, 535–572.
27. Thain, D., Tannenbaum, T., Livny, M. (2005). Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, *17*(2-4), 323–356.
28. Tian, J., Fu, R., Yuan, J. (2014). Online over time scheduling on parallel-batch machines: a survey. *Journal of the Operations Research Society of China*, *2*(4), 445–454.
29. Vielma, J.P. (2015). Mixed integer linear programming formulation techniques. *SIAM Review*, *57*(1), 3–57.
30. Witten, I.H., Frank, E., Hall, M.A. (2011). *Data mining: practical machine learning tools and techniques*, 3rd edn. San Mateo: Morgan Kaufmann.