Learning Sensitivity of RCPSP by Analyzing the Search Process

Marc-André Ménard , **Claude-Guy Quimper** and **Jonathan Gaudreault**
Université Laval, Québec, Canada
marc-andre.menard.2@ulaval.ca, {claude-guy.quimper, jonathan.gaudreault}@ift.ulaval.ca

Abstract

Solving the problem is an important part of optimization. An equally important part is the analysis of the solution where several questions can arise. For a scheduling problem, is it possible to obtain a better solution by increasing the capacity of a resource? What happens to the objective value if we start a specific task earlier? Answering such questions is important to provide explanations and increase the acceptability of a solution. A lot of research has been done on sensitivity analysis, but few techniques can be applied to constraint programming. We present a new method for sensitivity analysis applied to constraint programming. It collects information, during the search, about the propagation of the CUMULATIVE constraint, the filtering of the variables, and the solution returned by the solver. Using machine learning algorithms, we predict if increasing/decreasing the capacity of the cumulative resource allows a better solution. We also predict the impact on the objective value of forcing a task to finish earlier. We experimentally validate our method with the RCPSP problem.

1 Introduction

Scheduling problems are important for companies to execute their tasks on time and maximize their productivity. Once a schedule is computed, it is interesting to ask questions and to test different scenarios around this solution. For instance, how much time a company would save by increasing the capacity of a resource? Is it possible to complete a new order in time? What is the company's bottleneck? Which resource causes a slowdown in the production because? Answering these questions is important for continual improvement process. This corresponds to the sensitivity analysis of the solution. "The term *sensitivity analysis* refers to an analysis of the effect on the optimal solution to a linear programming problem of changes in the input-output coefficients, cost coefficients, and constant terms." [Dantzig, 1963]

The sensitivity analysis can be done by altering the model to encode a "what-if" scenario and solving the new problem. This process requires to solve a variant of the problem that was already solved which can be time consuming, especially if a distinct scenario is required for hundreds of resources or tasks. If the problem is a linear program, it is possible to use the dual solution to find a range of changes for the coefficients of the objective function or the constants on the right-hand side of the inequalities while keeping the optimality of the solution. In constraint programming, one can use the dual inference of an optimal solution to find the range of possible change of a parameter while keeping the optimality of the solution [Hooker, 1999]. The dual inference can be used to do the sensitivity analysis for any constraint satisfaction problem (CSP), integer programming, and mixed integer programming (MIP) [Hooker, 2006].

This article presents a method to predict the impact on the objective value of making changes to the solution or on constraints parameters. We test our method to predict whether increasing the capacity of a cumulative resource improves the solution and to predict the impact on the objective value of finishing a task earlier for a resource-constrained project scheduling problem (RCPSP). We do not just want to predict whether a change keeps the solution optimal, but how much it improves/deteriorates the solution. Particular care is given for making these predictions instantaneous, which prevent using a solver for conducting new optimization.

The method uses the information about the constraint CUMULATIVE [Aggoun and Beldiceanu, 1993] collected during the search and information about the resource usage in the solution found. It also uses information collected offline while solving similar instances obtained from historical data. Once a machine learning classifier is trained, it can quickly predict, for new instances of the problem, whether it is possible to obtain a better solution by increasing the capacity of the resource. This predictor does not need to run the solver a second time to evaluate "what if" scenarios. In fact, executing a trained classifier is instantaneous.

The rest of the paper is divided as follows. First, we present the background of this research. Second, we explain the methodology. Third, we present empirical results using a scheduling benchmark. Fourth, we conclude and present ideas for future work.

2 Background

We present background information about sensitivity analysis (what we want to do), lazy clause generation (with what we

work), and the resource-constrained project scheduling problem (our application).

2.1 Sensitivity Analysis

The sensitivity analysis is the study of the impact on the solution of changing the value of a parameter. This problem is widely studied in the literature [Geoffrion and Nauss, 1977; Greenberg, 1998].

Hall et al. [Hall and Posner, 2004] identify 4 questions that sensitivity analysis attempts to answer. What is the range of possible changes to each parameter to keep the solution optimal, i.e. what is the shadow price? What is the new objective value for a specific change of a parameter? What is the new optimal solution following a change to a parameter? What are the answers to the three previous questions if we simultaneously apply several changes to the parameter values? This paper focuses on the second question. Hall et al. present algorithms for the sensitivity analysis for some type of single-machine or identical parallel machines scheduling problems. They point out that it is more difficult to answer questions when the problem is NP-hard.

Hooker [Hooker, 1999] proposes an approach to do sensitivity analysis on linear or discrete problems. This method involves using the *dual inference* to obtain a proof of the optimality of the solution. Any optimization problem has a dual inference. Solving the dual inference of a problem consists in inferring the best possible bound on the optimal objective value from the constraints. With the proof found with the inference dual, it is possible to change the data of the problem as long as the proof remains valid. Hooker [Hooker, 1999] shows an example on a 0-1 linear programming problem. Dawande et al. [Dawande and Hooker, 2000] extend this idea to Mixed Integer linear programming problems. Dual inference has two limitations: it requires finding the optimal solution and exploring the entire search tree that proves that a better solution does not exist and it does not tell by how much the objective value changes if it does.

Hadzic et al. [Hadzic and Hooker, 2006] use binary decision diagrams (BDD) to enumerate solutions of a MIP. This can take time for big problems, but it allows to answer questions very quickly afterwards. It allows the user to answer questions about the right-hand side of an inequation but also about the variable domains. For instance, what is the best solution to the problem if a variable is set to a given value?

2.2 Lazy Clause Generation

Constraint solvers such as Chuffed [Chu, 2011] use the lazy clause generation [Stuckey, 2010] to take advantage of the high level modeling and understanding of the structure of the problem as well as the inference graph of SAT solvers. We recall how no-goods are generated since we collect information about these nogoods in order to perform sensitivity analysis.

In a solver using no-goods, propagators generate an explanation for each filtering. This explanation is represented as a clause which is a disjunction of literals. A literal is a Boolean variable which can be represented in the form $[\![X \leq v]\!]$ or $[\![X = v]\!]$ or its negation where X is a variable and v is a value. The literal $[\![X \leq v]\!]$ indicates that X is smaller than or equal to v. Consider the constraint $X + Y \leq 10$ with $\mathrm{dom}(X) = [2, 10]$ and $\mathrm{dom}(Y) = [2, 8]$. The filtering algorithm prunes the domain of X to $[2, 8]$ and generates the explanation $\neg[\![Y \leq 1]\!] \implies [\![X \leq 8]\!]$ (equivalent to a clause $[\![Y \leq 1]\!] \vee [\![X \leq 8]\!]$). When branching on variables and performing constraint propagation, the solver accumulates explanations in an implication graph. Upon failure, a cut in this graph generates a new clause, called *no-good*, that is added to the model. This new clause prevents the solver from performing a second time the branchings that led to the failure. In summary, a no-good is a clause computed from many filtering explanations generated by multiple constraints.

2.3 Resource-Constrained Project Scheduling Problem

The RCPSP has a set of non-preemptive tasks I and a set of cumulative resources R. A resource $r \in R$ has a capacity c_r and a task $i \in I$ consumes $h_{r,i}$ units of resource r. A task i executes for p_i units of time. One needs to find at what time s_i a task i can start without overloading the resources, i.e. $\sum_{i|s_i \leq t < s_i + p_i} h_{r,i} \leq c_r$ for all time t and resource r. Tasks might be subject to precedence constraints: $s_i + p_i \leq s_j$. We minimize the makespan. The model uses one CUMULATIVE constraint per resource [Baptiste and Pape, 2000].

3 Problem

We consider the problem of predicting the impact of changing a parameter of the problem on the objective value. Specifically, we answer these questions for a solution of the RCPSP:

1. By how much does the makespan change if one increases/decreases the capacity c_r of a resource r?
2. By how much does the makespan change if one forces a task i to complete before a deadline d?

As such questions can be raised for multiple resources and multiple tasks, we consider it impractical to run a solver of on a "what if" scenario for each of these tasks/resources. We allow running the solver once to obtain the initial solution of an instance. We also allow to do off-line computations on historical data. But once the solution is obtained from the solver, we should be able to ask the questions above and instantaneously obtain answers. Since these questions are NP-Hard to answer, one cannot expect to obtain exact answers in a short computation time, but rather approximations.

Answering these questions explains to the decider why the solver returned such a solution. E.g.: Why is completing a task so late necessary? It also helps determine what should be changed in the parameters of the problem in order to obtain better solutions. E.g.: Should the company sell unnecessary machines/resources in order to purchase missing ones?

4 Methodology

We train a machine learning classifier/predictor to predict the fluctuation of the makespan. We collect information on a collection of historical instances $\mathcal{H}$ similar to the ones on which predictions are made (e.g. same resources but different tasks). For each historical instance, we solve "what if" scenarios by changing the capacity of a resource. For an instance $H \in \mathcal{H}$, a resource $r \in R$, and a variation $\delta \in D$ for

$D = \{0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 1.5, 2\}$, we create a "what if" scenario $S^{\delta}_{H,r}$ equivalent to the instance H but for the capacity of the resource r set to $\delta \cdot c_r$.

We solve each scenario and record three types of features. The *search features* are collected during the search for a solution. The *solution features* are computed from the solution. The *instance features* are related to the instance such as the number of tasks and the capacity of the resources. All these features are presented in detail in the next sections.

We generate $|\mathcal{H}||R|\binom{|D|}{2}$ training examples for a classifier/predictor that predicts the variation of the makespan given the increase/decrease of a resource capacity. For each pair of scenarios $S^{\delta_1}_{H,r}$ and $S^{\delta_2}_{H,r}$ we use the search, solution, and instance features of scenario $S^{\delta_1}_{H,r}$. As an extra instance feature, we add the variation $\delta_1 - \delta_2$. The target value that needs to be predicted is given by the difference in the objective values (makespan) obtained for $S^{\delta_2}_{H,r}$ and $S^{\delta_1}_{H,r}$. If one rather wants to train a classifier that predicts whether the objective value changes or not, we use the label 0 when the objective values for $S^{\delta_1}_{r}$ and $S^{\delta_2}_{r}$ are equal and 1 otherwise.

To train a model that predicts the impact of imposing a deadline to a task, we construct the scenarios as follows. For each historical instance $H \in \mathcal{H}$, we create the original scenario $Z^{\infty}_{H,i}$. We solve this instance. For each task i that has no successor (the last task of a job), we randomly and uniformly select a deadline d between the earliest starting time of task i (taking into account the predecessors graph) and the starting time s_i in the solution of H. We add the constraint $s_i + p_i \leq d$ to the instance H and call this scenario $Z^{d}_{H,i}$.

For each pair of scenarios $Z^{\infty}_{H,i}$ and $Z^{d}_{H,i}$, we find a solution for both scenarios. We create a training example by using the search, solution, and instance features from scenario $Z^{\infty}_{H,i}$. We use, as an extra instance feature, the difference between the ending time of task i in the solution of S^{∞}_{i} and the deadline d that is imposed to the task in scenario S^{d}_{i}. The target value is the difference in the makespan if we train a predictor and a Boolean value if we only want to classify whether the makespan changes or not.

Creating the training data might take time as it requires to solve multiple instances, but once the classifier is trained, it can quickly predict whether increasing the capacity of a resource or imposing a deadline has an effect on the makespan.

4.1 Search Features

As the solver solves an instance, whether it is a scenario created from historical data or the actual instance on which one wants to perform sensitivity analysis, it needs to collect information about the search process. Since the RCPSP can easily be modeled using one CUMULATIVE constraint per resource, it is natural to collect statistics about the activity of this constraint. The intuition is that a CUMULATIVE constraint associated to a scarce resource is more often violated and causes more filtering during the search than an abundant resource.

We use two filtering algorithms for the CUMULATIVE constraint: the time-tabling [Beldiceanu and Carlsson, 2002] and the time-tabling-edge-finder [Vilím, 2011]. Other algorithms could also be used, but these two algorithms were already implemented in the solver Chuffed. For each resource r and for each filtering algorithm associated to its CUMULATIVE constraint, we define three search features:

1. the number of filtered values;
2. the number of times a variable domain gets filtered;
3. the average number of filtered values, i.e. the ratio of the two previous features.

It is important to link a no-good to the constraints that generated it. Without this link, information about the constraints is lost since no-goods can end up performing the majority of the filtering. We link a no-good to all the constraints that provided an explanation that contributed to its generation. To achieve this, we modified the solver to create a map between an explanation and the constraint that generated it. We link the no-good to the constraints by checking the explanations used to generate the no-good and the constraints that generated these explanations. If a no-good b is generated from an explanation of a no-good a, we bind the no-good b to the constraints that generated the no-good a. In the end, if a no-good induces a filtering or a backtrack, it is possible to give credits to the constraints that inferred that no-good. The activity of a no-good is the number of times that its explanations are used to produce another no-good, hence this search feature.

4. The sum of the activity of all no-goods linked to the CUMULATIVE constraint of resource r.

We collect information specific to the filtering of the objective value. For each resource r, we create an initialy empty list L_r. If the lower bound of the objective variable gets filtered to v, we analyze the implication graph that contains the explanations for this filtering. If one of these explanations is mapped to the CUMULATIVE constraint of the resource r or a no-good linked to this CUMULATIVE constraint, we insert the value v to the list L_r. Let M be the makespan of the solution returned by the solver at the end of the search. For each resource r, we have the three following search features.

5. The smallest value $\min(L_r) - M$;
6. The largest value $\max(L_r) - M$;
7. The average value $\text{avg}(L_r) - M$;

When training a model to predict how much the makespan changes when increasing/decreasing the capacity of a resource r, we only use the features related to the resource r.

When training a model to predict how much the makespan changes after imposing a deadline to a task i, we use the search features related to all resources. Moreover, we collect information about the filtering of the starting time variable s_i. These features are similar to features 1, 2, and 3 but specific to the variable s_i that is the focus of the prediction.

8. The number of times a value from $\text{dom}(s_i)$ is filtered;
9. The number of times $\text{dom}(s_i)$ is filtered;
10. The average number of filtered values from $\text{dom}(s_i)$.

The next three features are inspired from the last three features. Rather than reporting the number of times a filtering occurs, we report how the variable ranks compared to other starting time variables. For instance, if s_i is the starting time

variable that gets filtered the least often, the rank is 1. If it is the second least filtered variable, the rank is 2 and so on.

11. The rank with respect to how often s_i is filtered;
12. The rank with respect to how many values are filtered;
13. The rank with respect to the average number of filtered values when the variable gets filtered.

4.2 Solution Features

We design 5 features computed from the solution returned by the solver. The *utilization rate* U_r^{rate} of the resource r is the amount of energy $p_i \cdot h_{r,i}$ consumed by the tasks over the availability of the resource $c_r \cdot M$, where M is the makespan.

$$U_r^{rate} = \frac{\sum_{i \in I} p_i \cdot h_{r,i}}{c_r \cdot M} \tag{1}$$

The *saturation duration* U_r^{max} for a resource r is the time for which it is used at full capacity. Let $U_{r,t}$ be the usage of r at time t, we have:

$$U_{r,t} = \sum_{i \in I : s_i \leq t < s_i + p_i} h_{r,i}, \qquad U_r^{max} = |\{t \mid U_{r,t} = c_r\}| \tag{2}$$

The *maximum usage ratio* $U_r^{max\ ratio}$ of a resource r is the ratio between U_r^{max} and the makespan M.

$$U_r^{max\ ratio} = \frac{U_r^{max}}{M} \tag{3}$$

The *number of waiting tasks* W_r counts the tasks waiting after resource r to start, i.e. the tasks that could have started earlier if r was available. To get this information, we must make sure that the task is not waiting for a predecessor to complete. Let $pred_i$ be the set of predecessors of task i and ρ_i the time at which the last predecessor of i finishes.

$$\rho_i = \max_{j \in pred_i} (s_j + p_j)$$
$$W_r = |\{i \in I \mid s_i \neq \rho_i \wedge U_{r,s_i - 1} + h_{r,i} > c_r\}| \tag{4}$$

The *waiting time* $W_r^{duration}$ of resource r is the amount of time the tasks wait after resource r to start. Equation (5) checks if the task could be scheduled earlier, i.e. after its last predecessor but before the current starting time. If it is the case, we conclude the task does not wait after the resource r (it probably waits after another resource) and the waiting time of task i for resource r is set to $W_{r,i}^{duration} = 0$. Otherwise, we impute a waiting time of $s_i - \rho_i$ to the resource r, i.e. the time between the last predecessor finishes and the starting time of task i. Finally, equation (6) computes the total waiting time and it is that quantity that is used as the solution feature.

$$W_{r,i}^{duration} = \begin{cases} 0 & \text{if } \exists t \in [\rho_i, s_i) \forall k \in [0, p_i) \\ & \quad U_{r,t+k} + h_i \leq c_r \vee t + k \geq s_i \\ s_i - \rho_i & \text{otherwise} \end{cases} \tag{5}$$

$$W_r^{duration} = \sum_{i \in I} W_{r,i}^{duration} \tag{6}$$

Like for the search features, we only use solution features related to the resource r when we train a model to predict how much the makespan changes when increasing/decreasing the capacity of a resource r.

When training a model to predict how much the makespan changes after imposing a deadline to a task i, we use the solution features related to all resources in the problem. Moreover we add a solution feature indicating how many tasks finish before the task i. And a feature that is simply the makespan.

4.3 Instance Features

We use as instance features:

1. The number of tasks n;
2. The original capacity c_r of each resource r;

When training the model to predict how much the makespan changes when increasing/decreasing the capacity of a resource, we use these features.

3. The new capacity of the resources;
4. The ratio of increase/decrease of the resource capacity.

When training the model to predict how much the makespan changes when imposing a deadline to a task, we add as an instance feature:

5. The difference between the ending time in the original scenario and the deadline in the second scenario.

5 Experiments

We use the RCPSP benchmarks PSPLib [Kolisch and Sprecher, 1997] and Pack [Carlier and Néron, 2003]. PSPLib has instances of 30, 60, 90, and 120 tasks and 4 resources. The Pack benchmark has instances between 17 and 32 tasks and between 2 and 5 resources. There are more precedence constraints in PSPLib than Pack. We use the model provided by Minizinc [Stuckey *et al.*, 2014]. We use the time-tabling and time-tabling-edge-finder as filtering rules for the CUMULATIVE constraint. We use a timeout of 10 minutes per instance for PSBLib and 3 hours for Pack since Pack has harder instances. Reaching the timeout can add noise to the dataset since the results become solver dependant, but this situation is likely to occur in a real context.

We randomly separate the instances of a benchmark into a training and a testing set with ratio 80/20. Since we produce many scenarios from one instance, we end up producing many training examples from training instances and many testing examples from testing instances. In no case is the testing data contaminated with the training data or vice versa. Solving theses instances require 80 cpu-days for PSPLib and 48 cpu-days for Pack. As the results show later, not that many instances are necessary to obtain good results. We apply a min-max normalization on all features to scale them between 0 and 1 using the relation $x_i' = \frac{x_i - \min(\vec{x})}{\max(\vec{x}) - \min(\vec{x})}$.

We use the random forest classifier and the random forest regressor from Scikit-Learn [Pedregosa *et al.*, 2011]. We use the default parameters except for the number of trees (nEstimators) for which we set the value to 100.

We evaluate the quality of our predictions given which features are used and how many training examples are used. As it can take a lot of time to solve the "what if" scenarios, we

want to know the number of examples necessary in the training set to obtain sufficiently good predictions. We compare the accuracy and the f1-score for the classification problem and the mean squared error for the regression problem.

We start by training on 100 randomly chosen examples in the training set and make the predictions on the test set. Then we add 100 more randomly selected examples from the training set and make the predictions on the test set. We do so until we consider all examples in the training set or up to 10,000 examples. We repeat this process 100 times with different training sets and test sets and take the average of the results. We also calculate a 95% confidence interval to ensure that our results do not depend on the training sets or test sets chosen.

5.1 Predicting the Makespan When Changing a Resource Capacity

Figure 1 shows the accuracy for the classification problem (i.e. predicting whether the makespan changes or not) according to the number of examples in the training set and the features used for the PSPLib benchmark. For the PSPLib benchmark, 54% of the examples have no change to the makespan (label 0). We compare four groups of features: instance features with solution features (green), instance features with search features (red), all features (blue), and the top 5 features (yellow). The classifier gets better accuracy when using all features, reaching 91.14% with 10,000 examples obtained from solving 892 scenarios. The accuracy of the classifier is similar if it uses the solution features with instance features (88.18%) as it does when using the search features with instance features (88.36%). When using all the features, we obtain an accuracy of 87.32% with 1,000 examples and 90% with 3000 examples. We obtain an f1-score of 90.18% with all features, 86.89% with solution+ instance features and 87.2% with search+instance features.

These results show that the classifier can properly predict whether changing the capacity of the resource affects the makespan. All types of features contribute to the quality of the prediction. We did a feature ranking with recursive feature elimination (RFE) to find the 5 most important features for the random forest classifier. The five most important features are the increase/decrease in the capacity of the resource, the smallest value $\min(L_r) - M$, the utilization rate U_r^{rate}, the sum of the activity of all no-goods linked to the CUMULATIVE constraint, and the new capacity of the resource.

Figure 2 shows the accuracy and f1-score for the classification problem for the Pack benchmark. These instances are harder to solve and have 63% of examples with no change to the makespan (label 0). The classifier gets an accuracy of 90.69% with all features and 10,000 training examples which is slightly better than using only the solution+instance features (89.86%) or the search+instance features (85.41%). We obtain a f1-score of 87.41% when using all features, 86.37% with the solution+instance features, and 80.43% with the search+instance features. For Pack, the solution features seem to provide more information than the search features.

Figure 3 shows the mean squared error for the regression problem. For the PSPLib benchmark, with 10,000 examples in the training set, the regressor obtains a mean square error of 38.24 with all features, 41.9 with the solution+instance

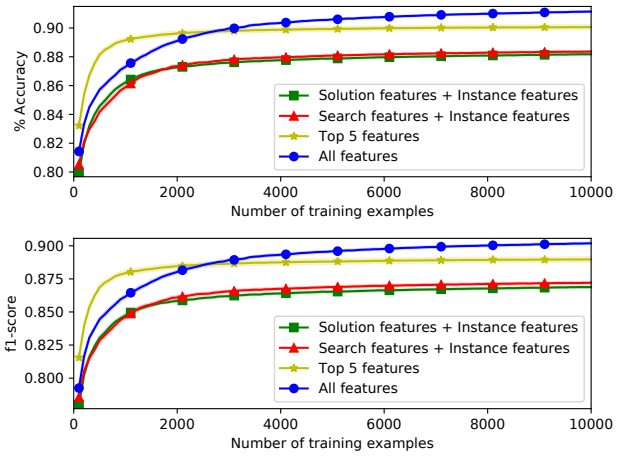

Figure 1: Does changing the resource capacity modify the makespan? Accuracy and f1-score for the classifier trained on the PSPLib benchmark.

features and 87.67 with search+instance features. For the Pack benchmark, with 10,000 examples, the regressor obtains a mean square error of 74.58 with all features, 75.41 with solution+instance features and 126.13 with search+instance features. For both benchmarks, the solution features provide more useful information than the search features. Although search features do help obtaining better results. Interestingly, the regressor made no error in its prediction for 48.98 % of the testing examples for PSPLib and 64.47 % for Pack.

5.2 Predicting the Makespan When Imposing a Deadline

For the experiments of predicting the impact on makespan of imposing a new deadline to a task, we have fewer training examples than for predicting the makespan when modifying the capacity of a resource. Also, for the Pack benchmark, for a large majority of tasks, the makespan remains unchanged when we constrain the task to finish earlier. For that reason, the Pack benchmark is less interesting and we omit results for this benchmark. This is not the case for the PSPLib benchmark. 67.17% of examples are labeled with 0. Tasks often cannot finish earlier due to the precedence constraints. There is therefore often no impact on the makespan to finish a task earlier. To rebalance the training set, we duplicate the examples with a label of 1. The average change in makespan after imposing a deadline on a task is 0.69 and the maximum is 17. Even though there are many examples with a label of 0, this benchmark is more interesting for evaluating our method.

Figure 4 shows the mean squared error for the regression problem for the PSPLib benchmark. We replicated these experiments 250 times to better see the difference between the different types of features. With 3,250 examples in the training set, the regressor obtains a mean square error of 3.51 when using all features, 3.57 with solution+instance features, and 3.55 with search+instance features. There is not much difference between using a group of features than another. But using all feature provides the best predictions.

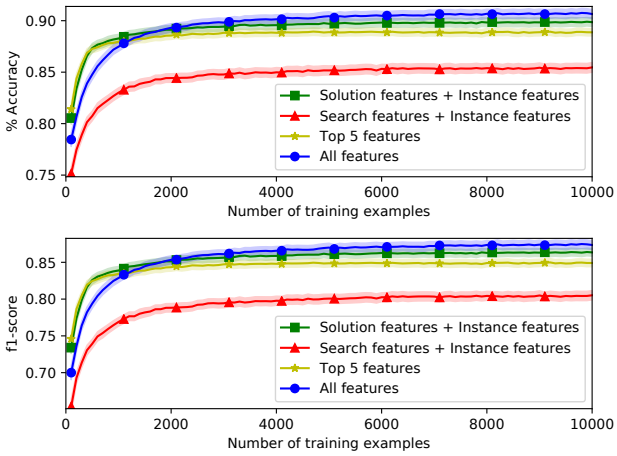

Figure 2: Does changing the resource capacity modify the makespan? Accuracy and f1-score for the classifier trained on the Pack benchmark

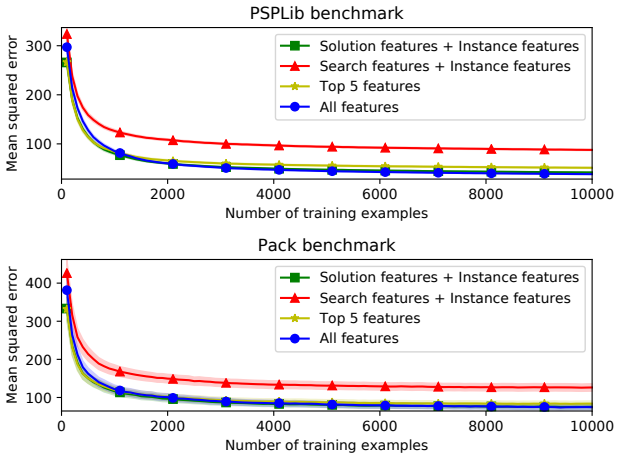

Figure 3: By how much does the makespan change if one changes the capacity of a resource? Mean squared error of the regressors.

Figure 5 shows the distribution of the error when the regressor trains on 3,250 examples. The regressor makes no error or only small errors for a majority of its predictions.

Figure 6 shows the accuracy and the f1-score for the classification problem for the PSPLib benchmark. Predicting a change on the makespan when imposing a deadline on a task seems more difficult than when changing the capacity of the resource. Indeed, the CUMULATIVE constraint provides much information about the impact of the capacity of the resource. No constraint in the original model provides information about the deadline, since there is initially no deadline. Moreover, the smaller dataset and especially the lack of examples with change to the makespan (label 1) makes it harder to train. We notice the search features provide more information than the feature solutions. However, it is by training on all the features that we obtain the best classifying results.

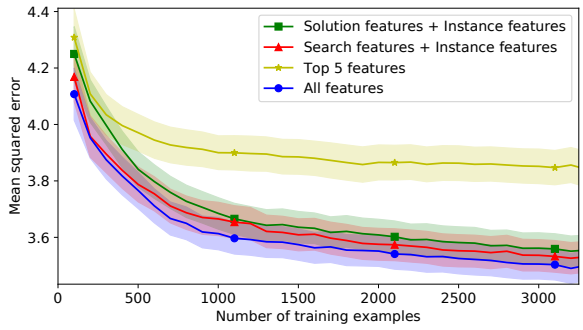

Figure 4: By how much does the makespan change after imposing a deadline? Mean squared error of the regressor trained on PSPLib

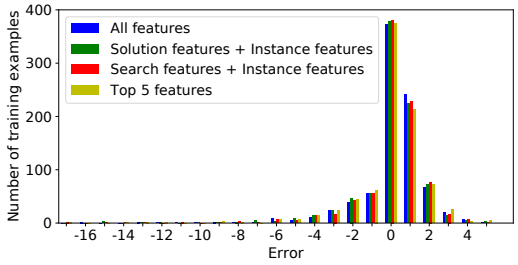

Figure 5: By how much does the makespan change after imposing a deadline? Error distribution of the regressor trained on PSPLib

6 Conclusion

We presented a method to predict how the makespan of a schedule is affected when the capacity of a resource is modified or a deadline is imposed on a task. Such predictions can help a decider to sell unnecessary resources to buy scarcer resources. It also tells whether a task could complete earlier. Our method takes advantage of the structure of a solution, but also some events that occur during the search for this solution.

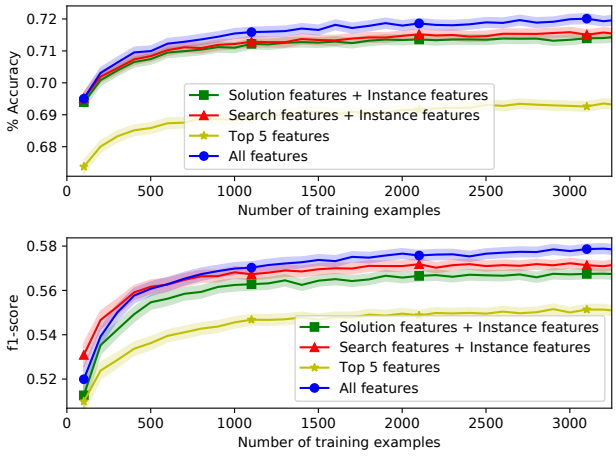

Figure 6: Does imposing a deadline to a task change the makespan? Accuracy and f1-score of the classifier trained on PSPLib

References

[Aggoun and Beldiceanu, 1993] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57 – 73, 1993.

[Baptiste and Pape, 2000] Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1):119–139, Jan 2000.

[Beldiceanu and Carlsson, 2002] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 63–79, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Carlier and Néron, 2003] Jacques Carlier and Emmanuel Néron. On linear lower bounds for the resource constrained project scheduling problem. *European Journal of Operational Research*, 149(2):314–324, 2003.

[Chu, 2011] Geoffrey Chu. Improving combinatorial optimization. PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

[Dantzig, 1963] George B. Dantzig. Linear programming and extensions. *Princeton University Press*, 1963.

[Dawande and Hooker, 2000] Milind Dawande and John N. Hooker. Inference-based sensitivity analysis for mixed integer/linear programming. *Operations Research*, 48(4):623–634, 2000.

[Geoffrion and Nauss, 1977] Arthur M. Geoffrion and Robert Nauss. Exceptional paper—parametric and postoptimality analysis in integer linear programming. *Management Science*, 23(5):453–466, 1977.

[Greenberg, 1998] Harvey J. Greenberg. *An Annotated Bibliography for Post-Solution Analysis in Mixed Integer Programming and Combinatorial Optimization*, pages 97–147. Springer US, Boston, MA, 1998.

[Hadzic and Hooker, 2006] Tarik Hadzic and John N. Hooker. Postoptimality analysis for integer programming using binary decision diagrams. In *GICOLAG Workshop (Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry), Vienna. Technical report, Carnegie Mellon University*, 2006.

[Hall and Posner, 2004] Nicholas G. Hall and Marc E. Posner. Sensitivity analysis for scheduling problems. *Journal of Scheduling*, 7(1):49–83, Jan 2004.

[Hooker, 1999] John N. Hooker. Inference duality as a basis for sensitivity analysis. *Constraints*, 4(2):101–112, May 1999.

[Hooker, 2006] John N. Hooker. Duality in optimization and constraint satisfaction. In J. Christopher Beck and Barbara M. Smith, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 3–15, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Kolisch and Sprecher, 1997] Rainer Kolisch and Arno Sprecher. Psplib - a project scheduling problem library: Or software - orsep operations research software exchange program. *European Journal of Operational Research*, 96(1):205 – 216, 1997.

[Pedregosa *et al.*, 2011] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[Stuckey *et al.*, 2014] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, Jun. 2014.

[Stuckey, 2010] Peter J. Stuckey. Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 5–9, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Vilím, 2011] Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In Tobias Achterberg and J. Christopher Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 230–245, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.