Embedding Decision Trees and Random Forests in Constraint Programming

Alessio Bonfietti, Michele Lombardi, and Michela Milano

DISI, University of Bologna
{alessio.bonfietti,michele.lombardi2,michela.milano}@unibo.it

Abstract. In past papers, we have introduced Empirical Model Learning (EML) as a method to enable Combinatorial Optimization on real world systems that are impervious to classical modeling approaches. The core idea in EML consists in embedding a Machine Learning model in a traditional combinatorial model. So far, the method has been demonstrated by using Neural Networks and Constraint Programming (CP). In this paper we add one more technique to the EML arsenal, by devising methods to embed Decision Trees (DTs) in CP. In particular, we propose three approaches: 1) a simple encoding based on meta-constraints; 2) a method using attribute discretization and a global TABLE constraint; 3) an approach based on converting a DT into a Multi-valued Decision Diagram, which is then fed to an MDD constraint. We finally show how to embed in CP a Random Forest, a powerful type of ensemble classifier based on DTs. The proposed methods are compared in an experimental evaluation, highlighting their strengths and their weaknesses.

1 Introduction

Combinatorial Optimization methods have been successfully applied to a broad range of industrial problems. Many of such approaches rely on the availability of some declarative model describing decisions, constraints on these decisions, their cost and their impact on the considered system. In short, they rely on an accurate problem model. However, in some application domains, the model is either not fully known, or it is described in a way that is not useful for combinatorial optimization. As an example, for many domains there are predictive models to forecast the temporal dynamic of a target system via differential equations, but those are unfortunately impossible to insert into a combinatorial optimization model without incurring in computational issues.

In these cases, it is likely that the domain expert proposes some heuristic knowledge on the problem that is a (non measurable) approximation of the effect that some decisions have on the system dynamic. We propose here an alternative approach, which is an instantiation of a general method called Empirical Model Learning that we proposed in [2]. We aim at learning part of the combinatorial optimization model and to embed the learned piece of knowledge in the combinatorial model itself. In this way, we have two advantages: the first

is that we can use this knowledge to reduce the search tree and the second is that we know the accuracy that we have obtained in the process.

In this paper, we consider the problem of embedding in a CP model two types of tree-based classifiers from Machine Learning, namely Decision Trees and Random Forests. Formally, a classifier is a function f mapping a tuple of values for a set of discrete or numeric *attributes* to a discrete *class*. We can embed a classifier in CP by introducing a vector of variables $\overline{x}$ to represent the attributes and a variable y to represent the class. Then we need to find a set of constraints such that they *guarantee* and *enforce some degree of consistency* on the relation:

$$\overline{x} = \overline{v} \wedge y = w \Leftrightarrow f(\overline{v}) = w \tag{1}$$

In other words, an effective embedding technique does not simply act as a function evaluator. Rather, it is capable to narrow the set of possible values for y given the current domain of $\overline{x}$ and vice-versa, i.e. it is capable of performing domain filtering.

Here, we show three CP encoding techniques for Decision Trees and Random Forests, in particular: 1) an approach based on rules and modeled via meta-constraints; 2) an encoding based the discretization of numeric attributes and a global TABLE constraint; 3) another approach relying on attribute discretization, but making use of an MDD constraint instead of TABLE. Each of the three approaches has its own merits: the rule-based encoding has the best scalability, but provides the weakest propagation. The TABLE and MDD approaches are both capable of enforcing GAC, but may suffer from scalability issues when dealing with large and complex trees.

We experiment our methods on a thermal-aware workload dispatching problem over an experimental multicore CPU by Intel, called Single-chip Cloud Computer (SCC, see [19]). Our goal is to map jobs so as to maximize the number of cores operating at high efficiency. The efficiency of each core depends on a number of complex factors, making it impossible to assess the effect of a job mapping via a traditional, expert designed, model. Hence, we obtained a model approximation by learning a set of Decision Trees (or Random Forests), each one trained to predict if a specific core will have high (class 1) or low (class 0) efficiency given a specific workload. We compare the behavior of the proposed techniques in a variety of conditions and we show how the EML approach is capable of providing improvements over a powerful Local Search method.

2 Background

In this section we discuss the basics of Decision Trees (DT) and Random Forests (RF), so as to establish the background to present our encoding techniques. A brief review of works that combine optimization and Machine Learning is provided in Section 5.

Decision Trees are a type of Machine Learning model typically employed for classification tasks. Each leaf of a Decision Tree (DT) is labeled with a *class*.

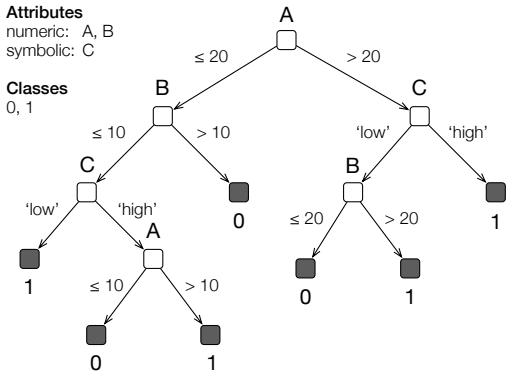

Fig. 1. Example of a simple Decision Tree.

Each node is labeled with one of a set of numeric or symbolic *attributes* that are used to described the DT input. The outgoing branches of a node are labeled with conditions over its attribute. The conditions are such that they form a partition of the attribute domain: in particular, branches over symbolic attributes are labeled with a set of symbolic values, while branches over numeric attributes are labeled with a splitting condition such as $x_i \leq \theta$ or $x_i > \theta$. An example of a simple Decision Tree is depicted in Figure 1.

Trees can be learned in a greedy manner by procedures like the C4.5 algorithm [26]. The learning process starts from a set of *examples*, i.e. (tuples of attribute values, associated with a class). Then, this training set is recursively split into subsets, according to the attribute that makes the classes in the subsets most homogeneous (e.g. that achieves minimum Gini index maximum Information Entropy). The recursive process terminates when the remaining subsets are sufficiently pure to conclude with a classification, which is then associated with a leaf of the tree. A new example is classified by starting from the root node and traversing the tree, always taking the branches whose condition is satisfied by the values of the example attributes. For a fully specified example, this process will lead to a single leaf, corresponding to the predicted class.

Decision trees are quick to train and easy to understand, they can provide class probabilities and error bars, they can handle wrong or missing attribute values. On the downside, they require relatively large training sets in order to be effective and they do not always reach satisfactory accuracy levels.

The last drawback can be overcome by using DTs in an ensemble learning method, leading to Random Forests [10, 18]. A Random Forest is a set of DTs and the forest output is the statistical mode of the classifications made by its components, i.e. the class predicted by the majority of the trees. Each DT is defined over a random subset of attributes and trained on a subset of the original examples (bagging). Additionally, randomization can also be employed for selecting the splitting value at each tree root. Such extensive use of randomization aims at breaking correlations between the trees, which greatly increases the

prediction ability of the forest. Random Forests are widely considered among the most powerful Machine Learning models.

3 Embedding Decision Trees and Random Forests in CP

In the following paragraphs we describe several techniques to guarantee the satisfaction of and to enforce consistency on Equation (1) from Section 1, i.e.:

$$\overline{x} = \overline{v} \wedge y = w \Leftrightarrow f(\overline{v}) = w \tag{2}$$

assuming the f is a Decision Tree or a Random Forest.

On this purpose, it is useful to introduce some formal notation. For sake of uniformity with other graphical structures mentioned in this paper, we view a DT as a (tree structured) directed acyclic graph $T = \langle N, A \rangle$. N is the set of the nodes n_i and A is the set of arcs a_j. The source and sink endpoints of an arc are referred to as $src(a_j)$ and $snk(a_j)$. Each node is associated to an attribute $x(n_i)$ and we refer as $x(a_j)$ to $x(src(a_j))$. Each arc is associated to a set $\lambda(a_j)$, called the arc *label*. The labels correspond to the arc conditions: an arc associated to the condition $x(a_j) \in \{0, 2\}$ has the label $\{0, 2\}$ (symbolic values can always be associated to integers); an arc with condition $x(a_j) \leq 3$ has the label $]-\infty, 3]$. Arcs having the same source always have disjoint labels. The leaf nodes of the DT are associated to a class from a set of classes $\mathcal{C}$.

3.1 Rule Based Encoding

A DT can be converted to a set of classification rules by interpreting each path from root to leaf as a logical implication. A simple approach to encode a DT in CP consists in translating each implication into a boolean meta-constraint.

Formally, let $\mathcal{P}$ be the set of root-to-leaf paths π in the tree, each path π being a sequence of arcs indices $\pi(0), \ldots, \pi(k)$ such that $snk(a_{\pi(i)}) = src(a_{\pi(i+1)})$ for all $i = 0, \ldots k-1$. Each path ends in a leaf node with a certain class, denoted here as $class(\pi)$. Then the DT can be encoded as a set of constraints:

$$\bigwedge_{\pi(i) \in \pi} \left[\left[x(a_{\pi(i)}) \in \lambda(a_{\pi(i)})\right]\right] \Rightarrow \left[\left[y = class(\pi)\right]\right] \qquad \forall \pi \in \mathcal{P} \tag{3}$$

where the notation $[[-]]$ refers to the reification of the constraint enclosed by the brackets. The notation $x(a_j)$ refers here to an attribute *variable* (attributes and attributes variables will often be considered interchangeable). If $x(a_{\pi(i)})$ is numeric, then the reified constraint $\left[\left[x(a_{\pi(i)}) \in \lambda(a_{\pi(i)})\right]\right]$ is defined as:

$$\left[\left[x(a_{\pi(i)}) \leq \theta\right]\right] \qquad \text{if } \lambda(a_{\pi(i)}) \text{ is in the form }]-\infty, \theta] \tag{4}$$

$$\left[\left[x(a_{\pi(i)}) > \theta\right]\right] \qquad \text{if } \lambda(a_{\pi(i)}) \text{ is in the form }]\theta, \infty[\tag{5}$$

while the constraint form for symbolic attributes is:

$$\bigvee_{\theta_j \in \lambda(a_{\pi(i)})} \left[\left[x(a_{\pi(i)}) = \theta_j\right]\right] \tag{6}$$

This encoding approach can be strengthened by observing that if the class variable takes a specific value, then one of the corresponding root-to-leaf paths in the DT must necessarily be true. This leads to this second, stronger, encoding:

$$[[y = w]] \Leftrightarrow \bigvee_{\substack{\pi \in \mathcal{P} \,:\\ class(\pi)=w}} \bigwedge_{\pi(i)\in\pi} \left[\left[x(a_{\pi(i)}) \in \lambda(a_{\pi(i)})\right]\right] \qquad \forall w \in \mathcal{C} \qquad (7)$$

which will serve as our reference rule-based encoding in the paper.

On solvers that do not provide support for logical constraints, sums can be used instead of $\vee$, multiplications instead of $\wedge$, and implications can be modeled as $\leq$ relations. A double implication such as the one in Equation (7) must be modeled using separate $\leq$ constraints for the two implication directions.

The rule-based encoding from Equation (7) is defined using a number of constraints that grows linearly with the number of leaves and logarithmically with the DT depth, yielding a space complexity of $O(|N| \log |N|)$. The encodings is simple, scalable, and easy to implement, but provides only a weak form of propagation. It makes therefore sense to investigate methods for embedding DTs in CP that strike a difference balance between propagation power and cost.

3.2 Table Based Encoding

The TABLE constraint in CP can be used to define a constraint in extensional form, i.e. by enumerating the allowed (or forbidden) tuples for its variables. The rules employed by our first encoding (corresponding to paths in the DT) are related to tuples in a TABLE constraint in that they specify conditions over the attribute and class variables. It makes therefore sense to investigate the possibility to embed a Decision Tree in CP using TABLE. For this to be possible, there are four important issues that should be addressed.

First, *(#1) DTs extracted via Machine Learning may feature numeric attributes*, for which enumerating the possible values is, strictly speaking, impossible. A straightforward solution consists in using integer variables to encode numerical attributes with finite precision. In fact, with many CP solver this is a mandatory step, since real valued variables are often not supported. However, with this form of discretization the variables corresponding to numeric attributes end up having a very large domain, dramatically reducing the scalability.

We address the issue via an interval-based discretization. Specifically, we traverse the DT and collect for each numeric attribute all the splitting thresholds. Let $\overline{\theta}_i$ be the sorted vector of the thresholds θ_i^k for a given numeric attribute x_i, to which we always add the values $\pm\infty$. For example, $\overline{\theta}_A = [-\infty, 10, 20, \infty]$ for DT from Figure 1. Then, we introduce a new integer variable $\delta(x_i)$ with domain $\left\{0, \ldots |\overline{\theta}_i| - 2\right\}$ such that:

$$\delta(x_i) = k \Leftrightarrow \theta_i^k < x_i \leq \theta_i^{k+1} \qquad (8)$$

Equation (8) can be enforced via specific channeling constraints (if supported by the solver) or by using reification. Then, we update the numeric labels in the

DT with the substitution:

$$\lambda(a_j) = \left\{k \in D(\delta(x_i)) : \ ]\theta_i^k, \theta_i^{k+1}] \subseteq \lambda(a_j)\right\} \tag{9}$$

Once this is done, we can use $\delta(x_i)$ as a replacement for x_i in the encoding.

Then, *(#2) arcs in the DT are labeled with sets* and each path on the tree corresponds to a *set* of tuples rather than to a single one. This problem can be (inefficiently) addressed by generating all possible combination of values for the attribute and class variables, and removing those that violate Equation (1). A better approach would be to generate directly the set of feasible tuples, which is intuitively related to the cartesian product of the path labels.

However, this requires some care, because *(#3) a path in a DT obtained via Machine Learning can specify multiple conditions on the same attribute*, which makes a straightforward computation of the cartesian product $\prod_{\pi(i)} \lambda(a_\pi(i))$ meaningless. However, the problem can be easily fixed by replacing all the labels defined over a specific attribute with their intersection. Formally, we introduce the term *refined label* to refer to:

$$L(\pi, x_i) = \bigcap_{\substack{\pi(i) \in \pi \ : \\ x(a_{\pi(i)}) = x_i}} \lambda(a_{\pi(i)}) \tag{10}$$

Even with the refined labels, the set of allowed tuples cannot be formulated as a cartesian product, since *(#4) a path in the DT may specify no label for one or more attributes*. When this happens, it means that such attributes are irrelevant for the associated classification. Speaking in terms of tuples, this means that all possible completions obtained by assigning values to the missing attributes are feasible. In other words, the set of all allowed tuples corresponding to a path π is given by the following cartesian product:

$$\{class(\pi)\} \times \prod_{x_i \in \overline{x}} \begin{cases} L(\pi, x_i) & \text{if a label over } x_i \text{ exists in } \pi \\ D(x_i) & \text{otherwise} \end{cases} \tag{11}$$

where $D(x_i)$ is the domain of the attribute variable x_i. We have a polynomial number of products in the form of Equation (11), each one compactly representing a set of allowed tuples and corresponding to a c-tuple in the terminology of [23]. In that paper the authors introduce an algorithm to enforce GAC on a table constraint formulated by using c-tuples rather then regular tuples. We plan to test such method for future research, while in this work we focus on investigating a more classical approach, namely using Equation (11) to generate all the allowed tuples for a "traditional" TABLE constraint. This method may have limited scalability, but it is readily applicable on off-the-shelf available solvers.

3.3 MDD Based Encoding

Graphical structures closely related to decision trees are employed by propagators for several global constraints. In [13], the authors propose a filtering algorithm that achieves GAC on TABLE and is based on a *trie*. A trie is a particular

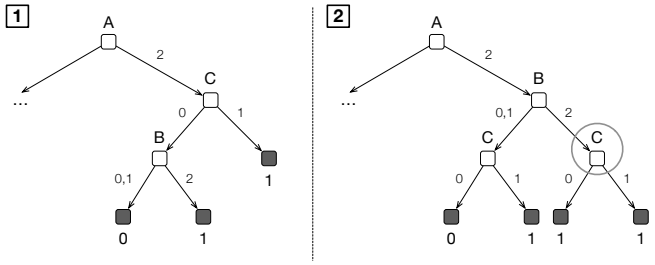

Fig. 2. Effect of attribute reordering.

type of decision tree where the attributes are always 1) discrete, 2) considered exactly once, and 3) in the same order.

The approach from [13] is improved in [11] by converting the trie into a MDD, which is then used to define an efficient GAC propagator. From the purpose of this paper, an MDD can be considered as a trie generalization, where the requirement to have a tree structure is relaxed and multiple arcs are allowed to point to the same node. Thanks to this ability, an MDD can be considerably more compact than a trie, reducing the complexity of enforcing GAC. This raises interest in using MDDs to encode Decisions Trees and perform propagation.

Both MDDs and tries differ from the Decision Trees employed in Machine Learning (after the attribute discretization) in two important respects: 1) DTs can consider the same attribute multiple times and 3) in DTs attributes can be considered in a different order along different paths[1]. Such differences have deep consequences when trying to encode a DT. In fact, the encoding requires to re-order attributes, which causes the graph size to grow. Consider Figure 2.1, which shows a portion of the DT from Figure 1 after the attribute discretization. Assume that the selected attribute order is A, B, C. Moving the "B" split to the correct position requires to copy part of the graph, as shown in Figure 2.2. The phenomenon is multiplicative, possibly leading to an exponential growth. MDD reduction algorithms can mitigate the issue by sharing identical graph portions (e.g. the children of the circled node in Figure 2.2), but as of today we have no proof that exponential growth can be avoided, and some evidence that it actually occurs in practice.

Our approach for encoding a DT into an MDD is similar to the one from [11]: we construct a trie-like structure, which is then reduced using the *mddReduce* algorithm. The term "trie-like" is used because we allow multiple outgoing arcs of a node to point to the same child, as an (effective) measure for mitigating the graph expansion: strictly speaking, this make our structure already an MDD.

The algorithm starts from the set of all c-tuples obtained via Equation (11) and assumes that the attribute order has been pre-specified. We recall that in this

[1] Additionally, DTs can skip attributes that are irrelevant for a specific path, which may lead to an incorrect behavior of the MDD propagator from [11]. Despite this, skipped attributes are still easy to handle using MDDs.

Algorithm 1 build_trie(c-tuples, pos, [parent], [label])

```
1: node = a new MDD node
2: if parent is defined then
3:    for v ∈ label do build an arc from parent to node with label v
4: if pos = |x̄| then return node
5: L = ∅, D = the set of all values of x_pos in the c-tuples
6: while |D| > 0 do {Cover the c-tuples values with a set L of disjoint labels}
7:    λ = ∅
8:    for v ∈ D do
9:       if if all c-tuples containing v at position pos contain also λ then
10:         λ = λ ∪ {v}, D = D \ {v}
11:   L = L ∪ λ
12: for λ ∈ L do
13:   T = set of c-tuples containing all values of λ at position pos
14:   build_trie(T, pos + 1, node, label)
15: return node
```

formalization the class variable is considered an additional attribute. Algorithm 1 describes our *build_trie* function, which takes as input a set of c-tuples, an index over the sequence of attributes, plus an MDD node (to serve as a parent) and a label. The parent node and the label are left blank at the first invocation.

The function builds a new node and connects it to the parent (lines 2-3). Then the process stops if all x_i have been considered (line 4). Otherwise, we identify the set D of all values appearing at position *pos* in the c-tuples, and then we build a set L of labels such that: 1) the labels are disjoint; 2) the set of values for x_{pos} in any c-tuple can be expressed as a union of labels in L; 3) the cardinality of L is minimal. This is done via a loop with $O(|D||\text{c-tuples}|)$ iterations (polynomial in the size of the DT). This steps allows to limit the trie growth by grouping equivalent children of the current node. Finally, for each of the identified labels we build a set of compatible rules and we make a recursive call to *build_trie*. When the whole recursive process is over, the first call to the function returns the root of the trie/MDD.

Figure 3.1 shows the result of the conversion for the DT from Figure 1, using 'A, B, C, class' as attribute order. The inflating effect of attributes reordering is apparent. The graph size can be considerably reduced by feeding the trie to the *mddReduce* procedure from [11] (the output of the process is shown in Figure 3.2). For enforcing GAC over the MDD we use the same approach as [11].

3.4 Embedding Random Forests in CP

Embedding a Random Forest in CP requires to 1) embed in the CP model each DT from the forest and 2) define a constraint model for the mode computation, i.e. for aggregating the DT results and obtain the final classification. Since step (1) has already been throughly discussed, we now focus on step (2).

The statistical mode of a sample is the value that occurs most often in the collection. Formally, let y_F be the output of the forest and let y_j be the class

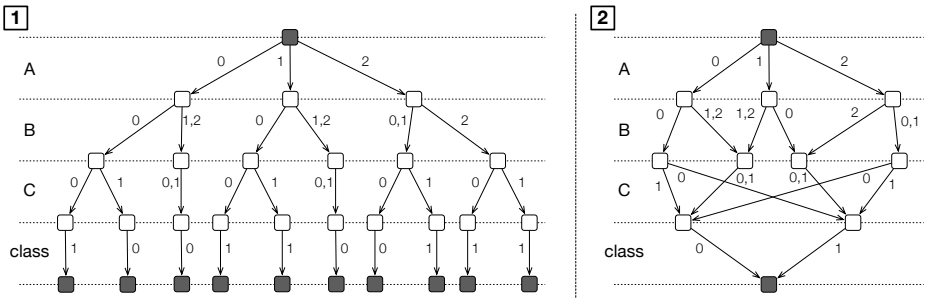

Fig. 3. [1] The trie-like structure for the DT from Figure 1. [2] The corresponding reduced MDD.

variable for the j-th DT. Then our goal is to enforce consistency on the following mathematical expression:

$$y_F = \text{argmax}_{w \in \mathcal{C}} \left(cnt_{\overline{y}}(w) \right) \tag{12}$$

where $cnt_{\overline{y}}(w) = |\{y_j \in \overline{y} \ : \ y_j = w\}|$. Our approach to modeling the mode uses a global constraint to compute the *cnt* functions and one for *argmax*. In particular, we employ the following pair of constraints:

$$\text{GCC}(\overline{y}, \mathcal{C}, \overline{z}) \tag{13}$$

$$\text{ELEMENT}(\mathcal{C}, y_F, \max(\overline{z})) \tag{14}$$

where we recall that $\mathcal{C}$ is the set of the classes w, treated here as a vector; $\overline{z}$ is a vector of fresh variables z_w, each representing (thanks to the GCC constraint) the cardinality of the value w in the vector $\overline{y}$. The ELEMENT constraint ensures that y_F is the index of the value $w \in \mathcal{C}$ with the largest z_w, i.e. the mode.

This approach is based on classical global constraints and very easy to implement. As a drawback, the mode computation can be incomplete in case of ties among the z_w variables. If this happens, the ELEMENT constraint ensures that the domain of y_F contains the indices of all w values having maximal z_w. This situation can be resolved in the case of binary classifications by using an odd number of trees. In general, the user can force the solver to pick a value by adding y_F to the branching variables.

4 Experimentation

Our experimentation is divided in two parts. First, we compare all the presented techniques to embed DTs and RFs, in order to assess their effectiveness and scalability. Second, we evaluate the practical performance of our CP approach against a powerful solver based on Local Search.

For both the comparisons we consider a workload dispatching problem over an experimental 48-core CPU by Intel called Single-chip Cloud Computer (SCC,

see [19]). The chip is designed to accept job batches and does not support process migration. We target a simulated version of the platform that has been augmented for research purpose with thermal controllers to slow down the cores in case they get too warm [1]. As a consequence, the efficiency of a core depends on its temperature. This in turns depends on many complex factors including the workload, the temperature of the other cores, the position on the silicon die, and the action of the thermal controllers. Our goal is to map jobs so as to maximize the number of cores operating at "high" efficiency ($\geq 97\%$).

Due to the complexity of the interactions determining the core behaviors, it is not possible to assess the effect of a job mapping on the efficiency via a traditional, expert designed, model. Hence, we obtained an approximation by generating a training set and then learning a Decision Tree (or Random Forest) for each core, to predict if it will have high (class 1) or low (class 0) efficiency given a specific workload. The input of each classifier is a set of four attributes (discussed later), whose values depend on the mapping decisions. We built the training set following a factorial design over a set of parameters identified in preliminary experiments. The training sets for each core range from 500 to 1000 tuples. Experiments were run on realistic sets of jobs.

Formally, there are n jobs that should be mapped to 48 cores. Each job i is characterized by its average Clocks Per Instruction (CPI) value cpi_i: jobs with low CPI are CPU-intensive and generate more heat, whereas jobs with high CPI are comparatively colder. All jobs are assumed to run indefinitely and hence, in order to avoid overloading, we require each core to run the same number of jobs. The mapping decisions are modeled via integer variables $core_i \in \{0, .., 47\}$ such that $core_i = k$ iff job i is mapped on core k. The overload prevention constraint is formulated using a single GCC with fixed cardinalities:

$$\text{GCC}(\overline{core}, [0, \dots 47], [m, \dots m]) \tag{15}$$

where $m = n/48$. The model objective is:

$$\max \sum_{k=0..47} \mathit{eff}_k \tag{16}$$

where eff_k is the integer class variable for the DT/RF associated to core k. The attributes for each DT are all numeric and correspond to:

- The average CPI of the jobs mapped on core k, i.e. $avgcpi_k$.
- The minimum CPI of the jobs mapped on core k, i.e. $mincpi_k$.
- The average $avgcpi_h$ for all the cores h that are within one tile from k.
- The average of $avgcpi_h$ for all the cores except k.

We model the attributes via integer variables (using a finite precision approximation). Attribute and mapping variables are connected by the following constraints:

$$avgcpi_k = \frac{1}{m} \sum_{i=0}^{n-1} cpi_i \cdot [[core_i = k]] \qquad \forall k = 0..47$$

$$mincpi_k = \min_{i=0..47} (\max(cpi_i) - [[core_i = k]] \, (\max(cpi_i) - cpi_i)) \quad \forall k = 0..47$$

		Closed Inst.			Time			Sol.%		
cores	**attr.**	**rls**	**tbl**	**mdd**	**rls**	**tbl**	**mdd**	**T>R**	**M>R**	**T>M**
3	4	100%	100%	100%	0.02	0.01	0.01	0.00%	0.00%	0.00%
	28	100%	100%	100%	0.06	0.04	0.06	0.00%	0.00%	0.00%
4	4	72%	98%	98%	19.27	2.22	2.51	1.67%	1.67%	0.00%
	28	57%	99%	99%	30.55	1.64	2.15	3.00%	3.00%	0.00%
5	4	13%	38%	34%	53.16	41.61	42.71	4.33%	4.00%	0.33%
	28	5%	38%	35%	57.47	40.60	42.05	8.33%	7.67%	0.67%
6	4	3%	14%	13%	58.30	52.40	52.59	9.67%	8.67%	1.00%
	28	1%	2%	2%	59.41	58.88	58.97	12.67%	11.67%	1.00%
7	4	0%	0%	0%	60.01	60.01	60.01	7.00%	7.00%	0.00%
	28	0%	1%	1%	60.01	59.47	92.54	10.00%	9.67%	0.67%

Table 1. Comparison of DT encodings.

$$neighcpi_k = \frac{1}{|N(k)|} \sum_{h \in N(k)} avgcpi_h \qquad \forall k = 0..47$$

$$allcpi_k = \frac{1}{47} \sum_{h \neq k} avgcpi_h \qquad \forall k = 0..47$$

The constraints connecting the attribute and class variable eff_k for each core are obtained via the techniques discussed in Section 3.

4.1 Comparing the Different Encodings

For comparing the proposed DT and RF encodings, we generated benchmarks of instance with controlled size. This was done by 1) selecting random groups of cores and 2) generating a set of 6 jobs per core with realistic CPIs. Each benchmark contains 100 instances. We then solved each instance to optimality using a static search strategy and a time limit of 60 seconds. All experiments are run on a 2.3 GHz Intel Core i7. Our approaches have all been implemented over the Google or-tools solver [25].

Comparing DT Encodings: We tested our Decision Trees encodings, investigating the effect of the problem size and the number of attributes. This was done by building benchmarks with different numbers of cores and by augmenting the initial set of four attributes with up to 24 similarly computed features. The trees were learned from a training set using the implementation of C4.5 in Weka [16].

The results for this first evaluation are displayed in Table 1. Columns *cores* and *attr* report the number of cores and attributes for the benchmark. Then for each encoding approach (based on rules, TABLE, and MDD) we show the number of closed instances and the average solution time (time-outs at 60 sec of search are included). The last columns report the fraction of instances where each approach (R for rules, T for TABLE, and M for MDD) managed to find more

	Res. ok			closed			Mem (MB)			Time/10^6 brnc		
attr	**rls**	**tbl**	**mdd**	**rls**	**tbl**	**mdd**	**rls**	**tbl**	**mdd**	**rls**	**tbl**	**mdd**
4	100%	100%	100%	87%	92%	87%	38.5	39.3	40.3	25.17	13.59	30.29
8	100%	100%	100%	87%	92%	85%	40.9	49.5	53.4	34.13	18.36	42.29
12	100%	57%	55%	92%	57%	55%	46.0	351.4	197.4	40.59	28.38	51.60
16	100%	–	–	97%	–	–	47.6	–	–	45.43	–	–
20	100%	–	–	95%	–	–	53.2	–	–	49.72	–	–

Table 2. Comparison of RF encodings, over different number of attributes

solutions than another: since we use a static search strategy, this is an indication of the size of the explored search space.

As a general trend, both approaches capable of enforcing GAC considerably outperform the rule-based encoding. The trend stays the same when the number of attributes is inflated from 4 to 28: since more attributes tend to yield considerably bigger trees, this result show that the TABLE and MDD provide good scalability, despite the potential risk of combinatorial explosion of their main data structures. A hint of this risk is given by the average time for solving 7-cores, 28-attributes instances with the MDD approach, which is higher than 60 seconds. The catch is that the 60 seconds timeout is enforced only on search, while the total solution time takes into account also the model construction. Basically, in such case building the MDDs took a very long time. This is in part due to inefficiencies of our implementation, but is also symptomatic of possible scalability issues. In general the performance of the rule based approach seems to be the one most affected by the increased number of attributes.

Comparing RF Encodings: Next, we investigated the effectiveness of our encodings when applied to Random Forests. This was done by generating: 1) a first group of benchmarks defined over random groups of three cores, with several number of attributes and a fixed number of trees per forest (seven); 2) a second group of benchmarks defined again over core triplets, but with fixed number of attributes (four) and variable number of trees. The RFs were learned with the default algorithm provided by Weka.

The results of the first evaluation (variable number of attributes) are show in Table 2. The training algorithm for RFs yields trees that are radically different from those of C4.5 and translates to *much* bigger tuple-sets and MDDs. As a consequence, the TABLE and MDD based encodings have considerable scalability problems *at model construction time*, before the search process evens starts. In practice, we decided to stop some runs before the model construction started to take an impractical amount of resources (memory or time). The fraction of complete runs for all approaches is reported in the "Res. ok" columns. The memory usage (column "Mem") was found to be the main bottleneck for the TABLE based approach, whereas for the MDD method the biggest issue was the model construction time. The larger number of trees and their size had a negative effect on the branching speed: the TABLE was the most affected approach, despite

	closed			Time			Time/10^6		brnc
trees	rls	tbl	mdd	rls	tbl	mdd	rls	tbl	mdd
3	95 %	97 %	97 %	10.71	8.58	12.22	14.58	10.01	17.05
5	92 %	97 %	90 %	14.08	9.86	15.95	18.07	10.81	22.39
7	87 %	92 %	87 %	17.07	11.83	18.39	24.51	13.58	29.60

Table 3. Comparison of RF encodings, over different number of attributes

being the one with the highest branching speed. Overall, the rule based approach managed to cope considerably better than any other with the scalability issues associated to RFs.

The effect of varying the number of trees can be observed in Table 3. The most striking finding is not reported in the table: basically, regardless of the number of trees, *all approaches reported exactly the same number of fails* in the instances they were able to close. This is probably due to the fact that the advantage of enforcing GAC on the individual trees gets lost at in the mode computation, performed by constraints (13) and (14). In such a situation, the best approach is the one with the lowest computation time. With this number of attributes (just 4) the TABLE encoding emerged as slightly faster, thanks also to the highly optimized implementation available in or-tools. Increasing the number of trees did not seem to have dramatic effects on the performance of the encodings.

4.2 Comparison with a State of the Art Local Search Approach

We performed a last set of experiments to evaluate our CP-based solution w.r.t. alternative approaches that can easily embed a Decision Tree in a model, but cannot benefit from constraint propagation. As a reference for the comparison, we used a model written and solved using Localsolver [6, 12]. The choice was motivated by the simplicity of use of Localsolver, and its effectiveness in solving problems with non-trivial constraints and non-linear objective functions. Our Localsolver model is similar to the CP one, but the $core_i$ variables are missing, the reified $[[core_i = k]]$ are replaced by binary variables, and the GCC is replaced by a set of bounded sums.

For the CP model, we developed a solution approach that works by: 1) generating a first mapping via a heuristic; 2) using Large Neighborhood Search [27] to relax and re-map the jobs allocated to a subset of cores. In particular, we always select a few "bad" cores having $\mathit{eff}_k = 0$ at random, plus a few "good" cores with a probability based on their $avgcpi_k$ value. For exploring each neighborhood we use Depth First Search with random variable/value selection and restarts. We used the rules and table encoding for embedding Decision Trees.

We tested both approaches on a benchmark of 20 instances with 48 cores (modeled using DTs with four input attributes) and 288 jobs each. Since all methods are randomized, we performed 10 runs per instance, with a time limit of 90 seconds. A time limit of 2 seconds was enforced on each LNS iteration for CP. Table 4 reports the average value of the problem objective over the 10 runs

ID	CP(rls) avg	CP(rls) std	CP(tbl) avg	CP(tbl) std	LS avg	LS std	ID	CP(rls) avg	CP(rls) std	CP(tbl) avg	CP(tbl) std	LS avg	LS std
0	29.60	0.98	30.50	0.50	27.70	0.46	10	35.60	0.98	36.80	0.87	33.20	0.88
1	33.60	0.80	35.60	0.80	32.40	0.67	11	24.30	0.78	24.20	0.75	24.00	0.00
2	28.70	1.01	30.00	0.78	28.10	0.70	12	33.10	0.83	34.20	1.17	31.30	0.78
3	25.10	0.70	25.60	1.02	25.10	0.54	13	24.00	1.00	25.30	0.78	24.00	0.00
4	23.30	0.46	23.80	1.08	23.50	0.50	14	29.60	0.66	30.90	0.70	28.60	0.92
5	30.50	1.12	31.80	0.60	29.00	0.63	15	32.90	0.83	35.00	0.78	30.80	0.60
6	39.10	0.70	40.00	0.63	36.90	0.70	16	25.10	0.70	25.90	0.70	24.70	0.64
7	28.60	0.80	30.30	0.64	27.90	1.04	17	38.90	0.54	40.10	0.54	36.60	1.02
8	32.90	1.30	34.10	0.54	30.70	0.64	18	26.40	1.02	26.80	0.60	25.20	0.40
9	37.30	0.64	38.70	0.90	34.90	1.04	19	32.90	1.14	34.80	0.98	31.80	0.75

Table 4. Comparison with a state-of-the-art hybrid solver based on Local Search

and its standard deviation. The CP and LS approach proved to work very well[2], but CP managed to find the best solutions for the majority of the instances. This result was made possible by the use of domain knowledge for selecting the LNS fragments, *and by the propagation performed by our DT encodings*. Using the rule encoding was sufficient to quickly close many ill-chosen fragments and speed up the search. The additional propagation granted by enforcing GAC allowed the TABLE encoding to work even better.

5 Other Related Work

The integration of Machine learning and CP (and in general combinatorial optimization) has received increased attention in the last decade. Fertilizations in both directions have been studied: on one hand the machine learning community has studied how constraints can be used during mining and learning; on the other way round, machine learning may allow to automatically tune an optimization approach, or to acquire constraints and objective functions from data.

Along the first line of research, works such as [5, 28] have studied the core optimization problems in Machine Learning algorithms and proposed efficient methods for extracting knowledge from huge volumes of data. On the other way round, some researchers have considered the problem of learning optimization problem instances for testing new techniques [17]. Clustering methods have been employed for automatic algorithm selection in [22]. Several Machine Learning techniques have been used for predicting the run time of optimization algorithms (e.g. [20]), and in general for algorithm selection (e.g. [24]).

The approaches that are closest in spirit to this paper are those that focus on learning parts of the model from data. Along this line several papers [7, 9, 4] show how to learn a constraint network from a set positive and negative examples, while the QuAcq system [8] requires only partial queries on subsets

[2] w.r.t. other approaches that are not reported here due to lack of space.

of problem variables, with no need of positive examples. All such approaches have a focus on learning an unknown problem while simultaneously trying to solve it. Conversely, in our approach we focus on embedding in combinatorial optimization a well-defined, pre-extracted, Machine Learning model, which may however lack a straightforward encoding.

Many approaches for optimizing non-linear functions over continuous domains rely on on-line learning techniques to reduce the number of required function evaluations. The authors of [21] introduced methods to fit a response surface based on a few sampled points: the surface is then employed to guide the search process. The OptQuest [14] system integrates in a closed loop simulation and metaheuristics and relies on a simple form of learning (a neural network accelerator) to avoid trivially bad solutions. Only a few works (e.g. [15, 29]) have resorted to using pre-extracted Machine Learning models to speed up the cost function evaluation. The LION book [3] proposes a similar approach, although in this case the goal is tackling problems where the cost function is difficult to model, rather than expensive to compute. The LION method focuses on extracting a function from available data, and then on obtaining solutions via model fitting. As a common feature, all such methods are designed for functions defined over an unconstrained (or loosely constrained) domain. Conversely, our method targets problems that mix a core combinatorial structure (typically having non-trivial constraints) with complex functions that we approximate via Machine Learning.

6 Concluding Remarks

The Empirical Model Learning (EML) method is aimed at learning part of the model from data or predictive tools. The learned component not only declaratively links decision variables with prediction/class variables, but contains an operational semantics enabling domain filtering and constraint propagation.

In this work, we have devised an additional component for EML: we have used Decision Trees and Random Forests as learning methods. We have provided three encodings of for DTs, respectively based on meta-constraints, on the global TABLE constraint, and on an MDD – the last two being able to enforce GAC. The experimentation on Decision Trees and Random Forests had quite different outcomes in terms of scalability, mainly because of differences between their learning algorithms. While for DTs the TABLE and MDD approaches are clearly the most effective, in RFs the TABLE constraint and MDD have serious scalability issues in terms of memory and model construction time respectively.

As part of future research, we plan to test the modified GAC-schema algorithm for c-tuples from [23], and to investigate methods to convert a DT into an MDD without the passing for an intermediate trie. We will also research the possibility to enforce GAC using the DT itself as the main data structure.

Finally, we are interested in finding ways to exploit the accuracy information provided by the Machine Learning models (e.g. class distributions in RFs). This is particularly important when combining different constraints, each representing an approximate relation between variables.

References

1. A. Bartolini, M. Cacciari, A. Tilli, and L. Benini. Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller. *IEEE Trans. Parallel Distrib. Syst.*, 24(1):170–183, 2013.
2. A. Bartolini, M. Lombardi, M. Milano, and L. Benini. Neuron constraints to model complex real-world problems. In *Proc. of CP*, pages 115–129, 2011.
3. R. Battiti and M. Brunato. *The LION way: Machine Learning plus Intelligent Optimization*. LIONlab, University of Trento, 2014.
4. N. Beldiceanu and H. Simonis. A model seeker: Extracting global constraint models from positive examples. In *Proc. of CP*, pages 141–157, 2012.
5. K. P. Bennett and E. Parrado-Hernández. The interplay of optimization and machine learning research. *Journal of Machine Learning Research*, 7:1265–1281, 2006.
6. T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua. Localsolver 1.x: a black-box local-search solver for 0-1 programming. *4OR*, 9(3):299–316, 2011.
7. C. Bessière, R. Coletta, E. C. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *Proc. of CP*, pages 123–137, 2004.
8. C. Bessière, R Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C. Quimper, and T. Walsh. Constraint acquisition via partial queries. In *Proc. of IJCAI*, 2013.
9. C. Bessière, R. Coletta, B. O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In *Proc of IJCAI*, pages 50–55, 2007.
10. L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
11. K. C. K. Cheng and R. H. C. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.
12. F. Gardi, T. Benoist, J. Darlay, B. Estellon, and R. Megel. *Mathematical Programming Solver Based on Local Search*. John Wiley & Sons, 2014.
13. I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proc. of AAAI*, pages 191–197, 2007.
14. F. Glover, J. P. Kelly, and M. Laguna. New Advances for Wedding optimization and simulation. In *Proc. of WSC*, pages 255–260, 1999.
15. K. Gopalakrishnan and A. M. Asce. Neural Network – Swarm Intelligence Hybrid Nonlinear Optimization Algorithm for Pavement Moduli Back-Calculation. *Journal of Transportation Engineering*, 136(6):528–536, 2009.
16. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.
17. L. Hernando, A. Mendiburu, and J. A. Lozano. Generating customized landscapes in permutation-based combinatorial optimization problems. In *Learning and Intelligent Optimization- LION*, volume 7997 of *Lecture Notes in Computer Science*, pages 299–303, 2013.
18. Tin Kam Ho. Random decision forests. In *Proc. of ICDAR*, pages 278–, 1995.
19. J. Howard and S. Dighe et al. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. *Proc. of ISSCC*, pages 108–109, February 2010.
20. F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artif. Intell.*, 206:79–111, 2014.

21. D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.
22. S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - instance-specific algorithm configuration. In *Proc. of ECAI*, pages 751–756, 2010.
23. G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proc. of CP*, pages 379–393, 2007.
24. L. Kotthoff, I.P. Gent, and I. Miguel. An evaluation of machine learning in algorithm selection for search problems. *AI Commun.*, 25(3):257–270, 2012.
25. L. Perron. Operations Research and Constraint Programming at Google. In *Proc. of CP*, page 2, 2011.
26. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
27. P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Proc. of CP*, pages 417–431, 1998.
28. S. Sra, S. Nowozin, and S. J. Wright. *Optimization for machine learning*. Mit Press, 2012.
29. A. H. Zaabab, Q. Zhang, and M. Nakhla. A neural network modeling approach to circuit optimization and statistical design. *Microwave Theory and Techniques, IEEE Transactions on*, 43(6):1349–1358, 1995.