Learning to Run Heuristics in Tree Search

Elias B. Khalil[1], Bistra Dilkina*[1], George L. Nemhauser[2], Shabbir Ahmed[2], Yufen Shao[3]
[1]School of Computational Science & Engineering, Georgia Institute of Technology, Atlanta, GA, USA
[2]School of Industrial & Systems Engineering, Georgia Institute of Technology, Atlanta, GA, USA
[3]ExxonMobil Upstream Research Company, Houston, TX, USA
{elias.khalil, bdilkina}@cc.gatech.edu, gn3@gatech.edu,
shabbir.ahmed@isye.gatech.edu, yufen.shao@exxonmobil.com

Abstract

"Primal heuristics" are a key contributor to the improved performance of exact branch-and-bound solvers for combinatorial optimization and integer programming. Perhaps the most crucial question concerning primal heuristics is that of *at which nodes they should run*, to which the typical answer is via hard-coded rules or fixed solver parameters tuned, offline, by trial-and-error. Alternatively, a heuristic should be run when it is most likely to succeed, based on the problem instance's characteristics, the state of the search, etc. In this work, we study the problem of deciding at which node a heuristic should be run, such that the overall (primal) performance of the solver is optimized. To our knowledge, this is the first attempt at formalizing and systematically addressing this problem. Central to our approach is the use of Machine Learning (ML) for predicting whether a heuristic will succeed at a given node. We give a theoretical framework for analyzing this decision-making process in a simplified setting, propose a ML approach for modeling heuristic success likelihood, and design practical rules that leverage the ML models to dynamically decide whether to run a heuristic at each node of the search tree. Experimentally, our approach improves the primal performance of a state-of-the-art Mixed Integer Programming solver by up to 6% on a set of benchmark instances, and by up to 60% on a family of hard Independent Set instances.

1 Introduction

Integer programming and combinatorial optimization are powerful tools that can model a myriad of complex decision-making tasks. Applications of Mixed Integer Programming (MIP) have thus spanned domains as diverse as aircraft routing [Barnhart *et al.*, 1998], wildlife conservation [Dilkina and Gomes, 2010], sports scheduling [Nemhauser and Trick, 1998], dose distribution [Lee *et al.*, 1999] and kidney exchange [Abraham *et al.*, 2007], to mention a few. As such, improving the performance of MIP solvers can have a dramatic impact across various domains. A recent trend that has shown strong promise for improving optimization is centered around integrating *artificial intelligence and machine learning* (ML) within solvers. For instance, deep learning can help tune gradient descent [Andrychowicz *et al.*, 2016], reinforcement learning is used for job-shop scheduling [Zhang and Dietterich, 1995], and classification is used for selecting an algorithm for QBF subproblems [Samulowitz and Memisevic, 2007]. In the context of MIP, ML has been used to find good parameter configurations for a solver [Hutter *et al.*, 2009], design improved node [Sabharwal *et al.*, 2012] or variable [Khalil *et al.*, 2016] selection strategies, or detect decomposable problem structure [Kruber *et al.*, 2016]. In the same spirit of augmenting exact solvers with ML, we propose a framework for learning when to run heuristics during branch-and-bound tree search, a task whose importance we motivate shortly.

The Primal Side of Integer Programming. There are two sides to any constrained optimization problem. On the one hand, we want to find *feasible solutions* to the problem instance at hand. On the other hand, we would like to *prove the optimality* of the best feasible solution found, i.e. to guarantee that no feasible solution with strictly better objective function value exists. These two sides are particularly prominent in MIP, the study of optimization problems with integer-valued variables. To use the terminology of MIP, the *primal* side refers to the quest for good feasible solutions, whereas the *dual* side refers to the search for a proof of optimality.

While proving optimality is a key trait of exact solvers for MIP, finding quality feasible solutions quickly is certainly at least as crucial. For example, consider a real-world MIP model that a company solves on a regular (e.g. daily) basis in order to plan its operations. When state-of-the-art MIP solvers require many hours to solve an instance to optimality, the user will expect good feasible solutions to be found much earlier in the solving process, so that they are able to act upon them and address their business needs promptly. An example of such a challenging real-world scenario is that of the maritime inventory routing problem (MIRP), described in [Papageorgiou *et al.*, 2014]. For the 28 MIRP instances, the solver Gurobi, with default settings (including parallel processing) and no warm-starting, is not capable of finding *any* feasible solutions for any of the instances in 24 hours [Papageorgiou *et*

*Corresponding author, bdilkina@cc.gatech.edu

al., 2014]. The delay in finding feasible solutions affects the decision-maker's ability to plan ahead and compare options before deployment.

The Impact of Primal Heuristics. In this work, we focus on the primal side of integer programming. We do note, however, that finding better feasible solutions while solving a MIP with branch-and-bound speeds up proving optimality by pruning nodes with worse lower bounds, assuming a minimization problem. The classical way by which the MIP solver finds feasible solutions is through linear programming (LP) relaxations: in a branch-and-bound search, after branching on a subset of the integer variables of a MIP instance, solving the LP relaxation of the restricted sub-problem may result in an integer-feasible solution.

However, the MIP community has recently realized the potential for combining *primal heuristics* with exact branch-and-bound search to improve solution finding. Primal heuristics are incomplete, bounded-time procedures that attempt to find a good feasible solution. Primal heuristics may be used as standalone methods, taking in a MIP instance as input, and attempting to find good feasible solutions, or as sub-routines inside branch-and-bound, where they are called periodically during the search. In this work, we focus on the latter, which we will expand on in the following paragraph.

A number of computational studies, with different MIP solvers, have demonstrated the large impact that primal heuristics have on branch-and-bound. An interesting finding reported in [Berthold, 2006] is that on 97 easy benchmark instances, the LP relaxation finds an optimal solution 59 times, whereas on 26 hard instances it finds an optimal solution only 3 times; for the remaining instances, one of the primal heuristics of the SCIP solver used by Berthold finds an optimal solution. Berthold's results show that the investment in developing effective primal heuristics has brought about significant returns, most notably for harder instances. Even stronger results confirming the impact of heuristics on the solver CPLEX are discussed in [Achterberg and Wunderling, 2013].

Our Problem Setting. Despite the success of primal heuristics within MIP solving, there remains a number of central issues pertaining to *when* and *what* heuristics should be run during the search. For instance, in SCIP (a state-of-the-art academic MIP solver), 43 primal heuristics have been implemented. In the default settings of the solver, some heuristics are turned off, others run very frequently (e.g. at every node), while yet another subset runs occasionally (e.g. every 10 or 20 nodes). Such rigid rules for running heuristics are static, instance-oblivious, context-independent, and are unable to adapt to the state of the search. Additionally, the algorithmic differences between primal heuristics result in substantial variation in performance. For instance, diving and neighborhood search heuristics are much more computationally expensive than their rounding counterparts, but are generally more likely to find quality feasible solutions.

Towards establishing a *dynamic, data-driven approach* to the use of primal heuristics in tree search, we address the problem of decision-making for primal heuristics. Assume that $P(t)$ is some primal performance measure, whose value at time point t indicates how successful the solver has been on the primal side up to t; the choice of the performance measure $P(.)$ will be discussed in detail in Section 2.2. In its simplest form, a formulation of the problem addressed here is:

> Given a primal heuristic H, a branch-and-bound solver with search tree $\mathcal{T}$, a time cutoff t_{max}, find the subset of nodes of $\mathcal{T}$ at which executing H results in the best primal performance possible, $P(t_{max})$.

To our knowledge, the systematic study of this problem is new. By "systematic", we mean that there is a well-defined *objective function* $P(t_{max})$ to optimize, and a clear *decision space*, namely executing H or not at each node. We refer to a procedure that decides when to run a primal heuristic as a *primal policy*. The proposed problem raises a number of interesting questions that span online decision-making under uncertainty and ML.

2 Definitions

2.1 Branch-and-Bound for MIP

Tree search is the *de facto* approach for solving Mixed Integer Linear Programming (MIP) problems of the form:

$$z^* = \min\{c^T x | Ax \leqslant b, x \in \mathbb{R}^n, x_j \in \mathbb{Z}\ \forall j \in I\}.$$

The vectors in the set $X_{MIP} = \{x \in \mathbb{R}^n | Ax \leqslant b, x_j \in \mathbb{Z}\ \forall j \in I\}$ are called *feasible solutions*. A feasible solution $x^* \in X_{MIP}$ is *optimal* if $c^T x^* = z^*$.

A MIP can be solved by Branch-and-Bound (B&B), a tree search algorithm that divides the original MIP into sub-problems organized in a binary tree. At each node of the tree, an LP relaxation of the sub-problem is solved. If the resulting solution x_N of the LP relaxation at a node N is integral, then it is also a feasible solution to the MIP, i.e. $x_N \in X_{MIP}$. If such an integral solution has an objective value that is better than the best one found so far, it is referred to as the *incumbent*, maintaining that designation until a better solution is found. Otherwise, the node is either pruned, if its lower bound is greater than the incumbent's value, or branched on, resulting in two child nodes that are added to the queue of nodes to be processed.

2.2 Primal Integral

Our goal is to improve the primal performance of tree search, i.e. the quality of and the speed at which feasible solutions are found. The *primal integral* is a primal performance criterion for MIP that was introduced in [Achterberg *et al.*, 2012] to formally capture these desired characteristics, and that we adopt as a main measure of primal performance.

Let x^* denote an optimal (or best known) solution for a MIP, and $\tilde{x}$ denote a feasible solution for the same MIP. The *primal gap* $\gamma \in [0, 1]$ of solution $\tilde{x}$ is defined as:

$$\gamma(\tilde{x}) := \begin{cases} 0, & \text{if } |c^T x^*| = |c^T \tilde{x}| = 0 \\ 1, & \text{if } c^T x^* \cdot c^T \tilde{x} < 0 \\ \frac{|c^T \tilde{x} - c^T x^*|}{\max\{|c^T \tilde{x}|, |c^T x^*|\}}, & \text{otherwise.} \end{cases} \quad (1)$$

Let $t_{\text{max}} \in \mathbb{R}_{\geq 0}$ be a limit on the solution time of the B&B MIP solver. Then, the *primal gap function*

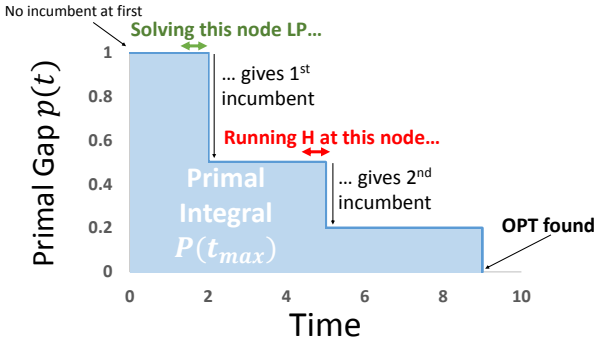

Figure 1: An illustration of the primal integral.

$p : [0, t_{\max}] \mapsto [0, 1]$ is defined as:

$$p(t) := \begin{cases} 1, & \text{if no incumbent is found until point } t, \\ \gamma(\tilde{x}(t)), & \text{with } \tilde{x}(t) \text{ the incumbent at point } t. \end{cases}$$

It is easy to see that $p(t)$ is a nonincreasing step function in $[0, 1]$ that changes whenever a new incumbent is found, and takes on a value of zero the moment an optimal solution is found – see Figure 1. The *primal integral* $P(T)$ of a branch-and-bound run until a point in time $T \in [0, t_{\max}]$ is defined as:

$$P(T) := \sum_{i=1}^{Inc+1} p(t_{i-1}) \cdot (t_i - t_{i-1}),$$

where Inc is the number of incumbents, $t_i \in [0, T]$ for $i \in 1, \ldots, Inc$ are the points in time when a new incumbent is found, $t_0 = 0$ and $t_{Inc+1} = T$. A graphical illustration of the primal integral is shown in Figure 1. Note that the *primal-dual integral* is another metric that is defined similarly to the primal integral, with $\gamma(\tilde{x}, \underline{z}) := \frac{c^T \tilde{x} - \underline{z}}{\max\{|c^T \tilde{x}|, |\underline{z}|\}}$, $PD(T) := \sum_{i=1}^{Chn} p(t_{i-1}) \cdot (t_i - t_{i-1})$, Chn the time points at which either the global lower bound $\underline{z}$ or upper bound $c^T \tilde{x}$ changed, and $p(t_{i-1}) = \gamma(\tilde{x}(t), \underline{z}(t))$ (or 1 if either bound is undefined). However, the primal-dual integral confounds the primal and dual sides, and is thus less relevant for our purposes.

Achterberg et al. suggest that the primal integral be used to measure the progress on the primal side during B&B [Achterberg *et al.*, 2012]. The smaller $P(t_{\max})$ is, the better the incumbent finding. As such, we will consider optimizing the *primal integral* directly, by means of making good decisions regarding whether a primal heuristic should be run at each node or not.

3 Theoretical Analysis

3.1 Problem Formulation

Should heuristic H be run at a given node N? An answer to this question must study the trade-off between the potential benefit of finding a better feasible solution if H is run at N, and the cost associated with running H (including the risk of failure). Running H at every node may be undesirable, as H may run for a long period of time, during which the MIP could be solved to optimality, irrespective of H. Thus, it is crucial to choose the right set of nodes at which to run H. We now give the first general formulation of the problem we call "Primal Integral Optimization" (PIO):

> **(PIO)** Given a primal heuristic H, a branch-and-bound MIP solver with search tree $\mathcal{T}$ and a time cut-off t_{max}, find the subset of nodes of $\mathcal{T}$ at which executing H results in a primal integral $P(t_{max})$ of minimum value.

The first step towards formalizing PIO lies in defining a simple, conceptual model of branch-and-bound, within which we can analyze the complexity of PIO, and the theoretical performance of approaches to solving it. We will distinguish two main settings:

- the *offline* setting, where the search tree $\mathcal{T}$ is *fixed and known* in advance, and PIO amounts to finding the best subset of nodes to run H at in hindsight;
- the *online* setting, where one must sequentially decide, at each node, whether H should be run, without any knowledge of the remainder of the tree or search.

In practice, the online setting is more relevant as it is representative of actual MIP solving. Thus, we will analyze online decision-making algorithms next, and bound their worst-case performance compared to an optimal solution obtained offline, in hindsight. Note that such an offline solution is easy to compute via dynamic programming if the B&B tree is known and fixed in advance. We do not provide the details of the offline algorithm, as it is not of practical interest.

Central to the algorithms that we analyze is the notion of an *oracle* which, when queried at a given node, tells the online algorithm whether heuristic H will find an incumbent or not. We will analyze the performance of an online algorithm under two different assumptions on the oracle's behavior. In the first setting, the oracle is assumed to be *perfect*, in that it returns the correct answer (i.e. whether H will find an incumbent or not) at every node at which it is queried. In the second setting, the oracle is assumed to be *faulty*, making a mistake with a given probability.

To see how this conceptual framework is tied to the proposed ML approach to PIO, notice that an ML model of heuristic success can be seen as a faulty oracle: the model is likely to miss a few incumbents, or wrongly predict incumbents at some nodes. Since the heuristic is run at nodes during B&B, any decision-making algorithm must act online, using only information about the solving process up to the given point in time. Our theoretical analysis aims at substantiating the practical ML approach to PIO of Section 4.

3.2 Competitive Ratio under a Perfect Oracle

The online PIO problem is a challenging one, since decisions regarding running H at a node must be made without any knowledge of the remainder of the search tree, and the incumbents that may be found later by LP or H. As a first result, we will analyze the following simple rule of thumb for deciding whether to run a primal heuristic at a node: if the oracle says that H can find an incumbent solution at node N_i, then run H; otherwise, do not run H. We refer to this rule of thumb as Run-When-Successful (RWS).

To analyze online algorithms, we resort to worst-case analysis using the *competitive ratio* [Albers, 2003]. Assume we are given a sequence of "requests" σ (in our case, each request is a node at which we run H or not). Let $A(\sigma)$ denote the cost incurred by a deterministic online algorithm A, and let $OPT(\sigma)$ denote the cost incurred by an optimal offline algorithm OPT. The algorithm A is called c-competitive if there exists a constant a such that $A(\sigma) \leq c \cdot OPT(\sigma) + a$, for $\forall\sigma$; c is the *competitive ratio*.

Theorem 1. *RWS achieves a competitive ratio of*

$$1 + \frac{t_H}{t_{LP}}$$

with respect to the optimal offline solution, where t_H, t_{LP} are the fixed running times of H and an LP relaxation solve, respectively.

Proof. Let P^{RWS} denote the primal integral value obtained on an instance I using the RWS rule, and P^{OPT} the optimal primal integral value for I. Assume the optimal primal policy for heuristic H goes through n nodes before finding an optimal solution to the MIP I. Notice that RWS will require at most n nodes as well, since RWS guarantees that at any node, its incumbent is the best possible up to that node. Let $\mathcal{T}_H^{RWS}$ be the set of time points at which RWS runs H, and $\mathcal{T}_{LP}^{OPT}$ the set of time points at which the optimal primal policy solves an LP. Then, P^{RWS} can be upper bounded as:

$$P^{RWS} \leq P^{OPT} + \sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H.$$

The upper bound is valid because the worst RWS can do is run for t_H seconds multiple times and not improve the incumbent. Dividing both sides of the above inequality by P^{OPT}, we obtain:

$$\begin{aligned}\frac{P^{RWS}}{P^{OPT}} &\leq 1 + \frac{\sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H}{P^{OPT}} \\ &\leq 1 + \frac{\sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H}{\sum_{t_i \in \mathcal{T}_{LP}^{OPT}} p(t_i) \cdot t_{LP}} \leq 1 + \frac{t_H}{t_{LP}}.\end{aligned}$$

The second inequality is valid because

$$P^{OPT} \geq \sum_{t_i \in \mathcal{T}_{LP}^{OPT}} p(t_i) \cdot t_{LP},$$

i.e. the optimal primal integral has value at least that of the LP solves weighted by the primal gap $p(t_i)$. The final inequality is valid because

$$\sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H \leq \sum_{t_i \in \mathcal{T}_{LP}^{OPT}} p(t_i) \cdot t_{LP},$$

as H is run at most as frequently as LPs are solved, and the primal gap value at any node at which RWS runs H is at worst equal to the corresponding gap value for the optimal primal policy. □

It is interesting to see how the bound in this theorem holds up in practice on real MIP instances from the Benchmark set of MIPLIB2010 [Koch *et al.*, 2011]. Table 1 shows the empirical competitive ratio of the RWS algorithm and the theoretical one, computed as in Theorem 1, for four sample instance-heuristic pairs. The optimal offline primal integral P^{OPT} is computed via dynamic programming in hindsight. The results of Table 1 show that the empirical performance of RWS is much better than the theory suggests. However, we do believe that the bound is tight up to an arbitrarily small constant.

Instance	Heuristic	Empirical $\frac{P^{RWS}}{P^{OPT}}$	Theoretical $\frac{P^{RWS}}{P^{OPT}}$
biella1	fracdiving	1.09	122.79
rail507	veclendiving	1.09	18.36
qiu	guideddiving	1.01	5.32
biella1	intshifting	1.00	3.57

Table 1: Sample results from empirical evaluation of RWS.

3.3 Competitive Ratio under a Faulty Oracle

Any practical oracle, such as one designed with ML, will not be perfect. A *faulty* oracle is one that makes a *mistake* at a node with some probability. We distinguish two types of mistakes: false positives and false negatives. Assume that the oracle incurs a false positive at a node with probability δ, i.e. the oracle states that H will find an incumbent at a node N when H does not, and a false negative with probability β, i.e. the oracle states that H will not find an incumbent at N when H does. When $\delta > 0, \beta = 0$, the bound from Theorem 1 also holds for the competitive ratio w.r.t. the *expected* primal integral, i.e. $\frac{\mathbb{E}[P^{RWS}]}{P^{OPT}} \leq 1 + \frac{t_H}{t_{LP}}$. The randomness is with respect to the nodes at which the false positives occur. Unfortunately, when $\beta > 0$, bounding the competitive ratio becomes much trickier: if H finds an optimal solution at N which cannot be found at any other node by either H or LP relaxation solves, and a false negative occurs at N, then $P^{RWS} = t_{max}$, the maximum possible value. As such, we believe that any such bound when $\beta > 0$ will be very loose. However, there may be suitable assumptions under which the bound is not as loose, and we consider this issue to be interesting for future research.

4 Learning a Success Oracle for Heuristics

In the previous section, we showed that despite its simplicity, the RWS rule provides theoretical guarantees under a simplified setting. However, in order to turn RWS into an operational policy, we must design a success oracle. Combining the designed oracle with RWS provides an online procedure for deciding when to run a heuristic during tree search. Recall that our aim is to *dynamically* decide whether to run the heuristic at a given node, based on the instance characteristics, node characteristics and state of the search. As such, we will design the oracle by *learning* a binary classifier which predicts whether heuristic H will find an incumbent solution at node N. The features used to describe node N will incorporate information about the instance, the node and the state of the search, as desired.

4.1 Realistic Data Collection

We now describe our method for collecting data on the heuristic's success across different instances. This aspect of oracle design warrants special attention, given the interplay between incumbent finding and tree search. More specifically, one has to make sure that the node data collected for heuristic H for training is similar to the node data to be encountered when using H's oracle, online, on a new problem instance.

Let $\mathcal{H}$ be the set of primal heuristics used during tree search, $H \in \mathcal{H}$ a given heuristic, and $\bar{\mathcal{H}} = \mathcal{H} - H$. We are given a set of MIP instances, $\mathcal{I}_{train}$, which can be used to collect data on H. A dataset D_I^H is obtained for each instance $I \in \mathcal{I}_{train}$, and the final training dataset for heuristic H is obtained by concatenating instance datasets together into $\mathcal{D}_{train}^H = \bigcup_{I \in \mathcal{I}_{train}} D_I^H$.

We now consider data collection at the individual instance level. Let $I \in \mathcal{I}_{train}$ be a MIP instance for which we want to collect data for heuristic H. We will run H at every node N of the search tree, and collect the binary classification label value, $y_H^N \in \{0, 1\}$, and the feature vector $\mathbf{x}^N \in \mathbb{R}^d$. The label y_H^N takes a value of 1 if H finds an incumbent at node N, and 0 otherwise. The key observation here is that the value of y_H^N depends on z_N^*, the objective function value of the incumbent when node N is considered. In turn, the value of z_N^* depends on the progress in the search up to N. If H is run at every node, then any incumbent that H finds affects the value $y_H^{N'}$ for all nodes N' that come after N. This interplay between the incumbents found by H and the data being collected for H is problematic, as the labels y_H^N in the training dataset assume that the oracle is perfect, i.e. H is run anytime it can find an incumbent. In practice, however, the oracle is a binary classifier that is unlikely to be perfect, meaning that it will not always run H, even when H can find an incumbent.

To deal with this complication, we devise a data collection procedure that is more likely to result in realistic datasets. First, for any heuristic H for which data is to be collected, H is run in "stealth" mode: any new incumbent that H finds does not replace the current incumbent. This measure is equivalent to not running H at all from the branch-and-bound perspective, while still obtaining useful data for H. Second, all other heuristics $H' \in \bar{\mathcal{H}}$ are run using their default solver frequencies, which simulates actual MIP solving, where many heuristics are interacting together.

4.2 Designing Node Features

So far, we have discussed collecting data that is realistic w.r.t. the target label. We now discuss the choice of features used to describe a given node N. We use a d-dimensional feature vector, $\mathbf{x}^N \in \mathbb{R}^d$, with $d = 49$, consisting of the features listed in Table 2. *Global features* describe the current state of the search using gap-related metrics. The (optimality) gap is defined as the relative difference between the global upper bound (i.e. the objective value of the best incumbent so far) and the global lower bound (i.e. the best possible objective value, due to LP relaxations). *Node LP features* use the solution of the LP relaxation at a node N to obtain certain indicative metrics. For instance, the feature "Num. of Active Constraints / Num. of Constraints" can be indicative of how sensitive the LP is to the fixing of additional variables, which is important for diving heuristics (an active constraint is one that is satisfied with equality at the LP solution). *Scoring Features for Fractional Variables* are inspired by the scoring functions that various diving heuristics use to select the next variable to fix. Details on the definitions of these functions are given in section 1.4.2 of [Hendel, 2011]. Essentially, for a given scoring function f : fractional variables $\rightarrow \mathbb{R}$, we compute the value of f for each fractional variable in the node's LP solution, compute statistics over the f values, and use those as features.

Global Features (4)
Optimality gap
Infinite gap?
Root LP value / Global Lower Bound
Root LP value / Global Upper Bound
Depth Features (2)
Node Depth / Max. Depth in Tree
Node Depth / Max. Possible Depth
Node LP Features (8)
Sum of variables' LP solution fractionalities / Num. of Fractional Variables
Num. of Fractional Variable / Num. of Integer Variables
Num. Variables Roundable Up (Down) / Num. of Integer Variables (x2)
Num. of Active Constraints / Num. of Constraints
Node is root?
Root LP value / Node LP value
Root LP value / Node Estimate
Scoring Features for Fractional Variables (35)
Number of Up Locks (x5) – Number of Down Locks (x5)
Normalized Objective Coefficient (x5)
Objective Gain (x5)
Distance to root LP solution (x5)
Vector Length (x5)
Pseudocost score (x5)

Table 2: List of the 49 features used. "Scoring features for fractional variables" are five statistics (mean, min., max., median, standard deviation) for each of seven metrics over fractional variables.

One trait of our features is that they are *naturally scaled*, i.e. each feature is appropriately divided by a scaling factor that depends only on the MIP instance (e.g. number of variables or constraints) or the node itself (e.g. number of fractional variables). Having appropriately scaled features is important for the convergence of many learning algorithms. However, that is not the only reason for emphasizing this aspect of our feature design. In fact, scaling is important in our setting because training data comes from multiple instances, each with its own dimensions, structure, etc. As such, the standard approach of scaling/normalizing data for training is not enough here: the scaling factors may not be directly applicable to a new instance's data. We have carefully crafted the features such that they can be scaled appropriately online, using only local information from the node and the instance.

5 Experimental Results

To evaluate the proposed framework, we modify the open-source MIP solver SCIP 3.2.1 [Gamrath *et al.*, 2016]; CPLEX 12.6.1 [IBM, 2014] is used as SCIP's LP solver. Machine

learning experiments use *scikit-learn* [Pedregosa *et al.*, 2011]. All experiments were run on a cluster of four 64-core machines with AMD 2.4 GHz processors and 264 GB RAM.

5.1 Oracle Learning

Heuristics. As a first phase, we learn a binary classifier that predicts incumbent success. We consider a set of ten heuristics implemented in SCIP, listed in the first column of Table 3. The ten heuristics were selected out of the 43 implemented in SCIP after excluding heuristics that are disabled by default (17), require a feasible solution (4), are too cheap (8), are for non-linear programs (3), or are for special constraints (1).

ML Setup. For a given heuristic H and a dataset $\mathcal{D}^H_{train} = \bigcup_{I \in \mathcal{I}_{train}} D^H_I$ collected from a set of training instances $\mathcal{I}_{train}$, we use *logistic regression* (LR) to learn a binary classification model, $\mathbf{w}_H \in \mathbb{R}^d$. The regularization parameter of LR is kept at a default of 1. Data points with label $y^N_H = 1$ are heavily weighted in the LR loss function to account for the extreme class imbalance we encounter [He and Garcia, 2009], as can be seen in column "Success Rate" of Table 3. The model $\mathbf{w}_H$ is simply a weight vector for the $d = 49$ node features described in Section 4.2, such that the dot product $\langle \mathbf{w}_H, \mathbf{x}^N \rangle$ between $\mathbf{w}_H$ and node N's feature vector $\mathbf{x}^N$ gives an estimate of the probability that heuristic H finds an incumbent at N. We have experimented with other ML models that have more capacity (and hyper-parameters) (SVM, gradient boosted trees), but have not observed any benefit from such models in either classification performance or learning/prediction time.

ML Results. We use leave-one-out cross-validation (LOOCV) on a per-instance basis: for each test instance I_{test}, a model is learned for a heuristic H using dataset $\mathcal{D}^H_{train}$, where $\mathcal{D}^H_{train}$ does not include any data from I_{test}; the model is then tested on I_{test}'s dataset, $D^H_{I_{test}}$.

Table 3 shows LOOCV results using 83 instances of the "Benchmark" set from MIPLIB2010 [Koch *et al.*, 2011], for which data was collected by running SCIP for 2 hours at most, per instance. First, observe that the datasets at hand are extremely imbalanced. For instance, the success rate (i.e. the fraction of nodes in the collected dataset for which the heuristic succeeds in finding an incumbent) of the *coefdiving* heuristic is 0.000192: only 1 in $5,000$ runs result in an incumbent. As such, the "Precision" of an ML success oracle must be better than random prediction (which succeeds with rate equal to the success rate). For each of the ten heuristics, Table 3 shows the average precision, recall and AUC-ROC, over instance datasets with at least one positive label data point (otherwise, these metrics are undefined). Fortunately, the average precision of the learned models is orders of magnitude better than the success rate: for *coefdiving*, the ML model is more than 100 times more precise in classifying incumbents than at random. Additionally, the recall of the models is satisfactory, with most incumbents being detected for most heuristics.

Despite the heterogeneous nature of the instances, our framework is able to learn oracle success models that are significantly better than random guessing, despite extreme class imbalance. Next, we study the impact of using the learned oracles, in conjunction with the RWS rule, on the solver's primal performance.

5.2 MIP Solving

Setup. While the ML results for the success oracles are positive, they are only of practical use if they can improve the performance of a state-of-the-art MIP solver w.r.t. primal metrics such as the primal integral. We use the learned oracles in conjunction with the Run-When-Successful rule to guide the decisions as to whether each of the ten heuristics of Table 3 should be run at each node. Specifically, for a given instance, the ten heuristics' models are loaded, and used to compute the probability of success of a heuristic given a node's feature vector. For other heuristics without ML oracles, we use their default settings in SCIP.

We compare our approach, referred to as ML, with the solver's default policy, DEF. For each of the 83 MIPLIB2010 "Benchmark" set instances, we run every policy with 5 different random permutations of the rows and columns of the instance; each instance-permutation pair is considered as a separate instance. This measure is a standard one for computational MIP studies, as it helps to control for the inherent "performance variability" in solvers – see [Lodi and Tramontani, 2013; Achterberg and Wunderling, 2013] for details.

MIP Results. Table 4 (left) summarizes the results. Table 4 (left) shows that our framework, ML, results in a reduction of 6% in the primal integral. Similarly, the time to the first and best incumbents are both improved by 22% and 1%, respectively. This is despite having an extremely heterogeneous set of training and testing instances. Our method makes better use of the heuristics it controls, as shown by the second set of rows in Table 4 (left): fewer calls are made to the ten heuristics, yet more incumbents are found by them on average, compared to DEF. Most notably, the success rate of ML-controlled heuristics is 1.79 times larger than that of DEF, and the number of incumbents found per second is 1.52 times larger. These figures, over a large set of benchmark instances, support our hypothesis: *dynamic decision-making for heuristics using the proposed framework improves the primal performance of an optimized state-of-the-art solver*.

5.3 Generalized Independent Set Problem

The experiments presented so far are on a heterogeneous set of MIP instances. However, in many real-world settings, one solves the same *homogeneous* family of problems, where instances differ only slightly in the number of constraints or variables, while maintaining the same overall combinatorial structure. To assess the effectiveness of our framework on a homogeneous instance set, we perform the same oracle learning and MIP solving experiments on instances of the *Generalized Independent Set Problem* (GISP) [Hochbaum and Pathria, 1997; Colombi *et al.*, 2016].

Recently, it has been shown that the GISP requires specialized techniques to obtain good feasible solutions [Colombi *et al.*, 2016], which motivated our choice of this problem. Given a graph $G(V, E)$, a subset of removable edges $E' \subseteq E$, revenues for each vertex and costs for each removable edge, GISP asks to select a subset of the vertices and a subset of removable edges that maximize the profit, i.e. the difference between selected vertex revenues and removable edge costs. No edge should exist between any two selected vertices u and v, i.e. $(u, v) \notin E$, or $(u, v) \in E'$ and (u, v) is removed.

Heuristic	Num. Instances	Num. Data Points	Success Rate	Precision		Recall		AUC-ROC	
				Mean +/- Std.	Median	Mean +/- Std.	Median	Mean +/- Std.	Median
coefdiving	44	2,635,296	0.0002	0.0264 +/- 0.0784	0.001	0.6552 +/- 0.3872	0.876	0.8543 +/- 0.1400	0.896
distributiondiving	51	2,721,704	0.0004	0.0255 +/- 0.0604	0.001	0.6715 +/- 0.3667	0.903	0.8075 +/- 0.1884	0.831
fracdiving	37	2,721,288	0.0001	0.0044 +/- 0.0093	0.001	0.6466 +/- 0.3810	0.688	0.7953 +/- 0.2184	0.878
intshifting	9	1,652,684	0.0001	0.1213 +/- 0.2291	0.001	0.4018 +/- 0.4347	0.177	0.8644 +/- 0.0823	0.865
linesearchdiving	34	2,552,685	0.0001	0.0170 +/- 0.0690	0.001	0.6794 +/- 0.3919	0.889	0.8187 +/- 0.1343	0.819
objpscostdiving	10	10,329	0.0127	0.3539 +/- 0.3814	0.131	0.5514 +/- 0.3769	0.486	0.8712 +/- 0.2247	0.996
pscostdiving	57	2,531,343	0.0007	0.0206 +/- 0.0430	0.002	0.6082 +/- 0.3636	0.716	0.7176 +/- 0.2359	0.773
rootsoldiving	6	6,047	0.0026	0.0990 +/- 0.1826	0	0.3333 +/- 0.4714	0	0.9599 +/- 0.0569	0.986
veclendiving	38	2,785,210	0.0002	0.0255 +/- 0.0687	0.003	0.7953 +/- 0.2799	0.936	0.7929 +/- 0.1923	0.829

Table 3: Leave-one-out cross-validation accuracy results for logistic regression on 10 primal heuristics in SCIP on MIPLIB2010 Benchmark. "AUC-ROC" is the area under the "receiver operating characteristic" curve. "Precision" is the fraction of points from the positive class out of all points classified as positive. "Recall" is the fraction of points from the positive class that are classified as positive. For both precision and recall, the results are using a threshold of 0.5 on the predicted probabilities.

MIPLIB – Num. Instances = 280	DEF	ML	ML/DEF
Primal integral	95.65	89.65	0.94
Time to first incumbent	34.23	26.60	0.78
Time to best incumbent	746.95	738.71	0.99
Total calls (ML heurs.)	755.19	514.77	0.68
Total time (ML heurs.)	124.38	101.88	0.82
Num. incumbents (ML heurs.)	1.85	2.45	1.33
Success Rate (ML heurs.)	0.00036	0.00064	1.79
Num. incs. per heur. sec. (ML heurs.)	0.00565	0.00860	1.52
Num. Instances Solved	170	172	1.01
Total time (BnB)	3,966.47	4,119.67	1.04
Total nodes (BnB)	27,458.77	27,904.43	1.02
Primal-dual integral	34,390.33	35,329.91	1.03

GISP – Num. Instances = 120	DEF	ML	ML/DEF
Primal integral	2,621.79	1,038.58	0.40
Time to first incumbent	0.19	0.19	1.00
Time to best incumbent	5,601.44	2,166.98	0.39
Total calls (ML heurs.)	49.37	63.59	1.29
Total time (ML heurs.)	194.42	610.64	3.14
Num. incumbents (ML heurs.)	1.48	2.69	1.82
Success Rate (ML heurs.)	0.02566	0.03710	1.45
Num. incs. per heur. sec. (ML heurs.)	0.00501	0.00319	0.64
Num. Instances Solved (% Gap)	0 (201.95)	0 (181.35)	N/A (0.90)
Total time (BnB)	7,200.00	7,200.00	1.00
Total nodes (BnB)	619.19	476.94	0.77
Primal-dual integral	520,674.41	454,162.12	0.87

Table 4: Summary of results on the MIPLIB2010 Benchmark set with 5 random permutations per instance (Left), and the GISP test set (Right); $t_{max} = 7,200$. For MIPLIB2010, instances requiring less than 10 minutes for either DEF or ML are excluded as too easy. Values shown are aggregates over instances: geometric means are used for all but *Num. Instances Solved* (count), *Num. incumbents*, *Success rate* and *Num. incs. per heur. sec.* (arithmetic means). For GISP, the *Primal integral* uses the best upper bound rather than the optimal solution.

We use the twelve graphs from the 1993 DIMACS Challenge [Johnson and Trick, 1996], also used in [Colombi *et al.*, 2016]. Six instances [1] are held out for data collection and training, and six others [2] for MIP testing. The graphs are dense, with training graphs having 125 – 300 nodes and 6963 – 20864 edges, testing graphs having 250 – 400 nodes and 21928 – 71819 edges. For each of the twelve graphs, we generate 20 GISP instances by randomizing the set of removable edges, as in [Colombi *et al.*, 2016]: each edge is in the set E' with probability α. We use $\alpha = 0.75$, each node has revenue 100 and each removable edge has cost 1, a configuration shown to be difficult w.r.t. finding feasible solutions (SET2-A in [Colombi *et al.*, 2016]). Note that, even for the same graph, its 20 instances have a different number of variables for removable edges and different constraints.

We collect data for eight diving heuristics (the heuristics listed in Table 3 except feaspump and intshifting, which SCIP did not run), and learn corresponding oracles. Then, we test the oracles on the 120 test instances that were not seen during learning. The primal integral requires an optimal or best integer solution, for which we use the best solution found by multi-threaded CPLEX 12.6.1 after 10 hours of solving. The results are shown in Table 4 (right).

A dramatic improvement in the primal integral can be observed, with ML costing only 0.4 of DEF. This improvement can be largely attributed to the reduction in the time to best incumbent, also down to 0.39 of DEF: ML needs around 1 hour less than DEF to find its best incumbent, over a time cutoff of 2 hours. As for the quality of the best incumbent, ML finds a better one than DEF in 93 of 120 of the instances (77%). For all 120 instances, ML has a better primal integral than DEF.

The larger reduction in the primal integral of GISP instances, as compared to the MIPLIB2010 Benchmark set, is consistent with the intuition that learning on homogeneous instances is easier than on heterogeneous ones. Note that the GISP training instances had fewer variables and constraints due to the smaller graphs, yet the classifiers were very effective on the larger test instances, indicating that generalization on homogeneous instance sets is possible.

6 Conclusions and Future Work

We have shown that intelligent decision-making for heuristics can boost the performance of a state-of-the-art optimization solver, even on instances for which the solver is already fine-tuned by experts. To our knowledge, this work represents the first systematic attempt at optimizing the use of heuristics in tree search. This work lays the ground for fruitful future extensions, such as more refined rules that take into account the running time of the heuristics and the amount of time remaining for the solver, approaches for more effective scheduling of heuristics within a node, and online learning of good rules.

[1] C125.9,keller4,brock200_2,p_hat300-1,gen200_p0.9_55,hamming8-4

[2] p_hat300-2, C250.9, p_hat300-3, brock400_2, MANN_a27, gen400_p0.9_75

References

[Abraham *et al.*, 2007] David J Abraham, Avrim Blum, and Tuomas Sandholm. Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges. In *Proceedings of the 8th ACM conference on Electronic commerce*, pages 295–304. ACM, 2007.

[Achterberg and Wunderling, 2013] Tobias Achterberg and Roland Wunderling. Mixed Integer Programming: Analyzing 12 Years of Progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*. Springer, 2013.

[Achterberg *et al.*, 2012] Tobias Achterberg, Timo Berthold, and Gregor Hendel. Rounding and propagation heuristics for mixed integer programming. In *Operations Research Proceedings 2011*, pages 71–76. Springer, 2012.

[Albers, 2003] Susanne Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1-2):3–26, 2003.

[Andrychowicz *et al.*, 2016] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[Barnhart *et al.*, 1998] Cynthia Barnhart, Natashia L Boland, Lloyd W Clarke, Ellis L Johnson, George L Nemhauser, and Rajesh G Shenoi. Flight string models for aircraft fleeting and routing. *Transportation science*, 32(3):208–220, 1998.

[Berthold, 2006] Timo Berthold. Primal Heuristics for Mixed Integer Programs. Master's thesis, Technische Universität Berlin, 2006.

[Colombi *et al.*, 2016] Marco Colombi, Renata Mansini, and Martin Savelsbergh. The generalized independent set problem: Polyhedral analysis and solution approaches. *European Journal of Operational Research*, 2016.

[Dilkina and Gomes, 2010] Bistra Dilkina and Carla P Gomes. Solving connected subgraph problems in wildlife conservation. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 102–116. Springer, 2010.

[Gamrath *et al.*, 2016] Gerald Gamrath, Tobias Fischer, Tristan Gally, Ambros M. Gleixner, Gregor Hendel, Thorsten Koch, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert Schwarz, Felipe Serrano, Yuji Shinano, Stefan Vigerske, Dieter Weninger, Michael Winkler, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 3.2. Technical Report 15-60, ZIB, Takustr.7, 14195 Berlin, 2016.

[He and Garcia, 2009] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[Hendel, 2011] Gregor Hendel. New rounding and propagation heuristics for mixed integer programming, 2011.

[Hochbaum and Pathria, 1997] Dorit S Hochbaum and Anu Pathria. Forest harvesting and minimum cuts: a new approach to handling spatial constraints. *Forest Science*, 43(4):544–554, 1997.

[Hutter *et al.*, 2009] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *JAIR*, 36(1):267–306, 2009.

[IBM, 2014] IBM. CPLEX User's Manual, Version 12.6.1, 2014.

[Johnson and Trick, 1996] David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

[Khalil *et al.*, 2016] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L Nemhauser, and Bistra N Dilkina. Learning to branch in mixed integer programming. In *AAAI*, pages 724–731, 2016.

[Koch *et al.*, 2011] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E Bixby, Emilie Danna, Gerald Gamrath, Ambros M Gleixner, Stefan Heinz, et al. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[Kruber *et al.*, 2016] Markus Kruber, Marco E Lübbecke, and Axel Parmentier. Learning when to use a decomposition. 2016.

[Lee *et al.*, 1999] Eva K Lee, Richard J Gallagher, David Silvern, Cheng-Shie Wuu, and Marco Zaider. Treatment planning for brachytherapy: an integer programming model, two computational approaches and experiments with permanent prostate implant planning. *Physics in Medicine and Biology*, 44(1):145, 1999.

[Lodi and Tramontani, 2013] Andrea Lodi and Andrea Tramontani. Performance variability in mixed-integer programming. *Tutorials in Operations Research: Theory Driven by Influential Applications*, pages 1–12, 2013.

[Nemhauser and Trick, 1998] George L Nemhauser and Michael A Trick. Scheduling a major college basketball conference. *Operations Research*, 46(1):1–8, 1998.

[Papageorgiou *et al.*, 2014] Dimitri J Papageorgiou, George L Nemhauser, Joel Sokol, Myun-Seok Cheon, and Ahmet B Keha. MIRPLib – A library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, 235(2):350–366, 2014.

[Pedregosa *et al.*, 2011] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Sabharwal *et al.*, 2012] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with uct. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 356–361. Springer, 2012.

[Samulowitz and Memisevic, 2007] Horst Samulowitz and Roland Memisevic. Learning to solve QBF. In *AAAI*, 2007.

[Zhang and Dietterich, 1995] Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120. Citeseer, 1995.