Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

Learning to Branch in Mixed Integer Programming

Elias B. Khalil[1], **Pierre Le Bodic**[2], **Le Song**[1], **George Nemhauser**[2], **Bistra Dilkina**[1]
[1]School of Computational Science & Engineering
[2]School of Industrial & Systems Engineering
Georgia Institute of Technology
{ekhalil3, lebodic, song9, gn3, bdilkina6}@gatech.edu

Abstract

The design of strategies for branching in Mixed Integer Programming (MIP) is guided by cycles of parameter tuning and offline experimentation on an extremely heterogeneous testbed, using the average performance. Once devised, these strategies (and their parameter settings) are essentially input-agnostic. To address these issues, we propose a machine learning (ML) framework for variable branching in MIP. Our method observes the decisions made by Strong Branching (SB), a time-consuming strategy that produces small search trees, collecting features that characterize the candidate branching variables at each node of the tree. Based on the collected data, we learn an easy-to-evaluate surrogate function that mimics the SB strategy, by means of solving a *learning-to-rank* problem, common in ML. The learned ranking function is then used for branching. The learning is instance-specific, and is performed on-the-fly while executing a branch-and-bound search to solve the instance. Experiments on benchmark instances indicate that our method produces significantly smaller search trees than existing heuristics, and is competitive with a state-of-the-art commercial solver.

Introduction

Recently, discrete optimization has been successfully leveraged to improve machine learning (ML) methodology (Roth and Yih 2005; Bertsimas, King, and Mazumder 2015). We will focus on the opposite direction of this fruitful cross-fertilization. We explore ways to harness ML approaches to improve the performance of branch-and-bound search for Mixed Integer Linear Programming (MIP). ML techniques have been successfully applied to a number of combinatorial search problems. For instance, UCT is a widely used online learning algorithm for Monte Carlo tree search (Kocsis and Szepesvári 2006), neural nets are used to combine heuristics in single-agent search (Samadi, Felner, and Schaeffer 2008), and the EM algorithm is used for solving constraint satisfaction problems (Hsu et al. 2007). In the context of MIP, some recent works propose ML techniques for constructing a portfolio of good parameter configurations for a MIP solver, and selecting the best configuration for a given instance (Hutter et al. 2009; Kadioglu et al. ; Xu et al. 2011). Alvarez et al. propose an ML approach to load balancing in parallel branch-and-bound for MIP (2015).

Variable selection for branching is considered to be a main component of modern MIP solvers (Linderoth and Savelsbergh 1999; Achterberg and Wunderling 2013). As part of the branch-and-bound algorithm for solving MIP problems (Nemhauser and Wolsey 1988), nodes in a search tree of partial assignments to variables must be expanded into (two) child nodes by selecting one of the unassigned variables and splitting its domain by adding additional constraints. Choosing good variables to branch on often leads to a dramatic reduction in terms of the number of nodes needed to solve an instance. In fact, a recent extensive computational study by researchers at IBM CPLEX, a leading commercial MIP solver, shows that using a naive variable selection strategy degrades performance by a factor of more than 8, compared to modern strategies (Achterberg and Wunderling 2013). However, in the same study, the authors note that "the results show that some progress has been achieved in branching variable selection since CPLEX 8.0 (2002 version), but certainly no break-through" (p. 458).

Traditional branching strategies fall into two main classes: Strong Branching (SB) approaches exhaustively test variables at each node, and choose the best one with respect to closing the gap between the best bound and the current best feasible solution value. Achterberg (2009) shows that SB can result in 65% fewer search tree nodes on average, compared to the state-of-the-art "hybrid branching" strategy. However, this comes at an increase of up to 44% in computation time, as more time is spent *per node*. On the other hand, Pseudocost (PC) branching strategies are engineered to imitate SB using a fraction of the computational effort, typically achieving a good trade-off between number of nodes and total time to solve a MIP. The design of such PC-based strategies has mostly been based on human intuition and extensive engineering, requiring significant manual tuning (initialization, statistical tests, tie-breaking, etc.). While that approach is important and constructive, we depart from it and propose to *learn* branching strategies directly from data.

We develop a novel framework for data-driven, on-the-fly design of variable selection strategies. By leveraging research in supervised ranking, we aim to produce strategies that gather the best of all properties: 1) using a small number of search nodes, approaching the good performance of SB, 2) maintaining a low computation footprint as in PC, and 3) selecting variables adaptively based on the properties of the

given instance. In the context of a single branch-and-bound search, in a first phase, we observe the decisions made by SB, and collect: features that characterize variables at each node of the tree, and labels that discriminate among candidate branching variables. In a second phase, we learn an easy-to-evaluate surrogate function that mimics SB, by solving a *learning-to-rank* problem common in ML (Liu 2009), with the collected data being used for training. In a third phase, the learned ranking function is used for branching.

Compared to recent machine learning methods for node and variable selection in MIP (He, Daumé III, and Eisner 2014; Alvarez, Louveaux, and Wehenkel 2014), our approach: 1) can be applied to instances *on-the-fly*, without an upfront offline training phase on a large set of instances, and 2) consists of solving a ranking problem, as opposed to regression or classification, which are less appropriate for variable selection. Its on-the-fly nature has the benefit of being instance-specific and of continuing the branch-and-bound seamlessly, without losing work when switching between learning and prediction. The ranking formulation is natural for variable selection, since the reference strategy (SB) effectively ranks variables at a node by a score, and picks the top-ranked variable, i.e. the score itself is not important.

We will show an instantiation of this framework using CPLEX, a state-of-the-art commercial MIP solver. We use a set of static and dynamic features computed for each candidate variable at a node, and SVM-based learning to estimate a two-level ranking of good and bad variables based on SB scores. Experiments on benchmark instances indicate that our method produces significantly smaller search trees than PC-based heuristics, and is competitive with CPLEX's default strategy in terms of number of nodes.

Background

Definition 1 (Mixed Integer Program) *Given matrix $A \in \mathbb{R}^{m\times n}$, vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, and a subset $I \subseteq \{1, ..., n\}$, the* mixed integer program *MIP = (A, b, c, I) is:*

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

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* of the MIP. A feasible solution $x^* \in X_{MIP}$ is called *optimal* if its objective value $c^T x^*$ is equal to z^*.

Definition 2 (LP relaxation of a MIP) *The* linear programming (LP) relaxation *of a MIP is:*

$$\check{z} = \min\{c^T x | Ax \leqslant b, x \in \mathbb{R}^n\}$$

When $X_{MIP} \neq \emptyset$, $\check{z}$ is a lower bound for z^*, i.e. $\check{z} \leqslant z^*$.

Branch-and-Bound

Branch-and-Bound (Nemhauser and Wolsey 1988) keeps a list of search *nodes*, each with a corresponding LP problem, obtained by relaxing the integrality requirements on the variables in I that have not been fixed to an integer value at an ancestor node. Let $\mathcal{L}$ denote the list of *active* nodes (i.e. nodes that have not been pruned nor branched on). Let $\bar{z}$ denote an upper bound on the optimum value z^*; initially, the bound $\bar{z}$ is set to ∞ or derived using heuristics that find an initial feasible solution. A lower (or dual) bound is derived by solving the LP relaxation of a MIP.

Two of the main decisions to be made during the algorithm are *node selection* and *variable selection*. In the former, the goal is to select an active node N_i from $\mathcal{L}$. Following that, the LP relaxation at N_i is solved, and its solution vector (if one exists) is $\check{x}^i$ with value $\check{z}^i$. N_i is pruned if its LP is infeasible, or if $\check{z}^i \geqslant \bar{z}$. If the LP solution $\check{x}^i$ is integer-feasible, i.e. $\check{x}^i \in X_{MIP}$, and $\check{z}^i < \bar{z}$, then z^* is updated to $\check{z}^i$, and $\check{x}^i$ is the new incumbent; the node is also pruned, as no better feasible solution can exist in its subtree.

If the selected node is not pruned, then it must be expanded into two child nodes. This is done by *branching* on an integer variable that has a fractional value in $\check{x}^i$, i.e. a variable $j \in I$ for which $\check{x}^i_j \notin \mathbb{Z}$, where $\check{x}^i_j$ denotes the value of variable j in the LP solution $\check{x}^i$. The two child nodes N^-_j and N^+_j are the result of branching on j downwards ($x_j \leqslant \lfloor \check{x}^i_j \rfloor$ in all descendants N_k) and upwards ($x_j \geqslant \lceil \check{x}^i_j \rceil$ in all descendants N_k). Variable selection deals with the problem of selecting that variable j from a set of possible candidates. Additional components of a MIP solver, such as cut generation and primal heuristics, are not discussed for simplicity.

In this work, we focus on learning a variable selection strategy. A generic variable selection strategy can be described as follows. Given a node N_i whose LP solution $\check{x}^i$ is not integer-feasible, let $\mathcal{C}_i \subseteq \{j \in I | \check{x}^i_j \notin \mathbb{Z}\}$ be the set of branching candidates. For all candidates $j \in \mathcal{C}_i$, calculate a score $s_j \in \mathbb{R}$, and return an index $j^* \in \mathcal{C}_i$ with $s_{j^*} = \max_{j\in\mathcal{C}_i} s_j$. Two standard approaches to computing the variable scores are briefly described next; we refer to (Achterberg 2009) for more details.

Strong Branching (SB)

Typically, the measure for the quality of branching on a variable x_j is the improvement in the dual bound. Consider a node N with LP value $\check{z}$, LP solution $\check{x}$, and candidate variable set $\mathcal{C}$. The two children N^-_j and N^+_j, resulting from branching on j downwards and upwards, have (feasible) LP values $\check{z}^-_j$ and $\check{z}^+_j$, respectively. If N^-_j (N^+_j) is infeasible, $\check{z}^-_j$ ($\check{z}^+_j$) is set to a very large value. The changes in objective value are then $\Delta^-_j = \check{z}^-_j - \check{z}$ and $\Delta^+_j = \check{z}^+_j - \check{z}$. To map these two values to a single score, let ϵ be a small constant (e.g. 10^{-6}), then:

$$SB_j = \text{score}\Big(\max\{\Delta^-_j, \epsilon\}, \max\{\Delta^+_j, \epsilon\}\Big) \qquad (1)$$

A product is typically used for scoring, i.e. $\text{score}(a, b) = a \times b$. SB attempts to find the variable with the maximum score (1), by simulating the branching process for the candidate variables in $\mathcal{C}$, and computing the scores as in (1).

While SB directly optimizes (1), it is computationally expensive: solving two LP problems for each candidate variable using the simplex algorithm is often time-consuming. The time spent per node ends up overshadowing the time saved due to a smaller search tree. Simpler but faster heuristics are hence preferred.

Pseudocost Branching (PC)

Pseudocosts are historical quantities aggregated for each variable during the search. The upwards (downwards) PC of a variable x_j is the average unit objective gain taken over upwards (downwards) branchings on x_j in previous nodes; we refer to this quantity as Ψ_j^+ (Ψ_j^-). Pseudocost branching at node N with LP solution $\check{x}$ consists in computing values:

$$PC_j = \text{score}\Big((\check{x}_j - \lfloor \check{x}_j \rfloor)\Psi_j^-, (\lceil \check{x}_j \rceil - \check{x}_j)\Psi_j^+\Big) \qquad (2)$$

and choosing the variable with the largest such value. As in SB, the product is used to combine the downwards and upwards values. One standard way to initialize the pseudocost values is by applying strong branching once for each integer variable, at the first node at which it is fractional (Linderoth and Savelsbergh 1999). We will refer to this PC strategy with SB initialization as *pseudocost branching* (PC).

Overall, PC-based strategies are input-agnostic, since the variable selection rule is always the same (branch on the variable with largest PC score (2)), and is dependent on extensive parameter tuning on instances that may be completely different in structure from the actual input. This is the case for modern strategies such as *reliability branching* (Achterberg, Koch, and Martin 2005) and *hybrid branching* (Achterberg and Berthold 2009). Additionally, experiments show that PC-based strategies are still far from matching the node-efficiency of SB, requiring 65% more nodes than the latter, on average (Table 5.1 in (Achterberg 2009)).

Overview of our Framework

We now introduce a framework for learning to branch in MIP. Our intuition is that by observing and recording the rankings induced by SB, we can learn a function of the variables' features that will rank them in a similar way, without the need for the expensive SB computations. Given a MIP instance and some parameters, we proceed in three phases:

1. Data collection: for a limited number of nodes θ, SB is used as a branching strategy. At each node, the computed SB scores are used to assign labels to the candidate variables; and corresponding variable features are also extracted. All information is compiled in a *training dataset*.
2. Model learning: the dataset is fed into a learning-to-rank algorithm that outputs a vector of weights for the features, such that some loss function is minimized over the training dataset.
3. ML-based branching: SB is no longer used, and the learned weight vector is used to score variables, branching on the one with maximum score until termination.

We highlight how the proposed method satisfies three desirable properties, before confirming so experimentally.

1. *Node-efficiency:* the learned ranking model uses SB scores and node-specific variable features as training data, and is thus expected to *imitate* the SB choices more closely than PC, yielding smaller search trees.
2. *Time-efficiency:* SB is used in the first phase only, typically for a few hundred nodes. The time required for learning (second phase) is small, and the third phase is dominated by feature computations, which are designed to be much cheaper than solving the LPs as does SB.
3. *Adaptiveness:* the learned ranking model is instance-specific, as it assigns different weights to features depending on the collected data.

Next, we describe each of the phases in more detail.

Data Collection

In this first phase, we aim to construct a training dataset from which we can learn a model that mimics SB's ranking. As such, the branch-and-bound algorithm is run with SB as the variable selection strategy, for a fixed number of nodes θ. If a node is fathomed (e.g. for infeasibility) during this phase, it does not count towards θ. At each node N_i, SB is run on a set of candidate variables $\mathcal{C}_i$, where $|\mathcal{C}_i| \leqslant \kappa$, and κ is typically in the range 10–20 in SB implementations (e.g. CPLEX). The variables in $\mathcal{C}_i$ are chosen among the fractional integer variables in the node's LP solution in standard ways (e.g. sorting by PC score (Achterberg, Koch, and Martin 2005)).

The training data then comprises:

- a set of search tree nodes $\mathcal{N} = \{N_1, ..., N_\theta\}$;
- a set of candidate variables $\mathcal{C}_i$ for a given node $N_i \in \mathcal{N}$;
- labels $\mathbf{y}^i = \{y_j^i \in \Omega | j \in \mathcal{C}_i\}$ for the candidate variables at each node i, where Ω is the domain of the labels;
- a feature map $\Phi : \mathcal{X} \times \mathcal{N} \to [0,1]^p$, where $\mathcal{X} = \{x_1, ..., x_n\}$. $\Phi(x_j, N_i)$ describes variable x_j at node N_i with p features.

Notice how the same variable x_j may appear in both $\mathcal{C}_i$ and $\mathcal{C}_k$ for $i \neq k$, yet with different labels and feature values. This is a result of the choice of feature map Φ, which maps a variable *at the node in question* to features, capturing different contexts encountered during the search.

The specification and representation of the labels and features is a core issue when modeling a problem using machine learning. We present intuitive, simple guidelines for doing so in the context of branching.

Labels

A label is a value assigned to each variable in $\mathcal{C}_i$, such that better variables w.r.t. to the SB score have larger labels. We consider the SB score to be a sort of "gold standard" for scoring variables, hence labels based on such a score are a good target for learning.

We propose a simple and intuitive *binary labeling* scheme, i.e. $\Omega = \{0, 1\}$. Let $\mathbf{SB}^i$ denote the vector of SB scores for variables in $\mathcal{C}_i$ of node N_i, and $SB_*^i = \max_{j \in \mathcal{C}_i}\{SB_j^i\}$. A label y_j^i is computed by transforming the corresponding SB score SB_j^i as follows:

$$y_j^i = \begin{cases} 1, & \text{if } SB_j^i \geqslant (1-\alpha) \cdot SB_*^i \\ 0, & \text{otherwise} \end{cases} \qquad (3)$$

where $\alpha \in [0, 1]$ is the fraction of the maximum SB score that a variable should have in order to get a '1' label. For instance, when $\alpha = 0.2$, variables whose SB scores are within 20% of the maximum score are assigned a label of '1'.

The labels resulting from the transformation in (3) emphasize our focus on the best variables w.r.t. SB, and are compatible with learning-to-rank formulations, as we will see in later sections. While other labeling schemes are possible (e.g. grading on a scale of 1 to 5), we prefer the simple binary labels for the purposes of this work. Note that although our labels are 0/1, our setting is that of *bipartite ranking* (i.e. ranking with 0/1 labels), and not binary classification. Having a binary labeling scheme, as opposed to using a full ranking among all candidate variables, helps to avoid learning to correctly rank variables with low SB scores relative to each other, a task that is irrelevant to making a good branching choice. In addition, instead of assigning only the variable with maximum SB score a label of '1' and all others a label of '0', our labeling scheme has a relaxed definition of "top" branching variable. This captures the fact that at some search nodes several candidate variables will have high SB scores, and will likely be good branching candidates.

Features

We compute a feature vector that describes a variable's state with respect to the node. The features can be split into two sets: *atomic features* are computed based on the node LP and candidate variable, whereas *interaction features* are the result of a product of two atomic features. The final feature vector can be seen as an explicit feature mapping, equivalent to the degree-2 polynomial kernel $K(y, z) = (y^T z + 1)^2$, in the space of atomic features.

The 72 atomic features are summarized in Table 1. They consist of counts and statistics (all or some of: mean, standard deviation (stdev.), minimum, maximum) capturing a variable's structural *role* within the node LP, as well as its historical performance. The atomic features are either *static* or *dynamic*. Static features are pre-computed once at the root node, and do not depend on the specific node LP.

For each feature, we normalize its values to the range $[0, 1]$ across the candidate variables at each node. This type of normalization is referred to as *query-based normalization* in the IR literature (Liu et al. 2007). This produces an additional layer of dynamism, as the final value of a feature for a given variable depends on the set of candidate variables being considered at that node. For example, this results in static features of a variable taking on different normalized values across different nodes.

All atomic features can either be accessed through the solver API in $O(1)$, or computed in $O(nnz(A))$, i.e. in time linear in the number of non-zero elements of the coefficient matrix A, which makes data collection efficient.

Learning a Variable Ranking Function

Given the training data, we would like to learn a linear function of the features $f : \mathbb{R}^p \rightarrow \mathbb{R}, f(\Phi(x_j, N_i)) = \mathbf{w}^T \Phi(x_j, N_i)$ that minimizes a loss function over the training data. In ML terms, this problem is one of *empirical risk minimization*.

We define $\hat{\mathbf{y}}^i \in \mathbb{R}^\kappa$ to be the vector of values resulting from applying f to every variable in $\mathcal{C}_i$, i.e. $\hat{y}^i_j = f(\Phi(x_j, N_i))$. Formally, the learning problem can be written as that of finding:

$$\mathbf{w}^* = \underset{\mathbf{w} \in \mathbb{R}^p}{\arg\min} \sum_{N_i \in \mathcal{N}} \ell(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \lambda \|\mathbf{w}\|_2^2 \qquad (4)$$

The (structured) *loss function* $\ell : \mathbb{R}^\kappa \times \mathbb{R}^\kappa \rightarrow \mathbb{R}$ measures the loss resulting from ranking the variables at a node N_i according to $\hat{\mathbf{y}}^\mathbf{i}$, as opposed to the true labels $\mathbf{y}^i$, and $\lambda > 0$ is a regularization parameter that helps to avoid overfitting.

Fortunately, this problem has been studied in the context of web search, where a system is given a set of queries, and must rank a set of documents by order of relevance. We leverage existing research in information retrieval (IR) to address our problem (4). Specifically, our choice of the loss function $\ell(.)$ is based on *pairwise loss*, a common IR measure (Joachims 2002; Liu 2009).

A Pairwise Ranking Formulation

For a node N_i, consider the set of pairs: $\mathcal{P}_i = \left\{(x_j, x_k) \mid j, k \in \mathcal{C}_i \text{ and } y^i_j > y^i_k\right\}$. Each pair in any set $\mathcal{P}_i$ includes two variables at node N_i: one with label 1, and another with label 0. In order to rank variables similarly to how SB ranks them, we could learn an f that violates as few as possible of the following *pairwise ordering* constraints:

$$\forall i \in \{1, ..., \theta\} : \forall (x_j, x_k) \in \mathcal{P}_i : \hat{y}^i_j > \hat{y}^i_k \qquad (5)$$

Violating a constraint in (5) is equivalent to learning a model for which $\hat{y}^i_j \leqslant \hat{y}^i_k$ for some node N_i and variables x_j, x_k: a variable with label $y^i_k = 0$ is ranked the same or higher than one with label $y^i_j = 1$. Minimizing such violations means that the model is more likely to rank good variables higher than bad ones, which is our goal in variable selection. Note that other formulations for the *bipartite ranking* problem are also possible, e.g. (Rudin 2009), but the one we adopt is simple, can be optimized efficiently, and works well in practice.

While minimizing the number of violated constraints in (5) is NP-hard, Joachims (2002) proposed a Support Vector Machine (SVM) approach that optimizes an upper bound on that number. SVM^{rank} is an efficient open-source package that implements that algorithm with the appropriate loss function ℓ (Joachims 2006). We use SVM^{rank} with a cost-sensitive loss, weighting each term in the sum in (4) by $1/|\mathcal{P}_i|$ (parameter "-l 2"). This variant is more suitable, as it reduces the bias towards nodes with more "good" variables.

Branching with the Learned Function

After θ nodes have been processed, and the vector $\mathbf{w}$ has been learned, we switch from SB to the function f as variable selection strategy. At each new node N_i, we compute the feature vector $\Phi(x_j, N_i)$ for each variable $j \in \mathcal{C}_i$, and branch on the variable j^* with maximum score $s_{j^*} = \max_{j \in \mathcal{C}_i} s_j$, where $s_j = f(\Phi(x_j, N_i))$. The complexity of this procedure is $O(nnz(A) + p \cdot \kappa)$, where the first term is due to the computation of features, and the second is due to feature normalization and scoring by dot product (using p features and at most κ candidate variables per node). Compared to SB, which requires many dual simplex iterations, experiments show that our approach is much more efficient.

Feature	Description	Count	Reference
Static Features (18)			
Objective function coeffs.	Value of the coefficient (raw, positive only, negative only)	3	
Num. constraints	Number of constraints that the variable participates in (with a non-zero coefficient)	1	
Stats. for constraint degrees	The *degree of a constraint* is the number of variables that participate in it. A variable may participate in multiple constraints, and statistics over those constraints' degrees are used. The constraint degree is computed on the root LP (mean, stdev., min, max)	4	
Stats. for constraint coeffs.	A variable's positive (negative) coefficients in the constraints it participates in (count, mean, stdev., min, max)	10	
Dynamic Features (54)			
Slack and ceil distances	$\min\{\check{x}^i_j - \lfloor \check{x}^i_j \rfloor, \lceil \check{x}^i_j \rceil - \check{x}^i_j\}$ and $\lceil \check{x}^i_j \rceil - \check{x}^i_j$	2	
Pseudocosts	Upwards and downwards values, and their corresponding ratio, sum and product, weighted by the fractionality of x_j	5	(Achterberg 2009)
Infeasibility statistics	Number and fraction of nodes for which applying SB to variable x_j led to one (two) infeasible children (during data collection)	4	
Stats. for constraint degrees	A dynamic variant of the static version above. Here, the constraint degrees are on the current node's LP. The ratios of the static mean, maximum and minimum to their dynamic counterparts are also features	7	
Min/max for ratios of constraint coeffs. to RHS	Minimum and maximum ratios across positive and negative right-hand-sides (RHS)	4	(Alvarez, Louveaux, and Wehenkel 2014)
Min/max for one-to-all coefficient ratios	The statistics are over the ratios of a variable's coefficient, to the sum over all other variables' coefficients, for a given constraint. Four versions of these ratios are considered: positive (negative) coefficient to sum of positive (negative) coefficients	8	(Alvarez, Louveaux, and Wehenkel 2014)
Stats. for active constraint coefficients	An active constraint at a node LP is one which is binding with equality at the optimum. We consider 4 weighting schemes for an active constraint: unit weight, inverse of the sum of the coefficients of all variables in constraint, inverse of the sum of the coefficients of only candidate variables in constraint, dual cost of the constraint. Given the absolute value of the coefficients of x_j in the active constraints, we compute the sum, mean, stdev., max. and min. of those values, for each of the weighting schemes. We also compute the weighted number of active constraints that x_j is in, with the same 4 weightings	24	(Patel and Chinneck 2007)

Table 1: Description of the atomic features.

Experimental Results

Setup. We use the C API of IBM ILOG CPLEX 12.6.1 to implement various strategies using control callbacks, in single-thread mode. To evaluate the performance of any variable selection strategy $\mathcal{A}$, the strategy is run on a set of instances with a time cut-off of t_{max} seconds. An instance $\mathcal{I}$ is *solved* by strategy $\mathcal{A}$ if and only if the run terminates within the tolerance gaps (we use default CPLEX values). If an instance $\mathcal{I}$ is not solved by the time cut-off, it is referred to as *unsolved*. All experiments were run on a cluster of four 64-core machines with AMD 2.4 GHz processors and 264 GB of memory; each run was limited to 2 GB of memory, and no run failed for memory reasons.

To isolate the effects of changing the variable selection strategy, we provide the optimal value as upper cutoff to CPLEX before the start of the search. This measure reduces the effect of node selection on the search, as the primal bound is given by the upper cutoff, and the order in which nodes are expanded has little impact on the tree itself. Additionally, cuts are allowed at the root only, and primal heuristics are disabled. These measures are common in branching studies (Linderoth and Savelsbergh 1999; Fischetti and Monaci 2012; Karzan, Nemhauser, and Savelsbergh 2009), since they eliminate the interference between variable selection and other components of the solver, such as node selection. This also reduces *performance variability*, which we discuss in the next section.

Instances. We use the "Benchmark" set from MIPLIB2010 as our test set; we refer to (Koch et al. 2011) for details. This set was designed to span a variety of problem classes, applications, dimensions, levels of difficulty, etc., and is routinely used for evaluating branching strategies. The "Benchmark" set consists of 87 instances that can be solved by at least one commercial solver within 2 hours on a high-end PC. Note that since we turn off multi-threading and cuts beyond the root, we cannot expect to solve all instances within 2 hours. Hence, we set the time cut-off t_{max} to 5 hours (18,000 seconds). Three infeasible instances are excluded.

For each of the 84 instances we consider, we run every strategy with 10 different random seeds, for every variable selection strategy. Recent studies have shown that MIP solvers can be very sensitive to seemingly performance-neutral perturbations to their inputs (Lodi and Tramontani 2013; Achterberg and Wunderling 2013). Therefore, runs with different seeds are necessary for obtaining meaningful results. In CPLEX, such perturbations can be induced by changing CPLEX's internal random seed via its C API.

Branching strategies. We experiment with five strategies. `CPLEX-D` is the strategy that branches on the variable chosen by the solver with its default variable selection rule (as set by `CPX_PARAM_VARSEL`); this is done within a callback, as for all other strategies. Up until 2013, CPLEX developers report that the default selection rule is "a version of *hybrid branching*" (Achterberg and Wunderling 2013). SB refers to Strong Branching, while PC refers to pseudocost branching with SB initialization of the PC values (Linderoth and Savelsbergh 1999). SB+PC is a hybrid of SB for the first

		CPLEX-D	SB	PC	SB+PC	SB+ML
Unsolved Instances	All (523)	11	129	66	63	**52**
	Easy (255)	0	12	15	14	**13**
	Medium (120)	2	43	22	22	**17**
	Hard (148)	9	74	29	27	**22**
Num. Nodes	All (523)	46,633	33,072	92,662	70,455	**59,223**
	Easy (255)	3,255	3,610	7,931	5,224	**5,124**
	Medium (120)	173,417	121,923	395,199	288,916	**234,093**
	Hard (148)	1,570,891	519,878	1,971,333	1,979,660	**1,314,263**
Total Time	All (523)	499	2,263	**960**	1,093	1,059
	Easy (255)	111	602	**243**	361	382
	Medium (120)	1,123	6,169	2,493	1,892	**1,776**
	Hard (148)	3,421	9,803	4,705	4,718	**4,039**

Table 2: Summary of experimental results. "Unsolved instances" are counts, "Num. nodes" and "Total time" (in seconds) are shifted geometric means over instances with shifts 10 and 1, respectively. Lower is better, and the best value in each row among PC, SB+PC and SB+ML is in bold.

$\theta = 500$ nodes, and PC afterwards; a similar strategy appears in (Fischetti and Monaci 2012). SB+ML is our proposed method with $\theta = 500$. We use $\alpha = 0.2$ and SVM^{rank} with a trade-off parameter $C = 0.1$ between training error and margin (λ in (4) is a function of C). We varied $\alpha \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ and $C \in \{0.001, 0.01, 0.1, 1\}$ (0.01 is the default in SVM^{rank}), and found that SB+ML performs similarly. For SB, SB+PC and SB+ML, κ is set to 10, and all SB calls are limited to 50 dual simplex iterations, as in (Fischetti and Monaci 2012).

We do not know what additional embellishments CPLEX uses, and report results for its default strategy as CPLEX-D. Even when CPLEX's default strategy is roughly known, callback implementations are much less node (and time) efficient; see Table 4 in (Fischetti and Monaci 2012) for an example. Hence, the main comparisons of our strategy are to PC and SB+PC. Most related to SB+ML is SB+PC, since both strategies share the same exact search tree up to θ nodes, then diverge by branching according to the variables selected by the learned ranking model and PC, respectively. Note that any extra information that CPLEX uses internally to score variables can be incorporated into our framework as features or labels, as can reliability branching scores, etc.

Results. We consider three metrics for evaluating branching strategies: the *number of unsolved instances*, the *number of nodes* to solve the instance and the *total time* to solve the instance. Since our hypothesis is that our strategy SB+ML is better at variable selection, the main criterion for comparison is the number of nodes. The case for focusing on nodes versus time in benchmarking branching methods is discussed in depth in (Hooker 1995). Total time is also important, and we have accordingly optimized our implementation of SB+ML to some extent. However, we believe that the time-efficiency of SB+ML may be improved, for example with access to CPLEX's internal data structures.

An instance with a different random seed is considered as a separate instance; this was suggested first by Danna (2008). Results for averages over seeds (per instance) are consistent with what we present here, but are not included due to space constraints. Of the 840 instances considered, we exclude: 184 instances solved by all strategies in 1,000 nodes (too easy), 82 instances not solved by any strategy in 5 hours (too hard), 3 instances that are flagged as infeasible by at least one strategy, and 48 instances for which CPLEX aborted. For Table 2, "All" refers to all instances considered, "Easy" and "Medium" refer to instances solved by CPLEX-D in less than 50,000 and 500,000 nodes, respectively; otherwise, an instance is classified as "Hard". When a strategy does not solve an instance, t_{max} is reported as the total time, and the number of nodes at termination is reported as the number of nodes. Note that this may be biased towards strategies that are slower, processing fewer nodes and solving fewer instances. This issue is inherent to MIP benchmarking (Achterberg and Wunderling 2013), and we will address it in Tables 3 and 4. Similar experimental procedures are used in recent work on branching (Fischetti and Monaci 2012; Karzan, Nemhauser, and Savelsbergh 2009).

Table 2 shows that SB+ML solves more instances than both PC and SB+PC. Most notably, SB+ML clearly outperforms PC and SB+PC, requiring respectively around 36% and 16% fewer nodes on average ("All" set). Currently, SB+ML spends, on average, 18 milliseconds (ms) per node, while SB, PC and SB+PC spend 68, 10 and 15 ms, respectively. Although SB+ML spends more time per node than SB+PC due to feature computations, it incurs a comparable total time, as it saves in the number of nodes. Compared to PC, our method is slower by 10% on average over all instances, but is 28% and 14% faster for instances in "Medium" and "Hard", respectively. A more optimized implementation of the feature computations of SB+ML is likely to make it faster than competitors for the "Easy" set too. It is clear that SB is not applicable, as it requires twice as much total time as the three competing methods, and times out on many more instances.

Table 3 addresses the bias in averaging over the number of nodes on unsolved instances in Table 2. Here, we con-

	CPLEX-D	SB	PC	SB+PC	SB+ML
CPLEX-D		1.39 (389)	0.64 (449)	0.84 (452)	0.97 (463)
SB	0.72 (389)		0.47 (389)	0.61 (388)	0.76 (389)
PC	1.56 (449)	2.11 (389)		1.34 (445)	1.59 (450)
SB+PC	1.20 (452)	1.63 (388)	0.75 (445)		1.22 (454)
SB+ML	1.03 (463)	1.32 (389)	0.63 (450)	0.82 (454)	

Table 3: Ratios for the shifted geometric means (shift 10) over nodes on instances solved by both strategies. The first value in a cell in row $\mathcal{A}$ and column $\mathcal{B}$ is the ratio of the average number of nodes used by $\mathcal{A}$ to that of $\mathcal{B}$. The second value is the number of instances solved by both $\mathcal{A}$ and $\mathcal{B}$.

	CPLEX-D	SB	PC	SB+PC
CPLEX-D				
SB	5/264/0/125/123			
PC	8/164/0/285/63	68/63/0/326/5		
SB+PC	8/227/0/225/60	72/66/7/315/6	15/320/0/125/12	
SB+ML	8/267/0/196/49	82/96/7/286/5	21/355/0/95/7	17/300/58/96/6

Table 4: Win-tie-loss matrix for the number of nodes. A quintuple in a cell in row $\mathcal{A}$ and column $\mathcal{B}$ has: the number of absolute wins, wins, ties, losses and absolute losses for $\mathcal{A}$ against $\mathcal{B}$, w.r.t. the number of nodes.

sider *only instances solved by both strategies*, for every pair of strategies, and compute shifted geometric means on that subset of the instances. The node ratios shown in Table 3 are in line with the previous two tables. SB+ML needs 37% and 18% fewer nodes than PC and SB+PC, respectively, and only 3% more nodes than CPLEX-D. Interestingly, SB+ML requires 32% more nodes than SB, while CPLEX-D requires 39%; PC and SB+PC are dramatically worse.

Table 4 shows *win-tie-loss* counts for each pair of strategies, comparing the number of nodes needed to solve an instance head-to-head, and avoiding the averaging used in the two previous tables. An *absolute win* for strategy $\mathcal{A}$ over $\mathcal{B}$ on instance $\mathcal{I}$ is recorded iff $\mathcal{A}$ solves $\mathcal{I}$ whereas $\mathcal{B}$ does not; a *win* occurs when both $\mathcal{A}$ and $\mathcal{B}$ solve $\mathcal{I}$, and $\mathcal{A}$ does so in strictly fewer nodes than $\mathcal{B}$; *absolute loss* and *loss* are defined analogously. A *tie* occurs when $\mathcal{A}$ and $\mathcal{B}$ solve $\mathcal{I}$ in the same number of nodes. Table 4 is consistent with Table 2, showing that SB+ML outperforms PC and SB+PC head-to-head, solving many more instances in fewer nodes. SB+ML has 21 absolute wins and 355 wins over PC, while PC has only 7 absolute wins and 95 wins over SB+ML in terms of number of nodes. SB+ML is on par with CPLEX-D in terms of overall wins, with fewer absolute wins (8 vs. 49), but more wins on instances solved by both (267 vs. 196).

Feature analysis. To evaluate whether the instance-specific branching models we use are warranted, we measure how similar the learned models are in terms of the feature ranking induced by the learned (absolute) feature weights. We analyze models for 57 different instances from one seed, and find that the average pairwise Spearman's rank correlation coefficient (a value between -1 and 1) is 0.16, indicating only little positive correlation. This implies that instance-specific variable ranking models are indeed useful. Examining which features are more informative, we find that combining structural (e.g. active constraint features) and historical features such as PC product results in interaction features that often have large (absolute) weights across models.

Conclusions and Future Directions

We have proposed the first successful ML framework for variable selection in MIP, an approach which may also benefit other components of the MIP solver such as cutting planes and node selection. The framework can be extended in several directions, such as dynamically adjusting the number of training nodes θ or learning models multiple times in adaptation to the search progress. Beyond the batch supervised ranking approach we used, online and reinforcement learning formulations may be interesting to explore, given the structured, sequential nature of the variable selection task.

Acknowledgments

The work depicted in this paper was partially sponsored by DARPA under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited.' L.S. is supported in part by NSF/NIH BIGDATA 1R01GM108341, ONR N00014-15-1-2340, NSF IIS-1218749, and NSF CAREER IIS-1350983. P.L. and G.N. are supported by AFOSR FA9550-12-1-0151 and NSF CCF-1415460.

References

Achterberg, T., and Berthold, T. 2009. Hybrid branching. In *CPAIOR*. Springer. 309–311.

Achterberg, T., and Wunderling, R. 2013. Mixed integer programming: Analyzing 12 years of progress. In Jünger, M., and Reinelt, G., eds., *Facets of Combinatorial Optimization*. Springer.

Achterberg, T.; Koch, T.; and Martin, A. 2005. Branching rules revisited. *Operations Research Letters* 33(1):42–54.

Achterberg, T. 2009. *Constraint Integer Programming*. Ph.D. Dissertation, Technische Universität Berlin.

Alvarez, A. M.; Louveaux, Q.; and Wehenkel, L. 2014. A supervised machine learning approach to variable branching in branch-and-bound. Technical Report, Université de Liège.

Alvarez, A. M.; Wehenkel, L.; and Louveaux, Q. 2015. Machine learning to balance the load in parallel branch-and-bound. Technical Report, Université de Liège.

Bertsimas, D.; King, A.; and Mazumder, R. 2015. Best subset selection via a modern optimization lens. *arXiv preprint arXiv:1507.03133*.

Danna, E. 2008. Performance variability in mixed integer programming. Presented at Workshop on Mixed Integer Programming, Columbia University, New York.

Fischetti, M., and Monaci, M. 2012. Branching on nonchimerical fractionalities. *Operations Research Letters* 40(3):159–164.

He, H.; Daumé III, H.; and Eisner, J. 2014. Learning to search in branch-and-bound algorithms. In *NIPS*.

Hooker, J. N. 1995. Testing heuristics: We have it all wrong. *Journal of Heuristics* 1(1):33–42.

Hsu, E. I.; Kitching, M.; Bacchus, F.; and McIlraith, S. A. 2007. Using expectation maximization to find likely assignments for solving CSP's. In *AAAI*, 224–230.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *JAIR* 36(1):267–306.

Joachims, T. 2002. Optimizing search engines using clickthrough data. In *KDD*, 133–142.

Joachims, T. 2006. Training linear SVMs in linear time. In *KDD*, 217–226.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. ISAC – instance-specific algorithm configuration. In *ECAI*.

Karzan, F. K.; Nemhauser, G. L.; and Savelsbergh, M. W. 2009. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation* 1(4):249–293.

Koch, T.; Achterberg, T.; Andersen, E.; Bastert, O.; Berthold, T.; Bixby, R. E.; Danna, E.; Gamrath, G.; Gleixner, A. M.; Heinz, S.; et al. 2011. MIPLIB 2010. *Mathematical Programming Computation* 3(2):103–163.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*.

Linderoth, J. T., and Savelsbergh, M. W. 1999. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing* 11(2):173–187.

Liu, T.-Y.; Xu, J.; Qin, T.; Xiong, W.; and Li, H. 2007. LETOR: Benchmark dataset for research on learning to rank for information retrieval. In *Proceedings of SIGIR Workshop on Learning to Rank for Information Retrieval*, 3–10.

Liu, T.-Y. 2009. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval* 3(3):225–331.

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

Nemhauser, G. L., and Wolsey, L. A. 1988. *Integer and Combinatorial Optimization*. John Wiley & Sons.

Patel, J., and Chinneck, J. W. 2007. Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming* 110(3):445–474.

Roth, D., and Yih, W. 2005. Integer linear programming inference for conditional random fields. In *ICML*, 736–743.

Rudin, C. 2009. The p-norm push: A simple convex ranking algorithm that concentrates at the top of the list. *JMLR* 10:2233–2271.

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Learning from multiple heuristics. In *AAAI*, 357–362.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2011. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *Proceedings of the 18th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 16–30.