Noname manuscript No.
(will be inserted by the editor)

Learning to Use Local Cuts

Timo Berthold · Matteo Francobaldi · Gregor Hendel

Received: date / Accepted: date

Abstract An essential component in modern solvers for mixed-integer (linear) programs (MIPs) is the separation of additional inequalities (cutting planes) to tighten the linear programming relaxation. Various algorithmic decisions are necessary when integrating cutting plane methods into a branch-and-bound (B&B) solver as there is always the trade-off between the efficiency of the cuts and their costs, given that they tend to slow down the solution time of the relaxation. One of the most crucial questions is: Should cuts only be generated globally at the root or also locally at nodes of the tree? We address this question by a machine learning approach for which we train a regression forest to predict the speed-up (or slow-down) provided by using local cuts. We demonstrate with an open implementation that this helps to improve the performance of the FICO Xpress MIP solver on a public test set of general MIP instances. We further report on the impact of a practical implementation inside Xpress on a large, diverse set of real-world industry MIPs.

Keywords Mixed Integer Programming · Cutting Planes · Machine Learning

Mathematics Subject Classification (2020) 90C11 · 90C05

1 Introduction

Cutting planes play an indispensable role in solving mathematical optimization problems, and they are at the heart of all competitive solvers for mixed-integer programming today. Consequently, there is a plethora of scientific papers on

T. Berthold, G. Hendel
Fair Isaac Germany GmbH
Takustr. 7, 14195 Berlin, Germany

M. Francobaldi
Department of Computer Science and Engineering, University of Bologna
Viale Risorgimento 2, 40136 Bologna, Italy

cutting planes algorithms, how to improve them, which theoretical characteristics they have, how to improve the numerical stability of cuts, which cuts to select from a given set and so on. We refer to the excellent surveys [17, 6] and the references therein for an overview of computationally efficient cutting plane methods for general mixed-integer programming. However, there is surprisingly little work on whether for a given MIP instance cuts should be continuously generated throughout the tree search or whether the tree search would be more efficiently handled by a pure B&B approach. The present paper will address this question and introduce a Machine Learning approach to identify MIP instances that benefit from not applying cutting planes locally inside the tree.

2 Problem discussion: To cut or not to cut

Cutting planes can be integrated into the B&B scheme in many different ways, each one determined by a combination of several algorithmic decisions [5, 18, 8, 2]. In this paper, we classify a cutting plane either as a *global cut* or as a *local cut*, according to whether it is generated at the root node or at an internal node of the B&C search tree.

As for any other routine involved in the solving process of a MIP, designing an efficient cutting strategy is a matter of trade-offs. On one side, the use of cuts improves the dual bound, hence increases the chances of pruning, and is consequently expected to reduce the number of explored nodes. On the other side, cut generation does not come for free; in particular, their inclusion enlarges the size of the LP relaxation, thereby slowing down the linear solver. Moreover, the use of local cuts, whose validity can not be guaranteed globally, prevents the use of conflict analysis [1, 27]. This might lead to a performance degradation on instances for which conflict analysis is beneficial.

In the present work, we address the question of whether to cut at internal nodes of the B&B tree, or to limit all cutting activity exclusively to the root node; equivalently, whether to use local cuts during the solve or not. We denote these two alternatives by `LC` and `NLC`, respectively. To the best of the authors' knowledge, up until the time of writing this paper, no efficient criterion has been discussed in the literature to provide an answer to this question for general MIP. According to a computational study conducted on our dataset with FICO Xpress, the `LC` method is, on average, 27% faster than `NLC`. However, on 22% of the test set, `NLC` was significantly faster than `LC`, against the general trend. If we had a perfect oracle to decide, for any given input instance, whether `NLC` works better than `LC`, then we could speed up the average runtime of the solver by 11%. This statistic indicates the potential for a method that could discriminate between instances that solve better with one or the other approach. At the same time, it suggests an upper limit on the improvement that we can expect by such a method. The goal of this paper is to train a machine learning model to predict whether the `NLC` or the `LC` approach will work better for a given instance.

3 Methodological Approach

We developed our ML approach by using the benchmark set MIPLIB 2017 [11] as problem set, and FICO XPRESS 8.9 as MIP solver. Note that, however, our methodology can be implemented for any MIP solver $\mathcal{S}$ (supporting both configurations `LC` and `NLC`) and problem set $\mathcal{P} \subseteq \mathscr{P}$, where $\mathscr{P}$ denotes the space of all MIP problems.

Feature Design. We represent a problem $p \in \mathscr{P}$ as a vector of 32 features, which condense the relevant pieces of information leading to an algorithmic discrimination between `LC` and `NLC`. Table 1 provides a comprehensive description of the resulting *feature space*, denoted by $\mathcal{F}_{\mathcal{S}} \subset \mathbb{R}^{32}$, together with the definition of each individual feature. We denote by A', b', c' the set of absolute values of the non-zero entries of the problem's matrix A, vectors b and c, respectively: $A' := \{|A_{i,j}|; (i,j) \in [m] \times [n], A_{i,j} \neq 0\}$, $b' := \{|b_i|; i \in [m], b_i \neq 0\}$, and $c' := \{|c_j|; j \in [n], c_j \neq 0\}$. The number of remaining rows and columns after presolving, instead, are denoted by $\tilde{m}$ and $\tilde{n}$, respectively. Finally, we consider the following objective values at the root node: the objective value of the initial LP optimum (initial bound), of the LP optimum at the end of the root node cutting loop (dual bound), and of the incumbent at the end of the root (primal bound), if already available. We denote them as c_{i}, c_{d}, and c_{p}, respectively.

Features can be classified as either *static* or *dynamic*. The former represent the mathematical formulation and combinatorial structure of the problem, hence they are completely solver-independent; the latter, instead, capture the algorithmic behavior and development of the problem, hence they are solver-dependent. Moreover, we divide the static features into three groups, that is, *Matrix*, *Variables* and *Constraints*, corresponding to different components of the problem formulation. The first group provides information about the size and the sparsity of the problem matrix, as well as the presence of eventual symmetries; the second and the third describe instead the variable and constraint composition of the problem [20], respectively. The dynamic features are split into 3 groups as well, that is, *Presolving*, *Scaling* and *Global Cutting*, corresponding to different stages of the solving process, in which the solution process is not yet affected by our decision. The first group describes the order of magnitude of the problem data after the scaling process, the second measures the effectiveness of the presolver, while the latter quantifies the impact of the global cutting loop.

Label Definition. The seemingly natural approach to solve our problem would be to formulate it as a *classification* task, given that the problem itself consists, essentially, in taking a binary decision. There are, however, two reasons which are in favor of taking a *regression* approach. Firstly, our ultimate goal is to improve the average runtime of the solver, which is a metric that is numerical and not categorical. Secondly, our focus is on getting the prediction right for those instances on which `LC` and `NLC` significantly diverge from each other.

	Feature	Definition
	∼ Matrix ∼	
Static	`Rows`	$\ln(m)$
	`Columns`	$\ln(n)$
	`NonZeros`	ratio of non-zeros, over $m \times n$
	`Symmetries`	1 if any symmetry, 0 otherwise
	∼ Variables ∼	
	`Binaries`	ratio of binary variables, over n
	`Integers`	ratio of integer variables, over n
	∼ Constraints ∼	
	`LessThan`	ratio of constraints per constraint type, over m
	`GreaterThan`	
	`Equality`	
	`SetPartitioning`	
	`SetPacking`	
	`SetCovering`	
	`Cardinality`	
	`KnapsackEquality`	
	`Knapsack`	
	`KnapsackInteger`	
	`BinaryPacking`	
	`VariableLowerBound`	
	`VariableUpperBound`	
	`MixedBinary`	
	`MixedInteger`	
	`Continuous`	
Dynamic	*∼ Scaling ∼*	
	`Coefficient_Oom`	$\ln(\max A' / \min A')$
	`RightHandSide_Oom`	$\ln(\max b' / \min b')$
	`Objective_Oom`	$\ln(\max c' / \min c')$
	∼ Presolving ∼	
	`PresolRows`	$\ln(\tilde{m})$
	`PresolColumns`	$\ln(\tilde{n})$
	`PresolIntegers`	ratio of presolved integer variables, over n
	∼ Global Cutting ∼	
	`DualInitial_Gap`	$\lvert c_\mathrm{d} - c_\mathrm{i}\rvert / \max(\lvert c_\mathrm{d}\rvert, \lvert c_\mathrm{i}\rvert, \lvert c_\mathrm{d} - c_\mathrm{i}\rvert)$
	`PrimalDual_Gap`	$\lvert c_\mathrm{p} - c_\mathrm{d}\rvert / \max(\lvert c_\mathrm{p}\rvert, \lvert c_\mathrm{d}\rvert, \lvert c_\mathrm{p} - c_\mathrm{d}\rvert)$
	`PrimalInitial_Gap`	$\lvert c_\mathrm{p} - c_\mathrm{i}\rvert / \max(\lvert c_\mathrm{p}\rvert, \lvert c_\mathrm{i}\rvert, \lvert c_\mathrm{p} - c_\mathrm{i}\rvert)$
	`Gap_Closed`	$1 - \frac{\texttt{PrimalDual_Gap}}{\texttt{DualInitial_Gap}}$

Table 1 Our feature space $\mathcal{F}_{\mathcal{S}}$. Note that, for some of these features (i.e., original and presolved rows and columns, as well as order of magnitude of problem data), we use the transformation ln(). This transformation, although irrelevant for our `RF`, was indeed convenient for different models that we used in preliminary experiments.

For instances where `NLC` and `LC` show a similar behavior, the classification will have a negligible impact on the solver performance. We define the label of a problem $p \in \mathcal{P}$ as the *speedup factor* between $Time_{\texttt{LC}}(p)$ and $Time_{\texttt{NLC}}(p)$, rescaled by means of the log_2 function:

$$y^p_{\mathcal{S}} = log_2 \left(\frac{Time_{\texttt{LC}}(p) + 1}{Time_{\texttt{NLC}}(p) + 1} \right) \in \mathbb{R},$$

where $Time_\phi(p)$ denotes the running time of $\mathcal{S}$ while solving p with configuration $\phi \in \{\texttt{LC}, \texttt{NLC}\}$. The runtimes of the two methods are both augmented by

1, to mitigate the impact of very small numbers, and to prevent the division by zero. The scatter plot in Figure 1 displays the instance-by-instance comparison between the runtimes of the two methods over MIPLIB 2017* (discussed in the following paragraph), computed with Xpress and colored continuously according to the speedup value. Now, we can provide a rigorous formulation of our regression problem: our task is to produce a model $M_{\mathcal{S}} : \mathcal{F}_{\mathcal{S}} \longrightarrow \mathbb{R}$ that, for each input vector $x^p_{\mathcal{S}}$, approximates the speedup factor $y^p_{\mathcal{S}}$ between the two runs, `LC` and `NLC`, of the solver $\mathcal{S}$ over $p \in \mathscr{P}$, that is, $M(x^p_{\mathcal{S}}) \approx y^p_{\mathcal{S}}$. In Section 4, we will confirm our expectations on the superiority of this regression formulation over a classification one.

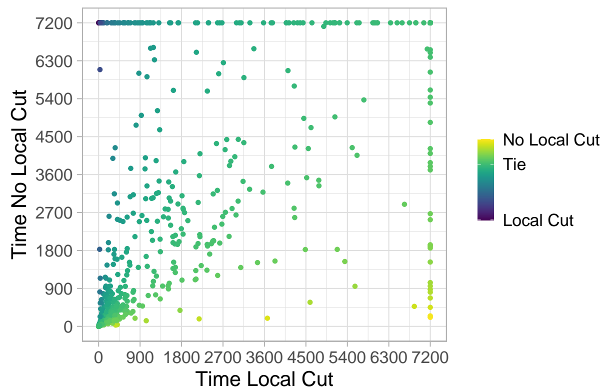

Fig. 1 The color varies continuously, from the upper-left corner to the lower-right one, between purple, representing those instances on which `LC` and `NLC` reach their minimum and maximum runtime, respectively, and yellow, corresponding to the opposite situation. Along the diagonal $y = x$, the color results from the interpolation of the two colors at the extremes, indicating that, on these points, the two methods perform similarly.

Data Collection. For the given solver $\mathcal{S}$ and problem set $\mathcal{P}$, we collect a ground dataset for our ML approach as follows. Firstly, we apply six random permutations to the elements of $\mathcal{P}$, each one associated to a seed $s = 0, \ldots, 5$ (with 0 identifying the identity permutation), in order to enlarge and diversify our ground problem set. We denote the expanded set by $\mathcal{P}^*$, and each of its elements by p_s, referred to as an *instance* of the problem set. Note that the six instances produced from p, although mathematically equivalent, can have a very different computational behavior [16]. Hence, from each instance p_s, we collect the solver-independent information that we use to compute the static features.

Then, we run the solver $\mathcal{S}$ over p_s twice, once with configuration `LC`, and once with `NLC`. From these runs, we retain the solver-dependent data that we use to compute both the dynamic features and the labels. Now, from the raw information collected, we construct the dataset of features-label observations $\mathcal{D} = \mathcal{D}(\mathcal{S}, \mathcal{P}) = \{(x^{p_s}_{\mathcal{S}}, y^{p_s}_{\mathcal{S}}) : \ p_s \in \mathcal{P}^*\} \subseteq \mathcal{F}_{\mathcal{S}} \times \mathbb{R}$, that we use for our learning

experiments. In particular, we split this set into a *training set*, $\mathcal{D}_{train}$, and a *test set*, $\mathcal{D}_{test}$, as follows: we use the permuted problems of $\mathcal{P}$ ($s = 1, \dots, 5$) for the former, and the original ones ($s = 0$) for the latter, corresponding to roughly a training-test split of 83% to 17% of the entire dataset. Using permutations for a training-test split has first been suggested in [4].

Training & Testing. Among the different ML models that we considered for our learning task, the best results were provided by a *random forest* (RF) [13], an ML model consisting of a multitude of *regression trees* [21,19], trained by means of ensemble techniques able to improve robustness and prevent over-fitting. An overview of the preliminary experiments that we conducted with other ML models can be found in the master's thesis of the second author [10].

We train RF by using the `caret` [14] package of `R` [23], with `method` set to `"rf"` in the `"train"` function. Moreover, in order to select a suitable number of regression trees to employ within the ensemble, we use the `trainControl` tool with `method` set to `"cv"` and `number` to `5`, which performs a 5-*fold cross-validation* with randomly generated folds [24, Section 11.2.4]. The produced model is a random forest with 500 regression trees.

We assess the quality of our trained model when deployed in a MIP solver by three different criteria: the running time (*Time*), the primal-dual integral (*PDI*) [3] and the number of explored nodes (*Nodes*). For each of these metrics Met, we compute the performance of RF, on a certain problem p, as

$$Met_{\mathtt{RF}}(p) = \begin{cases} Met_{\mathtt{LC}}(p), & \text{if } \mathtt{RF}(x^p_{\mathcal{S}}) \leq \tau \\ Met_{\mathtt{NLC}}(p), & \text{if } \mathtt{RF}(x^p_{\mathcal{S}}) > \tau, \end{cases}$$

where $Met_\phi(p)$ is the performance of our solver $\mathcal{S}$ when solving p with strategy ϕ. Here, $\tau \in \mathbb{R}$ is a *threshold* that we use as a switch between LC and NLC to convert our regressor into a binary classifier. As experimentally derived, we set the value of τ to 0, which corresponds to both methods being predicted to take the same run time.

We aggregate the performance of a strategy ϕ, over a set of problems $\mathcal{T} \subset \mathscr{P}$, by means of the *shifted geometric mean*,

$$Shm_{Met}(\phi) = \left(\prod_{p \in \mathcal{T}} (Met_\phi(p) + S) \right)^{\frac{1}{|\mathcal{T}|}} - S,$$

with *shift* S set to 10 for *Time* and *PDI*, and to 1000 for *Nodes*.

4 Computational Experiments

Our computational study consists of two parts. The first experiment was conducted on the dataset $\mathcal{D} = \mathcal{D}$(Xpress, Miplib 2017), collected by executing our data collection process (Section 3) with version 8.9 of the FICO Xpress MIP solver and the benchmark set of Miplib 2017 as instance set. We chose

a time limit of $T = 7200s$. As competitors of RF, we consider the following strategies: AlwaysLC, which always chooses LC, NeverLC, which always chooses NLC, and Oracle, which always takes the optimal choice, each time according to the particular metric used, between LC and NLC, hence representing the best possible performance that we can achieve with our selection method. The main goal for this first experiment was to analyze the potential of the approach and to evaluate the impact of different decisions within our framework.

For the second experiment, we implemented a learned model into FICO Xpress and tested it on an internal data set, which primarily consists of client instances. This implementation is used by default in the current release of Xpress. The main goal of this experiment is to demonstrate the impact of a practical implementation of our approach on a diverse set of real-world instances.

For the first experiment, we excluded from $\mathcal{D}$ all the instances that are not suitable for our study: the ones for which any of the strategies ran out of memory, the ones solved already at the root, and a few cases where one of the strategies ran into numerical issues. This cleaning process reduced the size of $\mathcal{D}$ from 1440 instances to 1155 instances on which the model was trained.

We compare our learned model with its competitors, on both $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$. The results of the evaluation are reported in Table 2, where **Imp.** refers to the improvement provided to the solver by our model RF, when compared against the better of our two competitors AlwaysLC and NeverLC, **Pot.** denotes the potential improvement, i.e., the performance gap between this competitor and Oracle. **Achiev.** reports the percentage of this potential improvement that RF is able to achieve. An immediate observation that we can make is that, between AlwaysLC and NeverLC, the former is certainly the more performant one, given that, on both sets, it significantly outperforms NeverLC in terms of all considered metrics. This confirms our claim from Section 2: if AlwaysLC and NeverLC were the only available strategies to take the LC/NLC decision, then we would choose the former. The results scored by our learned strategy RF, however, shows that a smarter strategy than AlwaysLC is achievable.

	Metric	RF	AlwaysLC	NeverLC	**Imp.** (%)	Oracle	**Pot.** (%)	**Achiev.** (%)
Train	*Time*	427.91	467.92	642.56	8.55	414.75	11.36	75.25
	PDI	79.50	84.39	99.62	5.79	76.21	9.69	59.79
	Nodes	22363.22	22693.97	43554.10	1.46	20363.27	10.27	14.19
Test	*Time*	420.09	434.61	593.21	3.34	384.72	11.48	29.09
	PDI	78.13	79.26	93.66	1.43	72.03	9.12	15.66
	Nodes	21243.26	19903.08	37812.65	-6.73	17919.92	9.96	-67.58

Table 2 The *Shm* performance of the competing strategies, in terms of different metrics.

When compared against AlwaysLC in terms of *Time*, RF provides a speedup of roughly 8.5% on the train set and of 3.3% on the test set, hence achieving about $3/4$ and $1/3$ of the *potential* improvement on the two sets, respectively. Similar considerations can be made by looking at the *PDI*, even though, in this

case, the contribution provided to the solver is less pronounced than the one observed in *Time*. In contrast, the results obtained in terms of *Nodes* seem to contradict the ones observed in the two metrics previously discussed. Indeed, with an improvement of less than 1.5% and an achievement of less than 15%, the *Nodes* performance of our solver is, on $\mathcal{D}_{train}$, almost unaffected by the use of our model, while it is even weakened by it on $\mathcal{D}_{test}$, where we can observe a 7% degradation and a 70% increment in the average node consumption.

This tendency, however, is explainable by the fact that the use of cutting planes, as discussed in Section 2, has the main advantage of reducing the number of explored nodes. This is why, when measured in terms of *Nodes*, the `AlwaysLC` strategy tends to show the smallest number of nodes. To provide a more detailed evaluation of the contribution given by our approach, we compare our model, `RF`, against our main competitor, `AlwaysLC`, on the entire dataset $\mathcal{D}$ and on different *brackets* of this set, as well as on the subset of the *affected* instances.

The results of this comparison are reported in Table 3. Here, the bracket $[t_1, t_2]$ is the set of instances of $\mathcal{D}$ satisfying the following: 1) they are solved (within the time limit T) by at least one of the two strategies, 2) the runtime of the slower strategy is between t_1 and t_2. We fix the right-hand side of each bracket at the time limit T, while we increase the left-hand one progressively, so to define a hierarchy of subsets of $\mathcal{D}$ of increasing difficulty. The affected instances, instead, are the ones that are solved by at least one of the two competitors, and for which these competitors make opposite choices, i.e., the ones on which `RF` decides to deactivate local cutting, as opposed to the default strategy.

		`RF`		`AlwaysLC`		
Bracket	**Instances**	**Solved**	**Time**	**Solved**	**Time**	**Imp.** (%)
All	1155	872	426.59	854	462.16	7.70
$[10, 7200]$	753	751	257.15	733	290.20	11.39
$[100, 7200]$	470	468	750.73	450	874.37	14.14
$[1000, 7200]$	226	224	2198.42	206	2703.86	18.69
Affected	310	308	132.45	290	180.63	26.67

Table 3 Comparison between `RF` and `AlwaysLC`, on different brackets of $\mathcal{D}$.

We observe that the learned model `RF` is able to solve 18 instances more than `AlwaysLC`, and to improve the average runtime by 7.7%. This increases to more than 11% on the instances of the bracket $[10, 7200]$, and keeps increasing together with the hardness of the evaluation set, until reaching an encouraging value of 18.7% on the most difficult among our problems. Finally, by restricting the test bed to the affected instances, we can observe a very promising improvement of more than 26%, cf. [10]. In an offline experiment, we also tested a classification version of our random forest training. While it also improved the performance, the benefit was clearly smaller than for the regression forest.

		RF		without RF		
Bracket	**Instances**	**Solved**	**Time**	**Solved**	**Time**	**Imp.** (%)
All	5530	5328	102.56	5319	104.04	1.44
$[10, 7200]$	4434	4431	131.01	4422	133.22	1.69
$[100, 7200]$	2226	2223	463.08	2214	474.39	2.44
$[1000, 7200]$	574	571	2093.23	562	2203.17	5.25
Affected	488	477	116.79	468	137.80	17.99

Table 4 Comparison between a version of Xpress with and without a RF model, on different brackets of an internal data set of FICO.

Finally, the results of our initial study were used to implement a new decision module into the FICO Xpress MIP solver [9] for our second experiment. It was trained and tested on FICO internal data sets which mainly consist of real-world instances provided by FICO customers. Naturally, this led to a model that is different in detail, but the applied training procedure was exactly the same. We restrained ourselves to using at most seven features and 50 trees for the final model for three reasons: to save memory, to speed up the RF evaluation, and to avoid overfitting. Both restrictions performed only slightly worse than a variant that used 500 trees and the full feature set. Given the known average superiority of LC over NLC, as an additional safeguard against performance losses, we skewed our decision towards the former approach as follows: in order to deactivate local cuts, we required not only the overall model prediction to be in favor of NLC, but also a majority of 70% of the individual trees. In other words, we kept generating local cuts even though the random forest decided to deactivate them, unless more than 70% of the ensemble agreed with this decision.

The results of running a pre-release version of FICO Xpress 8.13 can be seen in Table 4. The presented data set of 5530 test instances constitutes the standard performance evaluation set of Xpress. Note that it is significantly larger than the Miplib 2017 set we used for our initial experiment, and that this also includes instances which are solved at the root node. Hence, we expect the results to be less pronounced. Note further, that this is not the set that was used for training (and testing), but the result of a consequent verification.

We observe an overall improvement of 1.44% when using a random forest to decide on a local cutting strategy, again with increasing impact as the models get harder. For instances which take at least 1000 seconds to solve, we see an improvement of 5.25%. Additionally, there are nine more instances solved when using an automatic decision on local cuts. Again, the effect is much more pronounced when we restrict our attention to the affected models, hence those where we deactivate cutting now, but haven't done before. Here, we observe a speedup of almost 18%, which is comparable to the results of our initial experiment on Miplib 2017. Moreover, we used a Wilcoxon signed rank test [25] to evaluate the significance of the improvement. It rejected the null hypothesis with a p-value of less than 0.001, which can be interpreted as our observations indicating an actual speedup with more than 99.9% confidence.

5 Conclusion & Outlook

We introduced the first, to the best of our knowledge, methodology to predict for a given general MIP instance whether it would be better solved by a cut-and-branch or a branch-and-cut approach. Our methodology showed very promising results both for the Miplib 2017 benchmark set and for an internal performance evaluation set of FICO. Consequently, it has been implemented by the FICO Xpress developer team and is used by default as of the recent 8.13 release of Xpress.

As possible outlook of the present work, we suggest to learn the maximum depth of the B&B tree in which the cutting procedure should be stopped, rather than choosing only whether to deactivate the procedure after the root node or not. This seems particularly promising when connected with the concept of lifting local conflicts as suggested in [26].

References

1. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization **4**(1), 4–20 (2007). doi:10.1016/j.disopt.2006.10.006
2. Achterberg, T.: Constraint integer programming. phdthesis, Technische Universität Berlin (2007)
3. Berthold, T.: Measuring the impact of primal heuristics. Operations Research Letters **41**(6), 611–614 (2013). doi:10.1016/j.orl.2013.08.007
4. Berthold, T., Hendel, G.: Learning to scale mixed-integer programs. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 3661–3668 (2021)
5. Cordier, C., Marchand, H., Laundy, R., Wolsey, L.A.: bc-opt :A branch-and-cut code for mixed integer programs. Tech. rep., Université catholique de Louvain, Center for Operations Research and Econometrics (CORE) (1997)
6. Cornuéjols, G.: Valid inequalities for mixed integer programs. Mathematical Programming **112**, 3–44 (2008)
7. Dakin, R.J.: A tree-search algorithm for mixed integer programming problems. The Computer Journal **8**(3), 250–255 (1965). doi:10.1093/comjnl/8.3.250
8. Fügenschuh, A., Martin, A.: Computational integer programming and cutting planes. In: K. Aardal, G. Nemhauser, R. Weismantel (eds.) Discrete Optimization, *Handbooks in Operations Research and Management Science*, vol. 12, pp. 69–121. Elsevier (2005). doi:10.1016/S0927-0507(05)12002-7
9. FICO Xpress Optimizer. `https://www.fico.com/en/products/fico-xpress-solver`
10. Francobaldi, M.: Learning to use local cuts. Master's thesis, Freie Universität Berlin (2021). URL `https://opus4.kobv.de/opus4-zib/frontdoor/index/index/start/0/rows/10/sortfield/score/sortorder/desc/searchtype/simple/query/francobaldi/docId/8597`
11. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P.M., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H.D., Ozyurt, D., Ralphs, T.K., Salvagnin, D., Shinano, Y.: MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Mathematical Programming Computation **13**, 443–490 (2021). doi:10.1007/s12532-020-00194-3
12. Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **64**(5), 275–278 (1958). doi:978-3-540-68279-0_4
13. Ho, T.K.: Random decision forests. In: Proceedings of the Third International Conference on Document Analysis and Recognition, *ICDAR '95*, vol. 1, pp. 278–282. IEEE Computer Society (1995)
14. Kuhn, M.: caret: Classification and Regression Training (2020). URL `https://CRAN.R-project.org/package=caret`. R package version 6.0-86

15. Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28**(3), 497–520 (1960). doi:10.2307/1910129
16. Lodi, A., Tramontani, A.: Performance variability in mixed-integer programming. TutORials in Operations Research pp. 1–12 (2014). doi:10.1287/educ.2013.0112
17. Marchand, H., Martin, A., Weismantel, R., Wolsey, L.: Cutting planes in integer and mixed integer programming. Discrete Applied Mathematics **123**(1), 397 – 446 (2002). doi:https://doi.org/10.1016/S0166-218X(01)00348-1
18. Martin, A.: General mixed integer programming: Computational issues for branch-and-cut algorithms. Lecture Notes in Computer Science **2241**, 1–25 (2001). doi:10.1007/3-540-45586-8_1
19. Messenger, R., Mandell, L.: A modal search technique for predictive nominal scale multivariate analysis. Journal of the American Statistical Association **67**(340), 768–772 (1972). doi:10.1080/01621459.1972.10481290
20. Miplib 2017 website. URL `https://miplib.zib.de/index.html`
21. Morgan, J.N., Sonquist, J.A.: Problems in the analysis of survey data, and a proposal. Journal of the American Statistical Association **58**(302), 415–434 (1963). doi:10.1080/01621459.1963.10500855
22. Padberg, M., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Review **33**(1), 60–100 (1991). doi:10.1137/1033004
23. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing (2013). URL `http://www.R-project.org/`
24. Shalsev-Shwartz, S., Ben-David, S.: Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, USA (2014)
25. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics bulletin pp. 80–83 (1945)
26. Witzig, J., Berthold, T.: Conflict Analysis for MINLP. INFORMS Journal on Computing **33**(2), 421–435 (2021). doi:10.1287/ijoc.2020.1050
27. Witzig, J., Berthold, T., Heinz, S.: Computational aspects of infeasibility analysis in mixed integer programming. Mathematical Programming Computation (2021). doi:10.1007/s12532-021-00202-0