Integrated Integer Programming and Decision Diagram Search Tree with an Application to the Maximum Independent Set Problem

Jaime E. González[1], Andre A. Cire[2], Andrea Lodi[1], and Louis-Martin Rousseau[1]

[1]Department of Mathematics and Industrial Engineering, Polytechnique Montréal {jaime.gonzalez, andrea.lodi,louis-martin.rousseau}@polymtl.ca
[2]Department of Management, University of Toronto Scarborough andre.cire@utoronto.ca

October, 2019

Abstract

We propose an optimization framework which integrates decision diagrams (DDs) and integer linear programming (ILP) to solve combinatorial optimization problems. The hybrid DD-ILP approach explores the solution space based on a recursive compilation of relaxed DDs and incorporates ILP calls to solve subproblems associated with DD nodes. The selection of DD nodes to be explored by ILP technology is a significant component of the approach. We show how supervised machine learning can be useful to detect, on-the-fly, a subproblem structure for ILP technology. We use the maximum independent set problem as a case study. Computational experiments show that, in presence of suitable problem structure, the integrated DD-ILP approach can exploit complementary strengths and improve upon the performance of both a stand-alone DD solver and an ILP solver in terms of solution time and number of solved instances.

Keywords— Decision diagrams, Integrated methods, Integer linear programming, Supervised Learning.

1 Introduction

Several optimization problems can be addressed efficiently by leveraging multiple solution approaches with complementary strengths, such as integer linear programming (ILP) and constraint programming (CP) [23, 21]. Within this extensive area, successful hybrid techniques often involve an *a-priori* decomposition of the problem into two or more subproblems that are more amenable for existing techniques, such as in the case of logic-based Benders decomposition for machine scheduling [22] and CP-based branch and price [16, 17].

In this paper, we propose a novel hybrid methodology that combines elements from decision diagrams (DDs) (see, e.g., [1], [26], [13]) with ILPs for discrete optimization problems. While ILPs have an extensive body of work, only recently decision diagrams have been established as viable data structure in optimization solution approaches [10]. Specifically, DD techniques are based on discrete approximations of the solution space denoted by *relaxed decision diagrams* (relaxed DDs) [2].

We contribute to the literature on decision diagram-based approaches that integrate different optimization paradigms. Relaxed DDs are used as a replacement of the linear programming relaxation in branch-and-bound methods [7], and also to strengthen ILP models through their equivalent linear reformulation as shortest paths [5]. In addition, DDs have been embedded as global constraints in CP models for sequencing problems [25, 14], as a cutting plane method for integer programming [31], and to solve the pricing problem in a branch and price scheme [29], to name a few. Furthermore, in the context of two-stage stochastic programming, DDs have also been used to model second-stage decisions which are parameterized by the first-stage variables [28].

Our methodology, in turn, provides an alternative way of exploiting relaxed DDs by further emphasizing their role as approximations of the branch-and-bound search tree of a problem. In particular, we perceive the nodes of a relaxed DD as subproblems that may involve some structure that is more suitable to another technology (in our case, ILPs), which is invoked to prune the node in advance. This process therefore involves a *dynamic* identification of subproblems as opposed to an a-priori one, and may both strengthen the relaxed DD approximation (by removing undesired nodes) as well as provide primal solutions that can further speed-up solution time.

We represent each subproblem at a node by a dynamic programming state, currently the standard for a DD formulation. Using such a state representation, we follow a supervised learning methodology to get insights

and define criteria to detect a structure that is more efficiently solvable by ILP. Specifically, we describe an algorithm-selection problem where, given a node of a relaxed DD, we use a decision tree inferred from predictive models to determine the best method to solve its associated subproblem. This approach has similarities, e.g., with the concept of algorithm portfolios in SAT solvers [33].

We present a case study on the maximum independent set problem (MISP), which is typically used as a basis for novel DD methodologies due to its well-understood representation [8]. We show that, under particular structure, DD-ILP can dominate either the typical DD-based branch and bound or a leading commercial ILP solver. Most importantly, it suggests a way to enhance DD-based solvers when its relative effectiveness with respect to ILP can be evaluated efficiently and dynamically with Machine Learning.

The remainder of this paper is organized as follows. Section 2 describes preliminary concepts in decision diagrams for optimization. In Section 3, we introduce the proposed DD-ILP framework and the supervised learning methodology used to exploit the DD-ILP algorithm. Section 4 presents computational experiments on MISP instances. Finally, we conclude in Section 5.

2 Preliminaries and Notation

Maximum Independent Set Problem. Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, an independent set of $\mathcal{G}$ is a subset of vertices $\mathcal{S} \subseteq \mathcal{V}$ such that no edge in $\mathcal{E}$ has its two endpoints in $\mathcal{S}$. The maximum independent set problem (MISP) asks for the independent set with the largest cardinality [19]. For instance, the optimum MISP solution to the graph in Figure 1 is $\{1, 2, 5\}$. We let $n := |\mathcal{V}|$ and denote by $\mathcal{S}(\mathcal{G})$ the set of independent sets of $\mathcal{G}$ (i.e., the solution set of the problem).

A classical ILP model to address the maximum independent set problem is the so-called edge formulation [19]. Let x_i be a binary variable that takes the value of 1 if vertex i belongs to the maximum independent set and 0 otherwise. An integer programming formulation with $|\mathcal{E}|$ constraints is given by $\max \left\{ \sum_{i \in \mathcal{V}} x_i \;:\; x_i + x_j \leq 1, \forall (i,j) \in \mathcal{E}; x \in \{0,1\}^{|\mathcal{V}|} \right\}$. However, a stronger ILP formulation for the MISP is obtained by partitioning the vertices of $\mathcal{G}$ into cliques [19], i.e., subsets of $\mathcal{V}$ where the vertices in each subset are pairwise adjacent. Let x_i be a binary variable that takes value 1 if vertex i belongs to the maximum independent set and 0 otherwise. Let $\mathcal{K}$ be a collection of (inclusion-wise) maximal cliques that cover $\mathcal{V}$. The resulting

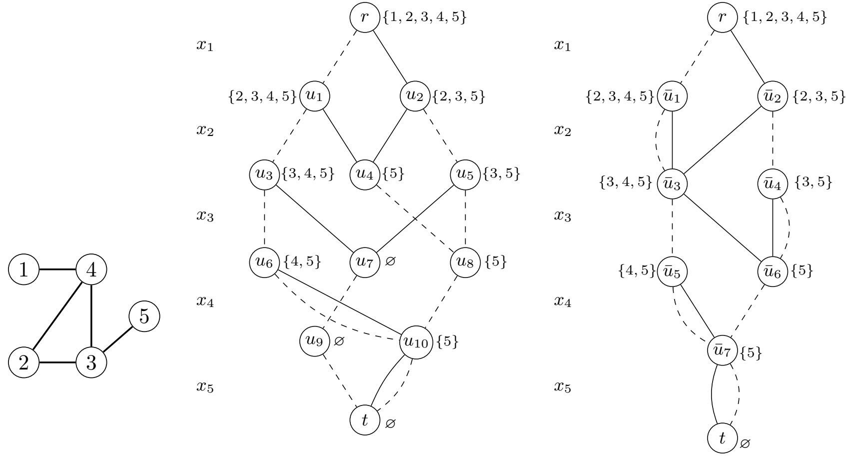


Figure 1: Undirected graph for the MISP

Figure 2: Exact DD for MISP instance of Figure 1

Figure 3: Relaxed DD for MISP instance of Figure 1

ILP formulation is $\max\left\{\sum_{i\in\mathcal{V}} x_i \;:\; \sum_{i\in K} x_i \leq 1, \forall K \in \mathcal{K}; x \in \{0,1\}^{|\mathcal{V}|}\right\}$.

Each maximal clique K is computed in a greedy fashion (as in [10]) by initially selecting the vertex with highest degree and then adding iteratively adjacent vertices (also with highest degree) to every vertex so far in K until no more additions are possible. Then, we add clique K to $\mathcal{K}$, remove from $\mathcal{G}$ all the edges belonging to the added clique, update the vertices degrees, and repeat the procedure.

Exact and Approximate Decision Diagrams. A decision diagram $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ is a directed graph with node set $\mathcal{N}$ and arc set $\mathcal{A}$ that encodes a set of solutions of an optimization problem. For the case of the MISP, the node set $\mathcal{N}$ is partitioned into $n+1$ layers $L_1, \ldots, L_{n+1}$. The first and last layers are singletons and contain the root node r and the terminal node t, respectively. Arcs emanating from nodes in $L_1, \ldots, L_n$ are associated with the variables $x_1, \ldots, x_n$, respectively. Let $l(u)$ be the index of the layer associated with node $u \in \mathcal{N}$. Each arc $a = (u, u') \in \mathcal{A}$ only connects adjacent layers, i.e., $l(u') = l(u) + 1$, and its layer corresponds to the one of its tail node, $l(u)$. Moreover, a label $d(a) \in \{0,1\}$ of an arc a at layer i represents the assignment $x_i = d(a)$; we denote an arc with $d(a) = 0$ and $d(a) = 1$ by 0-arc and 1-arc, respectively. An arc-specified path $(a_1, \ldots, a_n)$ from the root node to the terminal node encodes the solution $(d(a_1), \ldots, d(a_n))$ to

the MISP. Finally, a DD is called *exact* if the set of solutions encoded in $\mathcal{D}$ is $\mathcal{S}(\mathcal{G})$.

Dynamic programming (DP) is used as the conceptual basis to compile exact DDs. We follow the same DP formulation presented in [6] for the MISP. Let $N(i)$ be the neighborhood of vertex $i \in \mathcal{V}$ including i, i.e., $N(i) = \{i'|(i,i') \in \mathcal{E}\} \cup \{i\}$. With each node u of $\mathcal{D}$ we associate a state information $s(u) \subseteq \mathcal{V}$ that represents the vertices that can be added to the partial independent set encoded by any path from the root to u. The root node's state is fixed as $s(r) = \mathcal{V}$. Each node $u \in L_i$ has an outgoing 0-arc that leads to a node u' with state $s(u') := s(u) \setminus \{i\}$ (i.e., we exclude the vertex from the independent set). If $i \in s(u)$, then u also has an outgoing 1-arc that leads to a node u'' with state $s(u'') := s(u) \setminus N(i)$ (i.e., we include the vertex and remove its neighbourhood). No two nodes in a layer have the same state. Typically exact DDs are constructed in a top-down fashion, where layers are processed in the order $L_1, \ldots, L_n$ one at a time, adding arcs and merging nodes with the same state as required.

Exact DDs are in general of exponential size and, in practice, we prefer to manipulate approximate (controllable-size) decision diagrams such as the relaxed [7] and restricted [9] versions. A decision diagram is called *relaxed* if it over-approximates the solution set of a problem ($\mathcal{S}(\mathcal{G})$ for the MISP case). Figures 2 and 3 depict an exact and a relaxed DD for the graph in Figure 1, respectively, where solid arcs represent $d(a) = 1$ and dashed arcs represent $d(a) = 0$.

Relaxed DDs can be obtained similarly as exact DDs when using a top-down approach. In particular, if the number of nodes in a layer (i.e., the layer width) exceeds a given pre-specified limit W during its construction, two nodes u and u' are heuristically selected and merged into a new node u''. The state of this node is set as $s(u'') := s(u) \cup s(u')$ to ensure all valid independent sets are preserved. The strategy used to define which nodes are merged consists on selecting DD nodes u and u' with the partial longest path from the root node. Once nodes are merged, all previous incoming arcs to u and u' are directed to the new merged node. Figure 3, in particular, presents a relaxed DD with maximum width of 2 ($W \leq 2$). Notice that, if the length of an arc is set as $d(a)$, the longest path of an exact DD provides the optimal MISP, while the longest path value of a relaxed DD provides an upper bound to the problem.

Another type of approximate decision diagrams are the so-called restricted DDs which are under-approximations of the solution set of a problem. Restricted DDs can also be obtained in a top-down construction. Once the maximum width W is reached when constructing a layer, we heuristi-

cally remove nodes from such a layer. In this procedure, we evidently remove feasible solutions and possibly the optimal one too. However, a longest path computation on a restricted DD provides a feasible solution and a lower bound on the optimal solution value.

DD-based Branch and Bound. Relaxed DDs can play the role of a search tree in a branch-and-bound scheme [7]. The main idea is that the solution space of a problem can be divided and explored by branching recursively on decision diagram nodes as opposed to branching on a variable-value pair.

Specifically, consider a relaxed decision diagram $\bar{\mathcal{D}}$ of an optimization problem. For every pair of nodes $u, u' \in \bar{\mathcal{D}}$ such that $l(u) < l(u')$, we define $\bar{\mathcal{D}}_{uu'}$ as the decision diagram induced by all the nodes and arcs that lie on directed paths from u to u'; e.g., $\bar{\mathcal{D}}_{rt} = \bar{\mathcal{D}}$. We say that a node u in $\bar{\mathcal{D}}$ is *exact* if all $r-u$ paths lead to the same state $s(u)$, and is relaxed otherwise. A *cutset* of $\bar{\mathcal{D}}$ is a subset of nodes C such that any $r-t$ path in $\bar{\mathcal{D}}$ contains at least one node in C. In particular, C defines an exact cutset if all nodes in C are exact. Note that the removal of C from $\bar{\mathcal{D}}$ disconnects r and t. Several strategies exist for obtaining cutsets [10], such as *the frontier cutset (FC)* and *the last exact layer (LEL)*. The frontier cutset of a relaxed decision diagram $\bar{\mathcal{D}}$, FC($\bar{\mathcal{D}}$), is the set of exact nodes in $\bar{\mathcal{D}}$ such that, for each node, at least one of its outgoing arcs' heads is a relaxed node. For instance, FC($\bar{\mathcal{D}}$) in the relaxed DD of Figure 3 is defined by $\{\bar{u}_1, \bar{u}_4\}$. The LEL cutset LEL($\bar{\mathcal{D}}$), in turn, is defined by the last layer where all nodes are exact (i.e., where no two nodes were forcefully merged to impose the maximum width). For the case of Figure 3, LEL($\bar{\mathcal{D}}$) is defined by $\{\bar{u}_1, \bar{u}_2\}$.

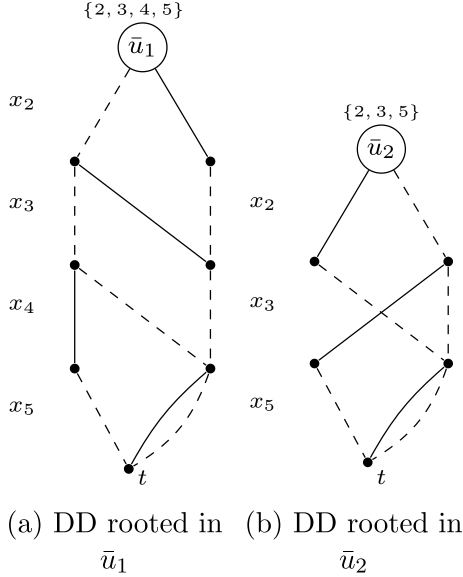
Let C be an exact cutset obtained either by FC or LEL. DD-based branch and bound explores each node in C separately to find and prove optimal solutions. Namely, for each $u \in C$, let $v^*(u)$ be the longest-path value from r to u. If z^*_u is the optimal value of the subproblem for which its solutions are exactly encoded in $\mathcal{D}_{ut}$, then $v^*(u) + z^*_u$ is the value of the best solution across all $r-t$ paths that contain u. Since all $r-t$ paths of a decision diagram must contain some node in C, we can solve the subproblems associated with $\mathcal{D}_{ut}$ separately for all $u \in C$ to search for the optimal solution, each subproblem leading to a smaller and hence more tractable DD. Such a procedure can be applied recursively for each u if required.

3 A Hybrid DD-ILP Approach

We propose a novel strategy named *ILP-based pruning* to integrate ILP technology into DD-based branch and bound. Once we define a cutset C and select a node $u \in C$ to explore, we can solve the subproblem associated with u using ILP as opposed to recursively relaxing and exploring $\mathcal{D}_{ut}$, since only its optimal solution z_u^* is required for our purposes. Such a decision is made based on the properties of the subproblem encoded by the state $s(u)$. In particular for the MISP, a subproblem corresponds to a vertex-induced subgraph defined by the vertices in $s(u)$. In a nutshell, once an exact cutset C is defined, we explore each node of C either by recursively applying DD-based branch and bound, or by directly invoking an ILP model to prune the node in advance.

For solving the MISP instance in Figure 1 using the DD-ILP framework, we initially compile a relaxed decision diagram $\bar{\mathcal{D}}$ (like the one in Figure 3) to compute an upper bound on the optimal solution. For this case, the longest path from r to t is equal to 4 which provides a dual bound on the optimal value of the objective function. Then, we identify all DD nodes in an exact cutset which are included in a list of branching nodes to explore further e.g., $C = \{\bar{u}_1, \bar{u}_2\}$. Next, in a pure DD solver scheme, we would select a DD node to branch on from the list of open branching nodes. For instance, Figures 4(a) and 4(b) present the decision diagrams recursively obtained when branching on $\bar{u}_1$ and $\bar{u}_2$. Following a branch and bound scheme, we would update the upper and lower bounds whereas we recursively explore the solution space until we prove that the incumbent solution is optimal. Note that, for instance, the DD rooted in $\bar{u}_1$ (Fig. 4(a)) is relaxed and following a pure DD-based exploration, we would have to compile new relaxed decision diagrams from a selected exact cutset in $\mathcal{D}_{\bar{u}_1 t}$.

Nevertheless, in the context of our proposed hybrid DD-ILP framework, the evaluation of properties of the subproblem encoded by a DD node can lead us to determine that such a node should be pruned by solving it with ILP technology. For instance, after evaluating DD nodes properties, one possible scenario could be applying the ILP-based pruning strategy to DD node $\bar{u}_1$ but not to DD node $\bar{u}_2$. Figure 5 illustrates how the solution space exploration is done within the DD-ILP approach. On the one hand, we generate a relaxed DD which is rooted in $\bar{u}_2$; on the other hand, we prune $\bar{u}_1$ by solving an ILP subproblem with a classical LP-based branch and bound tree (small subtree with green and red nodes rooted in $\bar{u}_1$). Note that solving to optimality the ILP subproblem associated with $\bar{u}_1$ allows us to prune the DD node, get a feasible solution (i.e., a lower bound equals to



(a) DD rooted in $\bar{u}_1$ (b) DD rooted in $\bar{u}_2$

Figure 4: Compiled DDs after branching on DD nodes in $C = \{\bar{u}_1, \bar{u}_2\}$

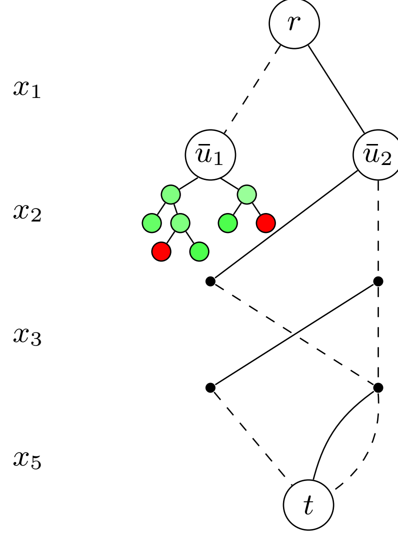


Figure 5: Integrated DD-ILP search tree.

$v^*(\bar{u}_1) + z^*_{\bar{u}_1}$), and avoid the compilation and further exploration of more relaxed DDs.

Furthermore, the DD-ILP framework can be enhanced by different strategies that combine information from the ILP and the partial solutions enumerated by the DD. For instance, assume that we define a time cut-off to solve the ILP at a node u, obtaining a lower bound $\underline{z}_u$ and an upper bound $\overline{z}_u$ as opposed to the optimal solution z^*_u. Recalling that $v^*(u)$ is the optimal longest-path value from r to u, we derive two simple methodologies:

ILP-cutoff Pruning. The value $v^*(u) + \overline{z}_u$ corresponds to an upper bound to the solutions encoded by $r - t$ paths crossing u. If such a value is lower than the current incumbent solution, the node can be pruned from the DD without losing optimality. We can also update the relaxed states of a DD after the removal of a node, which by itself can lead to additional pruning based on DD filtering methods.

ILP-cutoff Heuristic. Since u is exact, $v^*(u) + \underline{z}_u$ corresponds to a primal heuristic value to the MISP and can be used to update the current incumbent, also possibly triggering the pruning of other nodes.

When the optimal solution value z^*_u is found, we can immediately prune the node and update the incumbent solution, if needed. Otherwise, we can

use the bounds as above and proceed with DD-based branch and bound.

In addition, the complementarity offered by the hybrid DD-ILP scheme reveals an additional procedure to speed-up the subproblems' solution when calling the ILP solver. For any of the three ILP-based strategies, we can provide a global incumbent solution to the ILP solver, namely, a *lower cutoff* c. This lower cutoff indicates that the objective value of a given subproblem has to be at least c. Let LB be a global incumbent solution obtained from the hybrid DD-ILP exploration. Each time the ILP solver is called to solve a subproblem u, we include the constraint $z_u \geq c$ to the corresponding ILP model. In particular for the MISP, we include the constraint

$$\sum_{i \in s(u)} x_i \geq LB - v^*(u). \tag{1}$$

The constraint (1) takes into account that $c = LB - v^*(u)$ and $z_u = \sum_{i \in s(u)} x_i$ since the vertices considered in the subproblem are the ones defined by the state $s(u)$. As a result of this procedure, when exploring a subproblem, the ILP solver may terminate significantly earlier with the proof that no feasible solution exists, which leads to prune the DD node. Note that the ILP-based pruning strategy involves solving to optimality several ILP problems associated with DD nodes. In general, integer linear programming is NP-hard and a straightforward application of the ILP-based pruning could be computationally very expensive. Therefore, a systematic identification of subproblems that can be efficiently solved by ILP technology is key for the hybrid method to pay off. In Section 3.1, we define a machine learning approach useful to design the exploration mechanisms within the hybrid DD-ILP algorithm.

3.1 Supervised Learning to Identify Complementarity

Recently, machine learning (ML) has served as an important tool to enhance discrete optimization solvers, see, e.g., [24, 11] and [4] for a recent survey. We propose to use ML to build predictive models which lead us to define a decision-making tool that classifies when a node should be explored by an ILP model or by a DD-based branch and bound. This approach is similar to a portfolio-based algorithm selection [33]. The distinction, however, is that our subproblems are defined dynamically during search, as driven by our DD construction. The classification is based on the node state $s(u)$, which contains all the information required to define the subproblem associated with u (i.e., $\mathcal{D}_{ut}$).

We present three different learning experiments which lead to three different classifiers. The first two classifiers aim at predicting the most suitable method (either ILP or DD solver) for tackling a particular instance. They are closely related and presented in Section 3.1.1. Next, in a third experiment, we focus only on the ILP solver performance and learn a classifier to discriminate when an instance could be efficiently solved by an ILP solver. This classifier is presented in Section 3.1.2. The methodology for all the experiments consists of four steps: dataset generation, features design, label definition, and learning experiments.

3.1.1 Learning to classify between ILP or DD solver

We describe, for the case of MISP, the four steps in the methodology.

1. Dataset Generation. We consider a set of randomly defined test-cases to train our classifier. Since subproblems related to DD nodes correspond to vertex-induced subgraphs, we generate a dataset composed of random graphs according to different graph generator schemes, namely, the Erdös-Rényi (ER) [18], Watts-Strogatz (WS) [32], Barabási-Albert (BA) [3], and Holme-Kim (HK) [20] models. In addition, to include even more different graph structures in the test bed, we generate additional instances as follows. We create two graphs ($\mathcal{G}_1$, $\mathcal{G}_2$) resulting from different graph generator models. We then connect both graphs by defining a number of edges e which link randomly selected nodes from $\mathcal{G}_1$ and $\mathcal{G}_2$. We combine each pair of graph models from the named four generators, giving us a total of six new graph types, that we name ERWS (i.e., combination of an ER graph and a WS graph), ERBA, ERHK, WSBA, WSHK, and BAHK. The number of nodes (n) is defined as $n = \{100, 125, 150, 175, 200\}$ and the density (p), in percentage, takes values from the set $p = \{10, 20, 30, 50, 70\}$. Finally, our dataset has $18,500$ instances.

2. Feature Design. We define and extract features that could better discriminate the performance between ILP and DD-based branch and bound. As we compare two different representations of the problem, we only define features from graph metrics and properties of the instance. Since an instance for the MISP corresponds to an undirected graph, we select 17 specific graphical features: number of nodes (n), number of edges ($|\mathcal{E}|$), density (p), graph assortativity, vertex connectivity, edge connectivity, graph transitivity, the average clustering coefficient, and the average and maximum number of triangles. We also derive seven features that come from node degrees: mean, median, standard deviation (SD), minimum, maximum, the interquartile range (IR), and the variability score (SD/mean).

3. Label Definition. We now need to establish the performance of each method based on our instance set. We solve each MISP instance with both ILP and DD solvers. For ILP solver and to deal with performance variability issues [27], we solve each instance 5 times with a different random seed using `IBM-CPLEX 12.8`. Then, the ILP solution time for each instance is the average of the 5 runs. Finally, for each instance, we assign a label (either `ILP` or `DD`) according to the minimum solution time between both methods.

4. Supervised Learning Experiments. Finally, we construct the classifier using traditional supervised learning methodologies. We randomly split the dataset into a training set and a test set, with 13875 (75%) and 4625 instances (25%), respectively. Each feature is normalized to have a mean 0 and a standard deviation 1. Each experiment consists of a training phase with 5-fold cross validation to grid search the hyper-parameters, and a test phase on the neutral test set. We test Support Vector Machines (SVM) with RBF kernel [15] and Random Forests (RF) [12]. As a measure of baseline performance, we compare both methods versus a dummy classifier (DUM), which follows a stratified strategy.

We implemented this methodology using Python with Scikit-learn [30]. Table 1 presents the standard performance measures for binary classification, namely, accuracy, precision, recall and f1-score. We observe high accuracy scores obtained from both SVM and RF. This corroborates that there is a statistical pattern to be learned when selecting the best method between ILP and DD solvers. In addition, the designed features can capture such discrimination. In the experiments, we use RF due to its interpretability. Scikit-learn provides scores which rank features based on their importance for the RF prediction. We report them in Table 2 for features appearing in the top-5 of the experiment.

We observe in Table 2 that features related to density (1st), number of triangles (2nd), average clustering (3rd), graph transitivity (4th) and the degree of nodes (5th) significantly discriminate ILP and DD.

In a second learning experiment, we learn a classifier that predicts whether solving an instance with an ILP solver is *significantly easier* than using a DD solver. From the previous experiment, we slightly modify the label definition based on the solving times as follows. For each instance, we assign the label `ILP` if the ILP solver is x times faster than the DD solver, otherwise we assign the label `NO-ILP`. Following the same methodology for the previous learning experiment, we obtain the performance metrics presented in Table 3.

We observe that both the SVM and RF models achieve high performance

Table 1: Performance measures for the three classifiers when predicting `ILP` or `DD`

	DUM	SVM	RF
Accuracy	0.543	0.963	0.962
Precision	0.309	0.936	0.944
Recall	0.330	0.951	0.937
F1-score	0.319	0.943	0.941

Table 2: Top-5 features ranked by importance score from RF

Feature	Score
Density	0.2179
Avg. number of triangles	0.2147
Avg. clustering	0.1481
Graph transitivity	0.1090
Avg. degree	0.0661

Table 3: Performance measures for the three classifiers when predicting `ILP` or `NO-ILP`

	DUM	SVM	RF
Accuracy	0.656	0.989	0.989
Precision	0.222	0.983	0.981
Recall	0.212	0.967	0.969
F1-score	0.216	0.975	0.975

metrics in the classification. In addition, such metrics are significantly better than the baseline performance obtained by the dummy classifier showing that the learning models are actually learning something useful from the data.

3.1.2 Learning to classify easy/hard MISP instances for an ILP solver

In a third learning experiment, we aim at classifying whether a MISP instance will be efficiently solved by an ILP solver. For this experiment, we use the same dataset and features than in subsection 3.1.1, so steps 1 and 2 of the methodology are already described. However, in this case, we redefine the label to focus on the ILP solver explaining this in steps 3 and 4.

3. Label Definition. Each MISP instance is solved with `IBM-CPLEX 12.8` (5 times with a different random seed to deal with performance variability issues) to get the ILP solving time ILP_{time}. In addition, a threshold value t is defined to binarize the label. Finally, for each instance, we assign the

Table 4: Performance measures for the three classifiers when predicting E or H

	DUM	SVM	RF
Accuracy	0.505	0.976	0.977
Precision	0.461	0.954	0.956
Recall	0.461	0.997	0.997
F1-score	0.461	0.975	0.976

Table 5: Top-5 features ranked by importance score from RF

Feature	Score
Number of nodes	0.483
Number of edges	0.082
Degree SD	0.069
Avg. clustering	0.046
Degree IR	0.044

label E (*easy*, i.e., $ILP_{time} \leq t$) or H (*hard*, i.e., $ILP_{time} > t$).

4. Supervised Learning Experiments. We construct the classifier using the three supervised learning algorithms presented in subsection 3.1.1. In addition, we follow the same strategy for splitting the dataset, normalizing the features, and performing the training phase.

Table 4 presents the standard performance measures for binary classification. Similarly to the previous experiment, we remark that SVM and RF achieve high performance scores. Therefore, we conclude that both methods are able to capture enough about the ILP solver performance on training set to make meaningful predictions on test set. This is, most of the time, easy (E) instances are predicted as easy and, hard (H) instances tend to be predicted as hard.

Additionally, we report the top-5 most important features from RF in Table 5. We observe that the importance score of the number of nodes is, by far, the most involved feature in the discrimination of easy/hard instances for the ILP solver in this dataset.

From here, the three trained classifiers (i.e., ILP/DD, ILP/NO-ILP, and E/H) and the insights obtained from them are used to guide the node exploration into the hybrid method and benchmark its performance.

3.2 Learning to explore within a hybrid DD-ILP for MISP

The hybrid approach proposed for the MISP profits from the learning experiments in Section 3.1 to incorporate mechanisms to guide, on-the-fly, the exploration of the solution space. The hybrid is presented in Algorithm 1.

At the beginning, we include the initial DD root node r (which corresponds to the initial state) on the list of open nodes L. Next, we initialize the optimal solution value z_{opt} and the longest path from the root node

Algorithm 1 Hybrid DD-ILP for MISP

Input: MISP instance, `ILP`/`NO-ILP` classifier, decision tree (n_{ILP}, p_{ILP})
Output: z_{opt}

1: initialize $L = \{r\}$, where r corresponds to the DD root node (initial state)
2: let $z_{opt} = -\infty$ and $v^*(r) = 0$
3: extract features $ft(r)$ from subproblem associated with r
4: prediction $\leftarrow$ `ILP`/`NO-ILP` classifier$(ft(r))$
5: **if** prediction = `ILP` **then**
6: $\quad z_r^* \leftarrow$ solve_ILP(r)
7: $\quad z_{opt} = v^*(r) + z_r^*$
8: $\quad$ **return** z_{opt}
9: **while** $L \neq \emptyset$ **do**
10: $\quad u \leftarrow$ select_node(L), $L \leftarrow L \backslash \{u\}$
11: $\quad$ extract features size n_u and density p_u
12: $\quad$ **if** decision_tree(u) = `ILP` **then**
13: $\quad\quad z_u^* \leftarrow$ solve_ILP(u), $v^*(\mathcal{D}_{ut}) = v^*(u) + z_u^*$
14: $\quad\quad$ **if** $v^*(\mathcal{D}_{ut}) > z_{opt}$ **then**
15: $\quad\quad\quad z_{opt} \leftarrow v^*(\mathcal{D}_{ut})$
16: $\quad$ **else**
17: $\quad\quad$ create relaxed DD $\overline{\mathcal{D}}_{ut}$ with root u and $v_r = v^*(u)$
18: $\quad\quad$ **if** $\overline{\mathcal{D}}_{ut}$ is exact **then**
19: $\quad\quad\quad$ **if** $v^*(\overline{\mathcal{D}}_{ut}) > z_{opt}$ **then**
20: $\quad\quad\quad\quad z_{opt} \leftarrow v^*(\overline{\mathcal{D}}_{ut})$
21: $\quad\quad$ **if** $\overline{\mathcal{D}}_{ut}$ is not exact **then**
22: $\quad\quad\quad$ **if** $v^*(\overline{\mathcal{D}}_{ut}) > z_{opt}$ **then**
23: $\quad\quad\quad\quad$ let C be an exact cutset of $\overline{\mathcal{D}}_{ut}$
24: $\quad\quad\quad\quad$ **for all** u' in C **do**:
25: $\quad\quad\quad\quad\quad$ let $v^*(u') = v^*(u) + v^*(\overline{\mathcal{D}}_{uu'})$, add u' to L
26: $\quad\quad\quad$ create restricted DD $\mathcal{D}'_{ut}$ with root u and $v_r = v^*(u)$
27: $\quad\quad\quad$ **if** $v^*(\mathcal{D}'_{ut}) > z_{opt}$ **then** $z_{opt} \leftarrow v^*(\mathcal{D}'_{ut})$
28: return z_{opt}

to itself $v^*(r)$. For such initial subproblem, we want to predict if solving it by ILP technology could be much easier than using a pure DD branch-and-bound. For this purpose we use once the trained `ILP/NO-ILP` classifier developed in Section 3.1.1. If the classifier predicts `ILP`, we simply prune the root node with ILP technology solving the problem to optimality. This automated ML-driven feature of the hybrid framework allows to avoid an unnecessary DD-based exploration.

Conversely, if the classifier predicts the root node as `NO-ILP`, we start the DD-based exploration. While open DD nodes remain in L, we select a node $u \in L$ to be explored. The search strategy used to select nodes from L follows a best-first search algorithm which is based on the upper bound obtained from the relaxed DD in which u was created. Let $v^*(\mathcal{D}_{ru}) = v^*(u)$ be the longest path from the root node r to u and $v^*(\mathcal{D}_{ut})$ the longest path from u to the terminal node t. Since u is an exact node which was identified in an exact cutset from a previously computed relaxed DD, $v^*(u)$ is known. Next, we have to determine if (a) we solve node u to optimality by finding $v^*(\mathcal{D}_{ut})$ with an ILP solver, or (b) if we create a new relaxed DD $\overline{\mathcal{D}}_{ut}$ to compute an upper bound $v^*(\overline{\mathcal{D}}_{ut})$ on $v^*(\mathcal{D}_{ut})$.

Thus, we must predict whether we invoke the ILP solver to prune the node with the ILP-based pruning strategy or not. This prediction could possibly be performed by evaluating both the `ILP/DD` and `E/H` classifiers. However, these classifications require collecting, on-the-fly, expensive graph features for every selected DD node and the computational cost does not pay off for the overall DD-ILP performance. Therefore, we propose to get a proxy for the classifiers predictions by evaluating a unique and computationally inexpensive decision tree, specifically defined from significant features identified when training the classifiers.

The feature importance scores obtained during training of the `ILP/DD` and `E/H` classifiers are useful to get insights and select the features to use in a simple decision tree. We select the most important feature during training for each classifier, i.e., the density (Table 2) and size (number of nodes in Table 5) of the graph. Thus, in the DD-ILP framework, we only compute two features, the size n_u and density p_u of the subproblem (i.e., associated vertex-induced subgraph) encoded by each selected DD node u. In addition, we define two threshold values, the maximum size (n_{ILP}) and maximum density (p_{ILP}) to fully describe the decision tree. To sum up, the evaluation, on-the-fly, of the decision tree presented in Figure 6 determines whether the ILP solver is invoked to prune a particular DD node.

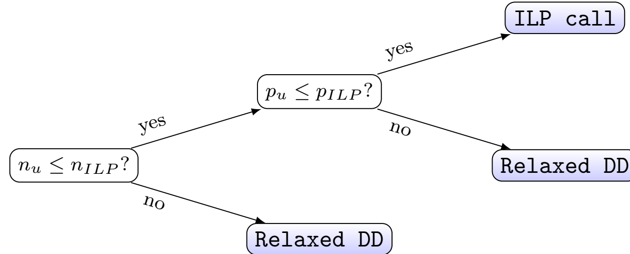


Figure 6: Decision tree to be evaluated at each DD node within the DD-ILP framework

In case the decision tree evaluates node u as `ILP call`, we prune the node by finding the optimal solution for such subproblem z_u^* with an ILP call. Solving the ILP subproblem associated with the DD node u provides $v^*(\mathcal{D}_{ut})$ which is used to compute z_u^* as $v^*(\mathcal{D}_{ut}) + v^*(u)$. Note that z_u^* is a lower bound on z_{opt} and no more exploration from u is needed. Otherwise, we create a relaxed DD $\overline{\mathcal{D}}_{ut}$. If $\overline{\mathcal{D}}_{ut}$ is exact (i.e., the maximum width W is never exceeded during compilation) there is no need of further branching from node u and we update the lower bound if necessary. On the contrary, if $\overline{\mathcal{D}}_{ut}$ is not exact, we compute an upper bound on the optimal solution for such subproblem u and identify an exact cutset C of $\overline{\mathcal{D}}_{ut}$ to include the nodes in C to the list L. In addition, we obtain a lower bound on the optimal solution of subproblem u by compiling a restricted DD $\mathcal{D}'_{ut}$.

4 Computational Experiments with the DD-ILP Approach

We perform experiments in order to compare the performance of the DD-ILP framework with respect to both the stand-alone DD solver proposed in [7] and a commercial ILP solver. We use `IBM-CPLEX 12.8` as ILP solver in single-thread mode with default parameter settings. We implement the DD-ILP approach in C++, solving the ILP subproblems also with `IBM-CPLEX 12.8`. All experiments are run on a Linux machine, Intel(R) Xeon(R) CPU E5-2637 v4 at 3.50GHz (16 threads) and 128 GB RAM.

Our purpose is to evaluate the DD-ILP approach performance against both the ILP solver and DD-based branch and bound, in order to verify the effectiveness of our hybrid framework. The only strategy used in the DD-ILP solver is the ILP-based pruning, i.e., we solve the subproblems without any

time limit when the ILP solver is called. We also consider the lower cutoff procedure. The variable ordering and exact cut selection strategies used for all the experiments, both for the DD-ILP and for the stand-alone DD solver, are *the minimum number of states (MIN)* and *the frontier cutset (FC)*, respectively. We refer the reader to [10] where different variable ordering heuristics and exact cutset strategies are defined. The maximum width W in both cases is 128.

From the experiments in Section 3.1, we determine that small-size ($n \leq 150$) instances are efficiently solved by either the DD solver (specially, dense cases) or the ILP solver (mainly in sparse cases). Instances which are very sparse ($p \leq 10\%$) represent the hardest case to be solved. On the other hand, really dense instances (where $p > 40\%$) are efficiently solved by both the ILP and, specially, the DD solver. For that reason, in this section, we focus on intermediate-size instances.

We generate random instances with $n = \{250, 300\}$ and $p = [15, 30]\%$ according to the different 10 families defined in Section 3.1, namely, ER, WS, BA, HK, ERWS ERBA, ERHK, WSBA, WSHK, and BAHK. We define a group as a tuple family-size-density and consider 25 instances for each of the 28 groups considered in this test bed, for a total of 700 MISP instances[1]. For example, group ER-250-20 corresponds to Erdös-Rényi instances with 250 nodes and density 20%.

All the instances are processed three times by solving them with: the ILP solver, the stand-alone DD solver, and our DD-ILP framework. Each solver run uses only one thread with a time limit of 7,200 seconds. For the DD-ILP solver, we exactly use the Algorithm 1 (presented in Section 3.2). Specifically, the SVM presented in Section 3.1.1 is used as the trained `ILP/NO-ILP` classifier for the DD root node. Regarding the decision tree (Figure 6) evaluated at each DD node explored in the DD-based branching tree, after experimentation with the threshold values, we set $n_{ILP} = 180$ and $p_{ILP} = 20$ for all groups of instances.

Table 6 compares the performance of the three methods for the 25 instances of each group. Column 1 presents the group label. Columns 2 and 3 are related to the ILP solver and show, for each family, how many instances out of 25 were solved to optimality within time limit and the average solving time of the 25 instances (in seconds). Columns 4, 5, and 6 are associated with the stand-alone DD solver and present the number of instances solved to optimality, the average number of DD nodes explored, and the average

[1] This set of instances is available at `https://github.com/jaimegonzalezj/Instances-MISP-DDILP-paper.git`

solving time, respectively. Finally, columns 7, 8, 9, 10, and 11 are related to the hybrid DD-ILP solver performance for each group of instances. Column 7 presents the number of instances solved to optimality within time limit. Column 8 shows the number of instances (out of 25) where the root DD node was predicted by the classifier as `ILP`. Columns 9, 10, and 11 present the average number of DD nodes explored, the average number of ILP subproblems solved, and the average solving time.

First we focus on analyzing the three solvers in terms of number of instances solved and average solving time (columns 2, 3, 4, 6, 7, and 11). For example, group ER-250-20 (i.e., Erdös-Rényi (ER) instances with 250 nodes and density 20%), ER-300-20, and BA-300-20 are the hardest groups of instances to be solved no matter the method. For all methods, the 25 instances of each group hit the time limit before being solved to optimality.

In 10 out of 28 groups, ILP outperforms both DD and DD-ILP. However, in 4 out of those 10 groups, namely BA-250-30, BA-300-30, HK-250-30, and HK-300-30, all instances are efficiently solved (on avg. in less than 8 seconds) by any of the three methods, solving to optimality the 25 instances in each of these groups.

On the other hand, in 4 out of the 28 groups (ER-250-30, ER-300-30, WS-250-30, WS-300-30), DD solver outperforms both ILP and DD-ILP solvers. Specifically, note that for group WS-300-30, the DD-ILP and the DD solvers perform almost equally and the average solving time is much better than the one obtained with the ILP solver.

Interestingly, the proposed DD-ILP approach outperforms both the ILP and the DD solvers in 11 out of 28 groups. For example, for group ERBA-300-20, the DD-ILP approach solves to optimality all 25 instances whereas the ILP solver and the DD solver solve, 18 and 19, respectively. It is even more remarkable if we observe the overall performance (700 instances), the proposed hybrid DD-ILP approach solves more instances to optimality than the other two methods with the smallest average solving time.

It is worth mentioning that we also tested the hybrid DD-ILP without providing a lower cutoff (i.e., constraint (1) in Section 3) each time the ILP solver is called. On average for all the groups, the version presented in Table 6 (with respect to the hybrid version without the lower cutoff) has an improvement of 11.8% on the solving time and for some groups this improvement is up to 30%.

In conclusion, for groups of instances where ILP seems much better than DD, the hybrid DD-ILP profits from the algorithm-portfolio feature incorporated through the `ILP/NO-ILP` trained classifier at the root DD node. Furthermore, the most interesting fact is that for some family of instances

Table 6: Comparison between ILP, DD, and Hybrid DD-ILP solvers for each group of instances

	ILP solver		DD solver			Hybrid DD-ILP				
Family group	# instances solved	Avg. CPU time (s)	# instances solved	Avg. DD nodes explored	Avg. CPU time (s)	# instances solved	# classified ILP at root	Avg. DD nodes explored	Avg. # of subILPs	Avg. CPU time (s)
ER-250-20	0	7200.00	0	4759587	7200.00	0	0	277653	2100	7200.00
ER-250-30	22	4598.24	25	1283024	225.30	25	0	1283024	0	233.03
ER-300-20	0	7200.00	0	3535260	7200.00	0	0	3679	322	7200.00
ER-300-30	0	7200.00	25	5880746	1235.96	25	0	5880746	0	1278.64
WS-250-20	25	1040.45	25	641324	2279.15	25	0	641324	0	2371.83
WS-250-30	25	656.78	25	47732	60.87	25	0	47732	0	62.74
WS-300-20	6	6708.84	9	711026	6464.05	9	0	711026	0	6457.05
WS-300-30	25	3878.02	25	160139	205.64	25	0	160139	0	213.75
BA-250-20	5	6610.66	0	2591519	7200.00	25	0	18125	1769	2368.19
BA-250-30	25	6.42	25	5377	6.56	25	0	4741	7	7.89
BA-300-20	0	7200.00	0	2398494	7200.00	0	0	14886	3165	7200.00
BA-300-30	25	2.62	25	4353	7.20	25	12	1925	2	4.92
HK-250-20	25	288.13	0	2020336	7200.00	25	25	1	1	322.09
HK-250-30	25	2.80	25	5165	8.11	25	19	1223	1	4.20
HK-300-20	16	4593.45	0	1787542	7200.00	6	6	76140	8368	5908.74
HK-300-30	25	1.95	25	3687	8.58	25	24	105	1	2.47
ERWS-250-20	25	1125.28	25	1424150	3543.42	25	17	8213	118	824.15
ERWS-300-21	4	6996.75	0	2208236	7200.00	25	0	361452	1562	3631.58
ERBA-250-20	25	754.79	25	1402841	1199.26	25	0	30519	304	486.39
ERBA-300-20	18	5650.41	19	6132721	5761.62	25	0	358830	1617	2706.06
ERHK-250-20	25	657.10	25	1497968	1217.93	25	3	25978	284	441.55
ERHK-300-20	22	5175.26	21	5708902	5313.47	25	0	358705	1457	2437.91
WSBA-250-14	25	158.00	24	493894	2201.70	25	25	1	1	127.64
WSBA-300-15	25	1873.90	7	508490	6290.94	20	17	157450	17	2770.48
WSHK-250-14	25	153.28	24	316572	1499.97	25	21	1822	7	127.72
WSHK-300-15	25	1508.87	12	511939	5947.28	24	21	52217	5	1603.00
BAHK-250-14	25	327.51	0	1108809	7200.00	25	25	1	1	311.65
BAHK-300-15	3	6977.28	0	1055527	7200.00	0	0	116538	10601	7200.00
Total	496	88546.79	416	48205360	108277.01	559	215	10594195	31710	63503.67
Average		3162.39			3867.04					2267.99

(e.g., ERWS, ERBA, ERHK), the algorithm-portfolio feature is not enough. Then, the hybrid mechanisms (i.e., ILP-based pruning strategy) exploit the complementary strengths when integrating two different representations to get the best performance.

4.1 Comparison versus a traditional portfolio-based algorithm selection approach

In this experiment, we compare the hybrid DD-ILP with a classic portfolio-based algorithm approach. For this purpose, we deploy into production the `ILP/DD` trained classifier (described in Section 3.1.1) to define the portfolio algorithm as follows.

At the root node, we classify the problem either as `ILP` or `DD`. Then, we simply solve the problem using the corresponding solver according to the prediction. The comparison of the portfolio version with the hybrid approach allows us to assess the impact of the ILP-based pruning strategy within the hybrid DD-ILP algorithm.

Table 7 presents the performance metrics for the hybrid DD-ILP and the portfolio approach. The instances are aggregated by family instead of group as in Table 6. Column 1 presents the family label and column 2, the number of instances per family. Columns 3, 4, and 5 are related to the hybrid DD-ILP algorithm whereas columns 6, 7, and 8 are associated with the portfolio approach. Column 3 shows, for each family, how many instances were solved to optimality within time limit. Column 4 presents the number of instances where the root DD node was predicted as `ILP` by the `ILP/NO-ILP` classifier. Column 5 shows the average solving time for each family. Column 6 presents the number of instances solved to optimality by the portfolio approach. Column 7 shows the number of instances predicted as `ILP` by the `ILP/DD` classifier. Finally, column 8 presents the average solving time.

Table 7: Comparison between DD-ILP and Portfolio approach on number of solved instances and average solving time for each family

Instances		Hybrid DD-ILP			Portfolio Approach		
Family	# of instances	# solved instances	`ILP/NO-ILP`	Avg. CPU time (s)	# solved instances	`ILP/DD`	Avg. CPU time (s)
ER	100	50	0	3977.92	50	50	3975.31
WS	100	84	0	2276.34	84	2	2216.68
BA	100	75	12	2395.25	56	97	3456.25
HK	100	81	74	1559.38	93	100	1088.19
ERWS	50	50	17	2227.87	32	50	3875.8
ERBA	50	50	0	1596.23	44	48	3040.47
ERHK	50	50	3	1439.73	46	49	2914.96
WSBA	50	45	42	1449.06	50	50	734.46
WSHK	50	49	42	865.36	50	50	610.45
BAHK	50	25	25	3755.83	31	50	3532.66
Total	700	559			536		

We observe the benefits of the hybrid DD-ILP, through its ILP-based pruning strategy, mainly for families BA, ERWS, ERBA, and ERHK. The performance on these families shows that there is a significant difference between applying a classic algorithm portfolio and the proposed hybrid approach.

4.2 Sensitivity analysis on the ILP-based pruning strategy

We also want to observe the effect of varying the threshold values of the decision tree (n_{ILP}, p_{ILP}) on the overall hybrid DD-ILP performance. To illustrate the algorithm performance on a per-instance basis, we take the 25 instances of group ERHK-300-20 and solve them with the DD-ILP framework by using 5 different (n_{ILP}, p_{ILP}) combinations. Note that for all these instances the `ILP/NO-ILP` classifier never leads us to prune the DD root node with ILP (column 7 in Table 6). Thus, we can analyze a similar behavior of the hybrid DD-ILP only considering the effect of the ILP-based pruning strategy in this group.

We consider the combination ($n_{ILP} = 180$, $p_{ILP} = 20$) as the base case since those were the threshold values set in the previous experiments. Then, we evaluate four additional threshold combinations: ($n_{ILP} = 180$, $p_{ILP} = 17$), ($n_{ILP} = 180$, $p_{ILP} = 15$), ($n_{ILP} = 150$, $p_{ILP} = 20$), and ($n_{ILP} = 210$, $p_{ILP} = 20$). We choose such values to aim at evaluating the effect of small variations of n_{ILP} and p_{ILP} with respect to the base case combination. Note that the n_{ILP} threshold must be between 0 and n. On the other hand, the density of induced subgraphs does not generally differ much from the density of the original instance graph so, p_{ILP} should be close to p.

Figure 7 compares the solution time of the ILP solver versus the DD-ILP

approach for the 25 instances of group ERHK-300-20. We only compare the running time with the ILP solver's time because, on average for this group, the ILP solver dominates the DD solver (Table 6). Each marker corresponds to a MISP instance where the color and shape indicates the (n_{ILP}, p_{ILP}) combination used in the DD-ILP algorithm. The marker location above the diagonal means that DD-ILP approach outperforms the ILP solver in such instance for the corresponding (n_{ILP}, p_{ILP}) combination.

In general, we can observe that the parameters (n_{ILP}, p_{ILP}) used in the decision tree do affect the DD-ILP framework due to the significant differences in performances between the five combinations. We remark that several points are located far and above from the diagonal implying that DD-ILP approach is much better in those instances in comparison with the ILP solver. We also notice that the DD-ILP approach with $(n_{ILP}, p_{ILP}) = (180, 20)$ (represented by blue rhombus) achieves the best performance. None of those 25 instances are located below the diagonal. Conversely, if we observe the DD-ILP performance with either $(n_{ILP}, p_{ILP}) = (180, 15)$ (green triangles) or $(n_{ILP}, p_{ILP}) = (150, 20)$ (yellow circles), we can observe more markers below the diagonal, i.e., for the related instances, ILP solver outperforms DD-ILP framework for such threshold combination.

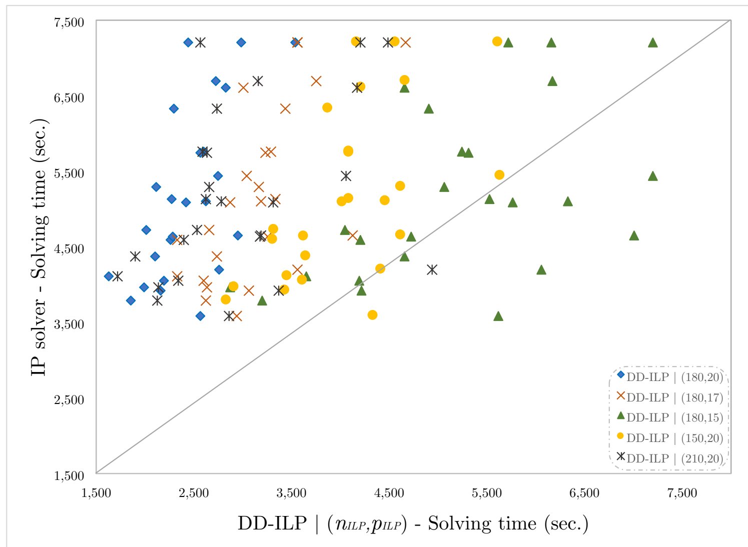


Figure 7: DD-ILP solver (using 5 different thresholds) versus the ILP solver in terms of solution time per instance of group ERHK-300-20

The solving time reduction obtained by the DD-ILP approach is ex-

plained by a smaller number of DD nodes explored in the DD-based branch and bound. Evidently, by the way in which the ILP-based pruning strategy is defined, there exists a trade-off between the number of ILP subproblems solved and the number of DD-nodes explored. The latter, in comparison with a pure DD solver. It is then worth mentioning the computation time spent solving the ILP subproblems within the hybrid DD-ILP. For the base case combination $(n_{ILP}, p_{ILP}) = (180, 20)$ in this group of instances, the average total solution time is $2,437.91$ seconds while the average total time solving the ILP subproblems is $1,555.29$. That is, the 64% of the algorithm time is consumed solving the ILP subproblems that are on average $1,457$ ILPs for this group.

The box plot presented in Figure 8 compares the stand-alone DD solver (red box) against the DD-ILP framework with the five different combinations varying the (n_{ILP}, p_{ILP}) thresholds. The plot shows, in log scale, the five-number summary (minimum, first quartile, median, third quartile, and maximum) of the number of DD nodes explored for the 25 instances of group ERHK-300-20. On the other hand, Figure 9 presents a box plot for the number of ILP subproblems solved for the same group (ERHK-300-20) using the different (n_{ILP}, p_{ILP}) combinations in the DD-ILP solver. To ease the analysis, note that we use the same color for each combination (n_{ILP}, p_{ILP}) in Figures 7, 8, and 9.

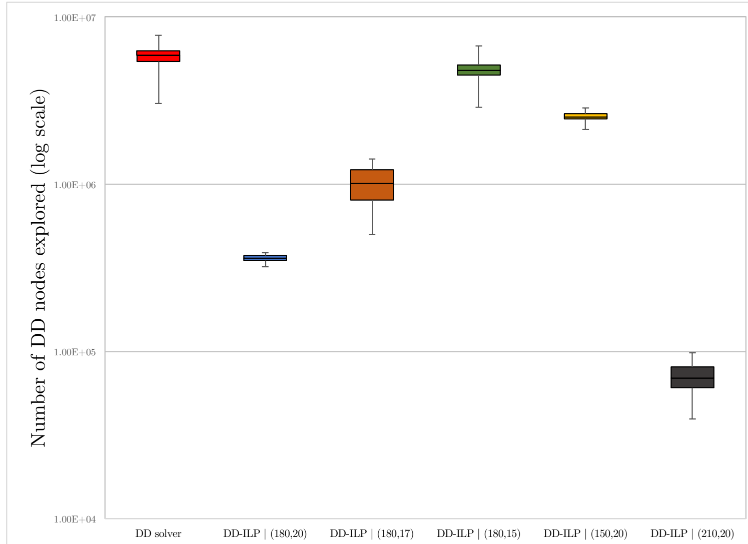


Figure 8: DD-ILP solver (using 5 different thresholds) versus the DD solver, in terms of the number of DD nodes explored for instances of group ERHK-300-20

From Figure 8, we observe that the number of DD-nodes explored with the pure DD solver is greater than such number for any combination of the DD-ILP approach. For example, on average, we can reduce 98.7% of the DD-nodes explored with the stand-alone DD solver (red box) by using the proposed DD-ILP approach with $(n_{ILP}, p_{ILP}) = (210, 20)$ (black box). If we compare the DD-ILP solver with $(n_{ILP}, p_{ILP}) = (180, 20)$ (blue box) and the DD-ILP solver $(n_{ILP}, p_{ILP}) = (210, 20)$ (black box), we note that the latter remarkably explores less DD-nodes. When we notice the number of ILP subproblems solved per instance by each configuration (Figure 9), DD-ILP $(n_{ILP}, p_{ILP}) = (210, 20)$ (black box) is close to the number reported by DD-ILP $(n_{ILP}, p_{ILP}) = (180, 20)$ (blue box). However, for the $(n_{ILP}, p_{ILP}) = (210, 20)$ combination, the ILP calls occur in DD nodes associated at subproblems with a larger number of vertices which are likely located in higher layers in the relaxed DD representations.

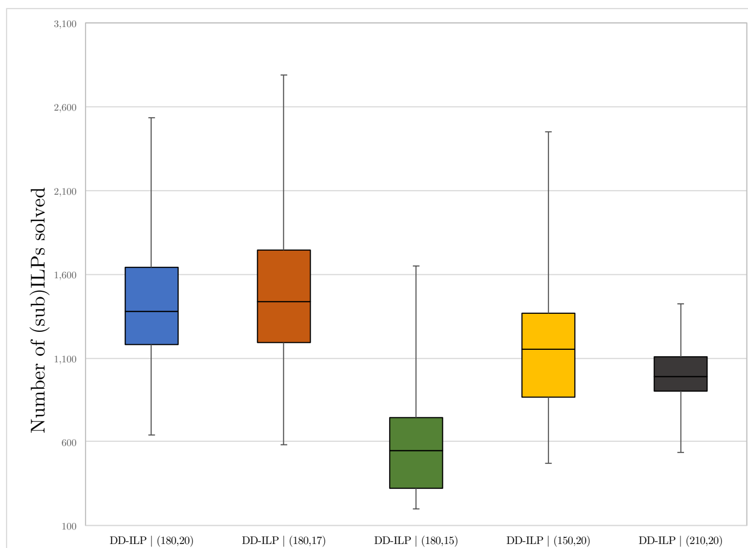


Figure 9: Number of subILPs solved in the DD-ILP solver (using 5 different thresholds) for instances of group ERHK-300-20

For this particular case, such earlier ILP calls (i.e., with the $((n_{ILP}, p_{ILP}) = (210, 20))$ combination) lead to prune more DD-nodes which is consistent with the behavior observed in Figure 8. However, by observing Figure 7, we note that this strategy does not pay off because DD-ILP with $(n_{ILP}, p_{ILP}) = (180, 20)$ outperfoms $(n_{ILP}, p_{ILP}) = (210, 20)$. In particular, on average for these instances, DD-ILP with $(n_{ILP}, p_{ILP}) = (180, 20)$ is faster than the DD-ILP with $(n_{ILP}, p_{ILP}) = (210, 20)$. We can observe such a superior performance in Figure 7 because in general the blue markers are located to the left of the black markers. This analysis indicates that a good approach is not just to prune as much as we can with the ILP calls but to use the ILP-based pruning in a more strategical way where it really exploits complementary strengths.

As we anticipated, the mechanism to classify the DD nodes in which the ILP-based pruning is performed is a critical component for the hybrid DD-ILP performance. In particular with this experiment, we can observe the significant impact when varying the threshold values in the classification. Moreover, the most remarkable fact is that the hybrid DD-ILP works effectively when using this simple decision tree.

5 Conclusion

In this paper, we propose a generic hybrid DD-ILP approach that is in principle suitable to any discrete optimization problem for which both a (mixed) integer programming formulation and a decision diagram representation are available. Our methodology consists of identifying DD nodes as subproblems through their associated states. We then introduce an original way to profit from an ILP representation when solving such subproblems. In addition, we use a supervised learning approach to derive a trained classifier and a decision-making tool which guide the exploration by verifying whether an ILP would efficiently prune a DD node.

The supervised learning experiments (Section 3.1) are essential for designing the proposed hybrid approach. From the trained classifiers and their feature importance scores, we incorporate mechanisms for node selection into the DD-ILP algorithm. The `ILP/NO-ILP` trained classifier is used at the DD root node, as a ML-driven black-box predictor to incorporate an algorithm-portfolio like feature that proves to be effective. Moreover, we profit from the feature importance scores (obtained when training the `ILP/DD` and `E/H` classifiers) to get insights and derive a decision tree that effectively proxies the classification needed for applying the ILP-based pruning strategy. Finally, the `ILP/DD` trained classifier is deployed in a traditional portfolio-based algorithm selection to benchmark the hybrid DD-ILP.

Computational results on the maximum independent set problem show that the DD-ILP approach can be effective if the DD representation reveals a problem structure that the ILP exploits well. In addition, we show that the framework works effectively even when using a simple decision tree to classify DD nodes. For the group of instances where the hybrid DD-ILP is shown to be superior, we observe that the problem structure is exploited by complementary strengths that are only leveraged through the proposed methodology. Then, it could be worth to include families of instances proposed in this paper, in other combinatorial optimization problems over graphs to compare the algorithms' performance.

We highlight that there is still room for research to define a unique and efficient trained classifier to identify DD nodes that can be pruned by ILP technology. In any case, the classification mechanism to be included within the hybrid algorithm is problem-specific and a feature where another machine learning approach (e.g., reinforcement learning) can be adopted. In addition, another research direction from the proposed hybrid DD-ILP is related to the search strategy. We experiment with a best-first search algorithm but novel search strategies could exploit the complementarity offered

by the hybrid approach.

This work suggests a research avenue where decision diagrams are used to explore and enumerate subproblems which can be tackled by other technologies, such as mixed-integer programming or constraint programming. In future research, we plan to extend the hybrid algorithm to other cases where a DD representation could be advantageous, such as in sequencing and scheduling problems.

Acknowledgements

We thank the anonymous referees for providing thoughtful, constructive and detailed comments which helped to improve this work.

References

[1] Akers, S.: Binary decision diagrams. Computers, IEEE Transactions on **C-27**(6), 509–516 (June 1978). https://doi.org/10.1109/TC.1978.1675141

[2] Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming – CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings. pp. 118–132. Springer Berlin Heidelberg (2007)

[3] Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999). https://doi.org/10.1126/science.286.5439.509

[4] Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. Preprint arXiv:1811.06128 (2018)

[5] Bergman, D., Cire, A.A.: Discrete nonlinear optimization by state-space decompositions. Management Science **64**(10), 4700–4720 (2018). https://doi.org/10.1287/mnsc.2017.2849

[6] Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.: Optimization bounds from binary decision diagrams. INFORMS Journal on

Computing **26**(2), 253–268 (2014), https://doi.org/10.1287/ijoc.2013.0561

[7] Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.: Discrete optimization with decision diagrams. INFORMS Journal on Computing **28**(1), 47–66 (2016), http://doi.org/10.1287/ijoc.2015.0648

[8] Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Variable ordering for the application of BDDs to the maximum independent set problem. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems. pp. 34–49. Springer Berlin Heidelberg (2012)

[9] Bergman, D., Cire, A.A., van Hoeve, W.J., Yunes, T.: Bdd-based heuristics for binary optimization. Journal of Heuristics **20**(2), 211–234 (Apr 2014). https://doi.org/10.1007/s10732-014-9238-1, https://doi.org/10.1007/s10732-014-9238-1

[10] Bergman, D., Cire, A.A., Hoeve, W.J.v., Hooker, J.: Decision Diagrams for Optimization. Springer Publishing Company, Incorporated, 1st edn. (2016)

[11] Bonami, P., Lodi, A., Zarpellon, G.: Learning a classification of mixed-integer quadratic programming problems. In: van Hoeve, W.J. (ed.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 595–604. Springer International Publishing, Cham (2018)

[12] Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324

[13] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (Aug 1986). https://doi.org/10.1109/TC.1986.1676819

[14] Cire, A.A., van Hoeve, W.J.: Multivalued decision diagrams for sequencing problems. Operations Research **61**(6), 1411–1428 (2013)

[15] Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20**(3), 273–297 (1995). https://doi.org/10.1007/BF00994018

[16] Cortés, C.E., Gendreau, M., Rousseau, L.M., Souyris, S., Weintraub, A.: Branch-and-price and constraint programming

for solving a real-life technician dispatching problem. European Journal of Operational Research **238**(1), 300–312 (2014). https://doi.org/10.1016/j.ejor.2014.03.006

[17] Easton, K., Nemhauser, G., Trick, M.: CP Based Branch-and-Price, pp. 207–231. Springer US, Boston, MA (2004). https://doi.org/10.1007/978-1-4419-8917-8_7

[18] Erdös, P., Rényi, A.: On the evolution of random graphs. In: Publications of the Mathematical Institute of the Hungarian Academy of Sciences. vol. 5, pp. 17–61 (1960)

[19] Grötschel, M., Lovász, L., Schrijver, A.: Stable Sets in Graphs, pp. 272–303. Springer Berlin Heidelberg (1988). https://doi.org/10.1007/978-3-642-97881-4_10, https://doi.org/10.1007/978-3-642-97881-4_10

[20] Holme, P., Kim, B.J.: Growing scale-free networks with tunable clustering. Physical Review E **65**(026107) (2002). https://doi.org/10.1103/physreve.65.026107

[21] Hooker, J.N., van Hoeve, W.J.: Constraint programming and operations research. Constraints **23**(2), 172–195 (Apr 2018). https://doi.org/10.1007/s10601-017-9280-3, https://doi.org/10.1007/s10601-017-9280-3

[22] Hooker, J.N.: Planning and scheduling by logic-based Benders decomposition. Operations Research **55**(3), 588–602 (2007)

[23] Hooker, J.N.: Integrated Methods for Optimization. Springer US, Boston, MA (2012). https://doi.org/10.1007/978-1-4614-1900-6

[24] Khalil, E.B., Dilkina, B., Nemhauser, G.L., Ahmed, S., Shao, Y.: Learning to run heuristics in tree search. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17. pp. 659–666 (2017). https://doi.org/10.24963/ijcai.2017/92

[25] Kinable, J., Cire, A.A., van Hoeve, W.J.: Hybrid optimization methods for time-dependent sequencing problems. European Journal of Operational Research **259**(3), 887–897 (2017). https://doi.org/10.1016/j.ejor.2016.11.035

[26] Lee, C.: Representation of switching circuits by binary-decision programs. Bell System Technical Journal, The **38**(4), 985–999 (July 1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x

[27] Lodi, A., Tramontani, A.: Performance variability in mixed-integer programming. In: Theory Driven by Influential Applications. pp. 1–12. INFORMS (2014). https://doi.org/10.1287/educ.2013.0112

[28] Lozano, L., Smith, J.C.: A binary decision diagram based algorithm for solving a class of binary two-stage stochastic programs. Mathematical Programming pp. 1–24 (2018), https://doi.org/10.1007/s10107-018-1315-z

[29] Morrison, D.R., Sewell, E.C., Jacobson, S.H.: Solving the pricing problem in a branch-and-price algorithm for graph coloring using zero-suppressed binary decision diagrams. INFORMS Journal on Computing **28**(1), 67–82 (2016). https://doi.org/10.1287/ijoc.2015.0667, https://doi.org/10.1287/ijoc.2015.0667

[30] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011), http://dl.acm.org/citation.cfm?id=1953048.2078195

[31] Tjandraatmadja, C., van Hoeve, W.J.: Target cuts from relaxed decision diagrams. INFORMS Journal on Computing **31**(2), 285–301 (2019). https://doi.org/10.1287/ijoc.2018.0830, https://doi.org/10.1287/ijoc.2018.0830

[32] Watts, D.J., Strogatz, S.H.: Collective dynamics of small-world networks. Nature **393**, 440–442 (1998). https://doi.org/10.1038/30918

[33] Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Int. Res. **32**(1), 565–606 (2008), https://doi.org/10.1613/jair.2490