Evaluating the Impact of AND/OR Search on 0-1 Integer Linear Programming

Radu Marinescu [a,*,1], Rina Dechter [b]

[a] *Cork Constraint Computation Centre, University College Cork, Ireland*

[b] *Donald Bren School of Information and Computer Science, University of California, Irvine, CA 92697, USA*

Abstract

AND/OR search spaces accommodate advanced algorithmic schemes for graphical models which can exploit the structure of the model. We extend and evaluate the *depth-first* and *best-first* AND/OR search algorithms to solving 0-1 Integer Linear Programs (0-1 ILP) within this framework. We also include a class of dynamic variable ordering heuristics while exploring an AND/OR search tree for 0-1 ILPs. We demonstrate the effectiveness of these search algorithms on a variety of benchmarks, including real-world combinatorial auctions, random uncapacitated warehouse location problems and MAX-SAT instances.

1 Introduction

A *constraint optimization problem* is the minimization (or maximization) of an objective function subject to a set of constraints on the possible values of a set of independent decision variables. An important class of optimization problems in operations research and computer science are the 0-1 Integer Linear Programming problems (0-1 ILP) [1] where the objective is to optimize a linear function of bi-valued integer decision variables, subject to a set of linear equality or inequality constraints defined on subsets of variables. The classical approach to solving 0-1 ILPs is the *Branch-and-Bound* method [2] which maintains the best solution found so far, while discarding partial solutions which cannot improve on the best.

* Corresponding author.
Email addresses: r.marinescu@4c.ucc.ie (Radu Marinescu), dechter@ics.uci.edu (Rina Dechter).

[1] This work was done while at the University of California, Irvine.

The AND/OR search space for graphical models [3] is a relatively new framework for search that is sensitive to the independencies in the model, often resulting in substantially improved performance. It is based on a pseudo tree that captures conditional independencies in the graphical model, resulting in a search tree exponential in the depth of the pseudo tree, rather than in the number of variables.

The AND/OR Branch-and-Bound search (AOBB) introduced in [4,5] is a Branch-and-Bound algorithm that explores the AND/OR search tree in a depth-first manner, while the AND/OR Branch-and-Bound search with caching algorithm (AOBB-C) [6,7] also saves previously computed results and retrieves them when the same subproblems are encountered again. The latter algorithm explores the context minimal search graph. A *best-first* AND/OR search algorithm (AOBF-C) that traverses the search graph was introduced subsequently [8,9,7]. Extensions to dynamic variable orderings were also presented and tested [10,8,5]. Two such extensions, depth-first AND/OR Branch-and-Bound with Partial Variable Ordering (AOBB+PVO) and best-first AND/OR search with Partial Variable Ordering (AOBF+PVO) were shown to have significant impact on several domains.

In this paper we apply the general principles of AND/OR search with context-based caching to the class of 0-1 ILPs, exploring both depth-first and best-first control strategies. We also extend dynamic variable ordering heuristics for AND/OR search and explore their impact on 0-1 ILPs.

We evaluate the impact of our advancement on several benchmarks for 0-1 ILP problems, including combinatorial auctions, random uncapacitated warehouse location problems and MAX-SAT problem instances. Our results show conclusively that these new algorithms improve dramatically over the traditional OR search, in some cases by several orders of magnitude. Specifically, we illustrate a tremendous gain obtained by exploiting problem decomposition (using AND nodes), equivalence (by caching), branching strategy (via dynamic variable ordering heuristics) and control strategy. We also show that the AND/OR algorithms are competitive and in some cases even outperform significantly commercial ILP solvers such as CPLEX.

The paper is organized as follows. Sections 2 and 3 provide background on 0-1 ILP and AND/OR search spaces, respectively. In Sections 4 and 5 we present the depth-first AND/OR Branch-and-Bound and the best-first AND/OR search algorithms for 0-1 ILP. Section 6 describes the AND/OR search approach that incorporates dynamic variable ordering heuristics. Section 7 is dedicated to our empirical evaluation, Section 8 overviews related work, while Section 9 provides a summary, concluding remarks and directions of future research.

The work we present here is based in part on two conference submissions [11,12]. It provides a detailed presentation of the proposed algorithms as well as an extended empirical evaluation.

2 Background

Notations The following notations will be used throughout the paper. We denote variables by uppercase letters ($e.g.$, $X, Y, Z, ...$), subsets of variables by bold faced uppercase letters ($e.g.$, $\mathbf{X}, \mathbf{Y}, \mathbf{Z}, ...$) and values of variables by lower case letters ($e.g.$, $x, y, z, ...$). An assignment $(X_1 = x_1, ..., X_n = x_n)$ can be abbreviated as $x = (\langle X_1, x_1\rangle, ..., \langle X_n, x_n\rangle)$ or $x = (x_1, ..., x_n)$. For a subset of variables $\mathbf{Y}$, $D_{\mathbf{Y}}$ denotes the Cartesian product of the domains of variables in $\mathbf{Y}$. $x_{\mathbf{Y}}$ and $x[\mathbf{Y}]$ are both used as the projection of $x = (x_1, ..., x_n)$ over a subset $\mathbf{Y}$. We denote functions by letters f, h, g etc., and the scope (set of arguments) of a function f by $scope(f)$.

DEFINITION **1 (constraint optimization problem)** *A finite* constraint optimization problem (COP) *is a four-tuple* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F}, z\rangle$, *where* $\mathbf{X} = \{X_1, ..., X_n\}$ *is a set of variables,* $\mathbf{D} = \{D_1, ..., D_n\}$ *is a set of finite domains,* $\mathbf{F} = \{F_1, ..., F_r\}$ *is a set of constraints on the variables and* $z(\mathbf{X})$ *is a global cost function defined over* $\mathbf{X}$ *(also called* objective function*) to be optimized (*$i.e.$*, minimized or maximized). The scope of a constraint* F_i*, denoted* $scope(F_i) \subseteq \mathbf{X}$*, is the set of arguments of* F_i*. Constraints can be expressed* extensionally*, through relations, or* intentionally*, by a mathematical formula (*$e.g.$*, equality or inequality) with two possible values,* $i.e.$*, it is satisfied or not. An* optimal solution *to a COP is a complete value assignment to all the variables such that every constraint is satisfied and the objective function is minimized or maximized.*

With every COP instance we can associate a *constraint graph* G which has a node for each variable and connects any two nodes whose variables appear in the scope of the same constraint.

DEFINITION **2 (induced graph, induced width [13])** *The* induced graph *of a constraint graph* G *relative to an ordering* d *of its nodes, denoted* $G^*(d)$*, is obtained as follows: nodes are processed from last to first; when node* X *is processed, all its preceding neighbors in the ordering are connected. A new edge that is added to the graph by this procedure is called an* induced edge*. Given a graph and an ordering of its nodes, the* width *of a node is the number of edges connecting it to nodes lower in the ordering. The* induced width *(or* treewidth*) of a graph, denoted* $w^*(d)$*, is the maximum width of nodes in the induced graph (for illustration see Example 1).*

DEFINITION **3 (linear program)** *A* linear program (LP) *consists of a set of* n *continuous non-negative variables* $\mathbf{X} = \{X_1, ..., X_n\}$ *and a set of* m *linear constraints (equalities or inequalities)* $\mathbf{F} = \{F_1, ..., F_m\}$ *defined on subsets of variables. The goal is to minimize a global linear cost function, denoted* $z(\mathbf{X})$*, subject to the constraints. One of the standard forms of a linear program is:*

$$min \quad z(\mathbf{X}) = \sum_{i=1}^{n} c_i \cdot X_i \tag{1}$$

$$\textit{s.t.} \quad \sum_{i=1}^{n} a_{ij} \cdot X_i \leq b_j, \quad \forall\, 1 \leq j \leq m \tag{2}$$

$$X_i \geq 0, \quad \forall\, 0 \leq i \leq n \tag{3}$$

where (1) *represents the linear objective function, and* (2) *defines the set of linear constraints. In addition,* (3) *ensures that all variables are non-negative.*

A linear program can also be expressed in a matrix notation, as follows:

$$min\{\mathbf{c}^{\top}\mathbf{X} \mid \mathbf{A} \cdot \mathbf{X} \leq \mathbf{b}, \mathbf{X} \geq 0\} \tag{4}$$

where $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and* $\mathbf{X} \in \mathbb{R}^n_+$*. Namely,* $\mathbf{c}$ *represents the cost vector and* $\mathbf{X}$ *is the vector of decision variables. The vector* $\mathbf{b}$ *and the matrix* $\mathbf{A}$ *define the* m *linear constraints.*

One of the most important constraint optimization problems in operations research and computer science is *integer programming*. Applications of integer programming include scheduling, routing, VLSI circuit design, combinatorial auctions, and facility location [1]. Formally:

DEFINITION **4 (integer linear program)** *An* Integer Linear Program (ILP) *is a linear program where all the decision variables are constrained to have non-negative integer values. Formally,*

$$\textit{min} \quad z(\mathbf{X}) = \sum_{i=1}^{n} c_i \cdot X_i \tag{5}$$

$$\textit{s.t.} \quad \sum_{i=1}^{n} a_{ij} \cdot X_i \leq b_j, \quad \forall\, 1 \leq j \leq m \tag{6}$$

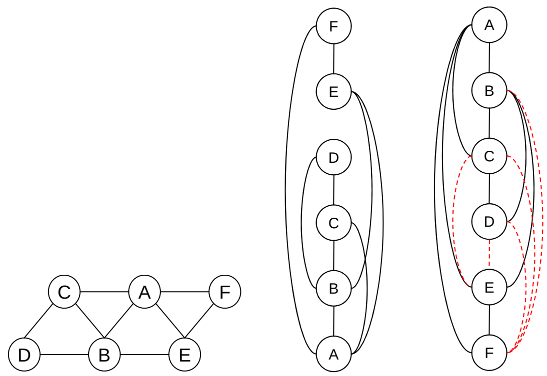
$$X_i \in \mathbb{Z}_+ \quad \forall\, 0 \leq i \leq n \tag{7}$$

If all variables are constrained to have integer values 0 or 1, then the problem is called 0-1 Integer Linear Program (0-1 ILP). *If not all variables are constrained to be integral (they can be real), then the problem is called* Mixed Integer Linear Program (MILP).

Example 1 *Figure 1(a) shows a 0-1 ILP instance with 6 binary decision variables* (A, B, C, D, E, F) *and 4 linear constraints* $F_1(A, B, C)$, $F_2(B, C, D)$, $F_3(A, B, E)$, $F_4(A, E, F)$*. The objective function to be minimized is defined by* $z = 7A + B - 2C + 5D - 6E + 8F$*. Figure 1(b) displays the constraint graph* G *associated with*

$$\begin{aligned}
&\text{minimize}: z = 7A + 3B - 2C + 5D - 6E + 8F \\
&\text{subject to}: \\
&\quad 3A - 12B + C \leq 3 \\
&\quad -2B + 5C - 3D \leq -2 \\
&\quad 2A + B - 4E \leq 2 \\
&\quad A - 3E + F \leq 1 \\
&\quad A, B, C, D, E, F \in \{0,1\}
\end{aligned}$$

(a) 0-1 Integer Linear Program



(b) Constraint graph (c) Induced graph along d_1 (d) Induced graph along d_2

Fig. 1. Example of a 0-1 Integer Linear Program.

this 0-1 ILP, where nodes correspond to the decision variables and there is an edge between any two nodes whose variables appear in the scope of the same constraint. Figures 1(c) and 1(d) show the induced graphs $G^*(d_1)$ and $G^*(d_2)$ obtained along the orderings $d_1 = (A, B, C, D, E, F)$ and $d_2 = (F, E, D, C, B, A)$, respectively. Notice that $G^*(d_1)$ does not contain any induced edges, while $G^*(d_2)$ contains 4 induced edges (dotted lines). The induced widths corresponding to the ordering d_1 and d_2 are 2 and 4, respectively.

While 0-1 integer linear programming, and thus integer linear programming and MILP are all NP-hard [14], there are many sophisticated techniques that can be used and allow solving very large instances in practice. We next briefly review the existing search techniques upon which we build our methods.

In **Branch-and-Bound** search, the best solution found so far (the *incumbent*) is kept in memory. Once a node in the search tree is generated, a lower bound (also known as a heuristic evaluation function) on the solution value is computed by solving a relaxed version of the problem, while honoring the commitments made on the search path so far. The most common method is to relax the integrality constraints of all undecided variables. The resulting *linear program* (LP) can be solved fast in practice using the *simplex* algorithm [15] (and in polynomial worst-case time using integer-point methods [16,17]). A path terminates when the lower bound is at least the value of the incumbent, or when the subproblem is infeasible or yields an integer solution. Once all paths have terminated, the incumbent is a provably optimal solution.

There are several ways to decide which leaf node of the search tree to expand next. In *depth-first* Branch-and-Bound, the most recent node is expanded next. In *best-first search* ($i.e.$, A^* search [18]), the leaf with the lowest lower bound is expanded next. A^* search is desirable because for any fixed branching variable ordering, no tree search algorithm that finds a provably optimal solution can guarantee expanding fewer nodes [19]. However, A^* requires exponential space. A variant of a best-first node-selection strategy, called *best-bound search*, is often used in MILP [20]. While in general A^* the children are evaluated when they are generated, in best-bound search the children are queued for expansion based on their parents' values and the LP of each child is solved only if the child comes up for expansion from the queue. Thus best-bound search needs to continue until each node on the queue has value no better than the incumbent. Best-bound search generates more nodes, but may require fewer (or more) LPs to be solved.

Branch-and-Cut Search for Integer Programming. A modern algorithm for solving MILPs is *Branch-and-Cut*, which was successful in solving large instances of the traveling salesman problem [21,22], and is now the core of the fastest commercial general-purpose integer programming packages. It is a *Branch-and-Bound* that uses the idea of *cutting planes* [1]. These are linear constraints that are deduced during search and, when added to the subproblem at a search node, may result in a smaller feasible space for the LP and thus a higher lower bound. Higher lower bounds can cause earlier termination of the search path, thus yielding smaller search trees.

Software Packages. CPLEX [2] is a leading commercial software product for solving MILPs. It uses Branch-and-Cut, and it can be configured to support many different branching algorithms ($i.e.$, variable ordering heuristics). It also makes available low-level interfaces ($i.e.$, APIs) for controlling the search, as well as other components such as the pre-solver, the cutting plane engine and the LP solver.

`lp_solve` [3] is an open source linear (integer) programming solver based on the *simplex* and the Branch-and-Bound methods. We chose to develop our AND/OR search algorithms in the framework of `lp_solve`, because we could have access to the source code. Unlike CPLEX, `lp_solve` does not provide a cutting plane engine nor a best-bound control strategy. We note however that open source LP solvers, with potentially better performance than `lp_solve`, such as: BPMPD [4], CLP [5], PCx [6], QSOPT [7], SOPLEX [8] or GLPK [9] are available online and any one of them can be used within our proposed framework to replace `lp_solve`.

[2] http://www.ilog.com/cplex/
[3] http://lpsolve.sourceforge.net/5.5/
[4] http://www-neos.mcs.anl.gov/neos/solvers/lp:bpmpd/MPS.html
[5] https://projects.coin-or.org/Clp
[6] http://www-fp.mcs.anl.gov/OTC/Tools/PCx/
[7] http://www.isye.gatech.edu/ wcook/qsopt/
[8] http://soplex.zib.de/
[9] http://www.gnu.org/software/glpk/

3 Extending AND/OR Search Spaces to 0-1 Integer Linear Programs

As mentioned earlier, the common way of solving 0-1 ILPs is by search, namely to instantiate variables one at a time following a static/dynamic variable ordering. In the simplest case, this process defines an OR search tree, whose nodes represent states in the space of partial assignments. However, this search space does not capture independencies that appear in the structure of the problem. To remedy this problem the idea of AND/OR search spaces [23] was recently introduced to general graphical models [3]. The AND/OR search space for a graphical model is defined using a backbone *pseudo tree* [24,25].

DEFINITION **5 (pseudo tree)** *Given an undirected graph $G = (\mathbf{V}, \mathbf{E})$, a directed rooted tree $\mathcal{T} = (\mathbf{V}, \mathbf{E}')$ defined on all its nodes is called a* pseudo tree *if any arc of G which is not included in $\mathbf{E}'$ is a back-arc, namely it connects a node to an ancestor in $\mathcal{T}$. The arcs of $\mathbf{E}'$ are not necessarily included in $\mathbf{E}$.*

We will next specialize the AND/OR search space for a 0-1 ILPs.

3.1 AND/OR Search Trees for 0-1 Integer Linear Programs

Given a 0-1 ILP instance, its constraint graph G and a pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree $S_{\mathcal{T}}$ has alternating levels of OR nodes and AND nodes. The OR nodes are labeled by X_i and correspond to the variables. The AND nodes are labeled by $\langle X_i, x_i \rangle$ (or simply x_i) and correspond to value assignments in the domains of the variables that are consistent relative to the constraints. The structure of the AND/OR tree is based on the underlying pseudo tree $\mathcal{T}$ of G. The root of the AND/OR search tree is an OR node, labeled with the root of $\mathcal{T}$. The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$, consistent along the path from the root. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in $\mathcal{T}$.

Semantically, the OR states represent alternative ways of solving the problem, whereas the AND states represent problem decomposition into independent subproblems, all of which need be solved. When the pseudo tree is a chain, the AND/OR search tree coincides with the regular OR search tree.

As usual [23,3], a *solution tree* T of an AND/OR search tree $S_{\mathcal{T}}$ is an AND/OR subtree such that: (i) it contains the root of $S_{\mathcal{T}}$, s; (ii) if a non-terminal AND node $n \in S_{\mathcal{T}}$ is in T then all of its children are in T; (iii) if a non-terminal OR node $n \in S_{\mathcal{T}}$ is in T then exactly one of its children is in T; (iv) all its terminal leaf nodes (full assignments) are consistent relative to the constraints of the 0-1 ILP.

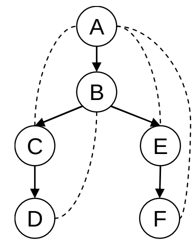
Example 2 *Consider the 0-1 ILP instance from Figure 2(a). A pseudo tree of the*

$$\text{minimize}: z = 7A + 3B - 2C + 5D - 6E + 8F$$

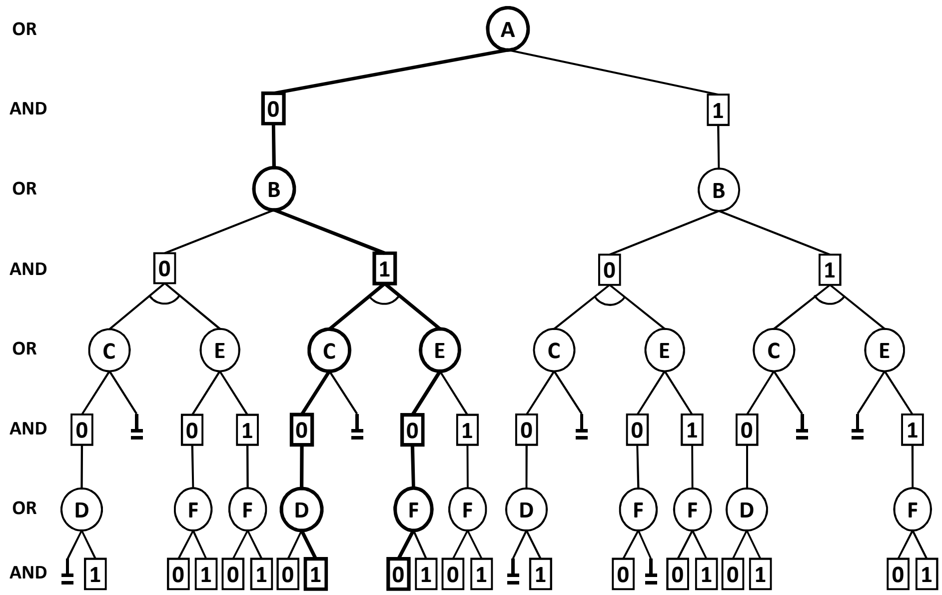
subject to:

$$3A - 12B + C \leq 3$$
$$-2B + 5C - 3D \leq -2$$
$$2A + B - 4E \leq 2$$
$$A - 3E + F \leq 1$$
$$A, B, C, D, E, F \in \{0,1\}$$

(a) 0-1 Integer Linear Program



(b) Pseudo tree



(c) AND/OR search tree

Fig. 2. AND/OR search tree for a 0-1 Integer Linear Program instance.

constraint graph, together with the back-arcs (dotted lines) are given in Figure 2(b). Figure 2(c) shows the corresponding AND/OR search tree. Notice that the partial assignment $(A = 0, B = 0, C = 0, D = 0)$ *which is represented by the path* $\{A, \langle A, 0\rangle, B, \langle B, 0\rangle, C, \langle C, 0\rangle, D, \langle D, 0\rangle\}$ *in the AND/OR search tree, is inconsistent because the constraint* $-2B + 5C - 3D \leq -2$ *is violated. Similarly, the partial assignment* $(A = 0, B = 0, C = 1)$ *is also inconsistent due to the violation of the same constraint for any value assignment of variable* D.

It was shown that:

THEOREM **1 (size of AND/OR search trees [3])** *Given a 0-1 ILP instance and a backbone pseudo tree* $\mathcal{T}$*, its AND/OR search tree* $S_{\mathcal{T}}$ *contains all consistent solutions, and its size is* $O(l \cdot 2^m)$ *where* m *is the depth of the pseudo tree and* l *bounds its number of leaves. If the 0-1 ILP instance has induced width* w^**, then there is a pseudo tree whose associated AND/OR search tree is* $O(n \cdot 2^{w^* \cdot \log n})$.

The arcs in the AND/OR search tree are associated with *weights* that are obtained from the objective function of the given 0-1 ILP instance.

DEFINITION **6 (weights)** *Given a 0-1 ILP with objective function* $\sum_{i=1}^{n} c_i \cdot X_i$ *and an AND/OR search tree* $S_{\mathcal{T}}$ *relative to a pseudo tree* $\mathcal{T}$*, the* weight $w(n,m)$ *of the arc from the OR node* n*, labeled* X_i *to the AND node* m*, labeled* $\langle X_i, x_i \rangle$*, is defined as* $w(n,m) = c_i \cdot x_i$.

Note that the arc-weights in general COPs are a more involved function of the input specification (see also [4,5] for additional details).

DEFINITION **7 (cost of a solution tree)** *Given a weighted AND/OR search tree* $S_{\mathcal{T}}$ *of a 0-1 ILP, and given a solution tree* T *having OR-to-AND set of arcs* $arcs(T)$*, the cost of* T*,* $f(T)$*, is defined by* $f(T) = \sum_{e \in arcs(T)} w(e)$.

With each node n of the search tree we can associate a *value* $v(n)$ which stands for the optimal solution cost of the subproblem below n, conditioned on the assignment on the path leading to it [3–5]. $v(n)$ was shown to obey the following recursive definition:

DEFINITION **8 (node value)** *The value of a node* n *in the AND/OR search tree of a 0-1 ILP instance is defined recursively by :*

$$v(n) = \begin{cases} 0 & \text{, if } n = \langle X, x \rangle \text{ is a terminal AND node} \\ \infty & \text{, if } n = X \text{ is a terminal OR node} \\ \sum_{m \in succ(n)} v(m) & \text{, if } n = \langle X, x \rangle \text{ is an AND node} \\ \min_{m \in succ(n)} (w(n,m) + v(m)) & \text{, if } n = X \text{ is an OR node} \end{cases}$$

where $succ(n)$ *denotes the children of* n *in the AND/OR tree.*

Example 3 *Figure 3 shows the weighted AND/OR search tree associated with the 0-1 ILP instance from Figure 2. The numbers on the OR-to-AND arcs are the weights corresponding to the objective function. For example, the weights associated with the OR-to-AND arcs* $(A, \langle A, 0 \rangle)$ *and* $(A, \langle A, 1 \rangle)$ *are* 0 *and* 7*, respectively. An optimal solution tree that corresponds to the assignment* $(A = 0, B = 1, C = 0, D = 0, E = 1, F = 0)$ *with cost* -3 *is highlighted. Note that inconsistent portions of the tree are pruned.*

Clearly, the value of the root node s is the minimal cost solution to the initial problem, namely $v(s) = \min_{\mathbf{X}} \sum_{i=1}^{n} c_i \cdot X_i$.

Therefore, search algorithms that traverse the AND/OR search tree and compute the value of the root node yield the answer to the problem. Consequently, depth-first search algorithms traversing the weighted AND/OR search tree are guaranteed to have time complexity bounded exponentially in the depth of the pseudo tree and can operate in linear space only.

$$\text{minimize}: z = 7A + 3B - 2C + 5D - 6E + 8F$$

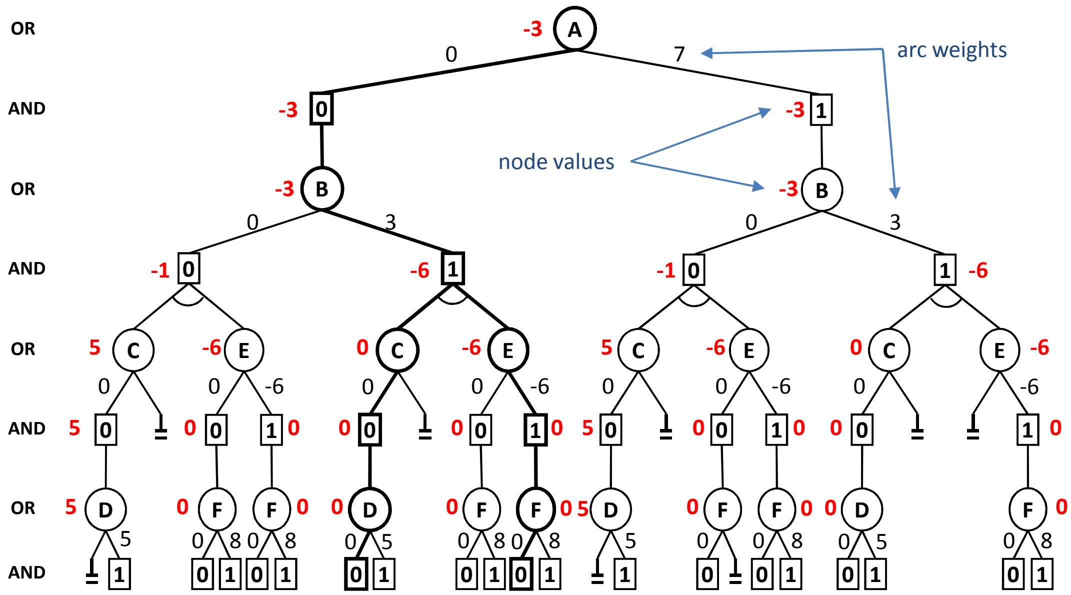
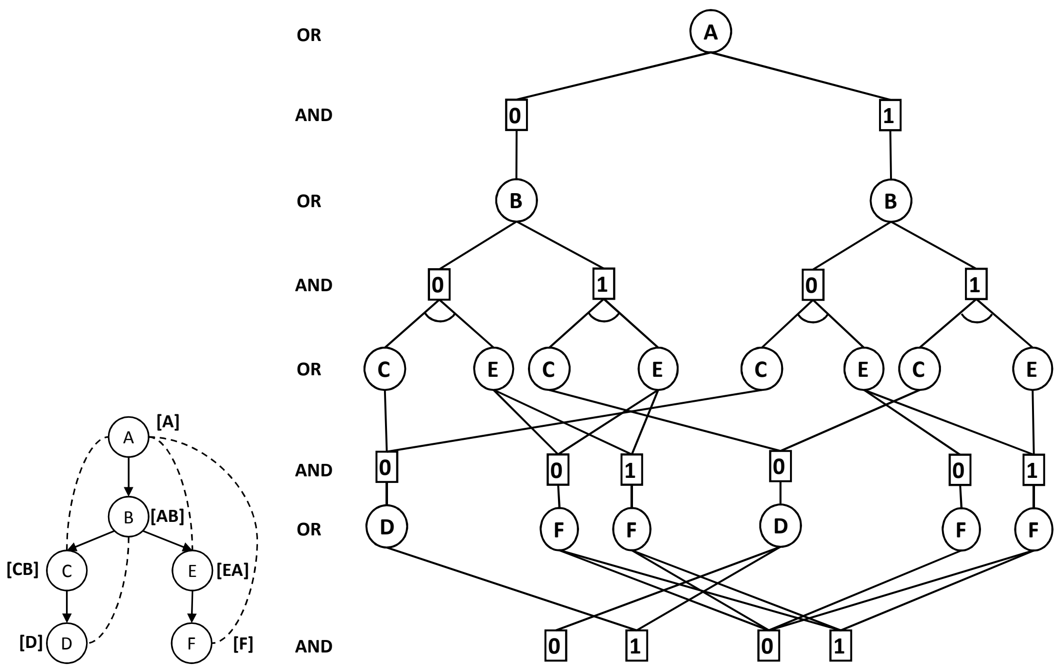


Fig. 3. Weighted AND/OR search tree for the 0-1 ILP instance from Figure 2.



(a) Pseudo tree (b) AND/OR search graph

Fig. 4. Context minimal AND/OR search graph for the 0-1 ILP instance from Figure 2.

3.2 AND/OR Search Graphs for 0-1 Integer Linear Programs

Often different nodes in the search tree root identical subtrees, and correspond to identical subproblems. Any two such nodes can be *merged*, reducing the size of the search space and converting it into a graph. Some of these mergeable nodes can be identified based on *contexts*, as described in [3] and as we briefly outline below.

Given a pseudo tree $\mathcal{T}$ of an AND/OR search space, the *context* of an AND node labeled $\langle X_k, x_k \rangle$, denoted by $context(X_k)$, is the set of ancestors of X_k in $\mathcal{T}$, including X_k, ordered descendingly, that are connected (in the induced graph) with descendants of X_k in $\mathcal{T}$. It is easy to see that $context(X_k)$ separates in the constraint graph, and also in the induced graph, the ancestors (in $\mathcal{T}$) of X_k from its descendants (in $\mathcal{T}$). Therefore, all subtrees in the AND/OR search tree that are rooted by the AND nodes labeled $\langle X_k, x_k \rangle$ are identical, given the same value assignment to the variables in $context(X_k)$. The *context minimal AND/OR graph* [3], denoted by $\mathcal{G}_{\mathcal{T}}$, can be obtained from the AND/OR search tree by merging all context mergeable nodes.

A depth-first search algorithm traverses the context minimal AND/OR graph by using additional memory. During search, it caches in memory all context mergeable nodes whose values have been determined. When the same nodes are encountered again ($i.e.$, corresponding to the same context instantiation), the algorithm retrieves from cache their previously computed values thus avoiding to explore the subspaces below them. This memoization process is referred to as *full caching*.

Example 4 *Figure 4(b) shows the context minimal AND/OR search graph corresponding to the 0-1 ILP from Figure 2, relative to the pseudo tree given in Figure 4(a). The square brackets next to each node in the pseudo tree indicate the AND contexts of the variables, as follows:* $context(A)$ = $\{A\}$, $context(B)$ = $\{A, B\}$, $context(C)$ = $\{B, C\}$, $context(D)$ = $\{D\}$, $context(E)$ = $\{A, E\}$ *and* $context(F)$ = $\{F\}$. *Consider for example variable* E *with* $context(E) = \{A, E\}$. *In Figure 3, the search trees below any appearance of* $(A = 0, E = 0)$ *(*$i.e.$*, corresponding to the subproblems below the AND nodes labeled* $\langle E, 0 \rangle$ *along the paths containing the assignments* $A = 0$ *and* $E = 0$*, respectively) are all identical, and therefore can be merged as shown in the search graph from Figure 4(b).*

It can be shown that:

THEOREM **2 (size of AND/OR graphs [3])** *Given a 0-1 ILP instance, its constraint graph* G*, and a pseudo tree* $\mathcal{T}$ *having induced width* $w^* = w_{\mathcal{T}}(G)$*, the size of the context minimal AND/OR search graph based on* $\mathcal{T}$*,* $\mathcal{G}_{\mathcal{T}}$*, is* $O(n \cdot 2^{w^*})$.

4 Depth-First AND/OR Branch-and-Bound Search for 0-1 ILPs

Traversing AND/OR search spaces by best-first algorithms or depth-first Branch-and-Bound was described as early as [23,26,27]. In a series of papers [4,6,10,8,9] we introduced extensions of these algorithms to AND/OR search spaces for constraint optimization tasks in graphical models. Our extensive empirical evaluations on a variety of probabilistic and deterministic graphical models demonstrated the power of these new algorithms over competitive approaches exploring traditional OR search spaces. In this section we revisit the notions of partial solution trees [23] to represent sets of solution trees, and heuristic evaluation function of a partial solution tree [4]. We will then recap the depth-first Branch-and-Bound algorithm for searching the AND/OR spaces, focusing on the specific properties for 0-1 ILPs.

We start with the definition of *partial solution tree* which is central to the algorithms.

DEFINITION **9 (partial solution tree)** *A* partial solution tree T' *of a context minimal AND/OR search graph* $\mathcal{G}_{\mathcal{T}}$ *is a subtree which: (1) contains the root node* s *of* $\mathcal{G}_{\mathcal{T}}$*; (2) if* n *is an OR node in* T' *then it contains one of its AND child nodes in* $\mathcal{G}_{\mathcal{T}}$*, and if* n *is an AND node it contains all its OR children in* $\mathcal{G}_{\mathcal{T}}$*. A node of* T' *is a* tip *node if it has no children in* T'*. A tip node of* T' *is either a* terminal *node (if it has no children in* $\mathcal{G}_{\mathcal{T}}$*), or a* non-terminal *node (if it has children in* $\mathcal{G}_{\mathcal{T}}$*).*

A partial solution tree represents $extension(T')$, the set of all full solution trees which can extend it. Clearly, a partial solution tree whose all tip nodes are terminal in $\mathcal{G}_{\mathcal{T}}$ is a solution tree.

Branch-and-Bound algorithms for 0-1 ILP are guided by the LP based lower bound heuristic function. The extension of heuristic evaluation functions to subtrees in an AND/OR search space was elaborated in [4,5]. We briefly introduce here the main elements and refer the reader for further details to the earlier references.

Heuristic Lower Bounds on Partial Solution Trees. We start with the notions of exact and heuristic evaluation functions of a partial solution tree, which will be used to guide the AND/OR Branch-and-Bound search.

The *exact evaluation function* $f^*(T')$ of a partial solution tree T' is the minimum of the costs of all solution trees extending T', namely: $f^*(T') = min\{f(T) \mid T \in extension(T')\}$. If $f^*(T'_n)$ is the exact evaluation function of a partial solution tree rooted at node n, then $f^*(T'_n)$ can be computed recursively, as follows:

1. If T'_n consists of a single node n then $f^*(T'_n) = v(n)$.
2. If n is an OR node having the AND child m in T'_n, then $f^*(T'_n) = w(n,m) + f^*(T'_m)$.
3. If n is an AND node having OR children $m_1, ..., m_k$ in T'_n, then $f^*(T'_n) = \sum_{i=1}^{k} f^*(T'_{m_i})$.

If each non-terminal tip node m of T' is assigned a heuristic lower bound estimate $h(m)$ of $v(m)$, then it induces a heuristic evaluation function on the minimal cost extension of T'. Given a partial solution tree T'_n rooted at n in the AND/OR graph $\mathcal{G}_{\mathcal{T}}$, the *tree-based heuristic evaluation function* $f(T'_n)$, is defined recursively by:

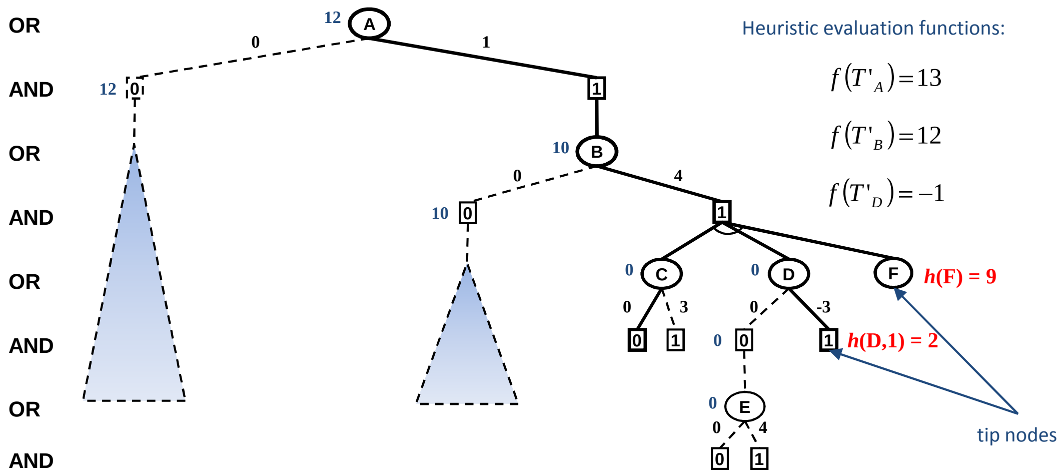


Fig. 5. Illustration of the pruning mechanism.

1. If T'_n consists of a single node n, then $f(T'_n) = h(n)$.
2. If n is an OR node having the AND child m in T'_n, then $f(T'_n) = w(n, m) + f(T'_m)$.
3. If n is an AND node having OR children $m_1, ..., m_k$ in T'_n, then $f(T'_n) = \sum_{i=1}^{k} f(T'_{m_i})$.

Clearly, by definition, $f(T'_n) \leq f^*(T'_n)$, and if n is the root of the context minimal AND/OR search graph, then $f(T') \leq f^*(T')$ [4,5].

During search, the algorithm maintains both an upper bound $ub(s)$ on the optimal solution $v(s)$, where s is the root of the search space, as well as the heuristic evaluation function $f(T')$ of the current partial solution tree T' being explored. Whenever $f(T') \geq ub(s)$, then searching below the current tip node t of T' is guaranteed not to yield a better solution cost than $ub(s)$ and, therefore, search below t can be halted.

In [4,5] we also showed that the pruning test can be sped up if we associate upper bounds with internal nodes as well. Specifically, if m is an OR ancestor of t in T' and T'_m is the subtree of T' rooted at m, then it is also safe to prune the search tree below t, if $f(T'_m) \geq ub(m)$.

Example 5 *Consider the partially explored weighted AND/OR search tree in Figure 5 (the weights and node values are given for illustration only). The current partial solution tree T' is highlighted. It contains the following nodes: A, $\langle A, 1\rangle$, B, $\langle B, 1\rangle$, C, $\langle C, 0\rangle$, D, $\langle D, 1\rangle$ and F. The nodes labeled by $\langle D, 1\rangle$ and by F are non-terminal tip nodes and their corresponding heuristic estimates are $h(\langle D, 1\rangle) = 2$ and $h(F) = 9$, respectively. The subtrees rooted at the AND nodes labeled $\langle A, 0\rangle$, $\langle B, 0\rangle$ and $\langle D, 0\rangle$ are fully evaluated, and therefore the current upper bounds of the OR nodes labeled A, B and D, along the active path, are $ub(A) = 12$, $ub(B) = 10$ and $ub(D) = 0$, respectively. The heuristic evaluation function of the partial solution tree rooted at the OR node A can be computed recursively, as follows:*

$$
\begin{aligned}
f(T'_A) &= w(A,1) + f(T'_{\langle A,1\rangle}) \\
&= w(A,1) + f(T'_B) \\
&= w(A,1) + w(B,1) + f(T'_{\langle B,1\rangle}) \\
&= w(A,1) + w(B,1) + f(T'_C) + f(T'_D) + f(T'_F) \\
&= w(A,1) + w(B,1) + w(C,0) + f(T'_{\langle C,0\rangle}) + w(D,1) + f(T'_{\langle D,1\rangle}) + h(F) \\
&= w(A,1) + w(B,1) + w(C,0) + 0 + w(D,1) + h(\langle D,1\rangle) + h(F) \\
&= 1 + 4 + 0 + 0 - 3 + 2 + 9 \\
&= 13
\end{aligned}
$$

The heuristic evaluation functions of the partial solution subtrees rooted at the OR nodes B and D along the current path can be computed in a similar manner, namely $f(T'_B) = 12$ and $f(T'_D) = -1$, respectively. Notice that while we could prune below $\langle D, 1\rangle$ because $f(T'_A) > ub(A)$, we could discover this pruning earlier by looking at node B only, because $f(T'_B) > ub(B)$. Therefore, the partial solution tree T'_A need not be consulted in this case.

The **Depth-First AND/OR Branch-and-Bound** search algorithm, `AOBB-C-ILP`, that traverses the context minimal AND/OR graph via full caching is described by Algorithm 1 and shown here for completeness. It specializes the Branch-and-Bound algorithm introduced in [6,7] to 0-1 ILPs. If the caching mechanism is disabled then the algorithm uses linear space only and traverses an AND/OR search tree [4,5].

The context based caching is done using tables [6,7]. For each variable X_i, a table is reserved in memory whose entries are indexed by each possible assignment to its context. Initially, each entry has a predefined value, in our case `NULL`. The fringe of the search is maintained by a stack called `OPEN`. The current node is denoted by n, its parent by p, and the current path by π_n. The children of the current node are denoted by $succ(n)$. The flag `caching` is used to enable the caching mechanism.

Each node n in the search graph maintains its current value $v(n)$, which is updated based on the values of its children. For OR nodes, the current $v(n)$ is an upper bound on the optimal solution cost below n. Initially, $v(n)$ is set to ∞ if n is OR, and 0 if n is AND, respectively. The heuristic function $h(n)$ in the search graph is computed by solving the LP relaxation of the subproblem rooted at n, conditioned on the current partial assignment along π_n ($i.e.$, $asgn(\pi_n)$) (lines 12 and 29, respectively). Notice that if the LP relaxation is infeasible, we assign $h(n) = \infty$ and in this case $v(n) = \infty$, denoting inconsistency. Similarly, if the LP has an integer solution, then $h(n)$ equals $v(n)$. In both cases, $succ(n)$ is set to the empty set, thus avoiding n's expansion (lines 7–8).

Before expanding the current AND node n, its cache table is checked (line 18). If the same context was encountered before, it is retrieved from the cache, and $succ(n)$ is set to the empty set, which will trigger the `PROPAGATE` step. Otherwise, the node is expanded in the usual way, depending on whether it is an AND or OR

Algorithm 1: `AOBB-C-ILP`: AND/OR Branch-and-Bound Search for 0-1 ILP

Input: A 0-1 ILP instance with objective function $\sum_{i=1}^{n} c_i X_i$, pseudo tree $\mathcal{T}$ rooted at X_1, AND contexts pas_i for every variable X_i, `caching` set to $true$ or $false$.
Output: Minimal cost solution.

```
1  create an OR node $s$ labeled $X_1$                          // Create and initialize the root node
2  $v(s) \leftarrow \infty$; $OPEN \leftarrow \{s\}$
3  if caching == true then
4      Initialize cache tables with entries "NULL"               // Initialize cache tables
5  while $OPEN \neq \emptyset$ do
6      $n \leftarrow top(OPEN)$; remove $n$ from $OPEN$; $succ(n) \leftarrow \emptyset$     // EXPAND
7      if $n$ is marked INFEASIBLE or INTEGER then
8          $v(n) \leftarrow \infty$ (if INFEASIBLE) or $v(n) \leftarrow h(n)$ (if INTEGER)
9      else if $n$ is an OR node, labeled $X_i$ then
10         foreach $x_i \in D_i$ do
11             create an AND node $n'$, labeled $\langle X_i, x_i \rangle$
12             $v(n') \leftarrow 0$; $h(n') \leftarrow LP(P_{n'})$           // Solve the LP relaxation
13             $w(n, n') \leftarrow c_i \cdot x_i$                        // Compute the arc weight
14             mark $n'$ as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
15             $succ(n) \leftarrow succ(n) \cup \{n'\}$
16     else if $n$ is an AND node, labeled $\langle X_i, x_i \rangle$ then
17         $cached \leftarrow false$; $deadend \leftarrow false$
18         if caching == $true$ and $Cache(asgn(\pi_n)[pas_i]) \neq NULL$ then
19             $v(n) \leftarrow Cache(asgn(\pi_n)[pas_i])$               // Retrieve value
20             $cached \leftarrow true$                              // No need to expand below
21         foreach OR ancestor $m$ of $n$ do
22             $lb \leftarrow$ evalPartialSolutionTree($T'_m$)
23             if $lb \geq v(m)$ then
24                 $deadend \leftarrow true$                         // Pruning
25                 break
26         if $deadend == false$ and cached == $false$ then
27             foreach $X_j \in children_{\mathcal{T}}(X_i)$ do
28                 create an OR node $n'$ labeled $X_j$
29                 $v(n') \leftarrow \infty$; $h(n') \leftarrow LP(P_{n'})$      // Solve the LP relaxation
30                 mark $n'$ as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
31                 $succ(n) \leftarrow succ(n) \cup \{n'\}$
32         else if $deadend == true$ then
33             $succ(p) \leftarrow succ(p) - \{n\}$
34     Add $succ(n)$ on top of $OPEN$                                // PROPAGATE
35     while $succ(n) == \emptyset$ do
36         if $n$ is an OR node, labeled $X_i$ then
37             if $X_i == X_1$ then
38                 return $v(n)$                                 // Search is complete
39             $v(p) \leftarrow v(p) + v(n)$                 // Update AND node value (summation)
40         else if $n$ is an AND node, labeled $\langle X_i, x_i \rangle$ then
41             if caching == $true$ and $v(n) \neq \infty$ then
42                 $Cache(asgn(\pi_n)[pas_i]) \leftarrow v(n)$        // Save AND node value in cache
43             if $v(p) > (w(p, n) + v(n))$ then
44                 $v(p) \leftarrow w(p, n) + v(n)$          // Update OR node value (minimization)
45         remove $n$ from $succ(p)$
46         $n \leftarrow p$
```

node (lines 9–33). The algorithm also computes the heuristic evaluation function for every partial solution subtree rooted at the OR ancestors of n along the path from the root (lines 21–25). The search below n is terminated if, for some OR ancestor m, $f(T'_m) \geq v(m)$, where $v(m)$ is the current upper bound on the optimal

Algorithm 2: Recursive computation of the heuristic evaluation function.

function: `evalPartialSolutionTree(`T'_n`)`
Input: Partial solution subtree T'_n rooted at node n.
Output: Heuristic evaluation function $f(T'_n)$.

if $succ(n) == \emptyset$ **then**
    return $h(n)$
else
    if n *is an AND node* **then**
        let $m_1, ..., m_k$ be the OR children of n in T'_n
        return $\sum_{i=1}^{k} evalPartialSolutionTree(T'_{m_i})$
    else if n *is an OR node* **then**
        let m be the AND child of n in T'_n
        return $w(n, m) + evalPartialSolutionTree(T'_m)$

cost below m. The recursive computation of $f(T'_m)$ is described in Algorithm 2.

The node values are updated by the `PROPAGATE` step (lines 35–46). It is triggered when a node has an empty set of descendants (note that as each successor is evaluated, it is removed from the set of successors in line 45). This means that all its children have been evaluated, and their final values are already determined. If the current node is the root, then the search terminates with its value (line 38). If n is an OR node, then its parent p is an AND node, and p updates its current value $v(p)$ by summation with the value of n (line 39). An AND node n propagates its value to its parent p in a similar way, by minimization (lines 43–44). Finally, the current node n is set to its parent p (line 46), because n was completely evaluated. Search continues either with a *propagation* step (if conditions are met) or with an *expansion* step.

`AOBB-C-ILP` is described relative to a static variable ordering determined by the underlying pseudo tree and explores the context minimal AND/OR search graph via *full caching*. However, if the memory requirements are prohibitive, rather than using full caching, `AOBB-C-ILP` can be modified to use a memory bounded caching scheme that saves only those nodes whose context size can fit in the available memory, as shown in [6,7].

5 Best-First AND/OR Search for 0-1 ILPs

We now direct our attention to a *best-first* rather than depth-first control strategy for traversing the context minimal AND/OR graph and present a best-first AND/OR search algorithm for 0-1 ILP. The algorithm uses similar amounts of memory as the depth-first AND/OR Branch-and-Bound with full caching. The algorithm was described in detail in [8,9,7] and evaluated for general constraint optimization problems. By specializing it to 0-1 ILP using the LP relaxation for h, we get `AOBF-C-ILP`. For completeness sake, we describe the algorithm again including minor modifications for the 0-1 ILP case.

Algorithm 3: `AOBF-C-ILP`: Best-First AND/OR Search for 0-1 ILP

Input: A 0-1 ILP instance with objective function $\sum_{i=1}^{n} c_i X_i$, pseudo tree $\mathcal{T}$ rooted at X_1, AND contexts pas_i for every variable X_i

Output: Minimal cost solution.

create an OR node s labeled X_1 // Initialize
$v(s) \leftarrow h(s)$; $\mathcal{G}'_{\mathcal{T}} \leftarrow \{s\}$
while *s is not labeled SOLVED* **do**
  $S \leftarrow \{s\}$; $T' \leftarrow \emptyset$; // Create the marked partial solution tree
  while $S \neq \emptyset$ **do**
    $n \leftarrow$ `top`(S); remove n from S
    $T' \leftarrow T' \cup \{n\}$
    let L be the set of marked successors of n
    if $L \neq \emptyset$ **then**
      add L on top of S
  let n be any nonterminal tip node of the marked T' (rooted at s) // **EXPAND**
  if *n is an OR node, labeled X_i* **then**
    foreach $x_i \in D_i$ **do**
      let n' be the AND node in $\mathcal{G}'_{\mathcal{T}}$ having context equal to pas_i
      if $n' == NULL$ **then**
        create an AND node n' labeled $\langle X_i, x_i \rangle$
        $h(n') \leftarrow LP(P_{n'})$; $v(n') \leftarrow h(n')$ // Solve the LP relaxation
        $w(n, n') \leftarrow c_i \cdot x_i$ // Compute the arc weight
        label n' as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
        if *n' is INTEGER or TERMINAL* **then**
          label n' as SOLVED
        else if *n' is INFEASIBLE* **then**
          $v(n') \leftarrow \infty$
      $succ(n) \leftarrow succ(n) \cup \{n'\}$
  else if *n is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
    foreach $X_j \in children_{\mathcal{T}}(X_i)$ **do**
      create an OR node n' labeled X_j
      $h(n') \leftarrow LP(P_{n'})$; $v(n') \leftarrow h(n')$ // Solve the LP relaxation
      label n' as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
      if *n' is INTEGER* **then**
        mark n' as SOLVED
      else if *n' is INFEASIBLE* **then**
        $v(n') \leftarrow \infty$
      $succ(n) \leftarrow succ(n) \cup \{n'\}$
  $\mathcal{G}'_{\mathcal{T}} \leftarrow \mathcal{G}'_{\mathcal{T}} \cup succ(n)$
  $S \leftarrow \{n\}$ // **REVISE**
  while $S \neq \emptyset$ **do**
    let m be a node in S such that m has no descendants in $\mathcal{G}'_{\mathcal{T}}$ still in S; remove m from S
    if *m is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
      $v(m) \leftarrow \sum_{m' \in succ(m)} v(m')$
      mark all arcs to the successors
      label m as SOLVED if all its children are labeled SOLVED
    else if *m is an OR node, labeled X_i* **then**
      $v(m) = min_{m' \in succ(m)} (w(m, m') + v(m'))$
      mark the arc through which this minimum is achieved
      label m as SOLVED if the marked successor is labeled SOLVED
    if *m changes its value **or** m is labeled SOLVED* **then**
      add to S all those parents of m such that m is one of their successors through a marked arc.
return $v(s)$ // Search terminates

The algorithm, denoted by `AOBF-C-ILP` (Algorithm 3), specializes Nilsson's `AO`* algorithm [23] to AND/OR search spaces for 0-1 ILPs. It interleaves forward expansion of the best partial solution tree (EXPAND) with a cost revision step (REVISE) that updates node values. The explicated AND/OR search graph, denoted by $\mathcal{G}'_{\mathcal{T}}$ is maintained, the current node is n, s is the root of the search graph and the current best partial solution subtree is denoted by T'. The children of a node n are denoted by $succ(n)$.

First, a top-down, graph-growing operation finds the best partial solution tree by tracing down through the marked arcs of the explicit AND/OR search graph $\mathcal{G}'_{\mathcal{T}}$ (lines 5–10). These previously computed marks indicate the current best partial solution tree from each node in $\mathcal{G}'_{\mathcal{T}}$. Before the algorithm terminates, the best partial solution tree, T', does not yet have all of its leaf nodes terminal. One of its non-terminal leaf nodes n is then expanded by generating its successors, depending on whether it is an OR or an AND node. Notice that when expanding an OR node, the algorithm does not generate AND children that are already present in the explicit search graph $\mathcal{G}'_{\mathcal{T}}$ (lines 14–16). All these identical AND nodes in $\mathcal{G}'_{\mathcal{T}}$ are easily recognized based on their contexts. Upon node's n expansion, a heuristic underestimate $h(n')$ of $v(n')$ is assigned to each of n's successors $n' \in succ(n)$ (lines 17 and 26). Again, $h(n')$ is obtained by solving the LP relaxation of the subproblem rooted at n', conditioned on the current partial assignment of the path to the root. As before, AOBF-C-ILP avoids expanding those nodes for which the corresponding LP relaxation is infeasible or yields an integer solution (lines 19–23 and 29–33).

The second operation in AOBF-C-ILP is a bottom-up, cost revision, arc marking, SOLVE-labeling procedure (lines 37–48). Starting with the node just expanded n, the procedure revises its value $v(n)$, using the newly computed values of its successors, and marks the outgoing arcs on the estimated best path to terminal nodes. This revised value is then propagated upwards in the graph. The revised value $v(n)$ is an updated lower bound estimate of the cost of an optimal solution to the subproblem rooted at n. During the bottom-up step, AOBF-C-ILP labels an AND node as SOLVED if all of its OR child nodes are solved, and labels an OR node as SOLVED if its marked AND child is also solved. The algorithm terminates with the optimal solution when the root node s is labeled SOLVED. We next summarize the complexity of both depth-first and best-first AND/OR graph search [3,6,8,7]:

THEOREM **3 (complexity)** *Given a 0-1 ILP and its constraint graph G, the depth-first AND/OR Branch-and-Bound and best-first AND/OR search algorithms guided by a pseudo tree $\mathcal{T}$ of G are sound and complete. Their time and space complexity is $O(n \cdot 2^{w^*})$, where w^* is the induced width of G along the pseudo tree.*

AOBB versus AOBF. Best-first search AOBF with the same heuristic function as depth-first Branch-and-Bound AOBB is likely to expand the smallest number of nodes [19], but empirically this depends on the optimal solution path itself. Second, AOBB can use far less memory by avoiding dead-caches for example ($e.g.$, when the context minimal search graph is a tree), while AOBF has to keep the explicated search graph in memory no matter what. Third, AOBB can be used as an anytime scheme, namely whenever interrupted, the algorithm outputs the best solution found so far, unlike `AOBF` which outputs a solution upon termination only. All the above points show that the relative merit of best-first versus depth-first over context minimal AND/OR search spaces cannot be determined by theory [19] and empirical evaluation is necessary.

6 Dynamic Variable Orderings

The depth-first and best-first AND/OR search algorithms presented in the previous sections assumed a static variable ordering determined by the underlying pseudo tree of the constraint graph. However, the mechanism of identifying unifiable AND nodes based solely on their contexts is hard to extend when variables are instantiated in a different order than that dictated by the pseudo tree. In this section we discuss a strategy that allows dynamic variable orderings in depth-first and best-first AND/OR search, when both algorithms traverse an AND/OR search tree. The approach called *Partial Variable Ordering (PVO)*, which combines the static AND/OR decomposition principle with a dynamic variable ordering heuristic, was described and tested also for general constraint optimization over graphical models in [10,5]. For completeness sake, we review it briefly next.

Variable Orderings for Integer Programming. At every node in the search tree, the search algorithm has to decide what variable to instantiate next. One common method in operations research is to select next the *most fractional variable*, $i.e.$, variable whose LP value is furthest from being integral [28]. Another relatively simple rule, which we used in our experiments, is *reduced-cost branching*. Specifically, the next fractional variable to instantiate has the smallest reduced cost ($i.e.$, dual value) [1] in the solution of the LP relaxation.

A more sophisticated approach, which is better suited for certain hard problems is *strong branching* [29]. This method performs a one-step lookahead for each variable that is non-integral in the LP at the node. The one-step lookahead computation solves the LP relaxations for each of the children of the candidate variable, and a score is computed based on the LP values of the children. The next variable to instantiate is selected as the one with the highest score among the candidates.

Pseudo-cost branching [30] is another rule that keeps a history of the success of the variables on which already has been branched. A score is then calculated for each variable based on its history, and the fractional variable with the highest score is instantiated next.

Partial Variable Ordering (PVO). *AND/OR Branch-and-Bound with Partial Variable Ordering* (resp. *Best-First AND/OR Search with Partial Variable Ordering*), denoted by `AOBB+PVO-ILP` (resp. `AOBF+PVO-ILP`), uses the static graph-based decomposition given by a pseudo tree with a dynamic semantic ordering heuristic applied over chain portions of the pseudo tree. For simplicity and without loss of generality we consider the *reduced cost* heuristic as our semantic variable ordering heuristic. Clearly, it can be replaced by any other heuristic.

Consider the pseudo tree from Figure 2 inducing the following variable groups (or chains): $\{A, B\}$, $\{C, D\}$ and $\{E, F\}$, respectively. This implies that variables $\{A, B\}$ should be considered before $\{C, D\}$ and $\{E, F\}$. The variables in each group can be dynamically ordered based on the semantic ordering heuristic.

`AOBB+PVO-ILP` (resp. `AOBF+PVO-ILP`) can be derived from Algorithm 1 (resp. Algorithm 3) with some simple modifications. As usual, the algorithm traverses an AND/OR search tree in a depth-first (resp. best-first) manner, guided by a pre-computed pseudo tree $\mathcal{T}$. When the current AND node n, labeled $\langle X_i, x_i \rangle$, is expanded in the forward step, the algorithm generates its OR successor m, labeled X_j, based on the semantic ordering heuristic. Specifically, m corresponds to the variable with the smallest reduced cost in the current pseudo tree chain. If there are no uninstantiated variables left in the current chain, namely variable X_i was instantiated last, then the OR successors of n are labeled by the variables with the smallest reduced cost from the variable groups rooted by X_i in $\mathcal{T}$.

7 Experimental Results

We evaluated the performance of the depth-first and best-first AND/OR search algorithms on 0-1 ILP problem classes such as combinatorial auction, uncapacitated warehouse location problems and MAX-SAT problem instances. We implemented our algorithms in C++ and carried out all experiments on a 2.4GHz Pentium IV with 2GB of RAM, running Windows XP.

Algorithms. The detailed outline of the experimental evaluation is given in Table 1. We evaluated the following 6 classes of AND/OR search algorithms:

1 Depth-first and best-first search algorithms using a static variable ordering and exploring the AND/OR tree, denoted by `AOBB-ILP` and `AOBF-ILP`, respectively.
2 Depth-first and best-first search algorithms using dynamic partial variable orderings and exploring the AND/OR tree, denoted by `AOBB+PVO-ILP` and `AOBF+PVO-ILP`, respectively.
3 Depth-first and best-first search algorithms with caching that explore the context minimal AND/OR graph and use static variable orderings, denoted by `AOBB-C-ILP` and `AOBF-C-ILP`, respectively.

All of these AND/OR algorithms use a *simplex* implementation based on the open-

Table 1
Detailed outline of the experimental evaluation for 0-1 ILP.

Problem classes	Tree search		Graph search	ILP solvers
	AOBB-ILP AOBF-ILP	AOBB+PVO-ILP AOBF+PVO-ILP	AOBB-C-ILP AOBF-C-ILP	BB (lp_solve) CPLEX 11.0
Combinatorial Auctions	$\surd$	$\surd$	$\surd$	$\surd$
Warehouse Location Problems	$\surd$	$\surd$	$\surd$	$\surd$
MAX-SAT Instances	$\surd$	$\surd$	$\surd$	$\surd$

source `lp_solve` library to compute the guiding LP relaxation. For this reason, we compare them against the OR Branch-and-Bound algorithm available from the `lp_solve` library, denoted by `BB`. The pseudo tree used by the AND/OR algorithms was constructed using the hypergraph partitioning heuristic described in [10,5] and outlined briefly below. `BB`, `AOBB+PVO-ILP` and `AOBF+PVO-ILP` used a dynamic variable ordering heuristic based on *reduced costs* and the ties were broken lexicographically.

We note however that the `AOBB-ILP` and `AOBB-C-ILP` algorithms support a restricted form of dynamic variable and value ordering. Namely, there is a dynamic internal ordering of the successors of the node just expanded, before placing them onto the search stack. Specifically, if the current node n is AND, then the independent subproblems rooted by its OR children can be solved in decreasing order of their corresponding heuristic estimates (variable ordering). Alternatively, if n is OR, then its AND children corresponding to domain values can also be sorted in decreasing order of their heuristic estimates (value ordering).

For reference, we also ran the ILOG CPLEX version 11.0 solver (with default settings), which uses a best-first control strategy, dynamic variable ordering heuristic based on strong branching, as well as cutting planes to tighten the LP relaxation. It explores however an OR search tree.

In the MAX-SAT domain we ran, in addition, three specialized solvers:

1 `MaxSolver` [31], a DPLL-based algorithm that uses a 0-1 non-linear integer formulation of the MAX-SAT problem,
2 `toolbar` [32], a classic OR Branch-and-Bound algorithm that solves MAX-SAT as a Weighted CSP problem [33], and
3 `PBS` [34], a DPLL-based solver capable of propagating and learning pseudo-boolean constraints as well as clauses.

`MaxSolver` and `toolbar` were shown to perform very well on random MAX-SAT instances with high graph connectivity [32], whereas `PBS` exhibits better performance on relatively sparse MAX-SAT instances [31]. These algorithms explore an OR search space.

Throughout our empirical evaluation we will address the following questions that govern the performance of the proposed algorithms:

1 The impact of AND/OR versus OR search.
2 The impact of best-first versus depth-first AND/OR search.
3 The impact of caching.
4 The impact of dynamic variable orderings.

Constructing the Pseudo Tree. Our heuristic for generating a low height balanced pseudo tree is based on the recursive decomposition of the dual hypergraph associated with the 0-1 ILP instance. The dual hypergraph of a 0-1 ILP with $\mathbf{X}$ variables and $\mathbf{F}$ constraints is a pair $(\mathbf{V}, \mathbf{E})$ where each constraint in $\mathbf{F}$ is a vertex $v_i \in \mathbf{V}$ and each variable in $\mathbf{X}$ is a hyperedge $e_j \in \mathbf{E}$ connecting all the constraints (vertices) in which it appears.

Generating heuristically good hypergraph separators can be done using a package called `hMeTiS` [10], which we used following [35]. The vertices of the hypergraph are partitioned into two balanced (roughly equal-sized) parts, denoted by $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$ respectively, while minimizing the number of hyperedges across. A small number of crossing edges translates into a small number of variables shared between the two sets of constraints. $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$ are then each recursively partitioned in the same fashion, until they contain a single vertex. The result of this process is a tree of hypergraph separators which can be shown to also be a pseudo tree of the original model where each separator corresponds to a subset of variables chained together [10,5].

Measures of Performance. We report the CPU time (in seconds) and the number of nodes visited. We also specify the number of variables (n), the number of constraints (c), as well as the induced width (w^*) and depth (h) of the pseudo trees obtained for each problem instances. The best performance points are highlighted. In each table, ”-” denotes that the respective algorithm exceeded the time limit. Similarly, ”out” stands for exceeding the 2GB memory limit.

7.1 Combinatorial Auctions

In **combinatorial auctions** (CA), an auctioneer has a set of goods, $M = \{1, 2, ..., m\}$ to sell and the buyers submit a set of bids, $\mathcal{B} = \{B_1, B_2, ..., B_n\}$. A bid is a tuple $B_j = (S_j, p_j)$, where $S_j \subseteq M$ is a set of goods and $p_j \geq 0$ is a price. The winner determination problem is to label the bids as winning or loosing so as to maximize the sum of the accepted bid prices under the constraint that each good is allocated to at most one bid. Combinatorial auctions are special cases of the classical set packing problem [1,36]. The problem can be formulated as a 0-1 ILP, as follows:

[10] Available at: http://www-users.cs.umn.edu/karypis/metis/hmetis

$$max \sum_{j=1}^{n} p_j x_j \quad (8)$$

$$\text{s.t.} \quad \sum_{j|i \in S_j} x_j \leq 1 \quad i \in \{1..m\}$$
$$x_j \in \{0,1\} \quad j \in \{1..n\}$$

Combinatorial auctions can also be formulated as binary Weighted CSPs [33], as described in [13]. Therefore, in addition to the 0-1 ILP solvers, we also ran `toolbar` which is a specialized OR Branch-and-Bound algorithm that maintains a level of local consistency called *existential directional arc-consistency* [37].

Figures 6 and 7 display the results for experiments with combinatorial auctions drawn from the *regions-upv* (Figure 6) and *arbitrary-upv* (Figure 7) distributions of CATS 2.0 test suite [38]. The *regions-upv* problem instances simulate the auction of radio spectrum in which a government sells the right to use specific segments of spectrum in different geographical areas. The *arbitrary-upv* problem instances simulate the auction of various electronic components. The suffix *upv* indicates that the bid prices were drawn from a *uniform* distribution. We looked at moderate size auctions having 100 goods and increasing number of bids. The number of bids is also the number of variables in the 0-1 ILP model. Each data point represents an average over 10 instances drawn uniformly at random from the respective distribution. The header of each plot in Figures 6 and 7 shows the average induced width and depth of the pseudo trees.

AND/OR vs. OR search. When comparing the AND/OR versus OR search regimes, we observe that both depth-first and best-first AND/OR search algorithms improve considerably over the OR search algorithm, `BB`, especially when the number of bids increases and the problem instances become more difficult. In particular, the depth-first and best-first AND/OR search algorithm using partial variable orderings, `AOBB+PVO-ILP` and `AOBF+PVO-ILP`, are the winners on this domain, among the `lp_solve` based solvers. For example, on the *regions-upv* auctions with 400 bids (Figure 6), `AOBF+PVO-ILP` is on average about 8 times faster than `BB`. Similarly, on the *arbitrary-upv* auctions with 280 bids (Figure 7), the difference in running time between `AOBB+PVO-ILP` and `BB` is about 1 order of magnitude. Notice that on the *regions-upv* dataset, `toolbar` is outperformed significantly by `BB` as well as the AND/OR algorithms. On the *arbitrary-upv* dataset, `toolbar` outperforms dramatically the `lp_solve` based solvers. However, the size of the search space explored by `toolbar` is significantly larger than the ones explored by the AND/OR algorithms. Therefore, `toolbar`'s better performance in this case can be explained by the far smaller computational overhead of the arc-consistency based heuristic used, compared with the LP relaxation based heuristic.

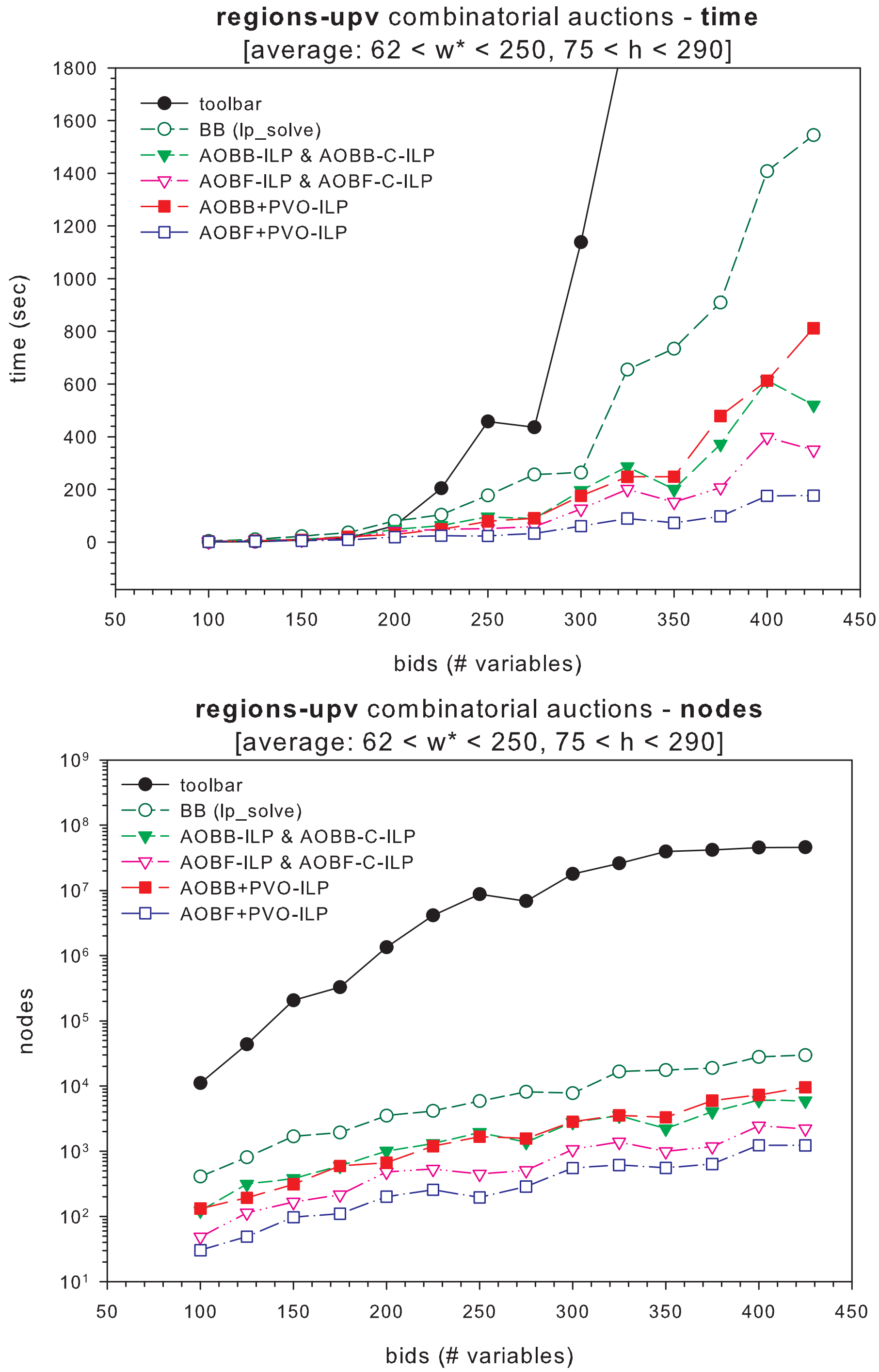


Fig. 6. Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***regions-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours. w^* is the average treewidth, h is the average depth of the pseudo tree.

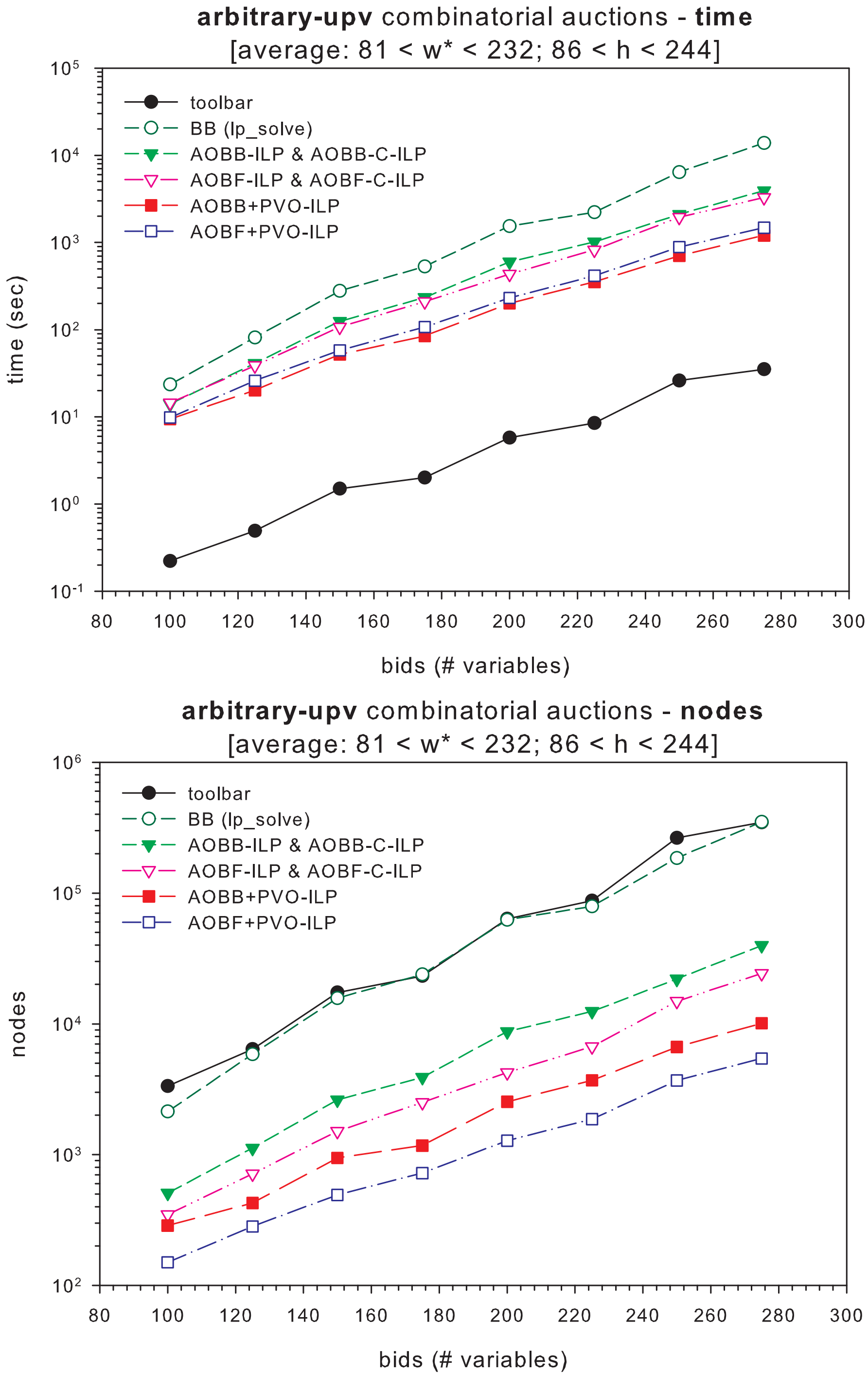


Fig. 7. Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***arbitrary-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours. w^* is the average treewidth, h is the average depth of the pseudo tree.

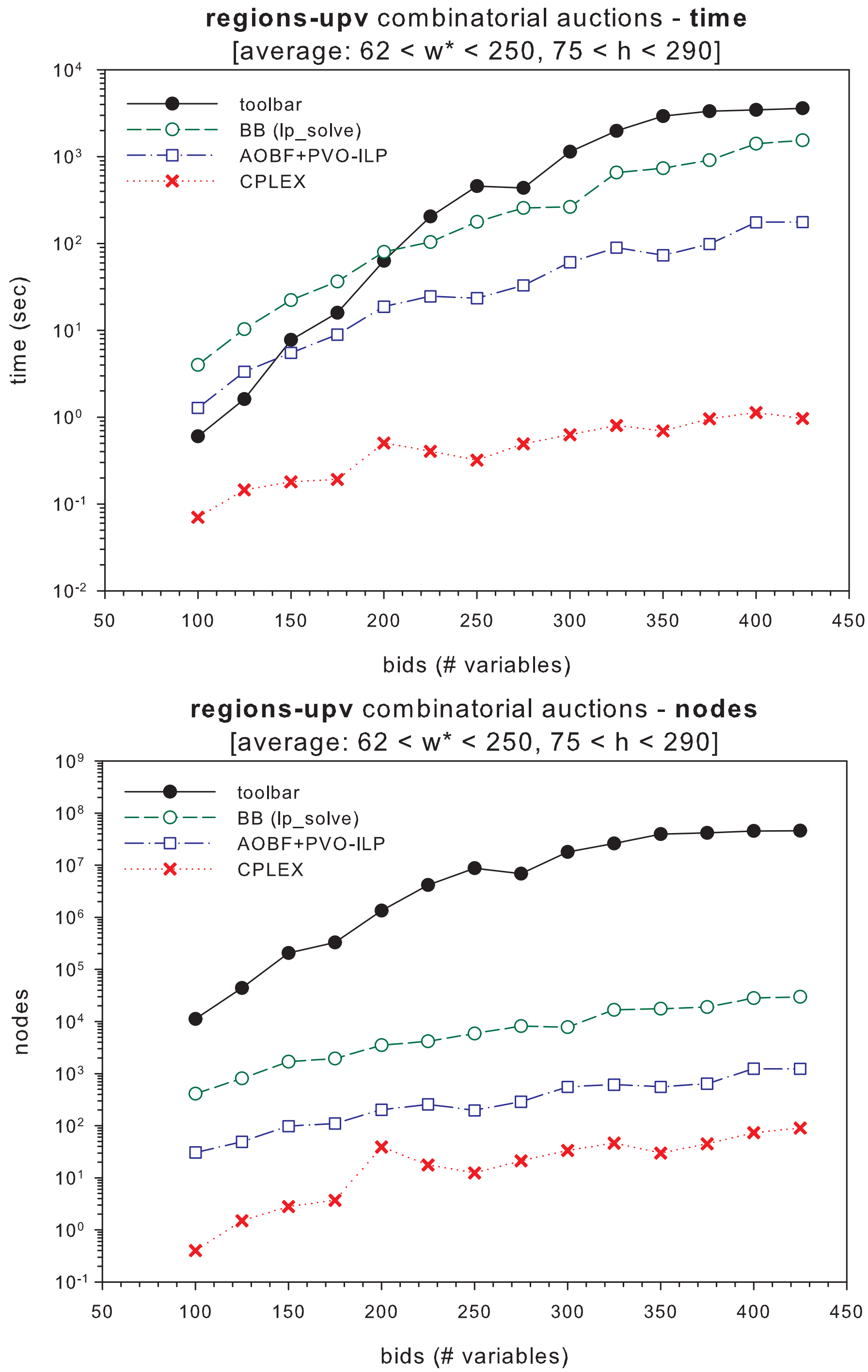


Fig. 8. Comparison with CPLEX. CPU time in seconds (top) and number of nodes (bottom) visited for solving combinatorial auctions from the ***regions-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

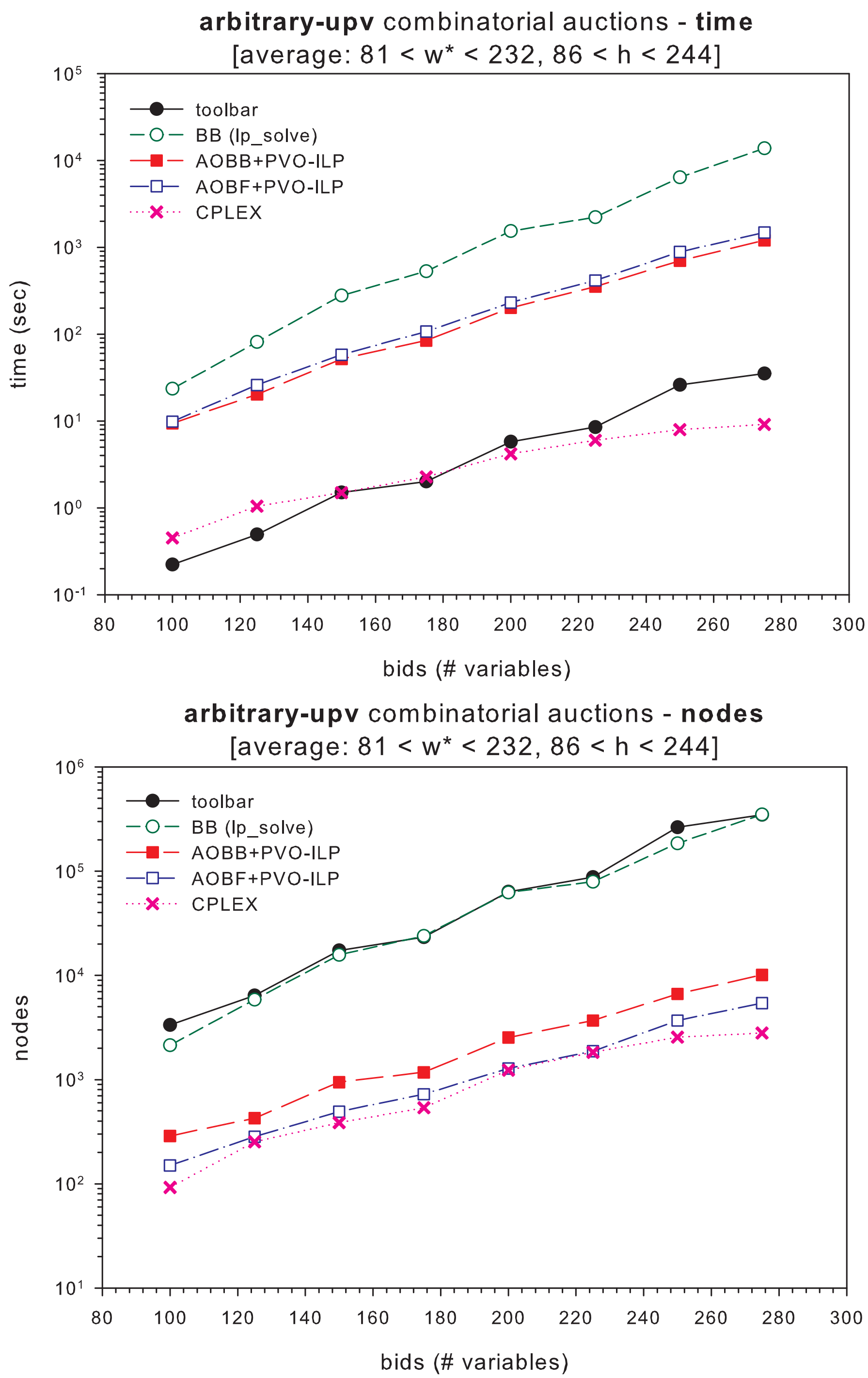


Fig. 9. Comparison with CPLEX. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***arbitrary-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

AOBB vs. AOBF. When comparing further best-first versus depth-first AND/OR search, we see that `AOBF-ILP` (resp. `AOBF+PVO-ILP`) improve considerably over `AOBB-ILP` (resp. `AOBB+PVO-ILP`), especially on the *regions-upv* dataset. The gain observed when moving from depth-first AND/OR Branch-and-Bound to best-first AND/OR search is primarily due to the optimal cost, which bounds the horizon of best-first more effectively than for depth-first search. Note that in this case `AOBF-ILP` (resp. `AOBF+PVO-ILP`) uses exponential space, unlike `AOBB-ILP` (resp. `AOBB+PVO-ILP`) which requires linear space only.

Impact of caching. When looking at the impact of caching on AND/OR search, we notice that the graph search algorithms `AOBB-C-ILP` and `AOBF-C-ILP` expanded the same number of nodes as the tree search algorithms `AOBB-ILP` and `AOBF-ILP`, respectively (see Figures 6 and 7). This indicates that, for this domain, the context minimal AND/OR search graph explored is a tree. Or, the LP relaxation is very accurate in this case and the AND/OR algorithms only explore a small part of the search space, for which the corresponding context-based cache entries are actually dead-caches.

Impact of dynamic variable orderings. We can see that using dynamic variable ordering heuristics improves the performance of best-first AND/OR search only. For depth-first AND/OR search, the performance deteriorated sometimes (see for example `AOBB-ILP` vs. `AOBB+PVO-ILP` on *regions-upv* auctions in Figure 6).

Comparison with CPLEX. In Figures 8 and 9 we contrast the results obtained with CPLEX, `toolbar`, `BB`, `AOBB+PVO-ILP` and `AOBF+PVO-ILP` on the *regions-upv* (Figure 8) and *arbitrary-upv* (Figure 9) distributions, respectively. Clearly, we can see that CPLEX is the best performing solver on these datasets. In particular, it is several orders of magnitude faster than the `lp_solve` based solvers, especially the baseline `BB` solver. Its excellent performance is leveraged by the powerful cutting planes engine as well as the proprietary variable ordering heuristic used. Note that on the *arbitrary-upv* dataset, `toolbar` is competitive with CPLEX only for relatively small number of bids.

We also experimented with combinatorial auctions derived from the *regions-npv* and *arbitrary-npv* distributions for which the bid prices were drawn from a *normal* distribution. The results displayed a similar pattern as those presented in this section and therefore we do not show them here. An extended version of the paper which contains these results is available online.

7.2 Uncapacitated Warehouse Location Problems

In the **uncapacitated warehouse location problem** (UWLP) a company considers opening m warehouses at some candidate locations in order to supply its n existing stores. The objective is to determine which warehouse to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized. Each store must be supplied by exactly one warehouse. The typical 0-1 ILP formulation of the problem is as follows:

$$min \sum_{j=1}^{n} \sum_{i=1}^{m} c_{ij} x_{ij} + \sum_{i=1}^{m} f_i y_i \tag{9}$$

$$\begin{array}{lll} \text{s.t.} & \sum_{i=1}^{m} x_{ij} = 1 & \forall j \in \{1..n\} \\ & x_{ij} \leq y_i & \forall j \in \{1..n\}, \forall i \in \{1..m\} \\ & x_{ij} \in \{0,1\} & \forall j \in \{1..n\}, \forall i \in \{1..m\} \\ & y_i \in \{0,1\} & \forall i \in \{1..m\} \end{array}$$

where f_i is the cost of opening a warehouse at location i and c_{ij} is the cost of supplying store j from the warehouse at location i.

Table 2 display the results obtained for 16 randomly generated UWLP instances [11] with 50 warehouses, 200 and 400 stores, respectively. The warehouse opening and store supply costs were chosen uniformly randomly between 0 and 1000. These are large problems with 10,050 variables and 10,500 constraints for the *uwlp-50-200* class, and 20,050 variables and 20,400 constraints for the *uwlp-50-400* class, respectively, having pseudo trees with induced widths of 50 and depths of 123.

AND/OR vs. OR search. When looking at AND/OR versus OR search, we can see that in almost all test cases the AND/OR algorithms dominate `BB`. On the *uwlp-50-200-013* instance, for example, `AOBF+PVO-ILP` has a speed-up of 186 over `BB`, and explores a search tree 1,142 times smaller. Similarly, on the *uwlp-50-400-001* instance, `AOBB+PVO-ILP` outperforms `BB` by almost 2 orders of magnitude in terms of running time and size of the search space explored. On this domain, the best performing algorithm among the `lp_solve` based solvers is best-first `AOBF+PVO-ILP`.

AOBB vs. AOBF. We observe only minor savings in running time in favor of best-first search. This can be explained by the already small enough search space traversed by the algorithms, which does not leave room for additional improvements due to the optimal cost bound exploited by best-first search.

Impact of caching. We see again that `AOBB-C-ILP` and `AOBF-C-ILP` visited the same number of nodes as `AOBB-ILP` and `AOBF-ILP`, respectively (see columns 3 and 5 in Table 2). This shows again that the context minimal AND/OR search graph explored by the `AOBB-C-ILP` and `AOBF-C-ILP` algorithms was a tree and therefore all cache entries were dead-caches.

[11] Problem generator from http://www.mpi-sb.mpg.de/units/ag1/projects/benchmarks/UflLib/

Table 2
CPU time in seconds and number of nodes visited for solving Uncapacitated Warehouse Location Problems with 50 warehouses 200 (top part) and 400 (bottom part) stores, respectively. No time limit. The best performance points among the lp_solve based solvers are shown in bold types, while the overall best performance points are boxed.

uwlp	BB (lp_solve)		AOBB-ILP		AOBB+PVO-ILP		AOBB-C-ILP	
	CPLEX		AOBF-ILP		AOBF+PVO-ILP		AOBF-C-ILP	
	time	nodes	time	nodes	time	nodes	time	nodes
50 warehouses 200 locations: (n=10,050, c=10,500), (w*=50, h=123)								
uwlp-50-200-004	61.08	142	46.39	46	17.47	10	46.42	46
	0.80	0	37.58	24	**15.49**	3	36.27	24
uwlp-50-200-005	1591.89	1,692	404.94	233	**125.81**	50	405.72	233
	9.91	81	287.64	97	145.53	37	270.99	97
uwlp-50-200-011	256.19	358	233.96	246	78.74	39	233.21	246
	7.97	37	88.22	41	**75.83**	22	83.75	41
uwlp-50-200-013	13693.76	14,846	116.19	44	78.86	24	116.25	44
	8.94	37	111.28	26	**74.53**	13	105.72	26
uwlp-50-200-017	711.04	998	123.14	118	18.17	9	124.70	118
	2.15	3	48.06	21	**16.84**	2	47.77	21
uwlp-50-200-018	1477.74	2,666	161.03	146	59.52	37	161.05	146
	5.74	8	54.58	21	**32.33**	8	52.41	21
uwlp-50-200-020	2179.39	3,668	190.77	138	68.91	36	190.81	138
	7.47	28	87.58	33	**48.33**	10	83.70	33
uwlp-50-200-021	3252.60	5,774	609.74	580	**37.63**	9	608.24	580
	6.66	25	80.55	30	46.80	7	92.08	30
50 warehouses 400 locations: (n=20,050, c=20,400), (w*=50, h=123)								
uwlp-50-400-001	13638.55	12,548	743.75	374	106.63	29	743.68	374
	10.76	12	130.03	20	**81.63**	8	126.39	20
uwlp-50-400-004	820.89	942	1114.47	794	55.10	10	1117.55	794
	6.52	6	126.97	25	**51.85**	3	123.19	25
uwlp-50-400-005	57532.67	32,626	2719.09	617	247.03	50	2722.26	617
	30.55	58	331.87	36	**131.58**	8	313.09	36
uwlp-50-400-006	365.93	632	48.41	11	**32.31**	1	48.44	11
	3.59	0	51.62	8	32.65	1	51.95	8
uwlp-50-400-008	599.49	560	175.60	49	96.66	21	175.67	49
	3.40	0	119.28	13	**60.27**	3	116.42	13
uwlp-50-400-009	17608.98	17,262	281.02	76	97.00	9	281.30	76
	9.02	6	132.27	14	**78.05**	2	128.58	14
uwlp-50-400-011	22727.61	22,324	193.91	77	**64.28**	5	193.89	77
	8.07	7	93.11	12	64.58	4	92.06	12
uwlp-50-400-012	5468.30	4,174	671.90	307	**52.22**	4	671.77	307
	4.49	0	164.64	32	52.95	2	159.28	32

Impact of dynamic variable orderings. We also observe that the dynamic variable ordering had a significant impact on performance in this case, especially for depth-first search. For example, on the *uwlp-50-200-021* instance, `AOBB+PVO-ILP` is 16 times faster than `AOBB-ILP` and expands 64 times fewer nodes. However, the difference in running time between the best-first search algorithms, `AOBF-ILP` and `AOBF+PVO-ILP`, is smaller compared to what we see for depth-first AND/OR search. This is because the search space explored by `AOBF-ILP` is already small enough and the savings in number of nodes caused by dynamic variable orderings cause only minor time savings.

Comparison with CPLEX. When looking at the results obtained with CPLEX (column 2 in Table 2), we notice again its excellent performance in terms of both running time and size of the search space explored. However, we see that in some cases `AOBF+PVO-ILP` actually explored fewer nodes than CPLEX ($e.g.$, *uwlp-50-200-021*). This is important because it shows that the relative worse performance of `AOBF+PVO-ILP` versus CPLEX is due mainly to lack of cutting planes as well as the naive dynamic variable ordering heuristic used.

7.3 MAX-SAT Instances

Given a set of Boolean variables the goal of **maximum satisfiability** (MAX-SAT) is to find a truth assignment to the variables that violates the least number of clauses. We experimented with problem classes *pret* and *dubois* from the SATLIB [12] library, which were previously shown to be difficult for 0-1 ILP solvers [32].

MAX-SAT can be formulated as a 0-1 ILP [39] or pseudo-Boolean formula [40,41]. In the 0-1 ILP model, a Boolean variable v is mapped to an integer variable x that takes value 1 when v is $True$ or 0 when it is $False$. Similarly, $\neg v$ is mapped to $1 - x$. With these mappings, a clause can be formulated as a linear inequality. For example, the clause $(v_1 \vee \neg v_2 \vee v_3)$ can be mapped to $x_1 + (1 - x_2) + x_3 \geq 1$. Here, the inequality means that the clause must be satisfied in order for the left side of the inequality to have a value no less than one.

However, a clause in a MAX-SAT may not be satisfied, so that the corresponding inequality may be violated. To address this issue, an auxiliary integer variable y is introduced to the left side of a mapped inequality. Variable $y = 1$ if the corresponding clause is unsatisfied, making the inequality valid; otherwise, $y = 0$. Since the objective is to minimize the number of violated clauses, it is equivalent to minimize the sum of the auxiliary variables that are forced to take value 1. For example, $(v_1 \vee \neg v_2 \vee v_3)$, $(v_2 \vee v_4)$ can be written as an 0-1 ILP of minimizing $z = y_1 + y_2$, subject to the constraints of $x_1 + (1 - x_2) + x_3 + y_1 \geq 1$ and $x_2 + (1 - x_4) + y_2 \geq 1$.

[12] http://www.satlib.org/

Table 3
CPU time in seconds and number of nodes visited for solving ***pret*** MAX-SAT instances. Time limit 10 hours. The best performance points among the lp_solve based solvers are shown in bold types, while the overall best performance points are boxed.

pret (w*, h)	BB (lp_solve) CPLEX time	nodes	MaxSolver time	toolbar PBS time	nodes	AOBB-ILP AOBF-ILP time	nodes	AOBB+PVO-ILP AOBF+PVO-ILP time	nodes	AOBB-C-ILP AOBF-C-ILP time	nodes
pret60-40	-	-	9.47	53.89	7,297,773	7.88	1,255	8.41	1,216	7.38	1,216
(6, 13)	676.94	3,926,422		**0.004**	565	7.56	1,202	8.70	1,326	**3.58**	568
pret60-60	-	-	9.48	53.66	7,297,773	8.56	1,259	8.70	1,247	7.30	1,140
(6, 13)	535.05	2,963,435		**0.004**	495	8.08	1,184	8.31	1,206	**3.56**	538
pret60-75	-	-	9.37	53.52	7,297,773	6.97	1,124	6.80	1,089	6.34	1,067
(6, 13)	402.53	2,005,738		**0.003**	543	7.38	1,145	8.42	1,149	**3.08**	506
pret150-40	-	-	-	-	-	95.11	6,625	108.84	7,152	75.19	5,625
(6, 15)	out			**0.02**	2,592	101.78	6,535	101.97	6,246	**19.70**	1,379
pret150-60	-	-	-	-	-	98.88	6,851	112.64	7,347	78.25	5,813
(6, 15)	out			**0.01**	2,873	106.36	6,723	102.28	6,375	**19.75**	1,393
pret150-75	-	-	-	-	-	108.14	7,311	115.16	7,452	84.97	6,114
(6, 15)	out			**0.02**	2,898	98.95	6,282	103.03	6,394	**20.95**	1,430

*7.3.1 **pret** Instances*

Table 3 shows the results for experiments with 6 *pret* instances. These are unsatisfiable instances of graph 2-coloring with parity constraints. The size of these problems is relatively small (60 variables with 160 clauses for *pret60* and 150 variables with 400 clauses for *pret150*, respectively).

AND/OR vs. OR search. We see again that the AND/OR algorithms improved dramatically over `BB`. For instance, on the *pret150-75* network, `AOBB-ILP` finds the optimal solution in less than 2 minutes, whereas `BB` exceeds the 10 hour time limit. Similarly, `MaxSolver` and `toolbar` could not solve the instance within the time limit. Overall, `PBS` offers the best performance on this dataset.

AOBB vs. AOBF. The best-first AND/OR search algorithms improve sometimes considerably over the depth-first ones, especially when exploring an AND/OR graph ($e.g.$, see `AOBF-C-ILP` versus `AOBB-C-ILP` in the leftmost column of Table 3). Moreover, the search space explored by `AOBF-C-ILP` appears to be the smallest. This indicates that the computational overhead of `AOBF-C-ILP` is mainly due to evaluating its guiding lower bounding heuristic evaluation function.

Impact of caching. When looking at the depth-first AND/OR Branch-and-Bound graph search algorithm we only observe minor improvements due to caching. This is probably because most of the cache entries were actually dead-caches. On the other hand, best-first `AOBF-C-ILP` exploits the relatively small size of the context-minimal AND/OR graph ($i.e.$, in this case the problem structure is captured by a very small context with size 6 and a shallow pseudo tree with depth 13 or 15) and achieves the best performance among the ILP solvers.

Impact of dynamic variable orderings. We also see that the dynamic variable ordering did not have an impact on search performance for both depth-first and best-first algorithms.

Comparison with CPLEX. Both depth-first and best-first AND/OR search algorithms outperformed dramatically CPLEX on this dataset. On the *pret60-40* instance, for example, `AOBF-C-ILP` is 2 orders of magnitude faster than CPLEX. Similarly, on *pret150-40*, CPLEX exceeded the memory limit.

*7.3.2 **dubois** Instances*

Figure 10 displays the results for experiments with random *dubois* instances with increasing number of variables. These are unsatisfiable 3-SAT instances with $3 \times degree$ variables and $8 \times degree$ clauses, each of them having 3 literals. As in the previous test case, the *dubois* instances have very small contexts of size 6 and shallow pseudo trees with depths ranging from 10 to 20.

AND/OR vs. OR search. As before, we see that the AND/OR algorithms are far superior to `BB`, which could not solve any of the test instances within the 3 hour time limit. `PBS` is again the overall best performing algorithm, however it failed to solve 4 test instances: on instance *dubois130*, for which $degree = 130$, it exceeded the 3 hour time limit, whereas on instances *dubois180*, *dubois200* and *dubois260* the clause/pseudo-boolean constraint learning mechanism caused the solver to run out of memory. We note that `MaxSolver` and `toolbar` were not able to solve any of the test instances within the time limit.

AOBB vs. AOBF. Best-first search outperforms again depth-first search, especially when exploring the AND/OR graph. However, the depth-first tree search algorithms `AOBB-ILP` and `AOBB+PVO-ILP` were better than the best-first tree search counterparts in this case. This was probably caused by the internal dynamic variable ordering used by `AOBB-ILP` and `AOBB+PVO-ILP` to solve independent subproblems rooted at the AND nodes in the search tree.

Impact of caching. We can see that `AOBF-C-ILP` takes full advantage of the relatively small context minimal AND/OR search graph and, on some of the larger instances, it outperforms its ILP competitors with up to one order of magnitude in terms of both running time and number of nodes expanded. On this dataset as well `AOBF-C-ILP` explores the smallest search space, but its computational overhead does not pay off in terms of running time when compared with `PBS`. The impact of caching on AND/OR Branch-and-Bound is not that pronounced as for best-first.

Impact of dynamic variable orderings. The dynamic variable ordering had a minor impact on depth-first AND/OR search only ($e.g.$, see `AOBB+PVO-ILP` versus

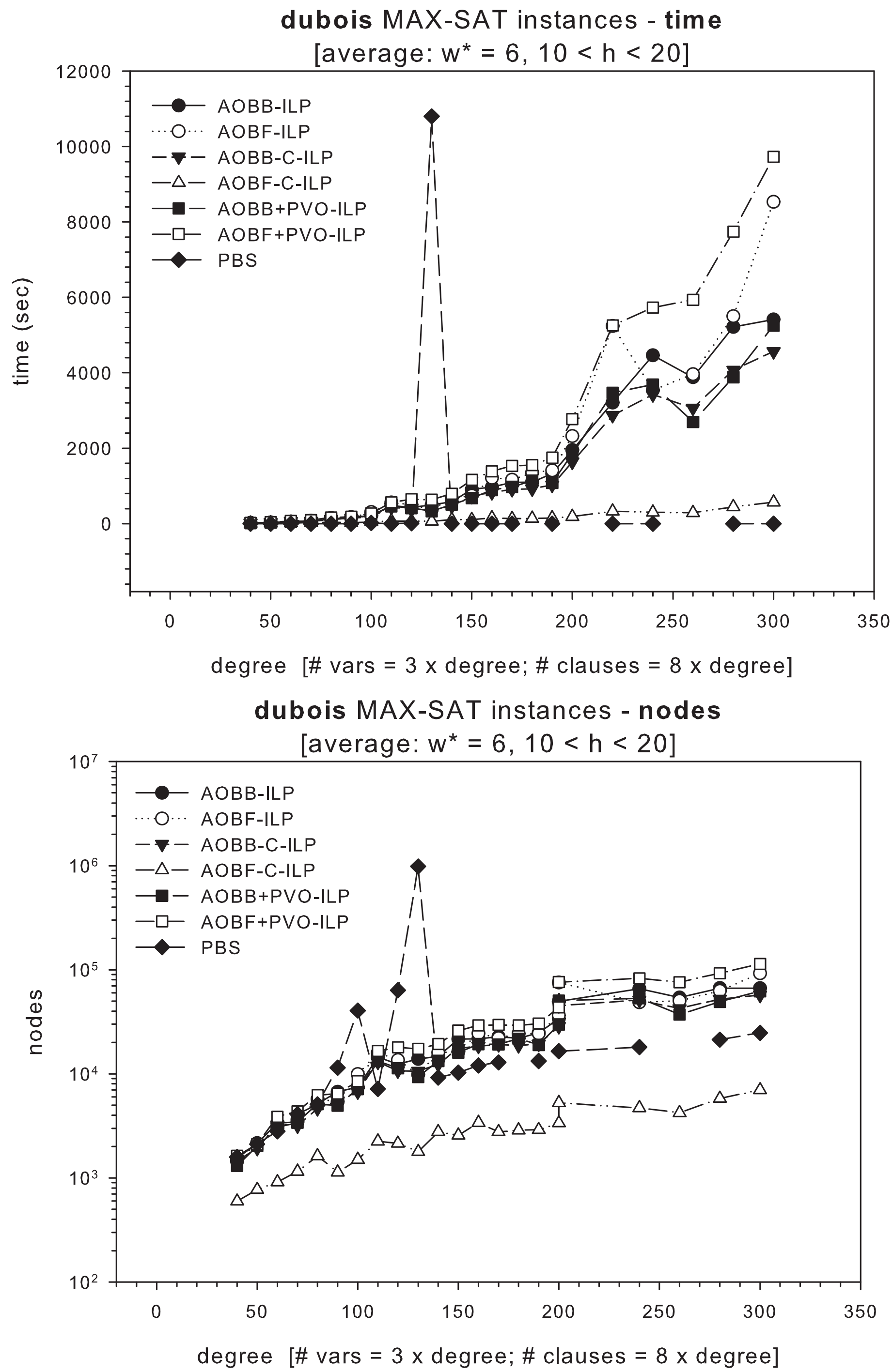


Fig. 10. Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving ***dubois*** MAX-SAT instances. Time limit 3 hours. CPLEX, `BB`, `toolbar` and `MaxSolver` were not able to solve any of the test instances within the time limit.

`AOBB-ILP` in Figure 10).

Comparison with CPLEX. The performance of CPLEX was quite poor on this dataset and it could not solve any of the test instances within the time limit.

8 Related Work

The idea of exploiting structural properties of the problem in order to enhance the performance of search algorithms is not new. Freuder and Quinn [24] introduced the concept of pseudo tree arrangement of a constraint graph as a way of capturing independencies between subsets of variables. Subsequently, *pseudo tree search* is conducted over a pseudo tree arrangement of the problem which allows the detection of independent subproblems that are solved separately. More recently, [42] extended pseudo tree search to optimization tasks in order to boost the Russian Doll search [43] for solving Weighted CSPs. Our depth-first AND/OR Branch-and-Bound and best-first AND/OR search algorithms for 0-1 ILPs are also related to the Branch-and-Bound method proposed by [27] for acyclic AND/OR graphs and game trees, as well as the depth-first and best-first AND/OR search algorithms for general constraint optimization over graphical models introduced in [4,6,10,8,9,5,7].

Dechter's graph-based back-jumping algorithm [44] uses a depth-first (DFS) spanning tree to extract knowledge about dependencies in the graph. The notion of DFS-based search was also used by [45] for a distributed constraint satisfaction algorithm. Bayardo and Miranker [25] reformulated the pseudo tree search algorithm in terms of back-jumping and showed that the depth of a pseudo tree arrangement is always within a logarithmic factor off the induced width of the graph.

In probabilistic reasoning, *Recursive Conditioning* (RC) [35] and *Value Elimination* (VE) [46] are search methods for likelihood and counting computations that can be viewed as exploring graph-based AND/OR search spaces.

In optimization, *Backtracking with Tree-Decomposition* (BTD) [47] is a memory intensive method for solving constraint optimization problems which combines search techniques with the notion of tree decomposition. This mixed approach can in fact be viewed as searching an AND/OR search space whose backbone pseudo tree is defined by and structured along the tree decomposition [3].

9 Summary and Conclusion

The paper investigates the impact of the AND/OR search spaces perspective to solving optimization problems from the class of 0-1 Integer Linear Programs. In earlier papers [4,6,8,9,5,7] we showed that the AND/OR search paradigm can improve general constraint optimization algorithms. Here, we demonstrate empirically the benefit of AND/OR search to 0-1 Integer Linear Programs.

Specifically, we extended and evaluated depth-first and best-first AND/OR search algorithm traversing the AND/OR search tree or graph for solving 0-1 ILPs. We also augmented the algorithms with dynamic variable ordering strategies. Our empirical evaluation demonstrated that the AND/OR search principle can improve 0-1 integer programming schemes sometimes by several orders of magnitude. We summarize next the most important factors influencing performance, including dynamic variable orderings, caching, as well as the search control strategy ($e.g.$, depth-first versus the best-first).

- **Depth-first versus best-first search.** Our results showed conclusively that the AND/OR search algorithms using a best-first control strategy and traversing either an AND/OR search tree or graph were able, in many cases, to improve considerably over the depth-first search ones ($e.g.$, combinatorial auctions from Figure 6, *dubois* MAX-SAT instances from Figure 10).
- **Impact of caching.** For problems with relatively small induced width and therefore small context, best-first AND/OR search was shown to outperform dramatically the traditional tree search algorithms ($e.g.$, *dubois* MAX-SAT instances from Figure 10). The impact of caching on the depth-first AND/OR Branch-and-Bound search algorithms was less prominent ($e.g.$, *pret* and *dubois* MAX-SAT instances from Table 3 and Figure 10, respectively) probably because most of the cache entries were dead-caches. Also, for problems with very large contexts ($e.g.$, combinatorial auctions from Figure 6, UWLP instances from Table 2) the context minimal AND/OR graph explored was a tree, and therefore caching had no impact.
- **Impact of dynamic variable orderings.** As with general AND/OR search where we showed that dynamic variable ordering schemes are powerful [10,5], for 0-1 ILPs too the AND/OR Branch-and-Bound with partial variable orderings sometimes outperformed the AND/OR Brach-and-Bound restricted to a static variable ordering by one order of magnitude ($e.g.$, UWLP instances from Table 2). Similarly, best-first AND/OR search with partial variable orderings for 0-1 ILP improved considerably over its counterpart using a static ordering ($e.g.$, combinatorial auctions from Figure 6).
- **AND/OR solvers versus CPLEX.** Our current implementation of the depth-first and best-first AND/OR search is far from being fully optimized with respect to commercial 0-1 ILP solvers such as CPLEX, as it relies on an open source implementation of the *simplex* algorithm, as well as a naive dynamic variable ordering heuristic. Nevertheless, we demonstrated that on selected classes of 0-1 ILPs the AND/OR algorithms outperformed CPLEX in terms of both the number of nodes explored ($e.g.$, UWLP instances from Table 2) and CPU time ($e.g.$, `pret` MAX-SAT instances from Table 3).

Our depth-first and best-first AND/OR search can be extended to accommodate all known enhancement schemes. In particular, it can be modified to incorporate *cutting planes* to tighten the linear relaxation of the current subproblem. The space required by the best-first AND/OR search can be enormous, because all nodes generated have to be saved. To remedy this problem, the algorithm can be equiped with a memory bounding scheme [23,48,49]. Our AND/OR search approach can be easily extended to accommodate mixed integer linear programs for which a subset of the decision variables are required to have integer values at the optimal solution.

Acknowledgments

This work was partially supported by the NSF grants IIS-0086529 and IIS-0412854, the MURI ONR award N00014-00-1-0617, the NIH grant R01-HG004175-02, the Marie Curie Transfer of Knowledge grant MTKD-CT-2006-042563 and by an IRC-SET Embark post-doctoral fellowship.

References

[1] G. Nemhauser and L. Wolsey. *Integer and combinatorial optimization.* Wiley, 1988.

[2] E. Lawler and D. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[3] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

[4] R. Marinescu and R. Dechter. AND/OR branch-and-bound for graphical models. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 224–229, 2005.

[5] R. Marinescu and R. Dechter. AND/OR branch-and-bound search for combinatorial optimization in graphical models. Technical report, University of California, Irvine, 2008.

[6] R. Marinescu and R. Dechter. Memory intensive branch-and-bound search for graphical models. In *National Conference on Artificial Intelligence (AAAI)*, 2006.

[7] R. Marinescu and R. Dechter. Memory intensive AND/OR search for combinatorial optimization in graphical models. Technical report, University of California, Irvine, 2008.

[8] R. Marinescu and R. Dechter. Best-first AND/OR search for graphical models. In *National Conference on Artificial Intelligence (AAAI)*, pages 1171–1176, 2007.

[9] R. Marinescu and R. Dechter. Best-first AND/OR search for most probable explanations. In *International Conference on Uncertainty in Artificial Intelligence (UAI)*, 2007.

[10] R. Marinescu and R. Dechter. Dynamic orderings for AND/OR branch-and-bound search in graphical models. In *European Conference on Artificial Intelligence (ECAI)*, pages 138–142, 2006.

[11] R. Marinescu and R. Dechter. AND/OR branch-and-bound search for pure 0/1 integer linear programming problems. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization (CPAIOR)*, pages 152–166, 2006.

[12] R. Marinescu and R. Dechter. Best-first AND/OR search for 0/1 integer programming. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization (CPAIOR)*, 2007.

[13] Rina Dechter. *Constraint Processing*. MIT Press, 2003.

[14] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations, Plenum Press, NY*, pages 85–103, 1972.

[15] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, 1951.

[16] L.G. Khachian. A polynomial algorithm in linear programming (in Russian. *Doklady Akad. Nauk SSSR*, 244(5):1093–1096, 1979.

[17] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[18] N. Nilsson P. Hart and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 8(2):100–107, 1968.

[19] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[20] J. Wolsey. *Integer Programming*. John Wiley & Sons, 1998.

[21] M. Padberg and G. Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch-and-cut. *Operations Research Letters*, 6(1):1–7, 1987.

[22] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[23] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga, 1980.

[24] E. Freuder and M. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1076–1078, 1985.

[25] R. Bayardo and D. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 558–562, 1995.

[26] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving.* Addison-Welsey, 1984.

[27] L. Kanal and V. Kumar. *Search in artificial intelligence.* Springer-Verlag., 1988.

[28] Laurence Wosley. *Integer Programming*. John Wiley & Sons, 1998.

[29] V. Chvatal D. Applegate, R. Bixby and W. Cook. Finding cuts in the TSP (a preliminary report). *Technical Report 95-05, DIMACS, Rutgers University*, 1995.

[30] M. Benichou, J.M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer programming. *Math. Programming*, 1(1):76–94, 1971.

[31] Z. Xing and W. Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Principles and Practice of Constraint Programming (CP)*, pages 660–705, 2004.

[32] S. de Givry, J. Larrosa, and T. Schiex. Solving Max-SAT as weighted CSP. In *Principles and Practice of Constraint Programming (CP)*, 2003.

[33] S. Bistarelli, U. Montanari, and F. Rossi. Semiring based constraint solving and optimization. *Journal of the ACM*, 44(2):309–315, 1997.

[34] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo-boolean solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[35] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.

[36] P. Cramton, Y. Shoham, and R. Steinberg. *Combinatorial Auctions*. MIT Press, 2006.

[37] S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *International Joint Conference in Artificial Intelligence (IJCAI)*, 2005.

[38] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM Electronic Commerce*, pages 66–76, 2000.

[39] S. Joy, J. Mitchell, and B. Borchers. A branch and cut algorithm for Max-SAT and weighted Max-SAT. In *Satisfiability Problem: Theory and Applications*, pages 519–536, 1997.

[40] J.P. Walser. *Integer Optimization Local Search*. Springer, 1999.

[41] H. Dixon and M.L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *National Conference on Artificial Intelligence (AAAI)*, pages 635–640, 2006.

[42] J. Larrosa, P. Meseguer, and M. Sanchez. Pseudo-tree search with soft constraints. In *European Conference on Artificial Intelligence (ECAI)*, pages 131–135, 2002.

[43] M. Lemaitre G. Verfaillie and T. Schiex. Russian doll search for solving constraint optimization problems. In *National Conference on Artificial Intelligence (AAAI)*, pages 298–304, 1996.

[44] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[45] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 318–324, 1991.

[46] F. Bacchus, S. Dalmao, and T. Pittasi. Value Elimination: Bayesian inference via backtracking search. In *Uncertainty in Artificial Intelligence (UAI)*, pages 20–28, 2003.

[47] P. Jegou and C. Terrioux. Decomposition and good recording for solving Max-CSPs. In *European Conference on Artificial Intelligence (ECAI)*, pages 196–200, 2004.

[48] P. Chakrabati, S. Ghose, A. Acharya, and S. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41(3):197–221, 1989.

[49] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.