

# AND/OR Branch-and-Bound Search for Pure 0/1 Integer Linear Programming Problems

Radu Marinescu and Rina Dechter

School of Information and Computer Science  
University of California, Irvine, CA 92697-3425  
{radum,dechter}@ics.uci.edu

**Abstract.** *AND/OR search spaces* have recently been introduced as a unifying paradigm for advanced algorithmic schemes for graphical models. The main virtue of this representation is its sensitivity to the structure of the model, which can translate into exponential time savings for search algorithms. In this paper we extend the recently introduced AND/OR Branch-and-Bound algorithm (AOBB) [1] for solving pure 0/1 Integer Linear Programs [2]. Since the variable selection can have a dramatic impact on search performance, we introduce a new dynamic AND/OR Branch-and-Bound algorithm able to accommodate variable ordering heuristics. The effectiveness of the dynamic AND/OR approach is demonstrated on a variety of benchmarks for pure 0/1 integer programming, including instances from the MIPLIB library, real-world combinatorial auctions and random uncapacitated warehouse location problems.

## 1 Introduction

A *constraint optimization problem* is the minimization/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 constraint optimization problems are the Integer Linear Programming problems (ILP) [2] where the objective is to optimize a linear function of integer-valued variables, subject to a set of linear equality or inequality constraints defined on subsets of variables. The classical approach to solving ILPs is the *branch-and-bound* method [3] which maintains the best solution found so far, while discarding partial solutions which cannot improve on the best.

The AND/OR search space for graphical models [4] is a newly introduced framework for search that is sensitive to the independencies in the model, often resulting in exponentially reduced complexities. It is based on a pseudo-tree that captures 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 algorithm (AOBB) is a new search method that explores the AND/OR search tree for solving optimization tasks in graphical models [1]. In this paper we present an extension of the algorithm for solving optimization problems from the class of pure 0/1 Integer Linear Programs [2]. A pure 0/1 integer linear program is a linear program where all the decision variables are restricted to be either 0 or 1 at the optimal solution.

Since variable selection can have a dramatic impact on search performance [2], we introduce a *dynamic* AND/OR Branch-and-Bound search algorithm that is able to accommodate variable ordering heuristics. There are two orthogonal approaches to incorporating dynamic orderings into AOBB. AND/OR Branch-and-Bound with Partial Variable Ordering (AOBB+PVO) improves AOBB by applying an ordering heuristic whenever the partial order dictated by the decomposition principle allows. The other, AND/OR Branch-and-Bound with Dynamic Variable Ordering (AOBB+DVO), gives priority to the variable ordering heuristic and applies problem decomposition as a secondary principle. We demonstrate empirically the practical efficiency of the dynamic AND/OR Branch-and-Bound approach on several benchmarks for pure 0/1 integer linear programming problems, including test instances from the MIPLIB library, combinatorial auctions simulating radio spectrum allocation and random uncapacitated warehouse location problems.

## 2 Background

### 2.1 Constraint Optimization Problems

A finite *Constraint Optimization Problem* (COP) is a four-tuple  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, z \rangle$ , where  $\mathcal{X} = \{X_1, \dots, X_n\}$  is a set of variables,  $\mathcal{D} = \{D_1, \dots, D_n\}$  is a set of finite domains,  $\mathcal{C} = \{C_1, \dots, C_m\}$  is a set of constraints on the variables and  $z$  is a global cost function (i.e. objective function) to be optimized. The scope of a constraint  $C_i$ , denoted  $\text{scope}(C_i) \subseteq \mathcal{X}$ , is the set of arguments of  $C_i$ . Constraints can be expressed *extensionally*, through relations, or *intentionally*, by a mathematical formula (equality or inequality). 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. The *induced graph* of  $G$  relative to an ordering  $d$  of its variables, denoted  $G^*(d)$ , is obtained by processing the nodes in reverse order of  $d$ . For each node all its earlier neighbors are connected, including neighbors connected by previously added edges. 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* of a graph, denoted  $w^*(d)$ , is the maximum width of nodes in the induced graph.

### 2.2 Integer Linear Programming

A *Linear Program* (LP) consists of a set of continuous variables and a set of linear constraints (equalities or inequalities). The goal is to optimize a global linear cost function subject to the constraints. One of the standard forms of a linear program is:

$$\min\{c^\top x \mid Ax \leq b, x \geq 0\} \quad (1)$$

where  $c \in \mathbb{R}^n$ ,  $b \in \mathbb{R}^m$ ,  $A \in \mathbb{R}^{m \times n}$  and  $x \in \mathbb{R}^n$ . Here  $c$  represents the cost vector and  $x$  is the vector of decision variables. The vector  $b$  and the matrix  $A$  define the  $m$  linear constraints. Linear programs are usually solved by Dantzig's *simplex* method [5].

A *Mixed Integer Linear Programming* (MILP) problem is a linear program where some of the decision variables are constrained to have only integer values at the optimal solution. An important special case is a decision variable  $x_i$  that is integer with  $0 \leq x_i \leq 1$ . This forces  $x_i$  to be either 0 or 1 at the solution. Variables like  $x_i$  are called *0/1* or *binary integer variables*. Subsequently, a MILP problem with binary integer variables is also called a *0/1 Mixed Integer Linear Programming* problem. A *pure 0/1 Integer Linear Programming* problem is a MILP where all the decision variables are binary. Pure 0/1 ILPs can formulate many practical problems such as capital budgeting [6], cargo loading [7], processor allocation in distributed systems [8] or combinatorial auctions [9, 10].

Clearly, any pure 0/1 integer linear program can be viewed as a finite COP instance  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, z \rangle$  with linear constraints and a linear objective function. In the remaining of the paper we will consider a *minimization* problem defined by  $z = \sum_{i=1}^n c_i X_i$  subject to  $m$  linear constraints  $\mathcal{C} = \{C_1, \dots, C_m\}$ , over  $n$  binary decision variables  $\mathcal{X} = \{X_1, \dots, X_n\}$ .

### 2.3 Branch-and-Bound Search for Constraint Optimization

*Branch-and-Bound* (BB) is a general *search* method for solving constraint optimization problems [3]. It traverses the search tree defined by the problem, where internal nodes represent partial assignments and leaf nodes denote complete ones, which may or may not be optimal. During the traversal, which is usually *depth first*, BB maintains an *upper bound*  $ub$ , the cost of the best solution found so far. At each internal node the algorithm computes a *lower bound*  $lb$  on the optimal extension of the current partial assignment. When  $lb \geq ub$ , the current best cost cannot be improved and the algorithm *backtracks* pruning the subtree below the current node. Otherwise, the algorithm moves forward and tries to instantiate the next variable in the ordering. In the context of pure 0/1 integer linear programs, the lower bound of a subproblem is obtained by solving its linear relaxation (i.e. relaxing the integrality restrictions). In this case the branching process can fail at a particular node for one of the following reasons: (i) the LP solution can be integer; or (ii) the LP problem can be infeasible; or (iii) the lower bound exceeds the upper bound (for more details see [2, 3]).

## 3 AND/OR Search Spaces

The classical way to do search is to instantiate variables one at a time, following a static/dynamic variable ordering. In the simplest case, this process defines a search tree (called here OR search tree), whose nodes represent states in the space of partial assignments. The traditional search space does not capture independencies that appear in the structure of the underlying graphical model. Introducing AND states into the search space can capture the structure, decomposing the problem into independent subproblems by conditioning on values [11, 4]. The AND/OR search space is defined using a backbone *pseudo-tree*.

**Definition 1 (pseudo-tree).** *Given an undirected graph  $G = (V, E)$ , a directed rooted tree  $T = (V, E')$  defined on all its nodes is called pseudo-tree if any arc of  $G$  which is not included in  $E'$  is a back-arc, namely it connects a node to an ancestor in  $T$ .*

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

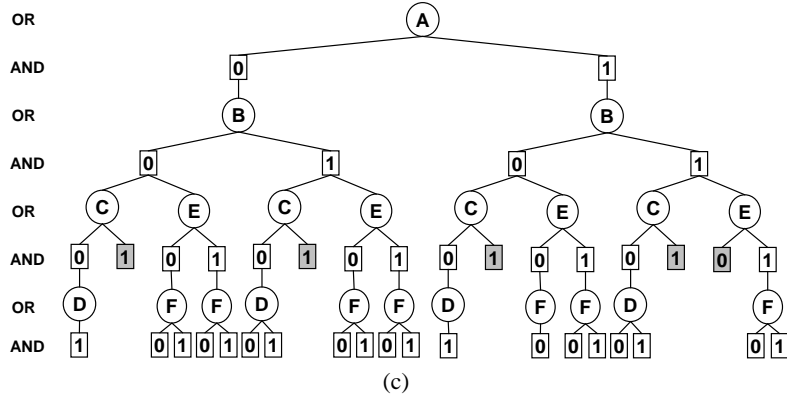
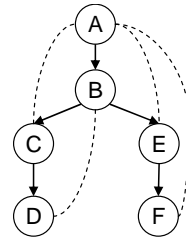


Fig. 1. The AND/OR search space.

### 3.1 AND/OR Search Trees

Given a COP instance  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, z \rangle$ , its constraint graph  $G$  and a pseudo-tree  $T$  of  $G$ , the associated AND/OR search tree  $S_T$  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  $\langle X_i, x_i \rangle$  and correspond to value assignments in the domains of the variables. The structure of the AND/OR tree is based on the underlying pseudo-tree  $T$  of  $G$ . The root of the AND/OR search tree is an OR node, labeled with the root of  $T$ .

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,  $path(x_i) = (\langle X_1, x_1 \rangle, \dots, \langle X_{i-1}, x_{i-1} \rangle)$ . The children of an AND node  $\langle X_i, x_i \rangle$  are OR nodes labeled with the children of variable  $X_i$  in  $T$ . In other words, the OR states represent alternative ways of solving the problem, whereas the AND states represent problem decomposition into independent sub-problems, all of which need to be solved. When the pseudo-tree is a chain, the AND/OR search tree coincides with the regular OR search tree.

A *solution subtree*  $Sol_{S_T}$  of  $S_T$  is an AND/OR subtree such that: (i) it contains the root of  $S_T$ ; (ii) if a nonterminal AND node  $n \in S_T$  is in  $Sol_{S_T}$  then all of its children are in  $Sol_{S_T}$ ; (iii) if a nonterminal OR node  $n \in S_T$  is in  $Sol_{S_T}$  then exactly one of its children is in  $Sol_{S_T}$ .

*Example 1.* For illustration consider the pure 0/1 integer program with 6 decision variables  $A, B, C, D, E, F$  and 4 linear constraints  $C_1(A, B, C), C_2(B, C, D), C_3(A, B, E),$

$C_4(A, E, F)$  from Figure 1(a). The objective function to be minimized is  $z = 7A+B-2C+5D-6E+8F$ . The pseudo-tree arrangement of the constraint graph, together with the back-arcs (dotted lines) are given in Figure 1(b). Figure 1(c) shows the corresponding AND/OR search tree (for AND nodes we only denote the value, namely  $\langle A, 0 \rangle$  is written as  $\boxed{0}$  child of  $A$ ). The shaded nodes represent dead-ends (i.e. inconsistent values).

The AND/OR search tree can be traversed by a depth-first search algorithm that is guaranteed to have a time complexity exponential in the depth of the pseudo-tree and can operate in linear space. The arcs from  $X_i$  to  $\langle X_i, x_i \rangle$  are annotated by appropriate *labels* of the objective function. The nodes in  $S_T$  can be associated with *values*, defined over the subtrees they root.

**Definition 2 (label).** Given a COP instance with objective function  $z = \sum_{i=1}^n c_i X_i$  and a corresponding AND/OR search tree  $S_T$ , the label  $l(X_i, x_i)$  of the arc from the OR node  $X_i$  to the AND node  $\langle X_i, x_i \rangle$  is defined as  $l(X_i, x_i) = c_i \cdot x_i$ .

**Definition 3 (value).** The value  $v(n)$  of a node  $n \in S_T$  is defined recursively as follows: (i) if  $n = \langle X_i, x_i \rangle$  is a terminal AND node then  $v(n) = l(X_i, x_i)$ ; (ii) if  $n = \langle X_i, x_i \rangle$  is an internal AND node then  $v(n) = l(X_i, x_i) + \sum_{n' \in \text{succ}(n)} v(n')$ ; (iii) if  $n = X_i$  is an internal OR node then  $v(n) = \min_{n' \in \text{succ}(n)} v(n')$ , where  $\text{succ}(n)$  are the children of  $n$  in  $S_T$ .

Clearly, the value of each node can be computed recursively, from leaves to root.

**Proposition 1.** Given an AND/OR search tree  $S_T$  of a COP instance  $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, z)$ , the value  $v(n)$  of a node  $n \in S_T$  is the minimal cost solution to the subproblem rooted at  $n$ , subject to the current variable instantiation along the path from root to  $n$ . If  $n$  is the root of  $S_T$ , then  $v(n)$  is the minimal cost solution to  $\mathcal{P}$ .

Therefore, we can traverse the AND/OR search tree in a depth-first manner to compute the value of the root. This approach would require linear space, storing only the current partial solution subtree. The algorithm expands alternating levels of OR and AND nodes, periodically evaluating the values of the nodes along the current path. It terminates when the root node is evaluated with the optimal cost.

**Theorem 1 (complexity).** The complexity of an algorithm that traverses an AND/OR search tree in a depth-first manner is linear space and time is  $O(n \cdot \exp(h))$ , where  $h$  is the depth of the pseudo-tree associated with the constraint graph. When the constraint graph has induced width  $w$ , the algorithm can be bounded by  $O(n \cdot \exp(w \cdot \log(n)))$ .

### 3.2 Pseudo-Trees Based on Recursive Hypergraph Decomposition

The performance of the AND/OR tree search algorithms is influenced by the quality of the pseudo-tree. Finding the minimal depth pseudo-tree is a hard problem [11, 12]. In this section we describe a heuristic for generating a low depth balanced pseudo-tree, based on the recursive decomposition of a hypergraph.

**Definition 4 (hypergraph).** Given a COP instance  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, z \rangle$ , its hypergraph  $\mathcal{H} = (V, E)$  has a vertex  $v_i \in V$  for each constraint in  $\mathcal{C}$  and each variable in  $\mathcal{X}$  is an edge  $e_j \in E$  connecting all the constraints in which it appears.

**Definition 5 (hypergraph separators).** Given a hypergraph  $\mathcal{H} = (V, E)$ , a hypergraph separator decomposition is a triple  $(\mathcal{H}, \mathcal{S}, \mathcal{R})$  where: (i)  $\mathcal{S} \subset E$ , and the removal of  $\mathcal{S}$  separates  $\mathcal{H}$  into  $k$  disconnected components (subgraphs)  $\mathcal{H}_1, \dots, \mathcal{H}_k$ ; (ii)  $\mathcal{R}$  is a relation over the size of the disjoint subgraphs (i.e. balance factor).

It is well known that the problem of generating optimal hypergraph partitions is hard. However heuristic approaches were developed over the years. A good approach is packaged in `hMeTiS`<sup>1</sup>. We will use this software as a basis for our pseudo-tree generation. This idea and software were also used by [13] to generate low width decomposition trees. Generating a pseudo-tree using `hMeTiS` is fairly straightforward. 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 is also a pseudo-tree of the original model since each separator corresponds to a subset of variables chained together.

## 4 AND/OR Branch-and-Bound Search

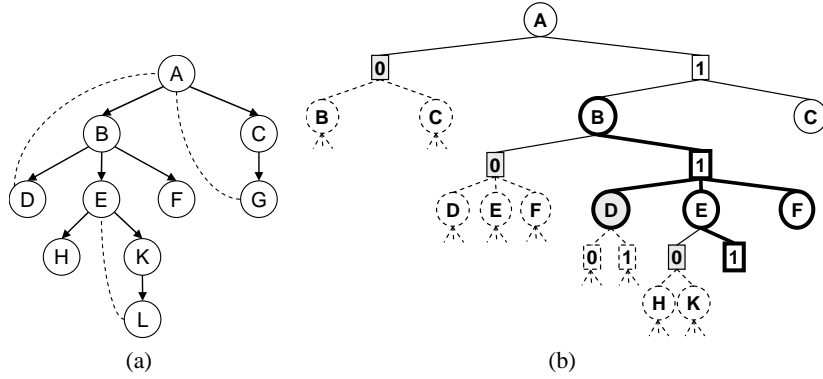
AND/OR Branch-and-Bound (AOBB) was recently proposed by [1] as a depth-first Branch-and-Bound that explores an AND/OR search tree for solving optimization tasks in graphical models. We first overview the static version of the algorithm (i.e. restricted to a static variable ordering induced by a pseudo-tree). Then, we introduce a new extension of the algorithm that traverses a dynamic AND/OR search tree, thus being able to use variable ordering heuristics.

### 4.1 Lower Bounds on Partial Trees

At any stage during search, a node  $n$  along the current path roots a current *partial solution subtree*, denoted by  $G_{sol}(n)$ , to the corresponding subproblem. By the nature of the search process,  $G_{sol}(n)$  must be connected, must contain its root  $n$  and will have a *frontier* containing all those nodes that were generated but not yet expanded. The leaves of  $G_{sol}(n)$  are called *tip* nodes. Furthermore, we assume that there exists a *static* heuristic evaluation function  $h(n)$  underestimating  $v(n)$  that can be computed efficiently when node  $n$  is first generated.

Given the current partially explored AND/OR search tree  $S_T$ , the *active path*  $\mathcal{AP}(t)$  is the path of assignments from the root of  $S_T$  to the current tip node  $t$ . The *inside context*  $in(\mathcal{AP})$  of  $\mathcal{AP}(t)$  contains all nodes that were fully evaluated and are children of nodes on  $\mathcal{AP}(t)$ . The *outside context*  $out(\mathcal{AP})$  of  $\mathcal{AP}(t)$ , contains all the frontier

<sup>1</sup> <http://www-users.cs.umn.edu/~karypis/metis/hmetis>



**Fig. 2.** A partially explored AND/OR search tree.

nodes that are children of the nodes on  $\mathcal{AP}(t)$ . The *active partial subtree*  $\mathcal{APT}(n)$  rooted at a node  $n \in \mathcal{AP}(t)$  is the subtree of  $G_{sol}(n)$  containing the nodes on  $\mathcal{AP}(t)$  between  $n$  and  $t$  together with their OR children. We can define now a *dynamic heuristic evaluation function* of a node  $n$  relative to  $\mathcal{APT}(n)$ , as follows.

**Definition 6 (dynamic heuristic evaluation function).** *Given an active partial tree  $\mathcal{APT}(n)$ , the dynamic heuristic evaluation function of  $n$ ,  $f_h(n)$ , is defined recursively as follows: (i) if  $\mathcal{APT}(n)$  consists only of a single node  $n$ , and if  $n \in in(\mathcal{AP})$  then  $f_h(n) = v(n)$  else  $f_h(n) = h(n)$ ; (ii) if  $n = \langle X_i, x_i \rangle$  is an AND node, having OR children  $m_1, \dots, m_k$  then  $f_h(n) = \max(h(n), l(X_i, x_i) + \sum_{i=1}^k f_h(m_i))$ ; (iii) if  $n = X_i$  is an OR node, having an AND child  $m$ , then  $f_h(n) = \max(h(n), f_h(m))$ .*

We can show that:

**Theorem 2.** (1)  $f_h(n)$  is a lower bound on the minimal cost solution to the subproblem rooted at  $n$ , namely  $f_h(n) \leq v(n)$ ; (2)  $f_h(n) \geq h(n)$ , namely the dynamic heuristic function is tighter than the static one.

*Example 2.* For illustration consider the pseudo-tree in Figure 2(a) and the partially explored AND/OR search tree in Figure 2(b). The active path has tip node  $\langle E, 1 \rangle$  and represents the partial assignment  $A = 1, B = 1, E = 1$ . The shaded nodes at the left of the active path belong to the inside context (their corresponding subtrees have already been explored). The outside context includes the nodes  $\{C, F\}$ , which are also in the search frontier. For the active partial subtree rooted at  $B$  (highlighted), the lower bound  $f_h(B)$  on  $v(B)$  is computed recursively as follows:  $f_h(B) = \max(h(B), f_h(\langle B, 1 \rangle))$ , where  $f_h(\langle B, 1 \rangle) = \max(h(\langle B, 1 \rangle), l(B, 1) + v(D) + f_h(E) + h(F))$ . Similarly,  $f_h(E) = \max(h(E), f_h(\langle E, 1 \rangle)) = \max(h(E), h(\langle E, 1 \rangle))$ , since  $f_h(\langle E, 1 \rangle) = h(\langle E, 1 \rangle)$ .

## 4.2 Static AND/OR Branch-and-Bound

A search algorithm traversing the AND/OR search space can calculate a *lower bound* on  $v(n)$  of a node  $n$  on the active path, by using  $f_h(n)$ . It can also compute an *upper*

```

function: AOBB(prev, vo, T,  $\mathcal{X}$ ,  $\mathcal{D}$ ,  $\mathcal{C}$ , z)
1 if  $\mathcal{X} = \emptyset$  then return 0;
2 else
3    $X_i \leftarrow \text{selectNextVar}(vo, prev, T, \mathcal{X})$ ;
4    $v(X_i) \leftarrow \infty$ ;
5   foreach  $a \in D_i$  do
6      $psol \leftarrow psol \cup (X_i, a)$ ;
7     foreach  $k = 1..q$  do
8        $lb_k \leftarrow \text{LB}(\mathcal{X}_k, \mathcal{D}_k, \mathcal{C}_k)$ ;
9        $\text{outside}(psol) \leftarrow \text{outside}(psol) \cup lb_k$ ;
10    end
11     $lb \leftarrow c_i a + \sum_{k=1}^q lb_k$ ;
12    if  $\text{findAncestorCut}(X_i, a, lb) = \text{false}$  then
13       $v(X_i, a) \leftarrow 0$ ;
14      foreach  $k = 1..q$  do
15         $v(X_i, a) += \text{AOBB}(X_i, vo, T, \mathcal{X}_k, \mathcal{D}_k, \mathcal{C}_k, z_k)$ ;
16         $\text{inside}(psol) \leftarrow \text{inside}(psol) \cup v(X_i, a)$ ;
17      end
18       $v(X_i) \leftarrow \min(v(X_i), v(X_i, a) + \text{label}(i, a))$ ;
19    end
20  end
21   $psol \leftarrow psol \cup \text{argmin}_{a \in D_i}(v(X_i))$ ;
22  return  $v(X_i)$ ;
23 end

```

**Fig. 3.** AND/OR Branch-and-Bound.

bound on  $v(n)$ , based on the portion of the search space below  $n$  that has already been explored. The upper bound  $ub(n)$  on  $v(n)$  is the current minimal cost solution subtree rooted at  $n$ . If for some node  $n$  along the active path  $f_n(n) \geq ub(n)$  then we are guaranteed that the current partial solution subtree cannot be extended to a better solution, and the current search path can be safely abandoned (for more details see [1]).

Figure 3 shows AOBB, which implements the depth-first AND/OR Branch-and-Bound search. AOBB assumes the *global* linear objective function  $z = \sum_{i=1}^n c_i X_i$ . The following notation is used:  $(\mathcal{X}, \mathcal{D}, \mathcal{C})$  is the problem with which the procedure is called,  $T$  is a pseudo-tree arrangement of the underlying constraint graph and  $prev$  denotes the variable instantiated in the previous step of the algorithm (initially  $prev = \text{NULL}$ ). The current partial solution subtree being explored is represented by  $psol$ . The contexts of the active path are denoted by  $\text{inside}(psol)$  and  $\text{outside}(psol)$ , respectively. The algorithm is restricted to a static variable ordering along a depth-first traversal of  $T$ . This is indicated by the input parameter  $vo = \text{SVO}$ .

If the set  $\mathcal{X}$  is empty, then the result is trivially computed (line 1). Else, AOBB selects a variable  $X_i$  (i.e. expands the OR node  $X_i$ ) and iterates over its values (lines 3-5) to compute the OR value  $v(X_i)$ . Each value  $a$  defines the current subproblem  $P = (X_i = a, \mathcal{X}, \mathcal{D}, \mathcal{C})$  that is decomposed into a set of  $q$  independent subproblems  $P_k = (\mathcal{X}_k, \mathcal{D}_k, \mathcal{C}_k, z_k)$ , with  $k = 1..q, q > 0$ , one per child  $X_k$  of  $X_i$  in the pseudo-

```

function: selectNextVar( $vo, prev, T, \mathcal{X}$ )
1 switch  $vo$  do
2   case SVO
3     if  $prev = NULL$  then  $next \leftarrow \text{getPseudoTreeRoot}(T)$ ;
4     else  $next \leftarrow \text{getPseudoTreeChild}(prev, T)$ 
5   case PVO
6      $candidates \leftarrow \text{getPseudoTreeSeparator}(prev, T)$ ;
7      $next \leftarrow \text{selectBestCandidate}(candidates)$ ;
8   case DVO
9      $next \leftarrow \text{selectBestCandidate}(\mathcal{X})$ ;
10 end
11 return  $next$ ;

```

**Fig. 4.** Variable selection procedure.

tree  $T$ . Each subproblem  $P_k$  is defined by the subset of variables  $\mathcal{X}_k$  corresponding to the descendants of  $X_k$  in  $T$  including  $X_k$ , the subset of constraints and constraint projections  $\mathcal{C}_k$  involving the variables in  $\mathcal{X}_k$ , subject to the current instantiation along the active path, and a *local* objective function denoted by  $z_k = \sum_{X_j \in \mathcal{X}_k} c_j X_j$ , which corresponds to the projection of  $z$  on the variables in  $\mathcal{X}_k$ . For each  $P_k$ , a lower bound  $lb_k$  of  $z_k$  is computed (line 8). The outside context of the active path is updated in line 9. The lower bound of  $P$  is  $lb$  (line 11), computed as the sum of independent lower bounds including the projection on  $X_i$  and value  $a$  of the objective function (i.e. the label of the corresponding AND node  $\langle X_i, a \rangle$ ).

Upon instantiating  $X_i$  with value  $a$  (i.e. expanding the AND node  $\langle X_i, a \rangle$ ) (line 12), AOBB successively updates the *lower bound function*  $f_h(m)$  for every ancestor node  $m$  along the active path (procedure `findAncestorCut` implements Definition 6). If the updated lower bound of an ancestor exceeds its current upper bound (i.e.  $f_h(m) \geq ub(m)$ ), then the algorithm backtracks and tries the next value in the domain of  $X_i$ . As the algorithm recursively solves independent subproblems (line 14) the AND value  $v(X_i, a)$  accumulates the results (line 15). The inside context of the active path is also updated with the actual solution of the current subproblem (line 16). Once all subproblems are solved, the OR value  $v(X_i)$  is also updated (line 18). After trying all feasible values of variable  $X_i$ , the cost of the optimal solution to the problem rooted by  $X_i$  remains in  $v(X_i)$ , which is returned (line 22).

In the context of pure 0/1 integer linear programming problems, the lower bound  $lb_k$  for each subproblem  $P_k$  is obtained by solving its linear relaxation (i.e. relaxing the integrality restrictions). If the respective linear program is infeasible, then  $lb_k$  is set to  $\infty$ . Moreover, if the linear relaxation of  $P_k$  has an integer solution, then there is no need to search the subtree below  $k$ . In this case,  $v(k) = lb_k$  and this is the value returned by the algorithm for  $P_k$ . For illustration, consider again the pure 0/1 integer program from Figure 1(a). Let  $A = 0$  and  $B = 0$  be the current partial assignment of the active path in the AND/OR search tree from Figure 1(c). The subproblem  $P_C$  rooted at node  $C$  in the search tree corresponds to minimizing the local objective function  $z_C = -2C + 5D$ ,

subject to the constraints and constraint projections involving variables  $C$  and  $D$  only (i.e.  $C \leq 3$  and  $5C - 3D \leq -2$ , respectively).

### 4.3 Dynamic AND/OR Branch-and-Bound

It is well known that the variable ordering can dramatically influence search performance [2, 14]. In this section we go beyond static orderings and introduce two new extensions of AOBB that are able to accommodate dynamic variable ordering heuristics.

**Partial Variable Ordering (PVO)** The pseudo-tree arrangement of a constraint graph can be viewed as a tree of *separators* which decompose the graph. For example, variables  $A$  and  $B$  chained together in the pseudo-tree from Figure 1(b) constitute a separator. Specifically, once  $A$  and  $B$  are instantiated (and therefore removed from the graph), the graph breaks into two separate components  $(C, D)$  and  $(E, F)$ , respectively.

Clearly, the variables belonging to a separator can be instantiated in an arbitrary order. Therefore, dynamic variable ordering can be incorporated into AOBB by letting the separators to be instantiated dynamically, according to a specific variable ordering heuristic. We will refer to this extension of AOBB as AND/OR Branch-and-Bound with Partial Variable Ordering (AOBB+PVO). In the context of integer linear programs, the traditional variable ordering heuristic is to choose the next variable to instantiate as the fractional variable with the best score resulted from solving the linear relaxation of the current subproblem.

AOBB+PVO is similar to its precursor AOBB described in Figure 3 in the sense that it is also guided by a precomputed pseudo-tree. The partial variable ordering strategy, indicated by the input parameter  $vo = PVO$ , is implemented by the `selectNextVar` procedure from Figure 4. AOBB+PVO selects the next variable as follows. Let  $prev$  be the variable instantiated in the previous step of the algorithm. The next variable  $X_i$  is chosen as the variable with the best score from the list of candidates represented by the uninstantiated variables of the current separator (i.e. the separator to which variable  $prev$  belongs to). The heuristic used for computing variables scores is implemented by the `selectBestCandidate` procedure.

**Dynamic Variable Ordering (DVO)** The partial variable ordering described in the previous section aims at enhancing a problem decomposition strategy with variable ordering heuristics. An orthogonal approach would be to improve a given variable ordering heuristic by applying problem decomposition as a secondary principle during search. Specifically, AND/OR Branch-and-Bound with Dynamic Variable Ordering (AOBB+DVO) instantiates variables according to a given ordering heuristic while constantly updating the problem graph structure. Whenever the graph breaks into disconnected components, their corresponded subproblems are solved separately. It is easy to see that in this case a variable may have the best heuristic to tighten the search space, yet, it may not yield a good decomposition for the remaining of the problem, in which case the algorithm would explore primarily an OR space.

AOBB+DVO is also based on the pseudo-code presented in Figure 3, where  $vo = DVO$ . The algorithm selects the next variable to instantiate as the one with the best

miplib	n	(w*,h)	BB		AOBB					
			time	nodes	SVO		PVO		DVO	
					time	nodes	time	nodes	time	nodes
p0033	33	(18, 20)	6.53	18,081	0.59	1,893	<b>0.39</b>	1,099	3.39	9,251
p0201	201	(120, 142)	37.41	15,575	57.88	25,284	<b>22.90</b>	8,988	42.46	14,463
lseu	89	(53, 69)	153.90	368,573	39.74	87,537	<b>38.94</b>	86,073	152.55	336,953

**Table 1.** Results for MIPLIB problem instances.

score from the uninstantiated variables of the current subproblem. After variable  $X_i$  is selected AOBB+DVO removes  $X_i$  from the graph and if the problem is decomposable, the independent components are solved in an AND/OR manner (lines 14-17).

## 5 Experiments

In this section we evaluate empirically the performance of the AND/OR Branch-and-Bound algorithms on several benchmarks for pure 0/1 integer linear programming including problem instances from the MIPLIB library<sup>2</sup>, combinatorial auctions and uncapacitated warehouse location problems. All our experiments were done on a 2.4GHz Pentium IV with 2GB of RAM, running Windows XP. Our C++ implementation of the AND/OR algorithms was based on the open source `lp_solve` library<sup>3</sup>.

We consider three classes of depth-first AND/OR Branch-and-Bound (AOBB) algorithms, each using a specific variable ordering strategy, as follows. As previously described, the class AOBB+SVO is restricted to a static variable ordering obtained from a depth-first traversal of the pseudo-tree. Similarly, the class AOBB+PVO is also restricted to a static pseudo-tree, except that the graph separators are instantiated dynamically using a variable ordering heuristic. The class AOBB+DVO gives priority to dynamic variable ordering and applies problem decomposition as a secondary principle. For comparison, we include results obtained with `lp_solve`'s classic OR depth-first Branch-and-Bound (BB). All competing algorithms used a variable ordering heuristic based on *reduced costs* [2]. Specifically, the next fractional variable to instantiate has the smallest reduced cost (ties are broken lexicographically). Furthermore, we did not attempt any simplification (or presolving) of the test instances used.

We report the average effort, as CPU time (in seconds) and number of nodes visited (which is equivalent to the number of time the `Simplex` routine was called to solve the linear relaxation of the current subproblem), required for proving optimality of the solution. We also record the number of variables (n), the depth of the pseudo-trees (h) and the induced width of the graphs (w\*) obtained for the test instances. As AOBB+SVO and AOBB+PVO algorithms use a non-deterministic algorithm for generating the pseudo-tree, the running time may vary significantly from one run to the next. We therefore ran these algorithms 5 times on each benchmark and provide an average of those runs. The best performance points are highlighted in all test cases.

<sup>2</sup> available at <http://miplib.zib.de/miplib2003.php>

<sup>3</sup> `lp_solve` 5.5.0.6 is available at [http://groups.yahoo.com/group/lp\\_solve/](http://groups.yahoo.com/group/lp_solve/)

auction	(w*,h)	BB			AOBB							
		time	nodes	wins	SVO		PVO		DVO			
		time	nodes	wins	time	nodes	wins	time	nodes	wins	time	nodes
reg-upv-b200g50	(145, 162)	<b>1.71</b>	602	4	3.28	938	0	2.98	888	6	1.97	602
reg-upv-b250g75	(166, 190)	16.27	3,472	0	7.32	1,209	3	<b>6.30</b>	1,110	7	18.36	3,472
reg-upv-b300g100	(173, 204)	63.29	7,997	2	52.75	4,855	4	<b>45.61</b>	4,801	4	69.18	7,997
reg-npv-b200g50	(140, 161)	1.27	443	1	1.78	514	1	<b>1.15</b>	302	8	1.45	443
reg-npv-b250g75	(160, 187)	<b>5.53</b>	1,150	2	5.97	1,085	4	5.96	1,144	4	6.24	1,150
reg-npv-b300g100	(172, 206)	58.61	7,342	1	21.54	1,904	4	<b>16.35</b>	1,748	5	63.74	7,342

**Table 2.** Results for combinatorial auction problem instances.

## 5.1 MIPLIB Library

MIPLIB is a library of mixed integer linear programming instances that is commonly used for benchmarking integer programming algorithms. For our purpose we selected 3 pure 0/1 integer instances of increasing difficulty. Table 1 reports a summary of the experiment. We see immediately that, overall, AOBB+PVO was the best performing algorithm, both in terms of CPU time and number of nodes visited. AOBB+DVO does indeed explore a smaller search space than BB in all test cases, but due to its computational overhead these savings do not reflect in the running time.

## 5.2 Combinatorial Auctions

In **combinatorial auctions** (CA), an auctioneer has a set of goods,  $M = \{1, 2, \dots, m\}$  to sell and the buyers submit a set of bids,  $\mathcal{B} = \{B_1, B_2, \dots, B_n\}$ . A bid is a tuple  $B_j = \langle S_j, p_j \rangle$ , 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 losing so as to maximize the sum of the accepted bid prices under the constraint that each good is allocated to at most one bid. We used the following pure 0/1 integer formulation of the problem:

$$\begin{aligned}
 & \max \sum_{j=1}^n p_j x_j & (2) \\
 & \text{s.t. } \sum_{j|i \in S_j} x_j \leq 1 \quad i \in \{1..m\} \\
 & \quad x_j \in \{0, 1\} \quad j \in \{1..n\}
 \end{aligned}$$

Table 2 shows results for experiments with combinatorial auctions drawn from the regions distribution of the CATS 2.0 test suite [10]. The suffixes npv and upv indicate that the bid prices were drawn from either a normal or uniform distribution. These 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 (for more details see [10]). We looked at moderate size auctions by varying the number of bids between 200 and 300, and the number of goods between 50 and 100. The number of bids is also the number of variables in the ILP model. For each value combination of bids and goods we drawn randomly 10 auctions from the respective distribution. For each algorithm we also report the number of wins out of the 10 runs. These instances

uwlp	(w*,h)	BB		AOBB					
		time	nodes	SVO		PVO		DVO	
				time	nodes	time	nodes	time	nodes
uwlp50-200-a	(50, 123)	6.27	27	15.72	70	<b>6.28</b>	12	7.23	27
uwlp50-200-b	(50, 123)	11.34	53	17.22	60	<b>5.78</b>	12	11.75	53
uwlp50-200-c	(50, 123)	73.66	469	15.78	58	<b>5.83</b>	10	77.94	469
uwlp50-200-d	(50, 123)	836.52	4,309	27.94	116	<b>11.97</b>	26	904.15	4,309
uwlp50-200-e	(50, 123)	2501.75	11,973	32.69	80	<b>16.98</b>	28	2990.19	12,733
uwlp50-200-f	(50, 123)	43.36	237	18.70	64	<b>8.03</b>	20	45.99	237
uwlp50-200-g	(50, 123)	1328.40	6,905	27.89	84	<b>8.53</b>	20	1515.48	7,265
uwlp50-200-h	(50, 123)	76.88	331	25.20	84	<b>13.70</b>	30	88.38	331
uwlp50-200-i	(50, 123)	224.33	1,003	46.06	194	<b>17.17</b>	50	367.14	1,533
uwlp50-200-j	(50, 123)	7737.65	31,003	28.03	64	<b>9.13</b>	10	9276.98	33,415

**Table 3.** Results for 10 uncapacitated warehouse location problem instances.

are highly connected with induced widths over 150. For this problem class AOBB+PVO outperforms its competitors, exploring the smallest search space. If we look for example at the 300 bid problem instances from the `reg-npv` distribution, AOBB+PVO is about 4 times faster than BB, exploring a search space 4 times smaller. Notice that AOBB+DVO explores the same number of nodes as BB, showing that in this case the dynamic variable ordering does not generate decomposable subproblems.

### 5.3 Uncapacitated Warehouse Location Problem

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 integer formulation of the problem is as follows.

$$\min \sum_{j=1}^n \sum_{i=1}^m c_{ij} x_{ij} + \sum_{i=1}^m f_i y_i \quad (3)$$

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

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 3 displays the results obtained on 10 randomly generated UWLP problem instances<sup>4</sup> with 50 warehouses and 200 stores. 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. The variable ordering heuristic that

<sup>4</sup> Problem generator from <http://www.mpi-sb.mpg.de/units/ag1/projects/benchmarks/UflLib/>

worked best in this case selects the next fractional variable whose value is closest to 0.5 (ties are broken lexicographically). We can see that AOBB+PVO dominates in all test cases, outperforming the classic BB with several orders of magnitude in terms of both running time and size of the search space explored. In `uwlp50-200-e` for example, one of the hardest instances, AOBB+PVO causes a speed-up of 147 over the classic OR Branch-and-Bound algorithm, exploring a search tree 428 times smaller. This is due to the problem's structure partially captured by a shallow pseudo-tree with depth 123. AOBB+DVO has a similar performance as BB on all test instances (it is slower than BB due to its computational overhead), indicating that these problems do not break into disconnected components when dynamic variable ordering has higher priority than problem decomposition.

## 6 Conclusion

In this paper we extended the AND/OR Branch-and-Bound search algorithm for solving pure 0/1 integer linear programming problems. The contribution of the paper is two-fold. First, we restricted the algorithm to a static variable ordering induced by a pseudo-tree of the constraint graph. Since the order in which variables are selected for instantiation can influence dramatically the search performance, we then proposed a dynamic version of the AND/OR Branch-and-Bound able to accommodate variable ordering heuristics. We looked at two orthogonal approaches to incorporating dynamic orderings into AOBB. On one hand, AOBB+PVO augments a static pseudo-tree based problem decomposition with variable ordering heuristics, by allowing the graph separators to be instantiated dynamically. On the other hand, AOBB+DVO gives priority to the variable ordering heuristic while constantly updating the graph structure and solving separately, in an AND/OR manner, disconnected components that may be discovered during search. Our empirical evaluation demonstrated on a variety of benchmark problems for pure 0/1 integer linear programming that AOBB+PVO is a promising candidate solver, outperforming the classic BB with several orders of magnitude in terms of both running time and size of the search space explored.

Our dynamic AND/OR approach leaves room for future improvements, which are likely to make it more effective in practice. For instance, it can be modified to explore the search tree in a *best-first* manner, rather than depth-first. This is desirable in the sense that no optimal tree search algorithm can guarantee expanding fewer nodes [15]. We also mention that the *Branch-and-Cut*, a more modern algorithm that generates *cutting planes* to tighten the LP relaxation of the current subproblem, can be adapted to traverse an AND/OR search tree. Finally, the AND/OR algorithms can be easily adapted for solving *mixed* 0/1 ILPs, where only a subset of the decision variables is restricted to integer values. In that case, the AND/OR search space is based on a *partial* pseudo-tree which spans only the integer variables.

**Related Work** AOBB is related to the Branch-and-Bound method proposed by [16] for acyclic AND/OR graphs and game trees, as well as the pseudo-tree search algorithm proposed in [17] for boosting Russian Doll search. The optimization method developed

in [18] for semi-ring CSPs can also be interpreted as an AND/OR graph search algorithm. Problem decomposition based on hypergraph separators was also explored by [19] and [20] for solving large real-world SAT problem instances.

## References

1. R. Marinescu and R. Dechter. And/or branch-and-bound for graphical models. *In IJCAI'05*.
2. G. Nemhauser and L. Wolsey. *Integer and combinatorial optimization*. Wiley, 1988.
3. E. Lawler and D. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.
4. R. Dechter and R. Mateescu. And/or search spaces for graphical models. *In AIJ*, 2006.
5. G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, 1951.
6. M. Vasquez and J. Hao. A hybrid approach for the 0/1 multidimensional knapsack approach. *Proceedings of IJCAI'01*, 2001.
7. W. Shih. A branch-and-bound method for the multiconstraint 0/1 knapsack problem. *Journal of the Operational Research Society*, 30:369–378, 1979.
8. B. Gavish and H. Pirkul. Allocation of data bases and processors in a distributed computing system. *Management of Distributed Data Processing*, 31:215–231, 1982.
9. T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. *Proc. of IJCAI'99*, pages 542–547, 1999.
10. K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. *ACM EC*, 2000.
11. E. Freuder and M. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. *Proc. of IJCAI'85*, 1985.
12. R. Bayardo and D. Miranker. On the space-time trade-off in solving constraint satisfaction problems. *Proc. of IJCAI'95*, 1995.
13. A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.
14. Rina Dechter. *Constraint Processing*. MIT Press, 2003.
15. J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Welsey, 1984.
16. L. Kanal and V. Kumar. *Search in artificial intelligence*. Springer-Verlag., 1988.
17. J. Larrosa, P. Meseguer, and M. Sanchez. Pseudo-tree search with soft constraints. *Proc. of ECAI'02*, 2002.
18. P. Jegou and C. Terrioux. Decomposition and good recording for solving max-csps. *In ECAI'04*.
19. J. Huang and A. Darwiche. A structure-based variable ordering heuristic. *Proceedings of IJCAI'03*, 2003.
20. W. Li and P. van Beek. Guiding real-world sat solving with dynamic hypergraph separator decomposition. *Proceedings of ICTAI'04*, 2004.