AND/OR Tree Search for Constraint Optimization

Radu Marinescu and Rina Dechter

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

Abstract. The paper presents and evaluates the power of a new framework for constraint optimization, based on the concept of AND/OR search trees. The virtue of the AND/OR search tree representation is that its size may be smaller than that of a traditional OR search tree. We introduce a new generation of depth first Branch-and-Bound algorithms that traverse an AND/OR search space and use the Mini-Bucket approximation scheme to generate heuristics to guide the search. Our preliminary experimental work shows that the new approach is competitive and in many cases superior to state of the art systematic search algorithms that explore the regular OR space.

1 Introduction

Constraint Satisfaction Problems (CSPs) provide a formalism for formulating many interesting real world problems as an assignment of values to variables, subject to a set of constraints. In Constraint Optimization Problems (COPs), some constraints (called *soft*) are cost functions indicating preferences. The task of interest is to find a complete assignment satisfying all hard constraints and minimizing the global cost. Solving constraint problems is NP-hard. Therefore, general algorithms are likely to require exponential time in the worst case.

Most complete algorithms for solving constraint problems typically fall within one of the following two categories: *search* and *dynamic programming*. Search algorithms transform a problem into a set of subproblems by selecting a variable and considering the assignment of each of its domain values. The subproblems are solved in sequence applying recursively the same transformation rule (often referred to as *conditioning*). These algorithms have a time complexity which is exponential in the number of variables, but can operate in polynomial space. Dynamic programming algorithms solve a problem by a sequence of transformations that reduce the problem size, while preserving the solution space of the problem. The time and space complexity of these methods is exponential in a topological parameter called *width* (always less than or equal to the number of variables). Due to their high space requirements, when the width is large, the latter methods are often impractical.

In this paper we focus on search. We adopt a new perspective of AND/OR search space [17] that allows exploiting the problem structure by search algorithms and in particular by depth first Branch and Bound (BnB) algorithms. The structure of a problem can dramatically influence the performance of a search algorithm. Whereas topological properties of the problem cannot be incorporated into regular OR search trees, we

recently showed that they are naturally captured by an AND/OR search tree [9, 10]. Within the AND/OR framework, AND nodes generally root independent subproblems that can be solved separately. For simplicity we develop our work for *weighted* CSP (WCSP) problems, where costs are natural numbers and global costs are computed by summing partial costs. The extension to other soft-constraint frameworks is straightforward. We introduce a new generation of *depth first* AND/OR Branch and Bound algorithms that traverse the AND/OR search tree and extend the Mini-Bucket approximation scheme for computing a heuristic evaluation function to guide the search [10, 12, 16].

The Mini-Bucket approximation uses a controlling parameter which allows adjustable levels of accuracy and efficiency. Rather than computing and recording functions on many variables as is often required by variable elimination algorithms, the Mini-Bucket scheme partitions function computations into subsets of bounded number of variables, i (the so-called i-bound), and records several smaller functions, instead. It can be shown that it outputs a *lower bound* (resp. *upper bound*) on the desired optimal value for the desired minimization (resp. maximization) task. This is a flexible scheme that can trade off complexity for accuracy; as the i-bound increases, both the complexity (which is exp(i)) and the accuracy increase (for details see [7]).

We experiment with random binary/non-binary CSPs as well as a number of real-world benchmarks. Our results show that indeed in many cases the Branch and Bound over the AND/OR space takes advantage of the structural properties of the problem and significantly improves over the traditional BnB. Impressive time savings are exhibited, especially for small i-bounds when all algorithms rely primarily on search, rather than on pruning. However, if larger i-bounds are possible, the AND/OR algorithms using pre-compiled heuristic information are overall superior.

The paper is organized as follows. Section 2 provides preliminaries and background on the Mini-Bucket algorithms, generic depth first Branch and Bound strategy and the Mini-Bucket based heuristics. In Section 3 we introduce the new paradigm of AND/OR search spaces. Section 4 is devoted to the depth first AND/OR Branch and Bound strategy. Section 5 presents empirical evaluation, Section 6 relates previous work and Section 7 concludes.

2 Preliminaries

2.1 Notations and Definitions

Constraint Networks provide a framework for formulating real-world problems as a set of constraints between variables. They are graphically represented by nodes corresponding to variables and undirected edges corresponding to constraints between variables.

Definition 1 (Constraint Satifaction Problem). *A* Constraint Satisfaction Problem *(CSP) is defined by a set of variables* $X = \{X_1, ..., X_n\}$*, associated with a set of discrete-valued domains,* $D = \{D_1, ..., D_n\}$*, and a set of constraints* $C = \{C_1, ..., C_m\}$*. Each constraint* C_i *is a pair* (S_i, R_i)*, where* R_i *is a relation* $R_i \subseteq D_{i1} \times ... \times D_{ik}$ *defined on a subset of variables* $S_i = \{X_{i1}, ..., X_{ik}\}$ *called the scope of* C_i*, consisting of all tuples of values for* $\{X_{i1}, ..., X_{ik}\}$ *which are compatible with each other. A* constraint network *can be represented by a constraint graph that contains a node for each variable,*

and an arc between two nodes iff the corresponding variables participate in the same constraint. A solution *is an assignment of values to variables* $x = (x_1, ..., x_n), x_i \in D_i$*, such that each constraint is satisfied. A problem that has a solution is termed* satisfiable *or* consistent*.*

Definition 2 (Constraint Optimization Problem). *A finite* Constraint Optimization Problem *is defined by a triple* $(\mathcal{X}, \mathcal{D}, \mathcal{F})$*, where* $\mathcal{X}$ *and* $\mathcal{D}$ *are as in the CSP case, and* $\mathcal{F}$ *is a set of* cost functions $\mathcal{F} = \{f_1, ..., f_m\}$ *which denote preferences among tuples. A cost function* f *is defined over its scope* $var(f)$ *and returns for each tuple a non-negative cost. The objective function, also called the* global cost function *is the sum of all individual cost functions,* $F(X) = \sum_{i=1}^{m} f_i(X)$*. The* solution *is the complete assignment that minimizes/maximizes* $F(X)$*.*

Problems with soft constraints can naturally be formulated as COPs. Observe that, without loss of generality, hard constraints can also be expressed in this model as functions returning two values: 0 for allowed tuples and ∞ for forbidden ones. In particular, the Max-CSP problem can be formulated as a COP using only 0/1 constraints.

2.2 Bucket and Mini-Bucket Elimination Algorithms

Bucket Elimination (BE) [6] is an algorithm for global optimization. Roughly, the algorithm starts by partitioning the set of constraints into n buckets, one per variable. Then variables are eliminated one by one. For each variable X_i a new constraint f_i is computed using the functions in its bucket, summarizing the effect of X_i on the rest of the problem. f_i is then placed in the bucket of the latest variable in its scope. The cost of the best solution to the problem is obtained after processing the last bucket. The bucket-elimination algorithm is time and space exponential in the induced-width of the constraint graph.

Mini-Bucket Elimination (MBE) [7] is an approximation of BE that mitigates its high time and space complexity. When processing variable X_i, its bucket is partitioned into *mini buckets*. Each mini-bucket is processed independently, producing bounded arity functions that are cheaper to compute and store.

2.3 Solving COP

Branch and Bound (BnB) is a general *search* schema for solving constraint optimization tasks [15]. 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*, BnB maintains the cost of the best solution found so far. In a minimization problem this is an *upper bound* (ub) on the problem's optimal cost. At each internal node, defined by its current partial assignment $\bar{x}_p$, the algorithm computes a *lower bound function* ($lb(\bar{x}_p)$), which underestimates the cost of the best solution that can be found by extending $\bar{x}_p$. When $ub \leq lb(\bar{x}_p)$, the current best cost cannot be improved by extending $\bar{x}_p$ and the algorithm *backtracks* pruning the subtree below $\bar{x}_p$. Otherwise, the algorithm moves forward and tries to instantiate the next variable in the ordering.

2.4 Using Bounded Inference to Guide Search

In general, the effectiveness of Branch and Bound greatly depends on the quality of the lower bound functions. Naturally, more accurate lower bounds imply a higher computational effort, hence the right trade-off between the computational overhead at each search tree node and the pruning power exhibited during search may be hard to predict. In the following, we overview a scheme for generating heuristic evaluation functions of varying strengths using the Mini-Bucket approximation [7].

Static Mini-Bucket Heuristics. The idea was first introduced in [12] and showed that the functions recorded by the Mini-Bucket algorithm can be used to assemble a heuristic function that estimates the cost of the completion of any partial assignment to a full solution, and therefore can serve as an evaluation function that can guide search. Briefly, given an ordered set of augmented buckets generated by the Mini-Bucket algorithm and any partial assignment $\bar{x}_p$, the heuristic function $h(\bar{x}_p)$ is defined as the combination (i.e. summation or multiplication) of all the functions that were generated in buckets $p+1$ through n and reside in buckets 1 through p.

Dynamic Mini-Bucket Heuristics. This idea of partitioning-based heuristics can be pushed one step further. Rather than pre-compiling the mini-bucket heuristic information, it is possible to generate it during search. Specifically, given a set of ordered buckets and any partial assignment $\bar{x}_p$, the heuristic function $h(\bar{x}_p)$ is defined as the bound (i.e. lower-bound for minimization, upper-bound for maximization) computed by the Mini-Bucket algorithm (MBE(i)) algorithm subject to the current assignment, restricted to buckets p through n.

3 AND/OR Search Trees Framework

We now move away from the traditional representation of the search space and introduce a family of depth first Branch and Bound algorithms over a recently introduced AND/OR search space paradigm for graphical models [9]. In this section we will give an overview of the main idea and in the next section we will introduce the new AND/OR Branch and Bound algorithm.

The classical way to do search (hereafter called *OR search*) is to instantiate variables following a static/dynamic linear ordering. This process defines a search tree, whose nodes represent states in a the space of partial assignments. In contrast with inference algorithms, the OR search space does not capture any of the structural properties of the model. One way to capture such independencies is to introduce *AND* nodes into the OR search space, which will decompose the problem into separate subproblems.

The *AND/OR search space* is a well known problem solving approach developed in the area of heuristic search [17, 18], that accommodates problem decomposition. The states of the AND/OR space are of two types: OR states which represent alternative ways of solving the problem, and AND states which usually represent problem decomposition into independent subproblems, all of which need be solved. We will next formally define the AND/OR search tree that applies for constraint networks.

The definition of an AND/OR search tree is guided by a tree structure that spans the original constraint graph. We can use a simple DFS spanning tree. However, the

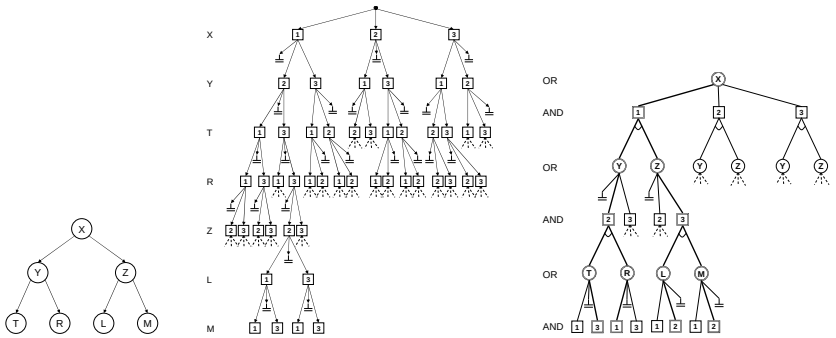

Fig. 1. OR vs. AND/OR search trees.

construction of the AND/OR tree can use a larger collection of spanning trees, called *pseudo-trees* [11], which includes in particular the DFS spanning trees. A pseudo-tree of a graph has the property that any arc of the graph that is not contained in the pseudo-tree is a *back-arc* (i.e. it connects a node to an ancestor in the tree).

Given a constraint graph and its pseudo-tree arrangement T, the associated AND/OR tree is defined as follows. The AND/OR *search tree* has alternating levels of AND and OR nodes. The OR nodes are labeled X_i corresponding to variables. The AND nodes are labeled $\langle X_i, v\rangle$ and correspond to values v assigned to variable X_i. The structure of an AND/OR search tree is based on the underlying pseudo-tree T. 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 its possible value assignments $\langle X_i, v\rangle$ which are consistent along the path from the root. The children of an AND node $\langle X_i, v\rangle$ are OR nodes labeled with the children of variable X_i in the pseudo-tree T. A *solution* of an AND/OR search tree G is not a path, but a *subtree* S which: (1) contains the root node of G; (2) if $n \in S$ is an OR node then it contains one of its child nodes in G and if $n \in S$ is an AND node it contains all its children in G.

Example 1. For illustration, consider the simple tree constraint network in Figure 1(a), over domains $\{1,2,3\}$ which represents a graph coloring problem. In this case the tree rooted at X will also serve as the pseudo-tree arrangement (it is also a DFS tree). Once variable X is assigned value 1, the search space it roots corresponds to two independent subproblems, one rooted by Y (contains variables Y, T, R) and another one rooted by Z (contains variables Z, L, M). These two subspaces do not interact. This can be captured by viewing the assignment $\langle X, 1\rangle$ as an AND state, having variables Y and Z as descendants. The same decomposition can be applied to other assignments of X. Applying the decomposition recursively to Y and Z and the rest of the variables, yields the AND/OR search tree in Figure 1(c). Notice that a full assignment of values to variables in the AND/OR search space is not a path, but a subtree. A solution subtree is highlighted in Figure 1(c).

4 AND/OR Tree Search for COP

The virtue of the AND/OR search tree representation is that its size can be far smaller than the traditional OR tree representation (compare the number of states in Figure 1(b) with that in 1(c)). It is bounded exponentially by the depth of the pseudo tree associated with the original constraint graph. Therefore, any algorithm that traverses the AND/OR search tree in a depth first manner is guaranteed to have a time bound exponential in the depth of the pseudo-tree only and can operate in linear space.

At a certain stage of the search, the current partial solution that is pursued is represented by a partial solution subtree Sol_T of the underlying AND/OR search tree S_T. Since we can only explore S_T by repeated node expansions starting from the root s, Sol_T must be connected, must contain s and will have a *frontier* containing all those nodes generated but not yet expanded. Moreover, Sol_T also contains an *active path* P from the root, which corresponds to the current partial assignment $\bar{v}_i = \langle X_1, v_1\rangle, ..., \langle X_i, v_i\rangle$. Each search tree node n is characterized by a *node value* $v(n)$, which represents the cost of the optimal solution to the subproblem associated with the subtree rooted at n, subject to the current variable instantiation along the active path from root to n. Each AND node $n = \langle X, v\rangle$ is associated with a *label* $l(n)$, representing the sum of all the cost functions for which variable X is contained in their scope and whose scope is contained in the active path from root to n. Based on the values of its successors, the value of a node can be computed recursively as follows:

Definition 3. *For every node* n *in the search tree* S_T *we define its* value *as follows:*

$$v(n) = \begin{cases} l(n) & \textit{if n is terminal AND node} \\ min_{n' \in succ(n)} v(n') & \textit{if n is OR node} \\ l(n) + \sum_{n' \in succ(n)} v(n') & \textit{if n is AND node} \end{cases}$$

where succ(n) *are the successors of* n *in the search tree and* l(n) *is the* label *of node* n.

Therefore, for any given node n in S_T, it is possible to compute its value $v(n)$ by evaluating the subtree rooted at n from the bottom up, as follows. The value of a leaf (terminal) AND node is equal to its label. The value of an internal OR node is obtained by minimizing the values of its successors. The value of an internal AND node is the sum of its own label and the values backed up by its successors.

Proposition 1. *Given an AND/OR search tree S_T, the value $v(n)$ of a node $n \in S_T$ represents the minimal cost solution to the subproblem rooted at n, subject to the current variable instantiation along the path to n from the root. If n is the root of S_T, then $v(n)$ represents the minimal cost solution to the initial problem.*

A *depth first* AND/OR tree search algorithm (hereafter called DF-AO) expands alternating levels of OR and AND nodes, starting from the root of T. When an OR node, $n = X_i$ is expanded, its successors are AND nodes represented by the values v in variable X_i's domain. The algorithm associates each of the child nodes, $n' = \langle X_i, v\rangle$ with its label $l(n')$. The label is calculated as the sum of the cost functions in $B(X_i)$, subject to the instantiation along its path. $B(X_i)$ denotes the *bucket* of variable X_i (for details

see [6]). In other words, given the path $\bar{v}_i = \langle X_1, v_1 \rangle, ..., \langle X_i, v_i \rangle$, node $n' = \langle X_i, v_i \rangle$ is labeled by $l(n') = \sum_{f \in B(X_i)} f(\bar{v}_i)$. If $B(X_i)$ is empty, $l(n')$ is set to 0. When an AND node, $n = \langle X_i, v \rangle$ is expanded, its successors are OR nodes represented by the children of X_i in T. There is no label associated with OR nodes.

At any search step, the algorithm attempts to evaluate the explicated portion of the search space. This is typically triggered by a node whose descendants are all evaluated, namely their values are already determined. An internal OR node minimizes the values propagated back from its children, while an internal AND node computes the value function by summing the label of the node, with the values backed up by its children. The algorithm terminates when the root node is evaluated.

Since the DF-AO algorithm explores each node in the AND/OR search tree in a depth first manner, exactly once, it follows that:

Theorem 1. *Algorithm DF-AO is sound and complete for constraint optimization. The complexity of DF-AO is linear space and time $O(nk^m)$, where m is the depth of the pseudo tree arrangement of the constraint graph [9].*

4.1 Specializing the AND/OR Tree Search Algorithm

The DF-AO algorithm must explore the entire AND/OR search space to find the optimal solution and this may be prohibitive in practice. In the following we describe a way of overcoming this problem by avoiding the exploration of unpromising portions of the search space, using a depth first AND/OR Branch and Bound algorithm.

For this purpose, each node n in the AND/OR search tree is assigned a heuristic estimate $h(n)$, which underestimates the cost of the optimal solution of the subproblem rooted at n, namely $v(n)$. During search, the OR nodes maintain *upper bounds* on their values $v(n)$, while the AND nodes are associated with *lower bounds* of $v(n)$. We start by defining the *inside*/*ouside* context of the active path P during search. Without loss of generality, we assume that P starts at the root s and terminates with an AND node.

Definition 4 (Inside/Outside of Active Path). *Given the current partial solution subtree Sol_T and its active path P, the* inside context of P, *denoted* in(P), *contains all the OR nodes that are evaluated and are children of AND nodes along P. Similarly, the* ouside context of P, *denoted* out(P), *contains all those OR nodes that belong to the frontier of Sol_T and are children of the AND nodes along P.*

Definition 5 (Upper Bound). *Given the active path P of the current partial solution subtree and an OR node $n \in P$, we define* ub(n) *to be the* upper bound *on the cost of the best solution of the subproblem rooted at n. Initially $ub(n)$ is ∞ and then, as search progresses, it is reduced by the values that are succesively propagated back from the AND children of n.*

For illustration, consider the AND/OR tree fragment in Figure2. Initially, the upper bound at the OR node $n = X$ is $ub(n) = \infty$. After exploring the subtree rooted at $n' = \langle X, 0 \rangle$, the value $v(n')$ is available and $ub(n)$ becomes $min(\infty, v(n')) = v(n')$. Similarly, after exploring the second subtree rooted at $n'' = \langle X, 1 \rangle$, the upper bound $ub(n)$ is updated to $min(v(n'), v(n''))$.

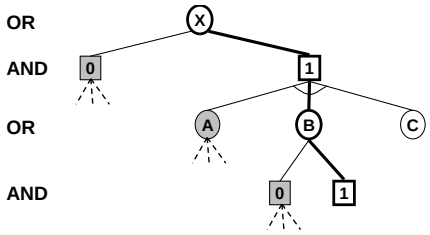

Fig. 2. Upper/Lower bounds computation

Definition 6 (Lower Bound). *Given the active path P of the current partial solution subtree and an AND node $n \in P$, the* lower bound lb(n) *on the cost of the best solution of the problem rooted at n is:*

$$lb(n) = \begin{cases} h(n) & \text{if n is terminal AND node} \\ max(h(n), l(n) + \sum_{n' \in succ(n)} e(n')) & \text{if n is non-terminal AND node} \end{cases}$$

where succ(n) *are the successors of* n, l(n) *is the label of node* n, h(n) *stands for the heuristic estimate of the value* v(n). *For any of the OR successors $n' \in succ(n)$, $e(n')$ is either: 1) the value $v(n')$ if $n' \in in(P)$, or 2) the estimate $h(n')$ if $n \in out(P)$, or 3) the lower bound $lb(n'')$, where n'' is the AND successor of n' on the active path.*

In other words, for any given AND node n along the active path P (that is a non-terminal AND node), it is possible to compute a lower bound $lb(n)$ on the cost of the solution rooted at n, bottom up, starting at the heuristic estimate associated with the tip node of P (that is a terminal AND node) and working upward along the path, until the desired lower bound is computed at n.

At any stage of the lower bound propagation, a test could be conducted to find out if the current partial solution subtree can be extended along its active path P to a better solution. Specifically, if the lower bound $lb(n)$ computed at some AND node n along the active path is greater than or equal to the current upper bound $ub(m)$ maintained at its OR parent m, then the active path is guaranteed not to lead to a better solution and the search can be safely discontinued below the tip node of P.

Theorem 2 (Pruning Rule). *Let P be the current active path such that it starts at the root node $s = X_1$ and ends at some AND node $t = \langle X_i, v_i \rangle$. Let $n = \langle X_j, v_j \rangle$ be an arbitrary AND node on the path and let $m = X_j$ be its OR parent.*

1. *The value $lb(n)$ is a lower bound on the cost of the solution rooted at m.*
2. *If $lb(n) \geq ub(m)$ then it is safe to prune the subtree rooted at the tip node t.*

Example 2. Figure 2 shows a portion of an AND/OR search tree rooted at X. The CLOSED list contains the shaded nodes and the AND node $\langle B, 1 \rangle$ is currently at the top of the OPEN list. The active path P is highlighted. The current upper bound at node B is $ub(B) = v(\langle B, 0 \rangle)$. Similarly, the upper bound at node X is $ub(X) = v(\langle X, 0 \rangle)$. The tip node $\langle B, 1 \rangle$ and the subtree below it can be pruned if either $lb(\langle B, 1 \rangle) \geq ub(B)$ or $lb(\langle X, 1 \rangle) \geq ub(X)$. First we calculate $lb(\langle B, 1 \rangle) = h(\langle B, 1 \rangle)$. If the pruning test

ALGORITHM: *dynamic*-AOMB($\mathcal{C}$, T)
Input: A cost network $\mathcal{C} = (\mathcal{X}, \mathcal{D}, \mathcal{F})$. A pseudo-tree T rooted at X_1. Bucket data structure along a depth-first traversal of T. $\bar{v}$ denotes the partial instantiation on the path from root to the current AND node.
Output: Minimal cost solution.

(1) Initialize OPEN $\leftarrow \{X_1\}$ (X_1 is an OR node); CLOSED $\leftarrow \phi$
(2) Get the first node n in OPEN
(3) Expand node n, generating all its immediate successors, $succ(n)$, as follows:
  (a) **if** (n is an OR node, i.e. $n = X_i$) **then**
    $succ(n) \leftarrow \{n' = \langle X_i, v\rangle | v \in domain(X_i)\}$
    foreach ($n' \in succ(n)$) **do**
      $l(n') = \sum_{f \in B(X_i)} f(\bar{v})$ (initialize local information)
      $h(n') = MBE(X_i, v)$ (assign heuristic estimates)
    $h(n) = min_{n' \in succ(n)} h(n')$
    $ub(n) = \infty$ (initialize upper bound)
  (b) **if** (n is an AND node, i.e. $n = \langle X_i, v\rangle$) **then**
    foreach ($n_a \in ancestors(n)$) **do**
      if (n_a is an AND node $n_a = \langle X_j, v_j\rangle$) **then**
        Evaluate $lb(n_a)$ using Definition 6 and let m be the OR parent of n_a
        if ($lb(n_a) \geq ub(m)$) **then**
          Remove n from OPEN (prune a subtree)
          goto Step (2)
    $succ(n) \leftarrow \{n' = Y | Y \in Children_T(X_i)\}$
  (c) Add $succ(n)$ on top of OPEN
  (d) Remove n from OPEN and place it on CLOSED
(4) Propagate bottom-up node values, as follows:
  (a) For a terminal AND node $n = \langle X, v\rangle$, $v(n) = l(n)$
  (b) For a non-terminal OR node $n = X$, such that $succ(n)$ are all evaluated:
    $v(n) = min_{n' \in succ(n)} v(n')$
  (c) For a non-terminal AND node $n = \langle X, v\rangle$ such that $succ(n)$ are all evaluated:
    $v(n) = l(n) + \sum_{n' \in succ(n)} v(n')$
    $ub(m) = min(ub(m), v(n))$ (update the upper bound at the OR parent m of n)
  (d) If the root node has been evaluated, **return** $v(X_1)$
  (e) Remove portion of CLOSED that is not relevant
(5) **goto** Step (2)

Fig. 3. *dynamic*-AOMB algorithm for Constraint Optimization

at node B fails, we move upward along the active path and calculate $lb(\langle X, 1\rangle) = max(h(\langle X, 1\rangle), l(\langle X, 1\rangle) + v(A) + h(C) + lb(\langle B, 1\rangle))$.

Figure 3 describes a specialized version of an AND/OR tree search algorithm that uses partitioning-based heuristic functions to guide the search. The algorithm, hereafter refered to as $dynamic$-AOMB traverses the AND/OR search tree in a depth first manner, starting from the root node $s = X_1$. A list OPEN simulates the recursion stack. The list CLOSED maintains the search frontier and $succ$ denotes the set of successors of a node in the search tree. When expanding an OR node $n = X_i$ (Step 3a), the algorithm

calculates a heuristic estimate $h(\cdot)$ for each of the possible value extensions $X_i = v$ and orders the corresponding AND successors in decreasing order of their estimates (value ordering). For this purpose, we use the Mini-Bucket approximation, restricted to the subproblem rooted at X_i (i.e. *dynamic mini-bucket heuristics*), but any other lower bounding function can be applied. Pruning occurs when the algorithm attempts to expand an AND node $n = \langle X_i, v \rangle$ (Step 3b). The lower bound function $lb(.)$ of n and all of its AND ancestors along the active path P is revised from the bottom up, using Definition 6. Search is discontinued below n as soon as $lb(n_a) \geq ub(m)$, where n_a is some AND ancestor of n (including n) and m is its OR parent. In Step 4, when the algorithm moves backward, propagating the nodes values, it also updates the upper bounds maintained at the OR nodes, according to Definition 5 (Step 4c).

The *static mini-bucket heuristics* can also be incorporated within the AND/OR search algorithm presented in Figure 3, yielding a new algorithm, called $static$-AOMB. It is possible to show that the independencies captured by the pseudo-tree associated with the network's graph are also present in the augmented bucket structure generated by the mini-bucket algorithm. As a consequence, those functions can be used to create heuristic estimates in a similar manner they were used in the regular OR search space (more details in [12, 8, 16]).

5 Empirical Evaluation

We have evaluated the performance of our AND/OR algorithms for solving the Max-CSP task on over-constrained binary random CSP. A binary random CSP class is characterized by $\langle N, K, C, T \rangle$, where N is the number of variables, K is the number of values per variable, C is the number of constraints and T is the constraint *tightness* defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected. Using this model, we have tested on connectivity regions where non-degenerated pseudo-trees (e.g. chains) could be constructed. Specifically, we have experimented on the following problem classes: $\langle 20, 5, 100, t \rangle$ (medium connectivity) and $\langle 50, 5, 80, t \rangle$ (sparse problems). For each problem class and each parameter setting we generated samples of 20 instances.

Each problem is solved by four algorithms using partitioning-based heuristic information: s-BBMB, d-BBMB, s-AOMB and d-AOMB. s-BBMB [12] uses static MB heuristics and is restricted to a static variable ordering. d-BBMB is our new depth first Branch and Bound algorithm that uses dynamic MB heuristics at each node of the search space. It is also restricted to a static variable ordering. These two algorithms explore the traditional OR space. s-AOMB/d-AOMB use static/dynamic MB heuristics and explore a static AND/OR search tree.

The pseudo-tree was computed as follows. We used the *min-fill* heuristic for computing the induced graph. It places variables with the smallest *fill set* (i.e. the number of induced edges that need be added to fully connect the neighbors of a node) at the end of the ordering, connects all of its neighbors, removes the variable from the graph and repeats the whole procedure. The *pseudo-tree* associated with the induced graph was created as a DFS traversal of the induced graph, starting with the variable that initiated the *min-fill* ordering, always preferring as successor of a node the earliest adjacent node

s-AOMB d-AOMB s-BBMB d-BBMB i=2 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=4 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=6 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=8 % / time / nodes	PFC-RDAC PFC-MRDAC PFC-MPRDAC
N=20, K=5, C=100, T=40%, w*=12, H=15.15				
40 / 152.2 / 2.4M	100 / 53.32 / 1.4M	100 / 8.138 / 250K	100 / 3.657 / 29K	100 / 0.316 / 36.3K
100 / 12.62 / 14K	**100 / 14.3 / 1.8K**	100 / 41.69 / 290	70 / 119.8 / 73	**100 / 0.284 / 21K**
20 / 158.9 / 6.2M	100 / 26.31 / 1.4M	**100 / 3.126 / 181K**	**100 / 3.135 / 30K**	100 / 0.318 / 21K
100 / 12.68 / 38K	100 / 16.3 / 4.9K	100 / 48.21 / 714	70 / 121.6 / 106	
N=20, K=5, C=100, T=60%, w*=12, H=15.4				
0 / 180 / 2.4M	30 / 150.3 / 4M	95 / 53.31 / 1.8M	100 / 9.742 / 200K	100 / 1.249 / 137K
95 / 61.5 / 72K	100 / 26.88 / 3.3K	95 / 70.56 / 481	60 / 146.3 / 81	**100 / 1.101 / 79.7K**
0 / 180 / 6.3M	95 / 82.63 / 4.1M	**100 / 7.508 / 424K**	**100 / 3.874 / 74K**	100 / 1.18 / 79.7K
100 / 53.62 / 180K	**100 / 20.82 / 6.6K**	100 / 69.1 / 1K	35 / 153.9 / 129	

Table 1. MAX-CSP (medium connectivity). Each table entry reports the average percentage of exactly solved instances (%), average CPU time in seconds (time) and average number of search tree nodes expanded (nodes). 180 seconds time limit.

s-AOMB d-AOMB s-BBMB d-BBMB i=2 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=4 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=6 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=8 % / time / nodes	PFC-RDAC PFC-MRDAC PFC-MPRDAC
N=50, K=5, C=80, T=60%, w*=7.75, H=15.5				
70 / 88.65 / 1.2M	100 / 3.093 / 76K	**100 / 0.131 / 2.4K**	**100 / 0.731 / 87**	100 / 3.142 / 227K
100 / 4.17 / 10.7K	**100 / 0.791 / 250**	100 / 0.838 / 80	100 / 1.717 / 53	**100 / 1.849 / 92K**
0 / 180 / 6.5M	75 / 68.19 / 2.8M	100 / 2.743 / 146K	100 / 0.744 / 1.1K	100 / 2.307 / 92K
75 / 74.32 / 444K	100 / 1.788 / 1.2K	100 / 0.634 / 80	100 / 1.673 / 50	
N=50, K=5, C=80, T=80%, w*=7.65, H=15.8				
5 / 176.6 / 3.4M	100 / 12.81 / 310K	**100 / 0.716 / 18K**	**100 / 0.632 / 491**	100 / 17.87 / 1M
95 / 28.37 / 56K	**100 / 1.327 / 287**	100 / 1.152 / 101	100 / 1.713 / 55	**100 / 10.87 / 422K**
0 / 180 / 6.1M	70 / 82.23 / 3.9M	95 / 23.01 / 1.1M	100 / 0.686 / 4.8K	100 / 13.21 / 422K
45 / 117.1 / 636K	100 / 1.776 / 902	100 / 1.113 / 115	100 / 1.441 / 55	

Table 2. MAX-CSP (sparse problems). Each table entry reports the average percentage of exactly solved instances (%), average CPU time in seconds (time) and average number of search tree nodes expanded (nodes). 180 seconds time limit.

in the induced graph. The variable ordering used by the algorithms (except the ones that have dynamic variable ordering) was the one resulted from a DFS traversal of the pseudo-tree arrangement.

Tables 1 and 2 show results for experiments with two classes of problems. Each table contains two horizontal blocks, each corresponding to a particular constraint tightness (T). For each class we also report the average induced width (w*) and the average height of the pseudo-tree (H). In each column, indexed by the i-bound of the mini-bucket heuristic, we have results for s-AOMB(i), d-AOMB(i), s-BBMB(i), d-BBMB(i), as well as for three algorithms based on PFC [13]. Each entry in the table gives the percentage of problems that were solved exactly within a time bound, the average CPU time in seconds required for solving these problems, as well as the average number of search tree nodes expanded (we only report AND nodes for AOMB). We have highlighted the best performance point in each row and in each column.

We observe that for the first problem class (Table 1), the algorithms based on the AND/OR search tree are inferior to those exploring the regular OR space. This, we

network (N,C)	w*	H	d-AOMB d-BBMB i=2 time / nodes	d-AOMB d-BBMB i=4 time / nodes	d-AOMB d-BBMB i=6 time / nodes	d-AOMB d-BBMB i=8 time / nodes	d-AOMB d-BBMB i=10 time / nodes	d-AOMB d-BBMB i=12 time / nodes	s-AOMB s-BBMB i=16 time / nodes
54b (31,144)	11	20	**6.922 / 15.4K** 8.485 / 28.2K	1.359 / 1.8K **0.406 / 836**	0.625 / 346 **0.562 / 445**	**0.157 / 47** 0.453 / 148	**0.14 / 31** 0.141 / 31	**0.188 / 31** 0.203 / 31	- -
404 (100,610)	19	41	**3.156 / 4.8K** - / 5.7M	**0.281 / 303** 234.8 / 3.3M	**0.328 / 257** 23.22 / 26.5K	**0.125 / 100** 4.234 / 3K	**0.141 / 100** 1.219 / 774	**0.172 / 114** 1.906 / 1K	- -
503b (99,390)	8	35	**116.3 / 2.9M** - / 9M	**5.406 / 9.1K** 8.969 / 8.8K	**0.813 / 1.5K** 2.718 / 2.5K	0.156 / 149 **0.094 / 99**	**0.062 / 99** 0.094 / 99	**0.063 / 99** 0.109 / 99	- -
505 (240,2002	22	67	- -	- -	- -	- -	- -	- -	**225.1 / 5.8M** - / 11.5M

Table 3. Results for SPOT5 benchmarks. 1 hour time limit.

speculate, is because the expected gain H=15 vs. N=20 is small and does not offset the overhead in AND nodes in the AND/OR space. The best performance is offered by the PFC class of algorithms. However, for the second class (Table 2) the AND/OR algorithms clearly pay off, offering the best performance and outperforming the PFC class. Here, the induced-width is low, so the performance with large i is likely to be similar to Bucket Elimination.

Our real-life domain consists of several problem instances from the SPOT5 benchmark [3]. These are over-constrained real scheduling problems for Earth observing satellites. The original problem formulation associates a positive real-valued weight with each variable. The task is to find a partial feasible assignment (i.e. meets all relevant constraints) which maximizes the sum of the weights. However, for our purpose, we only consider a Max-CSP variant, namely finding a complete assignment to variables that violates the least number of constraints. For each problem instance we provide the name (model), the size as number of variables (N) and number of constraints (C), the induced width of the constraint graph (w*) and the height of the corresponding pseudo-tree arrangement (H). Table 3 compares primarily the dynamic versions of AOMB and BBMB on several hard enough instances, whereas the last column of the table compares the static versions of those two algorithms, s-AOMB and s-BBMB respectively. We observe a clear dominance of the AND/OR algorithms over the regular OR ones. For instance, when solving problem instance 404, d-AOMB(2) required only 3.156 seconds whereas d-BBMB(2) exceeded its time limit of one hour. The same observation can be made for the static case, where s-AOMB(16) is superior to s-BBMB(16) on the most difficult problem instance (i.e. 505). We did not compare with PFC algorithms because the problem instances involve both binary and ternary constraints.

5.1 Belief Networks

We also experimented with optimization tasks defined over belief networks. *Belief Networks* (BN) [19] provide a formalism for reasoning about partial beliefs under conditions of uncertainty. They are defined as directed acyclic graphs over nodes representing variables of interest. The arcs signify the existence of direct causal influences between linked variables. Formally, a BN is defined by a triple (X, D, P), where X and D are as in the CSP formalism, and P is a set of functions. A function $p_i = P(X_i|pa_i)$, encodes a *conditional probability distribution* of variable X_i given its parents (in the graph) pa_i. The belief network represents a joint probability distribution over X having

s-AOMB d-AOMB s-BBMB d-BBMB BBBT i=2 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB BBBT i=4 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB BBBT i=6 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB BBBT i=8 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB BBBT i=10 % / time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB BBBT i=12 % / time / nodes
N=100, K=2, P=2, C=90, w*=16.3, H=25.8					
58 / 121.6 / 4M	95 / 34.07 / 1.1M	100 / 8.659 / 269K	100 / 2.368 / 72.5K	100 / 0.776 / 26.5K	**100 / 0.188 / 6.3K**
100 / 9.583 / 29.3K	**100 / 0.992 / 1.7K**	**100 / 0.520 / 731**	**100 / 0.365 / 381**	**100 / 0.382 / 225**	100 / 0.531 / 181
1 / 179.9 / 9M	57 / 108.9 / 6.4M	96 / 24.19 / 1.6M	99 / 5.54 / 363K	100 / 2.606 / 179K	100 / 0.34 / 25K
70 / 85.04 / 229K	100 / 2.045 / 2.8K	100 / 0.616 / 698	100 / 0.467 / 387	100 / 0.465 / 229	100 / 0.653 / 191
41 / 136.2 / 29.4K	98 / 31.71 / 6.4K	100 / 6.354 / 1,058	100 / 1.848 / 245	100 / 1.474 / 135	100 / 1.623 / 112
N=100, K=2, P=2, C=98, w*=18.2, H=27.7					
26 / 162.2 / 5.1M	87 / 64.19 / 2.1M	99 / 15.97 / 481K	100 / 5.868 / 172K	100 / 1.625 / 52K	**100 / 0.689 / 22.6K**
100 / 25.77 / 63.1K	**100 / 2.673 / 4.1K**	**100 / 1.277 / 1.3K**	**100 / 0.962 / 615**	**100 / 1.044 / 356**	100 / 1.415 / 243
0 / 180.0 / 8.9M	43 / 131.3 / 7,4M	85 / 47.36 / 2,8M	95 / 18.22 / 1,2M	96 / 11.54 / 712K	100 / 3.344 / 222K
70 / 80.48 / 215K	100 / 4.774 / 8K	100 / 1.520 / 1.6K	100 / 1.016 / 631	100 / 1.046 / 360	100 / 1.462 / 253
17 / 162.6 / 33K	74 / 80.59 / 14.2K	97 / 23.53 / 3,265	99 / 11.07 / 848	99 / 7.392 / 298	100 / 5.057 / 144

Table 4. MPE (medium connectivity): Average number of exactly solved instances (%), average CPU time in seconds (time) and average number of search tree nodes expanded (nodes). 180 seconds time limit.

Network (# vars, avg dom, max dom)	w*	H	s-AOMB d-AOMB s-BBMB d-BBMB i=2 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=3 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=4 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=5 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=6 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=7 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=8 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=9 time / nodes	s-AOMB d-AOMB s-BBMB d-BBMB i=10 time / nodes
Barley (48,8,67)	7	17	- / 5.1M - / 281.6K - / 12.8M - / 2.1M	- / 10.7M **144.9 / 39.9K** - / 9M - / 829.5K	- / 11.4M **13.60 / 976** - / 6.8M 34.75 / 5.7K	266.1 / 5.4M **49.11 / 639** 541.9 / 7.3M - / 56.8K	**1.094 / 5.2K** 49.28 / 122 1.642 / 6.7K 57.06 / 143	**9.361 / 410** 95.44 / 99 9.774 / 523 107.1 / 100			
Mildew (35,17,100)	4	15	479.1 / 5.1M **91.47 / 19.5K** - / 7.9M 152.9 / 64.9K	149.8 / 822K 36.73 / 1.7K **31.72 / 282K** 42.83 / 3K	0.312 / 43 1.781 / 35 **0.297 / 76** 1.86 / 71						
Munin1 (189,5,21)	11	24	292.3 / 3.3M **78.31 / 223K** - / 2.9M - / 311K	39.27 / 480K **15.42 / 9.7K** - / 3.2M - / 327K	17.10 / 255K **12.49 / 2.3K** - / 4.3M - / 185K	**3.768 / 62.2K** 14.51 / 802 - / 3.7M 14.90 / 804	**2.549 / 39.1K** 17.65 / 433 - / 4M 17.47 / 437	**2.736 / 37.6K** 44.04 / 349 - / 3.9M 43.11 / 352	**2.361 / 11.3K** 47.57 / 605 105.4 / 376K 48.63 / 661	**10.30 / 6.3K** 77.45 / 387 111.2 / 366K 78.50 / 427	**18.34 / 1.4K** 101.7 / 378 19.34 / 1.9K 100.6 / 417
Munin2 (1003,5,21)	7	35	- / 1.9M - / 3.1M - / 424K - / 25K	- / 3.6M - / 952K - / 581K - / 75K	- / 5.6M - / 270.1K - / 137K - / 64.2K	**2.984 / 32.9K** 48.47 / 3.9K - / 137.8K - / 16.3K	**0.906 / 7.4K** 35.11 / 1.1K - / 135.1K 121.5 / 1.2K	**0.641 / 1K** 6.203 / 1K - / 170K 101.9 / 1K			
Munin3 (1044,5,21)	7	25	- / 2.9M - / 2.3M - / 371K - / 25.2K	- / 3.1M **91.5 / 62.6K** - / 405K - / 82.4K	5.844 / 53.8K **4.578 / 5.9K** - / 172K - / 38.3K	**0.64 / 6.8K** 3.515 / 3.8K - / 432K - / 23.7K	**0.61 / 5.3K** 4.328 / 3.1K - / 364.9K 166.8 / 3.1K	**0.875 / 1K** 3.282 / 1K 38.94 / 1K 49.89 / 1K			

Table 5. Results for experiments with 5 real world belief networks. Time until completion (seconds) and number of nodes. 600 seconds time limit.

the product form $P_B(X) = \prod_{i=1}^{n} P(X_i|pa_i)$. One popular query over belief networks is finding the *most probable explanation* (MPE), that is finding a complete assignment to all variables having maximum probability, given some evidence e: $P(x_1, ..., x_n) = max_{x_1,...,x_n} \prod_{i=1}^{n} P(x_i, e|pa_i)$.

However, the classical MPE problem can be reformulated as a constraint optimization problem. Each original conditional probability table $P(X_i|pa_i)$ is replaced by a cost function $f_i(X_i, pa_i) = -log(P(X_i|pa_i))$, defined over the same scope. The global cost function that has to be maximized becomes $F(X) = \sum_{i=1}^{n} f_i(X)$.

We have evaluated the performance of our algorithms on random uniform belief networks. They were generated as in [16], using parameters $\langle N, K, C, P \rangle$, where N is the number of variables, K is the domain size, C is the number of conditional probability tables (CPTs) and P is the number of parents in each CPT. [16] provides an extensive

empirical study of the BBMB/BBBT classes of algorithms, for solving the Bayesian MPE problem. It was observed that over a wide range of problem classes, both random and real-world benchmarks, that BBMB/BBBT algorithms are superior to a number of state-of-the-art solvers. BBBT [8, 16] is a regular BnB algorithm that uses MBTE-based heuristics, dynamically at each search node. Unlike the BBMB class, it is not restricted to a static variable ordering.

Table 4 shows experiments with random uniform networks having N=100, K=2, P=2. The results are reported in a similar fashion. We observe the same trend as in the previous experiments. AND/OR search algorithms are superior to the OR algorithms, for all reported i-bounds. The time savings are again more dramatic for small i-bounds. This may be significant because small i-bounds require restricted space.

We also experimented with 5 real world belief networks from the Bayesian Network Repository[1]. We ran one instance of each network in order to compute the most probable explanation, without any evidence. Each algorithm was allowed 10 minutes (600 secs) to prove optimality of the solution. Table 5 summarizes the results. We observe again the same trend, d-AOMB is superior for small i-bounds (e.g. Barley, Mildew, Munin1, Munin3), while s-AOMB dominates for larger i-bounds. We conclude that for the MPE domain and for medium connected and sparse belief networks, the algorithms based on the AND/OR tree representation of the search space provide the best performance.

6 Related Work

The idea of exploiting structural properties of the problem in order to enhance the performance of search algorithms in constraint satisfaction is not new. Freuder and Quinn [11] introduced the concept of pseudo-tree arrangement of a constraint graph as a way of capturing independencies between subsets of variables. Pseudo-tree search [11] is conducted over a pseudo-tree arrangement of the problem which allows the detection of independent subproblems that are solved separately. Dechter's graph-based backjumping algorithm [5] uses a DFS spanning tree to extract knowledge about dependencies in the graph. The notion of DFS-based search was also used by Collin et al. [4] for a distributed constraint satisfaction algorithm. Bayardo and Miranker [2] reformulated the pseudo-tree search algorithm in terms of backjumping and showed that the depth of a pseudo-tree arrangement is always within a logarithmic factor off the induced width of the graph. Larrosa et al. [14] introduced BnB search that exploits a pseudo-tree arrangement of the constraint graph to boost the Russian Doll search for WCSP.

7 Conclusions

The paper investigates the impact of the AND/OR search paradigm for graphical models on Branch-and-Bound algorithms. Since the depth of an AND/OR search tree can be shown to be smaller than the depth of an equivalent OR search tree, search algorithms that explore an AND/OR space can exhibit exponential savings when compared

[1] http://www.cs.huji.ac.il/labs/compbio/Repository

with their OR space counterparts. We propose two new algorithms, s-AOMB and d-AOMB, which extend recent schemes of Branch-and-Bound with mini-bucket heuristics, s-BBMB and d-BBMB, to the new AND/OR search framework. Our empirical work was concentrated on the Max-CSP task in constraint processing and the MPE problem in belief networks, and shows that for some problem classes the new AND/OR scheme improves dramatically over the regular OR space algorithms, especially when the structure of the problem facilitates the construction of pseudo-trees with relatively small heights.

Our approach leaves room for future improvements, which are likely to make it more effective in practice. For instance, it can be modified to traverse an AND/OR graph, rather than a tree, which would facilitate caching. We did not study the effect of the ordering in which the independent subproblems are solved. Similarly, we used a rather simple scheme of generating pseudo-tree arrangements, probably having highly non-optimal height. All these issues represent out current and future work.

References

1. Arnborg, S. A. 1985. Efficient algorithms for combinatorial problems on graphs with bounded decomposability. BIT 25:2–23.
2. Bayardo, R. J. and Miranker, D. P. 1995. On the space-time trade-off in solving constraint satisfaction problems. In Fourteen IJCAI-95.
3. Bensana, E., Lemaitre, M. and Verfaillie, G. 1999. Earth observation satellite management. In Constraints 4 (1999), 293-299 .
4. Collin, Z., Dechter, R. and Katz, S. 1991. On the feasibility of distributed constraint satisfaction. In Proceedings of IJCAI-91.
5. Dechter, R. 1990. Constraint networks - a survey. In Encyclopedia of AI, 1990.
6. Dechter, R. 1999. Bucket elimination: A unifying framework for reasoning. Artificial Intelligence, 113:41–85.
7. Dechter, R. and Rish, I. 2003. Mini-Buckets: A general scheme for approximating inference. In Journal of ACM.
8. Dechter, R., Kask, K. and Larrosa, J. 2001. A general scheme for multiple lower bound computation in constraint optimization. In Proceedings of CP-01.
9. Dechter, R. 2004. AND/OR Search Spaces for Graphical Models. Technical Report.
10. Dechter, R. and Mateescu, R. 2004. Mixtures of deterministic and probabilistic networks. In Proceedings of UAI-04.
11. Freuder, E.C. and Quinn, M.J. 1985. Taking advantage of stable sets of variables in constraint satisfaction problems. In Proceedings of IJCAI-85.
12. Kask, K. and Dechter, R. 1999. Branch and Bound with Mini-Bucket heuristics. In IJCAI-99.
13. Larrosa, J. and Meseguer, P. 1999. Partition-based lower bound for Max-CSP. In CP-99.
14. Larrosa, J., Meseguer, P. and Sanchez, M. 2002. Pseudo-tree search with soft constraints. In Proceedings of ECAI-02.
15. Lawler, E.L. and Wood, D.E. 1966. Branch-and-bound methods: A survey. *Operations Research* **14(4)**, 699-719.
16. Marinescu, R., Kask, K., Dechter, R. 2003. Systematic vs. non-systematic search algorithms for solving the MPE task in Bayesian networks. In Proceedings of UAI-03.
17. Nillson, K.L. 1980. Principles of Artificial Intelligence. Tioga, Palo Alto, CA.
18. Pearl, J. 1984. Heuristics: Intelligent search strategies for computer problem solving. Addison-Welsey, 1984.
19. Pearl, J. 1988. Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, 1988.