Memory Intensive AND/OR Search for Combinatorial Optimization in Graphical Models

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

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

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

Abstract

In this paper we explore the impact of caching during search in the context of the recent framework of AND/OR search in graphical models. Specifically, we extend the depth-first AND/OR Branch-and-Bound *tree search* algorithm to explore an AND/OR *search graph* by equipping it with an adaptive caching scheme similar to good and no-good recording. Furthermore, we present *best-first* search algorithms for traversing the same underlying AND/OR search graph and compare both algorithms empirically. We focus on two common optimization problems in graphical models: finding the Most Probable Explanation (MPE) in belief networks and solving Weighted CSPs (WCSP). In an extensive empirical evaluation we demonstrate conclusively the superiority of the memory intensive AND/OR search algorithms on a variety of benchmarks.

Key words: search, AND/OR search, decomposition, graphical models, Bayesian networks, constraint networks, constraint optimization

1 Introduction

This is the second of two articles describing and evaluating the power of AND/OR search spaces for combinatorial optimization in graphical models. The first paper [1] focused on extending Branch-and-Bound algorithms to AND/OR search spaces which have no cycles, namely to AND/OR search trees. The virtue of the AND/OR

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

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

representation is that the search space size may be far smaller than that of a traditional OR representation which often translates to significant time savings. In the current paper we improve efficiency further by using more memory, exploring what we refer to as the context minimal AND/OR search graph.

Specifically, we extend the AND/OR Branch-and-Bound tree search algorithm introduced in [1–3] to explore the context minimal AND/OR search graph using a flexible caching mechanism that can adapt to memory limitations. The caching scheme is similar to good and no-good recording [4,5] which were used in several recent schemes such as Recursive Conditioning [6], Valued Backtracking [7] and Backtracking with Tree Decompositions [8]. Our contributions beyond those schemes is in presenting these ideas in an independent manner using the notion of AND/OR search spaces and extending optimization techniques to this framework. Finally, we carried out an extensive empirical study on which we report.

Clearly, the AND/OR search space can be explored by any traversal algorithm. So we next investigated the other most common search approach which is Best-First search. Best-First search is known to be superior among memory intensive search algorithms [9]. We therefore present a new AND/OR search algorithm that explores the context minimal AND/OR search graph in a best-first manner. Under conditions of admissibility and monotonicity of the heuristic function, best-first search is known to expand the minimal number of nodes, at the expense of using additional memory [9]. We will show that these savings in number of nodes often translate into significant time savings.

The efficiency of both depth-first and best-first AND/OR search methods also depends on the accuracy of the guiding heuristic function. We used the Mini-Bucket heuristic [10] which is extracted from the functional specification of the graphical model using the Mini-Bucket approximation algorithm [11]. These heuristics were explored in [1] in the context of AND/OR search trees. Following [1,2], we continue to explore empirically the efficiency of static and dynamic mini-bucket heuristics within the cache-based search spaces.

As in our earlier work [1–3], we apply the algorithms to finding the Most Probable Explanation (MPE) in belief networks [12] and to solving Weighted CSPs [13]. Our results show conclusively that the memory intensive AND/OR search algorithms improve dramatically over competitive approaches, especially when the heuristic estimates are less accurate and do not prune the search space effectively. We demonstrate the impact of caching, the impact of the strength of the guiding evaluation function, as well as the impact of best-first versus depth-first search regimes. We also investigate other factors that impact the performance of any search algorithm such as: the availability of hard constraints ($i.e.$, determinism), the availability of good initial upper bounds, and the availability of good ordering schemes ($e.g.$, pseudo trees).

The paper is organized as follows. Sections 2 and 3 provide background on graphical models and on the AND/OR search spaces. Sections 4 and 5 present the new depth-first and best-first AND/OR search algorithms exploring the context minimal AND/OR graph. Section 6 reviews the mini-bucket heuristics for AND/OR search. In Section 7 we present an extensive empirical evaluation of the proposed memory intensive search methods, while Section 8 provides concluding remarks and directions of future research. The relevant related work is discussed in detail in [1]. This paper is based in part on [14–16].

2 Background

2.1 Preliminaries

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

DEFINITION **1 (directed, undirected graphs)** *A* directed graph *is defined by a pair $G = \{\mathbf{V}, \mathbf{E}\}$, where $\mathbf{V} = \{X_1, ..., X_n\}$ is a set of vertices (nodes), and $\mathbf{E} = \{(X_i, X_j)|X_i, X_j \in V\}$ is a set of edges (arcs). If $(X_i, X_j) \in \mathbf{E}$, we say that X_i points to X_j. The degree of a vertex is the number of incident arcs to it. For each vertex X_i, $pa(X_i)$ or pa_i, is the set of vertices pointing to X_i in G, while the set of child vertices of X_i, denoted $ch(X_i)$, comprises the variables that X_i points to. The family of X_i, denoted F_i, includes X_i and its parent vertices. A directed graph is acyclic if it has no directed cycles. An* undirected *graph is defined similarly to a directed graph, but there is no directionality associated with the edges.*

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

2.2 Graphical Models

A graphical model is defined by a collection of functions $\mathbf{F}$, over a set of variables $\mathbf{X}$, conveying probabilistic or deterministic information, whose structure is captured by a graph. We used the formalism presented in [17].

DEFINITION **3 (graphical model, primal graph)** *A* graphical model *is a 4-tuple* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$, *where: 1.* $\mathbf{X} = \{X_1, ..., X_n\}$ *is a set of variables; 2.* $\mathbf{D} = \{D_1, ..., D_n\}$ *is a set of finite domains of values; 3.* $\mathbf{F} = \{f_1, ..., f_r\}$ *is a set of real valued functions, each defined over a subset of variables* $S_i \subseteq \mathbf{X}$ *(i.e., the scope); 4.* $\otimes_i f_i \in \{\prod_i f_i, \sum_i f_i\}$ *is a combination operator. The graphical model represents the combination of all its functions, namely* $\otimes_{i=1}^r f_i$. *When the combination operator is irrelevant we denote* $\mathcal{R}$ *by* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$. *The* primal graph *of a graphical model is an undirected graph that has the variables as its vertices and edges connecting any two variables that appear in the scope of the same function.*

There are various queries (tasks) that can be posed over graphical models. We refer to all as *automated reasoning problems*. In general, an optimization task is a reasoning problem defined as a function from a graphical model to a set of elements, most commonly, the real numbers.

DEFINITION **4 (constraint optimization problem)** *A* constraint optimization problem *is a pair* $\mathcal{P} = \langle \mathcal{R}, \Downarrow_{\mathbf{X}} \rangle$, *where* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$ *is a graphical model. If* S *is the scope of function* $f \in \mathbf{F}$ *then* $\Downarrow_S f \in \{max_S f, min_S f\}$. *The optimization problem is to compute* $\Downarrow_{\mathbf{X}} \otimes_{i=1}^r f_i$. *The min/max (*$\Downarrow$*) operator is called an* elimination *operator because it removes the arguments from the input functions' scopes.*

For a detailed description and examples of graphical models such as constraint networks, cost networks and belief networks we refer the reader to [17,18,1].

3 AND/OR Search Spaces for Graphical Models

The usual way to do search in graphical models is to instantiate variables in turn, following a static or dynamic variable ordering. In the simplest case, this process defines a search tree (called here OR search tree), whose nodes represent states in the state of partial assignments. This search space does not capture the structure of the underlying graphical model. To remedy this problem, an AND/OR search space was recently introduced in the context of general graphical models [18]. It specializes the AND/OR space introduced in [19] to graphical models. The AND/OR search space is defined using a backbone *pseudo tree* [20,5]. In subsections 3.1 and 3.2 we will give a brief overview of searching the AND/OR search trees by Branch-and-Bound, which was presented in detail in [1].

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

As in [1], we consider in the remainder of the paper an optimization problem $\mathcal{P} = \langle \mathcal{R}, min \rangle$ over a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$ for which the combination and elimination operators are *summation* and *minimization*, respectively.

3.1 AND/OR Search Trees for Graphical Models

In this subsection we overview briefly the AND/OR search tree for graphical models which was introduced in [18,1]. Given a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, its primal graph G and a pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree, denoted $S_{\mathcal{T}}(\mathcal{R})$, has alternating levels of OR and AND nodes. The OR nodes are labeled X_i and correspond to the variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ and correspond to the values in the domains of the variables. The structure of the AND/OR search tree is based on the underlying pseudo tree. The root of $S_{\mathcal{T}}(\mathcal{R})$ is an OR node labeled with the root of $\mathcal{T}$. The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in the pseudo tree $\mathcal{T}$. A path from the root of the search tree $S_{\mathcal{T}}(\mathcal{R})$ to a node n is denoted by π_n. The assignment sequence along π_n, denoted $asgn(\pi_n)$, is the set of value assignments associated with the AND nodes along π_n (see Fig. 1 in [1] for an example of an AND/OR tree).

A *solution tree* of an AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is an AND/OR subtree T such that: 1) it contains the root s of $S_{\mathcal{T}}(\mathcal{R})$; 2) if a non-terminal AND node $n \in S_{\mathcal{T}}(\mathcal{R})$ is in T then all of its children are in T; 3) if a non-terminal OR node $n \in S_{\mathcal{T}}(\mathcal{R})$ is in T then exactly one of its children is in T; 4) all its leaf (terminal) nodes are consistent.

Based on earlier work [18], it can be shown that given a graphical model $\mathcal{R}$ and a pseudo tree $\mathcal{T}$, the size of the AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is $O(n \cdot k^m)$ where m is the depth of the pseudo tree, n is the number of variables, and k bounds the domain size. Moreover, a graphical model that has treewidth w^* has an AND/OR search tree whose size is $O(n \cdot k^{w^* \cdot \log n})$.

The arcs from nodes X_i to $\langle X_i, x_i \rangle$ in an AND/OR search tree are annotated by *weights* derived from the cost functions in $\mathbf{F}$.

DEFINITION **6 (arc weight)** *The* weight $w_{(n,m)}(X_i, x_i)$ *(or simply* $w(n, m)$*) of the arc* (n, m)*, where* X_i *labels* n *and* $\langle X_i, x_i \rangle$ *labels* m*, is the* combination *(*$i.e.$*,* sum*)*

of all the functions whose scope includes X_i and is fully assigned along the path from the root to m, evaluated at the values along the path.

With each node n of the weighted AND/OR search tree we can associate a *value* $v(n)$ which stands for the optimal solution cost of the subproblem below n, conditioned on the assignment on the path leading to it [18,1]. It was shown that $v(n)$ obeys the following recursive definition:

DEFINITION **7 (node value)** *The* value $v(n)$ *of a node* n *in a weighted AND/OR tree is defined recursively as follows (where* $succ(n)$ *are the children of* n*):*

$$v(n) = \begin{cases} 0 & \text{, if } n = \langle X, x \rangle \text{ is a terminal AND node} \\ \infty & \text{, if } n = X \text{ is a terminal OR node} \\ \sum_{m \in succ(n)} v(m) & \text{, if } n = \langle X, x \rangle \text{ is an AND node} \\ \min_{m \in succ(n)} (w(n,m) + v(m)) & \text{, if } n = X \text{ is an OR node} \end{cases} \quad (1)$$

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

3.2 AND/OR Branch-and-Bound Search on AND/OR Trees

In [1–3] we introduced a new generation of linear space Branch-and-Bound search algorithms that exploit the underlying structure of the graphical model by traversing in a depth-first manner an AND/OR search tree associated with the graphical model. During search, the algorithm maintains the cost of the best solution found so far, which is an upper bound ub on the minimal cost solution. In addition, each node n in the search tree is also associated with a static heuristic function $h(n)$ that underestimates the minimal cost solution $v(n)$ to the subproblem below n, and it can be either pre-compiled or computed during search. The current partial solution being pursued is represented by a partial solution tree, T'. Given the current T', the algorithm then computes a heuristic lower bounding estimate $f(T')$ on the optimal extension of T' to a complete solution tree and, if $f(T') \geq ub$, it prunes the search space below the current tip node.

The efficiency of the algorithm depends heavily on its guiding heuristic function. In [1,2] we investigated the power of a heuristic generation scheme based on the Mini-Bucket approximation [11], in both static and dynamic setups. Since the Mini-Bucket algorithm is controlled by a bounding parameter, it allows heuristics having varying degrees of accuracy and results in a spectrum of search algorithms that can trade off heuristic computation and search.

We evaluated empirically the AND/OR Branch-and-Bound algorithm with the mini-bucket heuristics for probabilistic and deterministic optimization tasks [1,2]. The results showed conclusively that the scheme improves dramatically over the traditional OR approaches, in many cases yielding several orders of magnitude improvements in time and size of the search space explored.

In the following subsection we overview the notion of AND/OR search *graph* for general graphical models, which was presented in [18].

3.3 AND/OR Search Graphs for Graphical Models

It is often the case that a search space that is a tree can become a graph if identical nodes that root identical search subspaces and which correspond to identical reasoning subproblems are identified. Any two identical nodes can be *merged*, thus reducing the size of the search space. Some of these nodes can be identified based on graph-based *contexts*.

First, we present the notion of *induced width of a pseudo tree* of a graph G [18] which is necessary for bounding the size of the AND/OR search graphs. We denote by $d_{DFS}(\mathcal{T})$ a linear DFS ordering of a tree $\mathcal{T}$.

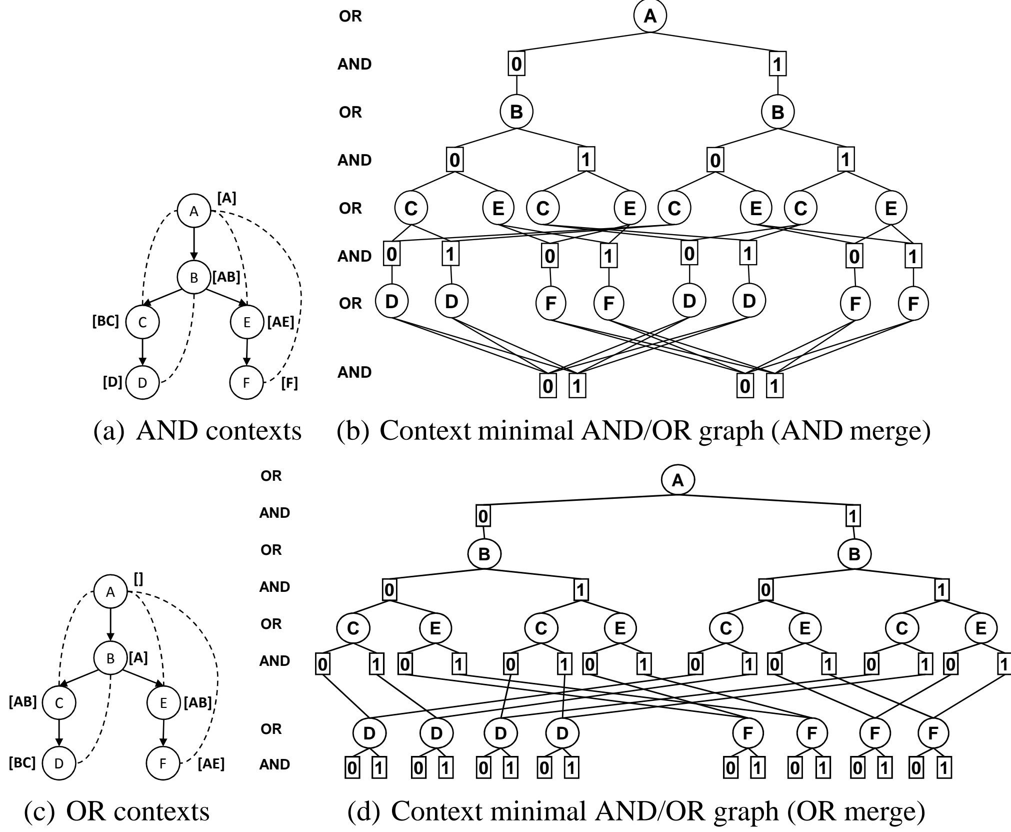
DEFINITION **8 (induced width of a pseudo tree)** *Given a graph G, the induced width of G relative to a pseudo tree $\mathcal{T}$, $w_{\mathcal{T}}(G)$, is the induced width along the $d_{DFS}(\mathcal{T})$ ordering of $G^{\mathcal{T}}$, the extended graph of G relative to $\mathcal{T}$.*

We next provide definitions which allow identifying nodes that can be merged. The idea is to determine a minimal set of predecessor variables to X_i, whose assignment completely determines the subproblem below X_i along the current path. Since a path to an OR node X_i and to an AND node $\langle X_i, x_i \rangle$ differs by the assignment x_i to X_i, these minimal assignments that we seek can differ. Indeed, the following definitions distinguish between two types of context-based caching which are quite subtle. In these definitions, ancestors and descendants are with respect to the pseudo tree $\mathcal{T}$, while the connectivity is with respect to the primal graph G.

DEFINITION **9 (parents)** *Given a primal graph G and a pseudo tree $\mathcal{T}$ of a reasoning problem $\mathcal{P}$, the* parents *of an OR node X_i, denoted by pa_i or pa_{X_i}, are the ancestors of X_i which are connected to X_i or to descendants of X_i in G.*

DEFINITION **10 (parent-separators)** *Given a primal graph G and a pseudo tree $\mathcal{T}$ of a reasoning problem $\mathcal{P}$, the* parent-separators *of X_i (or of $\langle X_i, x_i \rangle$), denoted by pas_i or pas_{X_i}, are formed by X_i and its ancestors that are connected in G to descendants of X_i (not only to X_i).*

It follows from these definitions that the parents of X_i, pa_i, separate in the primal



(a) AND contexts (b) Context minimal AND/OR graph (AND merge)

(c) OR contexts (d) Context minimal AND/OR graph (OR merge)

Fig. 1. AND/OR search graph for graphical models.

graph G (and also in the extended graph $G^{\mathcal{T}}$ and in the induced extended graph $G^{\mathcal{T}^*}$) the ancestors of X_i from its descendants. Similarly, the parent-separators set of X_i, pas_i, separate the ancestors of X_i from its descendants. It is also easy to see that each variable X_i and its parents pa_i form a clique in the induced graph $G^{\mathcal{T}^*}$. As was shown in [18], there exists the following relation between pa_i and pas_i: (1) if Y is the single child of X in $\mathcal{T}$, then $pas_X = pa_Y$; (2) if X has children $Y_1, ..., Y_k$ in $\mathcal{T}$, then $pas_X = \cup_{i=1}^{k} pa_{Y_i}$.

THEOREM **1 (context based merge [18])** *Given $G^{\mathcal{T}^*}$, let π_{n_1} and π_{n_2} be any two paths in an AND/OR search graph, ending with two nodes, n_1 and n_2.*

(1) If n_1 and n_2 are AND nodes labeled by $\langle X_i, x_i \rangle$ and $asgn(\pi_{n_1})[pas_{X_i}] = asgn(\pi_{n_2})[pas_{X_i}]$ then the AND/OR search subtrees rooted by n_1 and n_2 are identical. The $asgn(\pi_{n_i})[pas_{X_i}]$ is called the ***AND context*** *of n_i.*

(2) If n_1 and n_2 are OR nodes labeled by X_i and $asgn(\pi_{n_1})[pa_{X_i}] = asgn(\pi_{n_2})[pa_{X_i}]$ then the AND/OR search subtrees rooted by n_1 and n_2 are identical. The $asgn(\pi_{n_i})[pa_{X_i}]$ is called the ***OR context*** *of n_i.*

DEFINITION **11 (context minimal AND/OR search graph [18])** *The AND/OR search graph of $\mathcal{R}$ based on the backbone pseudo tree $\mathcal{T}$ that is closed under the context-based merge operator is called the* context minimal AND/OR search graph *and is*

denoted by $C_{\mathcal{T}}(\mathcal{R})$.

We should note that we can in general merge nodes based both on AND and OR contexts. However, it was shown in [18] that doing just one of them renders the other unnecessary (namely, yielding a small constant factor only). In this paper we will use AND context based merging.

THEOREM **2 (size of context minimal AND/OR search graphs [18])** *Given a graphical model $\mathcal{R}$, its primal graph G, and a pseudo tree $\mathcal{T}$ having induced width $w^* = w_{\mathcal{T}}(G)$, the size of the context minimal AND/OR search graph based on $\mathcal{T}$, $C_{\mathcal{T}}(\mathcal{R})$, is $O(n \cdot k^{w^*})$, where k bounds the domain size.*

Example 1 *Consider the example given in Fig. 1 which is based on Example 1 from [1]. The AND contexts of each node in the pseudo tree is given in square brackets in Fig. 1(a). The context minimal AND/OR search graph (based on AND merging) is given in Fig. 1(b). Its size is far smaller than that of the AND/OR search tree from Fig. 1 in [1] (16 vs. 54 AND nodes). Similarly, Fig. 1(d) shows the context minimal AND/OR graph based on the OR contexts given in Fig. 1(c). Its size is larger than that of the AND based graph (38 vs. 16 AND nodes) in this case. Consider for example variable C with AND-context $\{B, C\}$ from Fig. 1(a). In Fig. 1 from [1], the search subtrees below any appearance of $(B = 0, C = 0)$ ($i.e.$, corresponding to the subproblems below the AND nodes labeled $\langle C, 0\rangle$ along the paths containing the assignments $B = 0$ and $C = 0$, respectively) are all identical, and therefore can be merged, as shown in the search graph from Fig. 1(b).*

4 AND/OR Branch-and-Bound with Caching

Traversing AND/OR search spaces by depth-first Branch-and-Bound or by best-first search algorithms was described as early as [19,21,22] in the context of general search spaces. In the following two sections we revisit the definitions needed to describe the algorithms. We will then introduce two classes of memory intensive search algorithms that explore the context minimal AND/OR search graph of graphical models, in either a *depth-first* or *best-first* manner, for finding optimal solution trees. The algorithms extend those presented in [1] for exploring AND/OR search trees to algorithms exploring AND/OR search graphs.

DEFINITION **12 (partial solution tree)** *A* partial solution tree *T' of a context minimal AND/OR search graph $C_{\mathcal{T}}(\mathcal{R})$ is a subtree which: (1) contains the root node s of $C_{\mathcal{T}}(\mathcal{R})$; (2) if n in T' is an OR node then it contains one of its AND child nodes in $C_{\mathcal{T}}(\mathcal{R})$, and if n is an AND node it contains all its OR children in $C_{\mathcal{T}}(\mathcal{R})$. A node in T' is called a* tip *node if it has no children in T'. A tip node is either a* terminal *node (if it has no children in $C_{\mathcal{T}}(\mathcal{R})$), or a* non-terminal *node (if it has children in $C_{\mathcal{T}}(\mathcal{R})$).*

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

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

Heuristic Lower Bounds on Partial Solution Trees. We start with the notions of exact heuristic evaluation functions of a partial solution tree [1,2], which will be used to guide the AND/OR Branch-and-Bound.

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

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

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

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

Clearly, by definition, $f(T'_n) \leq f^*(T'_n)$, and if n is the root of the context minimal AND/OR search graph, then $f(T') \leq f^*(T')$ [1].

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

Algorithm 1: AOBB-C: AND/OR Branch-and-Bound Graph Search

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, pseudo-tree $\mathcal{T}$ rooted at X_1, parent separator sets pas_i (AND-context) for every variable X_i, heuristic function $h(n)$.

Output: Minimal cost solution and an optimal solution assignment.

```
1  create an OR node s labeled X_1                          // Create and initialize the root node
2  v(s) ← ∞; ST(s) ← ∅; OPEN ← {s}
3  Initialize cache tables with entries "NULL"              // Initialize cache tables
4  while OPEN ≠ ∅ do
5      n ← top(OPEN); remove n from OPEN                    // EXPAND
6      if n is an OR node, labeled X_i then
7          foreach x_i ∈ D_i do
8              create an AND node n', labeled ⟨X_i, x_i⟩
9              v(n') ← 0; ST(n') ← ∅
10             w(n, n') ← Σ_{f ∈ B_T(X_i)} f(asgn(π_n))     // Compute the OR-to-AND arc weight
11             succ(n) ← succ(n) ∪ {n'}
12     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
13         cached ← false; deadend ← false
14         if Cache(asgn(π_n)[pas_i]) ≠ NULL then
15             v(n) ← Cache(asgn(π_n)[pas_i]).value         // Retrieve value
16             ST(n) ← Cache(asg(π_n)[pas_i]).assignment;   // Retrieve optimal assignment
17             cached ← true                                // No need to expand below
18         foreach OR ancestor m of n do
19             f(T'_m) ← evalPartialSolutionTree(T'_m)
20             if f(T'_m) ≥ v(m) then
21                 deadend ← true
22                 break
23         if deadend == false and cached == false then
24             foreach X_j ∈ children_T(X_i) do
25                 create an OR node n' labeled X_j
26                 v(n') ← ∞; ST(n') ← ∅
27                 succ(n) ← succ(n) ∪ {n'}
28         else if deadend == true then
29             succ(p) ← succ(p) − {n}
30     Add succ(n) on top of OPEN                           // PROPAGATE
31     while succ(n) == ∅ do
32         if n is an OR node, labeled X_i then
33             if X_i == X_1 then
34                 return (v(n), ST(n))                     // Search is complete
35             v(p) ← v(p) + v(n)                           // Update AND node value (summation)
36             ST(p) ← ST(p) ∪ ST(n)                        // Update solution tree below AND node
37         else if n is an AND node, labeled ⟨X_i, x_i⟩ then
38             Cache(asgn(π_n)[pas_i]).value ← v(n)         // Save AND node value in cache
39             Cache(asgn(π_n)[pas_i]).assignment ← ST(n);  // Save optimal assignment
40             if v(p) > (w(p, n) + v(n)) then
41                 v(p) ← w(p, n) + v(n)                    // Update OR node value (minimization)
42                 ST(p) ← ST(n) ∪ {(X_i, x_i)}             // Update solution tree below OR node
43         remove n from succ(p)
44         n ← p
```

In [1] we also showed that the pruning test can be sped up if we associate upper bounds with internal nodes as well. Specifically, if m is an OR ancestor of t in T' and T'_m is the subtree of T' rooted at m, then it is also safe to prune the search tree below t, if $f(T'_m) \geq ub(m)$. For illustration, see also Section 6 in [1].

Algorithm 2: Recursive computation of the heuristic evaluation function.

```
function: evalPartialSolutionTree(T'_n, h(n))
Input: Partial solution subtree T'_n rooted at node n, heuristic function h(n).
Output: Heuristic evaluation function f(T'_n).
1 if succ(n) == ∅ then
2 |   return h(n)
3 else
4 |   if n is an AND node then
5 |   |   let m_1, ..., m_k be the OR children of n in T'_n
6 |   |   return Σ_{i=1}^{k} evalPartialSolutionTree(T'_{m_i}, h(m_i))
7 |   else if n is an OR node then
8 |   |   let m be the AND child of n in T'_n
9 |   |   return w(n, m) + evalPartialSolutionTree(T'_m, h(m_i))
```

The **Depth-First AND/OR Branch-and-Bound** algorithm, AOBB-C, for searching AND/OR graphs for graphical models, is described by Algorithm 1. It interleaves a forward expansion step of the current partial solution tree (EXPAND) with a backward propagation step (PROPAGATE) that updates the node values. This part is identical to the tree-based variant [1] and we describe it here for completeness.

The context-based caching uses a table representation. For each variable X_i, a table is reserved in memory for each possible assignment to its parent-separator set pas_i ($i.e.$, AND context). During search, each table entry records the optimal solution (both the cost and an optimal solution tree) to the subproblem below the corresponding AND node. Initially, each entry has a predefined value, in our case NULL. The fringe of the search is maintained by a stack called OPEN. The current node is denoted by n, its parent by p, and the current path by π_n. The children of the current node are denoted by $succ(n)$.

Each node n in the search graph maintains its current value $v(n)$, which is updated based on the values of its children. For OR nodes, the current $v(n)$ is an upper bound on the optimal solution cost below n. Initially, $v(n)$ is set to ∞ if n is OR, and 0 if n is AND, respectively. A data structure $ST(n)$ maintains the actual best solution tree found in the subgraph rooted at n. The node based heuristic function $h(n)$ of $v(n)$ is assumed to be available to the algorithm, either retrieved from a cache or computed during search.

Since we use AND caching, before expanding the current AND node n, its cache table is checked (line 14). If the same context was encountered before, it is retrieved from the cache, and $succ(n)$ is set to the empty set, which will trigger the PROPAGATE step. The algorithm also computes the heuristic evaluation function for every partial solution subtree rooted at the OR ancestors of n along the path from the root (lines 18–22). The search below n is terminated if, for some OR ancestor m, $f(T'_m) \geq v(m)$, where $v(m)$ is the current upper bound on the optimal cost below m. The recursive computation of $f(T'_m)$ is described in Algorithm 2.

If a node is not found in cache, it is expanded in the usual way, depending on whether it is an AND or OR node (lines 6–29). If n is an OR node, labeled X_i,

then its successors are AND nodes represented by the values x_i in variable X_i's domain (lines 6–11). Each OR-to-AND arc is associated with the appropriate weight. Similarly, if n is an AND node, labeled $\langle X_i, x_i \rangle$, then its successors are OR nodes labeled by the child variables of X_i in $\mathcal{T}$ (lines 23–27). There are no weights associated with AND-to-OR arcs.

The node values are updated by the PROPAGATE step (lines 31–44). It is triggered when a node value has an empty set of descendants (note that as each successor is evaluated, it is removed from the set of successors in line 43). This means that all its children have been evaluated, and their final values are already determined. If the current node is the root, then the search terminates with its value and an optimal solution tree (line 34). If n is an OR node, then its parent p is an AND node, and p updates its current value $v(p)$ by summation with the value of n (line 35). An AND node n propagates its value to its parent p in a similar way, by minimization (lines 37–42). It also saves in cache the value and optimal solution subtree below it (lines 38–39). Finally, the current node n is set to its parent p (line 44), because n was completely evaluated. Each node in the search graph also records the current best assignment to the variables of the subproblem below it. Specifically, if n is an AND node, then $ST(n)$ is the union of the optimal trees propagated from n's OR children (line 36). Alternatively, if n is an OR node and n' is its AND child such that $n' = argmin_{m \in succ(n)}(w(n,m)+v(m))$, then $ST(n)$ is obtained from the label of n' combined with the optimal solution tree below n' (line 42). Search continues either with a *propagation* step (if conditions are met) or with an *expansion* step. Clearly, since the size of the context minimal AND/OR search graph is bounded exponentially by the induced width of the primal graph, it follows that:

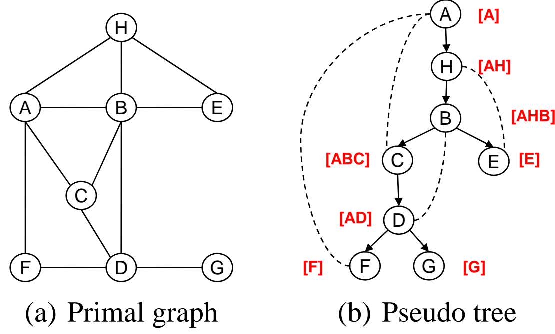
THEOREM **3 (complexity)** *AOBB-C traversing the context minimal AND/OR search graph relative to a pseudo tree $\mathcal{T}$ is sound and complete. Its time and space complexity is $O(n \cdot k^{w^*})$, where w^* is the induced width of the pseudo tree and k bounds the domain size.*

The space required by AOBB-C can sometimes be prohibitive. We next present two caching schemes that can adapt to the memory limitations. They use a parameter called *cache bound* (or simply j-bound) to control the amount of memory used for storing identical nodes.

4.1 Naive Caching

The first scheme, called *naive caching* and denoted by AOBB-C(j), stores nodes at the variables whose context size is smaller than or equal to the cache bound j. It is easy to see that, when j equals the induced width of the pseudo tree, the algorithm explores the context minimal AND/OR graph via full caching.

As we mentioned earlier, a straightforward way of implementing the caching scheme



(a) Primal graph (b) Pseudo tree

Fig. 2. An example of a primal graph and its pseudo tree.

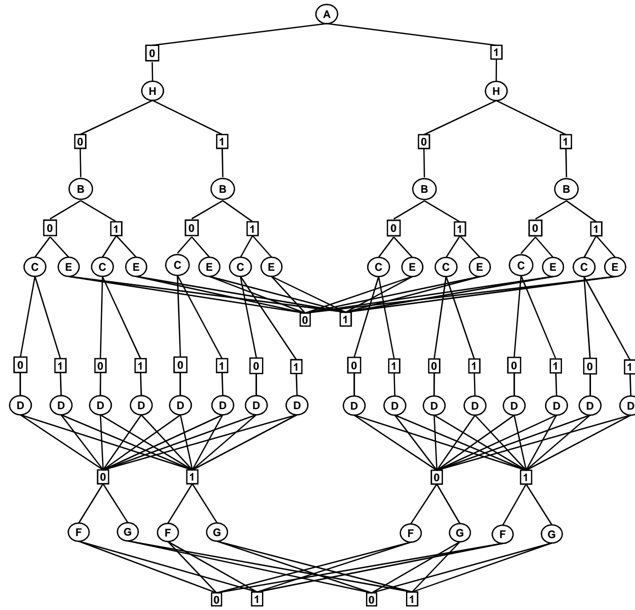


Fig. 3. Illustration of naive caching used by `AOBB-C(2)` on the problem from Fig. 2.

is to have a *cache table* for each variable X_k recording the context. Specifically, lets assume that the context of X_k is $context(X_k) = \{X_1, ..., X_k\}$ and $|context(X_k)| \leq j$. A cache table entry corresponds to a particular instantiation $\{x_1, ..., x_k\}$ of the variables in $context(X_k)$ and records the minimal cost solution to the subproblem rooted at the AND node labeled $\langle X_k, x_k \rangle$.

However, some tables might never get cache hits. These *dead-caches* [6,18] appear at nodes that have only one incoming arc in the context minimal graph. `AOBB-C`(j) needs to record only nodes that are likely to have additional incoming arcs, and some of these nodes can be determined by inspecting the pseudo tree (for example, when the context of a node does not include that of its parent).

Example 2 *Figure 3 displays the AND/OR search graph obtained with the naive caching scheme* `AOBB-C(2)`*, relative to the pseudo tree given in Figure 2(b). Notice that there is no need to create cache tables for variables H and B, because their AND contexts include those of their respective parents in the pseudo tree, namely*

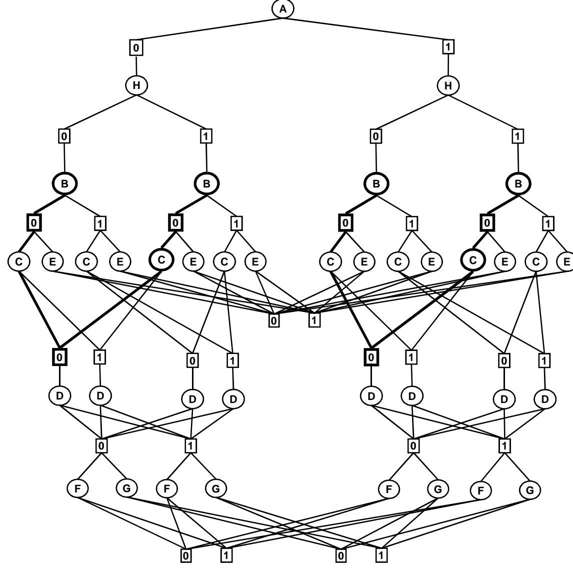


Fig. 4. Illustration of adaptive caching used by `AOBB-AC(2)` on the problem from Fig. 2.

$context(A) \subseteq context(H) \subseteq context(B)$, respectively. Moreover, `AOBB-C(2)` *does not cache any of the AND nodes corresponding to variable C because its corresponding cache table, which is defined on 3 variables ($e.g.$, A, B and C), cannot be stored in memory.*

4.2 Adaptive Caching

The second scheme, called *adaptive caching* and denoted by `AOBB-AC`(j), is inspired by the AND/OR cutset conditioning scheme and was first explored in [23]. It extends the naive scheme by allowing caching even at nodes with contexts larger than the given cache bound, based on *adjusted contexts*.

Specifically, consider the node X_k in the pseudo tree $\mathcal{T}$ with $context(X_k) = \{X_1, ..., X_k\}$, where $k > j$. During search, when variables $\{X_1, ..., X_{k-j}\}$ are instantiated, they can be viewed as part of a cutset. The problem rooted by X_{k-j+1} can be solved in isolation, like a subproblem in the cutset scheme, after variables $X_1, ..., X_{k-j}$ are assigned their current values in all the functions. In this subproblem, conditioned on the values $\{x_1, ..., x_{k-j}\}$, $context(X_k) = \{X_{k-j+1}, ..., X_k\}$ (we call this the *adjusted context* of X_k), so it can be cached within j-bounded space. However, when `AOBB-AC`(j) retracts to variable X_{k-j} or above, the cache table for variable X_k needs to be purged, and will be used again when a new subproblem rooted at X_{k-j+1} is solved. This caching scheme requires only a linear increase in additional memory, compared to the naive `AOBB-C`(j), but it has the potential of exponential time savings, as shown in [23].

Example 3 *Figure 4 shows the AND/OR graph traversed using the adaptive caching*

scheme `AOBB-AC(2)`. *In contrast to the naive scheme displayed in Figure 3,* `AOBB-AC(2)` *caches the AND level corresponding to variable C based on its adjusted context. The adjusted AND context of C is $\{C, B\}$ and a flag is installed at variable A, indicating that the cache table must be purged whenever A is instantiated to a different value.*

5 Best-First AND/OR Search

We now direct our attention to a *best-first* control strategy for traversing the context minimal AND/OR graph. The best-first search algorithm uses similar amounts of memory as the depth-first AND/OR Branch-and-Bound with full caching and therefore the comparison is warranted.

Best-first search expands the nodes in order of their heuristic evaluation function. Its main virtue is that it never expands nodes whose cost is beyond the optimal one, unlike depth-first search algorithms, and therefore is superior among memory intensive algorithms employing the same heuristic evaluation function [9].

Best-First AND/OR search, denoted by `AOBF-C`, that traverses the context minimal AND/OR search graph is described in Algorithm 3. It specializes Nilsson's $\texttt{AO}^*$ algorithm [19] to AND/OR search spaces for graphical models and interleaves forward expansion of the best partial solution tree (`EXPAND`) with a cost revision step (`REVISE`) that updates node values, as detailed in [19]. The explicated AND/OR search graph is maintained by a data structure called $C'_{\mathcal{T}}$, the current node is n, s is the root of the search graph and the current best partial solution subtree is denoted by T'. The children of the current node are denoted by $succ(n)$.

First, a top-down, graph-growing operation finds the best partial solution tree by tracing down through the marked arcs of the explicit AND/OR search graph $C'_{\mathcal{T}}$ (lines 4–10). These previously computed marks indicate the current best partial solution tree from each node in $C'_{\mathcal{T}}$. Before the algorithm terminates, the best partial solution tree, T', does not yet have all of its leaf nodes terminal. One of its non-terminal leaf nodes n is then expanded by generating its successors, depending on whether it is an OR or an AND node. If n is an OR node, labeled X_i, then its successors are AND nodes represented by the values x_i in variable X_i's domain (lines 12–21). Notice that when expanding an OR node, the algorithm does not generate AND children that are already present in the explicit search graph $C'_{\mathcal{T}}$, but rather links to them. All these identical AND nodes in $C'_{\mathcal{T}}$ are easily recognized based on their contexts. Each OR-to-AND arc is associated with the appropriate weight (see Definition 6). Similarly, if n is an AND node, labeled $\langle X_i, x_i \rangle$, then its successors are OR nodes labeled by the child variables of X_i in $\mathcal{T}$ (lines 22–26). There are no weights associated with AND-to-OR arcs. Moreover, a heuristic underestimate $h(n')$ of $v(n')$ is assigned to each of n's successors $n' \in succ(n)$.

Algorithm 3: AOBF-C: Best-First AND/OR Graph Search

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, pseudo tree $\mathcal{T}$ rooted at X_1, parent separator sets pas_i (AND-context) for every variable X_i, heuristic function $h(n)$.
Output: Minimal cost solution and an optimal solution assignment.

```
1  create an OR node s labeled X_1                                         // Initialize
2  v(s) ← h(s); C'_T ← {s}
3  while s is not labeled SOLVED do
4      S ← {s}; T' ← {};                       // Create the marked partial solution tree
5      while S ≠ ∅ do
6          n ← top(S); remove n from S
7          T' ← T' ∪ {n}
8          let L be the set of marked successors of n
9          if L ≠ ∅ then
10             add L on top of S
11     let n be any nonterminal tip node of the marked T' (rooted at s)     // EXPAND
12     if n is an OR node, labeled X_i then
13         foreach x_i ∈ D_i do
14             let n' be the AND node in C'_T having context equal to pas_i
15             if n' == NULL then
16                 create an AND node n' labeled ⟨X_i, x_i⟩
17                 v(n') ← h(n')
18                 w(n, n') ← Σ_{f ∈ B_T(X_i)} f(asgn(π_n))
19                 if n' is TERMINAL then
20                     label n' as SOLVED
21             succ(n) ← succ(n) ∪ {n'}
22     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
23         foreach X_j ∈ children_T(X_i) do
24             create an OR node n' labeled X_j
25             v(n') ← h(n')
26             succ(n) ← succ(n) ∪ {n'}
27     C'_T ← C'_T ∪ {succ(n)}
28     S ← {n}                                                              // REVISE
29     while S ≠ ∅ do
30         let m be a node in S such that m has no descendants in C'_T still in S; remove m from S
31         if m is an AND node, labeled ⟨X_i, x_i⟩ then
32             v(m) ← Σ_{m_j ∈ succ(m)} v(m_j)
33             mark all arcs to the successors
34             label m as SOLVED if all its children are labeled SOLVED
35         else if m is an OR node, labeled X_i then
36             v(m) = min_{m_j ∈ succ(m)} (w_(m,m_j) + v(m_j))
37             mark the arc through which this minimum is achieved
38             label m as SOLVED if the marked successor is labeled SOLVED
39         if m changes its value or m is labeled SOLVED then
40             add to S all those parents of m such that m is one of their successors through a marked arc.
41     return v(s)                                                          // Search terminates
```

The second operation in AOBF-C is a bottom-up, cost revision, arc marking, SOLVE-labeling procedure (lines 28–40). It aims at updating the evaluation function of any subtree that might be affected, and marks the best one. Starting with the node just expanded n, the procedure revises its value $v(n)$, using the newly computed values of its successors, and marks the outgoing arcs on the estimated best path to terminal nodes. This revised value is then propagated upwards in the graph. The revised value $v(n)$ is an updated lower bound on the cost of an optimal solution to the sub-problem rooted at n. If we assume the monotone restriction on h, cost revisions can

only be cost increases [24,19]. Therefore, not all ancestors need have cost revisions, but only those ancestors having best partial solution trees containing descendants with revised values (lines 39–40). During the bottom-up step, `AOBF-C` labels an AND node as `SOLVED` if all of its OR child nodes are solved, and labels an OR node as `SOLVED` if its marked AND child is also solved. The algorithm terminates with the optimal solution when the root node s is labeled `SOLVED`.

If $h(n) \leq v(n)$, the exact cost at n, for all nodes, and if h satisfies the monotone restriction, then algorithm `AOBF-C` will terminate with an optimal solution tree [24,19]. The optimal solution tree can be obtained by tracing down from s through the marked connectors at termination and its optimal cost is equal to the value $v(s)$ of s at termination. Since the algorithm explores every node in the context minimal graph just once, it is the case that:

THEOREM **4 (complexity)** *The best-first AND/OR search algorithm traversing the context minimal AND/OR graph has time and space complexity of $O(n \cdot k^{w^*})$, where w^* is the induced width of the pseudo tree and k bounds the domain size.*

AOBB versus AOBF. We highlight next the main differences between depth-first AND/OR Branch-and-Bound (`AOBB-C`) and best-first AND/OR search (`AOBF-C`) traversing the context minimal AND/OR search graph.

First, `AOBF-C` with the same heuristic function as `AOBB-C` is likely to expand the smallest number of nodes [9], but empirically this depends on how quickly `AOBB-C` will find an optimal solution that it will use as upper bound. Secondly, `AOBB-C` can use far less memory by avoiding dead-caches for example ($e.g.$, when the search graph is a tree), while `AOBF-C` has to keep the explicated search graph in memory. Third, `AOBB-C` can be used as an anytime scheme, namely whenever interrupted, the algorithm outputs the best solution found so far, unlike `AOBF-C` which outputs a complete solution upon termination only. All the above points show that the relative merit of best-first versus depth-first over context minimal AND/OR search spaces cannot be determined by sheer theory [9] and therefore empirical evaluation is essential.

6 Overview of the Mini-Bucket Lower Bound Heuristics for AND/OR Search

The effectiveness of both depth-first AND/OR Branch-and-Bound and best-first AND/OR search algorithms greatly depends on the quality of the heuristic evaluation functions. The primary heuristic that we used in our experiments is the Mini-Bucket heuristic, which we presented in [1,2]. For completeness, we review it briefly next.

Mini-Bucket Elimination (MBE(i)) [11] is an approximation algorithm designed

to avoid the high time and space complexity of *Bucket Elimination* (BE) [25], by partitioning large buckets into smaller subsets, called *mini-buckets*, each containing at most i (called i-bound) distinct variables. The mini-buckets are then processed separately. The algorithm outputs not only a bound on the optimal solution cost, but also a collection of augmented buckets, which form the basis for the heuristics generated. The complexity is time and space $O(exp(i))$. Both Bucket and Mini-Bucket Elimination can also be viewed as message passing from leaves to root along a *bucket tree* [17].

Static Mini-Bucket Heuristics. In [1,2,10] we showed that the intermediate functions generated by MBE(i) can be used to compute a heuristic function that underestimates the minimal cost solution to the current subproblem. Specifically, given an ordered set of augmented buckets $\{B(X_1), ..., B(X_n)\}$ generated by MBE(i) along the bucket tree $\mathcal{T}$ (which is also a pseudo tree [18]), and given a node n in the AND/OR search tree, the *static mini-bucket heuristic* function $h(n)$ is computed as follows: (1) if n is an AND node labeled $\langle X_p, x_p \rangle$, then $h(n)$ is the sum of all intermediate functions that were generated in buckets corresponding to the descendents of X_p in $\mathcal{T}$ and reside in bucket $B(X_p)$ or the buckets corresponding to the ancestors of X_p in $\mathcal{T}$; (2) if n is an OR node labeled by X_p, then $h(n) = min_m(w(n,m) + h(m))$, where m is the AND child of n labeled with value x_p of X_p.

Dynamic Mini-Bucket Heuristics. It is also possible to generate the mini-bucket heuristic information dynamically, during search. The idea is to compute MBE(i) conditioned on the current partial assignment [1,2]. Specifically, given a bucket tree $\mathcal{T}$, with buckets $\{B(X_1), ..., B(X_n)\}$, a node n in the AND/OR search tree and given the current partial assignment $asgn(\pi_n)$ along the path to n, the *dynamic mini-bucket heuristic* function $h(n)$ is computed as follows: (1) if n is an AND node labeled $\langle X_p, x_p \rangle$, then $h(n)$ is the sum of the intermediate functions that reside in bucket $B(X_p)$ and were generated by MBE(i), conditioned on $asgn(\pi_n)$, in the buckets corresponding to the descendants of X_p in $\mathcal{T}$; (2) if n is an OR node labeled X_p, then $h(n) = min_m(w(n,m) + h(m))$, where m is the AND child of n labeled with value x_p of X_p. Given an i-bound, the dynamic mini-bucket heuristic implies a much higher computational overhead compared with the static version. However, the bounds generated dynamically may be far more accurate since some of the variables are assigned and will therefore yield smaller functions and less partitioning.

7 Experimental Results

In [1,2] we evaluated empirically AND/OR search algorithms for AND/OR *trees* only. We now extend this evaluation to algorithms presented in this paper exploring the context minimal AND/OR search graphs. As in [1,2], we have conducted a num-

ber of experiments on the two common optimization problems classes in graphical models: finding the Most Probable Explanation in Bayesian networks and solving Weighted CSPs. We implemented our algorithms in C++ and ran all experiments on a 2.4GHz Pentium IV with 2GB of RAM, running Windows XP.

7.1 Overview and Methodology

Algorithms We evaluated the following classes of memory intensive AND/OR search algorithms:

- Depth-first AND/OR Branch-and-Bound search algorithms with full caching, using static and dynamic mini-bucket heuristics, denoted by `AOBB-C+SMB(`i`)` and `AOBB-C+DMB(`i`)`, respectively.
- Best-first AND/OR search algorithms using static and dynamic mini-bucket heuristics, denoted by `AOBF-C+SMB(`i`)` and `AOBF-C+DMB(`i`)`, respectively.

We compare these algorithms with those traversing the AND/OR search tree (without caching), denoted by `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)`, introduced in [1,2]. In addition, we also ran the traditional OR Branch-and-Bound search algorithms with full caching, denoted by `BB-C+SMB(`i`)` and `BB-C+DMB(`i`)`, respectively. In all cases, the parameter i represents the mini-bucket i-bound and controls the accuracy of the heuristic.

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

1 The impact of graph versus tree on AND/OR Branch-and-Bound search.
2 The impact of best-first versus depth-first AND/OR search regimes.
3 The impact of the mini-bucket i-bound.
4 The impact of the cache bound j on naive and adaptive caching.
5 The impact of the pseudo tree quality on AND/OR search.
6 The impact of determinism present in the network.
7 The impact of non-trivial initial upper bounds.

MPE Task for Bayesian Networks We tested the performance of the depth-first AND/OR Branch-and-Bound and best-first AND/OR search algorithms on the following types of problems [2]: random coding networks, grid networks, Bayesian networks derived from the ISCAS'89 digital circuits benchmark, genetic linkage analysis networks and Bayesian networks used in the UAI'06 Inference Evaluation contest. We report here in detail the results obtained for grid networks and genetic

[2] Available online at *http://graphmod.ics.uci.edu/group/Repository*

linkage analysis networks only, but we summarize the results over the entire set of benchmarks, and refer the reader to [26,27] for additional details.

In our experiments, we also consider an extension of the AND/OR Branch-and-Bound with caching that exploits the determinism present in the Bayesian network by constraint propagation. For reference, we also compared with the SAMIAM version 2.3.2 software package [3]. SAMIAM contains an implementation of Recursive Conditioning [6] which can also be viewed as an AND/OR search algorithm. It uses a context-based caching mechanism similar to our scheme. This version of recursive conditioning also explores a context minimal AND/OR search graph [18] and therefore its space complexity is exponential in the treewidth. Note that when we use mini-bucket heuristics with high values of i, we use space exponential in i for the heuristic calculation and storing, in addition to the space required for caching.

Weighted CSPs We evaluated the algorithms on: scheduling problems from the SPOT5 benchmark, networks derived from the ISCAS'89 digital circuits and instances of the popular game of Mastermind. We report here detailed results for SPOT5 problem instances and Mastermind game instances only. We also provide a summary of the results obtained on the other types of problems, and refer the reader to [26,27] for the full results.

For reference, we also report results obtained with the state-of-the-art solvers called `toolbar` [28] and `toolbar-BTD` [29] [4]. `toolbar` is an OR Branch-and-Bound algorithm that maintains during search a form of soft local consistency called Existential Directional Arc Consistency (EDAC). `toolbar-BTD` extends the *Backtracking with Tree Decomposition* (BTD) algorithm [8] and computes the guiding heuristic information as well by enforcing EDAC during search. It can be shown that BTD explores a context minimal AND/OR search graph, relative to a pseudo tree corresponding to the given tree decomposition. In addition, we also compare with the depth-first AND/OR Branch-and-Bound tree search algorithms with EDAC heuristics and with variable orderings such as: AOEDAC+PVO using partial variable orderings, DVO+AOEDAC using full dynamic variable ordering, and AOEDAC+DSO using dynamic separator orderings, respectively. For a detailed description of these ordering heuristics and their evaluation, see [1,3].

The dynamic variable ordering heuristic used by the OR and AND/OR Branch-and-Bound algorithms with EDAC heuristics was the *min-dom/ddeg* heuristic, which selects the variable with the smallest ratio of the domain size divided by the future degree. Ties were broken lexicographically.

[3] Available at http://reasoning.cs.ucla.edu/samiam. We used the `batchtool 1.5` provided with the package.
[4] Available at: *http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP*

Measures of Performance In all our experiments we report the CPU time in seconds and the number of nodes visited for solving the problems. We also specify the problems' parameters such as the number of variables (n), number of evidence variables (e), maximum domain size (k), the induced width (w^*) and depth (h) of the pseudo trees. When evidence is asserted in the network, w^* and h are computed after the evidence nodes were removed from the graph. We also report the time required by the Mini-Bucket algorithm MBE(i) to pre-compile the heuristic information. The pseudo trees that guide the AND/OR search algorithms were generated using the min-fill and hypergraph partitioning heuristics described in [1,6]. In our experiments we ran the min-fill heuristic just once and broke the ties lexicographically. The best performance points are highlighted. In each table, "-" denotes that the respective algorithm exceeded the time limit. Similarly, "out" indicates that the 2GB memory limit was exceeded.

7.2 *Results for Empirical Evaluation of Bayesian Networks*

Our results reported in [1] demonstrated conclusively that the AND/OR Branch-and-Bound *tree* search algorithms with static mini-bucket heuristics were the best performing algorithms on this domain when compared with traditional OR search algorithms. The difference between `AOBB+SMB(`i`)` and the OR tree search counterpart `BB+SMB(`i`)` was more pronounced at relatively small i-bounds (corresponding to relatively weak heuristic estimates) and amounted to two orders of magnitude in terms of both running time and size of the search space explored. For larger i-bounds, when the heuristic estimates are strong enough to prune the search space substantially, the difference between AND/OR and OR Branch-and-Bound tree search decreased. We also showed that `AOBB+SMB(`i`)` was in many cases able to outperform dramatically the current state-of-the-art solvers for belief networks such as SAMIAM and SUPERLINK (for genetic linkage analysis). The AND/OR Branch-and-Bound with dynamic mini-bucket heuristics `AOBB+DMB(`i`)` proved competitive only for relatively small i-bounds due to the computational overhead. In this section we extend the empirical evaluation to memory intensive depth-first and best-first AND/OR search algorithms.

7.2.1 *Grid Networks*

In random grid networks, the nodes are arranged in an $N \times N$ square and each CPT is generated uniformly randomly. We experimented with problem instances initially developed by [30] for the task of weighted model counting. For these problems N ranges between 10 and 38, and, for each instance, 90% of the CPTs are deterministic, namely they contain only 0 and 1 probability entries. All the variables are bi-valued.

Table 1
CPU time in seconds and nodes visited for solving **grid networks** using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour. The two horizontal blocks of the table show different ranges of the mini-bucket i-bounds.

min-fill pseudo tree

grid (w*, h) (n, e)	SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16	
	time	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.02		0.03		0.03		0.06		0.06	
90-10-1		0.23	3,297	0.06	373	**0.05**	102	0.06	102	0.06	102
(13, 39)	0.13	0.33	8,080	0.11	2,052	**0.05**	101	0.06	101	0.06	101
(100, 0)		0.14	2,638	0.06	819	**0.05**	101	0.06	101	0.06	101
		0.27	2,012	0.11	661	**0.05**	100	0.06	100	0.06	100
		0.03		0.03		0.08		0.14		0.44	
90-14-1		126.69	1,233,891	121.00	1,317,992	1.52	16,547	0.42	2,770	0.61	1,450
(22, 66)	11.97	8.00	130,619	6.59	100,696	1.06	17,479	0.33	3,321	0.61	2,938
(196, 0)		4.22	55,120	3.66	48,513	0.45	5,585	**0.23**	1,361	0.53	1,210
		3.20	18,796	2.70	15,764	0.55	2,899	0.30	898	0.63	857
		0.05		0.05		0.11		0.31		0.63	
90-16-1		-	-	-	-	40.05	345,255	2.38	16,942	1.23	5,327
(24, 82)	147.19	666.68	10,104,350	173.49	2,600,690	14.36	193,440	2.97	39,825	2.08	23,421
(256, 0)		209.60	2,695,249	35.45	441,364	4.23	50,481	1.19	11,029	**0.95**	4,810
		25.70	126,861	10.59	54,796	4.47	22,993	1.42	6,015	1.22	3,067

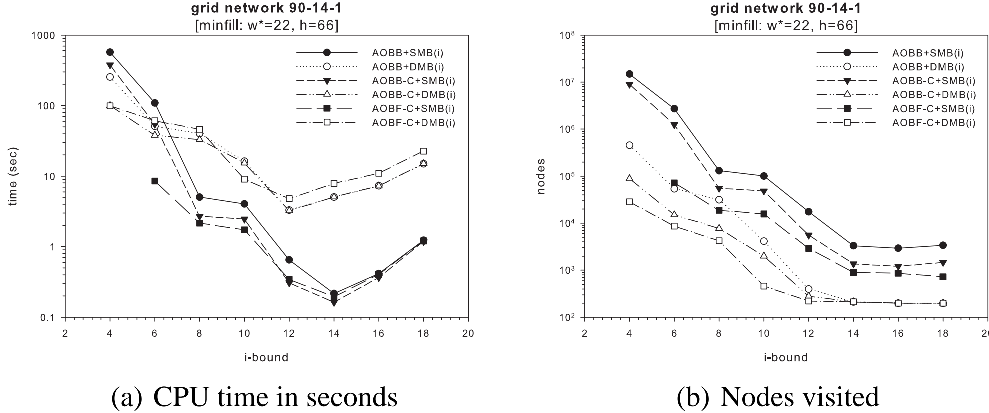
grid (w*, h) (n, e)	SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=18		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20	
	time	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.28		0.64		1.69		4.60		19.14	
90-24-1		-	-	-	-	-	-	-	-	-	-
(33, 111)	out	-	-	2338.67	24,117,151	1548.09	18,238,983	138.67	1,413,764	146.85	1,308,009
(576, 20)		-	-	1273.09	9,047,518	596.27	4,923,760	70.42	473,675	74.99	412,291
		out		21.94	75,637	10.59	33,770	**6.06**	5,144	23.80	17,291
		0.33		0.72		2.14		7.09		22.02	
90-26-1		-	-	-	-	395.67	1,635,447	-	-	67.09	277,685
(36, 113)	out	311.89	2,903,489	369.49	3,205,257	8.42	59,055	22.99	165,182	22.56	5,777
(676, 40)		146.97	878,874	152.80	962,484	4.36	15,632	12.92	46,489	22.13	2,242
		19.06	65,271	24.39	79,619	**4.27**	7,190	8.05	3,777	22.44	1,435
		0.47		0.98		2.77		7.98		30.44	
90-30-1		-	-	-	-	-	-	-	-	-	-
(43, 150)	out	1131.07	9,445,224	386.27	3,324,942	350.28	3,039,966	149.69	1,358,569	97.09	485,300
(900, 60)		652.15	3,882,300	165.74	1,070,823	155.20	956,837	40.14	212,963	59.28	174,715
		158.97	534,385	46.73	157,187	47.27	154,496	**21.06**	45,201	57.97	100,800
		0.63		1.25		3.72		11.66		40.00	
90-34-1		-	-	-	-	-	-	-	-	-	-
(45, 153)	out	-	-	-	-	-	-	-	-	478.10	1,549,829
(1154, 80)		-	-	-	-	-	-	-	-	369.36	823,604
		out		out		243.63	596,978	270.88	667,013	**71.19**	67,611
		0.78		1.67		4.20		12.36		43.69	
90-38-1		-	-	-	-	-	-	-	-	-	-
(47, 163)	out	2032.33	6,835,745	-	-	807.38	2,850,393	568.69	2,079,146	369.31	1,038,065
(1444, 120)		969.02	2,623,971	1753.10	3,794,053	203.67	614,868	165.45	488,873	113.06	214,919
		101.69	174,786	103.80	146,237	54.00	95,511	**53.44**	78,431	73.10	59,856

Table 2
CPU time in seconds and nodes visited for solving **grid networks** using **dynamic mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour. The two horizontal blocks of the table show different ranges of the mini-bucket i-bounds. Grid instances **90-30-1**, **90-34-1** and **90-38-1** could not be solved within the time limit.

min-fill pseudo tree										
grid	BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)	
	AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)	
(w*, h)	AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)	
(n, e)	AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)	
	i=8		i=10		i=12		i=14		i=16	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-10-1	0.66	303	0.47	197	0.33	102	0.41	102	0.38	102
(13, 39)	0.31	344	0.28	241	0.25	101	0.30	101	0.28	101
(100, 0)	0.28	235	0.25	170	**0.23**	101	0.28	101	0.30	101
	0.39	135	0.36	115	0.36	100	0.41	100	0.41	100
90-14-1	128.92	16,176	37.34	2,590	7.44	340	8.61	211	11.72	199
(22, 66)	56.66	31,476	23.61	4,137	4.69	397	7.25	211	10.19	199
(196, 0)	46.94	7,641	22.72	1,996	**4.67**	281	7.20	211	10.19	199
	54.09	4,007	12.84	462	6.83	221	11.94	211	16.05	199
90-16-1	639.91	42,786	388.47	12,563	112.44	1,913	103.14	1,017	39.16	262
(24, 82)	975.58	462,180	296.76	47,121	70.81	3,227	50.36	719	25.03	260
(256, 0)	382.78	44,949	245.50	11,855	65.41	1,430	48.61	525	**24.52**	260
	194.08	11,453	252.99	6,622	94.88	1,061	75.41	413	38.46	258
grid	BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)	
	AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)	
(w*, h)	AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)	
(n, e)	AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)	
	i=12		i=14		i=16		i=18		i=20	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-24-1	-	-	-	-	2586.38	3,243	1724.68	700	2368.83	601
(33, 111)	-	-	-	-	1367.38	2,739	1979.42	1,228	1696.56	598
(576, 20)	-	-	-	-	**781.21**	1,058	1211.99	788	1693.00	598
	3456.77	11,818	1834.71	2,728	1153.48	855	1871.03	759	2573.08	591
90-26-1	-	-	-	-	-	-	-	-	-	-
(36, 113)	-	-	-	-	1514.18	2,545	2889.49	1,191	-	-
(676, 40)	2801.39	35,640	2593.74	10,216	**892.88**	1,178	1698.70	861	2647.60	687
	1262.76	5,392	1737.01	2,585	1347.54	1,049	2587.10	828	-	-

Tables 1 and 2 show detailed results for experiments with 8 grids of increasing difficulty, using static and dynamic mini-bucket heuristics. The columns are indexed by the mini-bucket i-bound. Each table is organized into two horizontal blocks, each corresponding to a different range of i-bound values. For each instance we ran a single MPE query with e nodes picked randomly and instantiated as evidence. The guiding pseudo trees were generated using the min-fill heuristic.

Tree vs. graph AOBB. First, we observe that AOBB-C+SMB(i) using full caching improves significantly over the tree version of the algorithm, especially for relatively small i-bounds which generate relatively weak heuristic estimates. For example, on the 90-16-1 grid in Table 1, AOBB-C+SMB(8) is 3 times faster than AOBB+SMB(8) and explores a search space 5 times smaller. Notice also the significant additional reduction produced by the best-first search algorithm AOBF-C+SMB(8). While overall AOBF-C+SMB(i) is superior to AOBB-C+SMB(i) with the same i-bound, the best performance on this network is obtained by AOBB-C+SMB(16).



(a) CPU time in seconds (b) Nodes visited

Fig. 5. Comparison of the impact of static and dynamic mini-bucket heuristics. Shown are the CPU time in seconds (a) and the number of nodes visited (b) on the **`90-14-1` grid network** from Tables 1 and 2, respectively.

The algorithm is two times faster than the cache-less `AOBB+SMB(16)`, and 155 times faster than SAMIAM, respectively. When looking at the algorithms using dynamic mini-bucket heuristics (Table 2) we observe a similar pattern, namely the graph search AND/OR Branch-and-Bound algorithm improves sometimes significantly over the tree search one. For instance, on the `90-24-1` grid, `AOBB-C+DMB(16)` is about two times faster than `AOBB+DMB(16)`. Notice also that the AND/OR algorithms with dynamic mini-buckets could not solve the last 3 test instances due to exceeding the time limit. The OR Branch-and-Bound search algorithms with caching `BB-C+SMB(`i`)` (resp. `BB-C+DMB(`i`)`) are inferior to the AND/OR Branch-and-Bound graph search, especially on the harder instances ($e.g.$, `90-30-1`).

AOBF vs. AOBB. When comparing further the best-first and depth-first search algorithms, we see again the superiority of `AOBF-C+SMB(`i`)` over `AOBB-C+SMB(`i`)`, especially for relatively weak heuristic estimates (see also Figure 5). For example, on the `90-38-1` grid, one of the hardest instances, best-first search with the smallest reported i-bound ($i = 12$) is 9 times faster than `AOBB-C+SMB(12)` and visits 15 times less nodes. The difference between best-first and depth-first search is not too prominent when using dynamic mini-bucket heuristics, perhaps because these heuristics are far more accurate than the pre-compiled ones yielding a small enough search space.

Static vs. dynamic mini-bucket heuristics. When comparing the static versus dynamic mini-bucket heuristics, we see as before, that the former are more powerful for relatively large i-bounds, whereas the latter are cost effective only for relatively small i-bounds. Figures 5(a) and 5(b) plot the CPU time and size of the search space explored, as a function of the mini-bucket i-bound, on the `90-14-1` grid from Tables 1 and 2, respectively. Focusing on `AOBB-C+SMB(`i`)`, for example, we see that its running time, as a function of i, forms a U-shaped curve. At first ($i = 4$) it is high, then as the i-bound increases the total time decreases (when $i = 14$ the time is 0.23), but then as i increases further the time starts to increase again because the

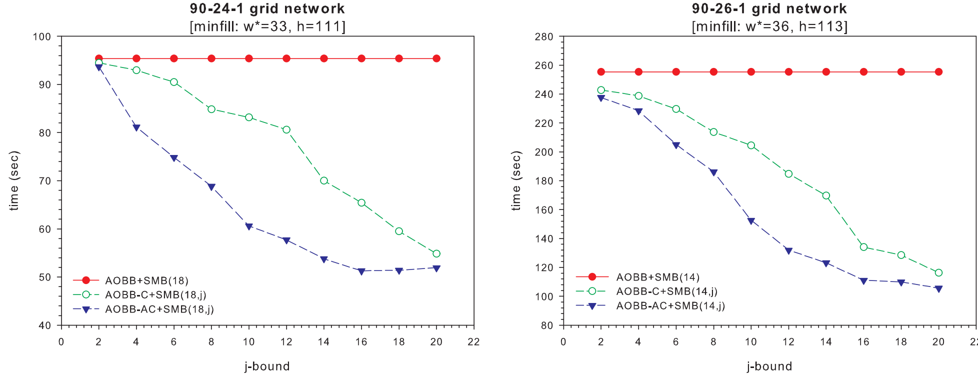


Fig. 6. Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **grid networks**. Shown is the CPU time in seconds.

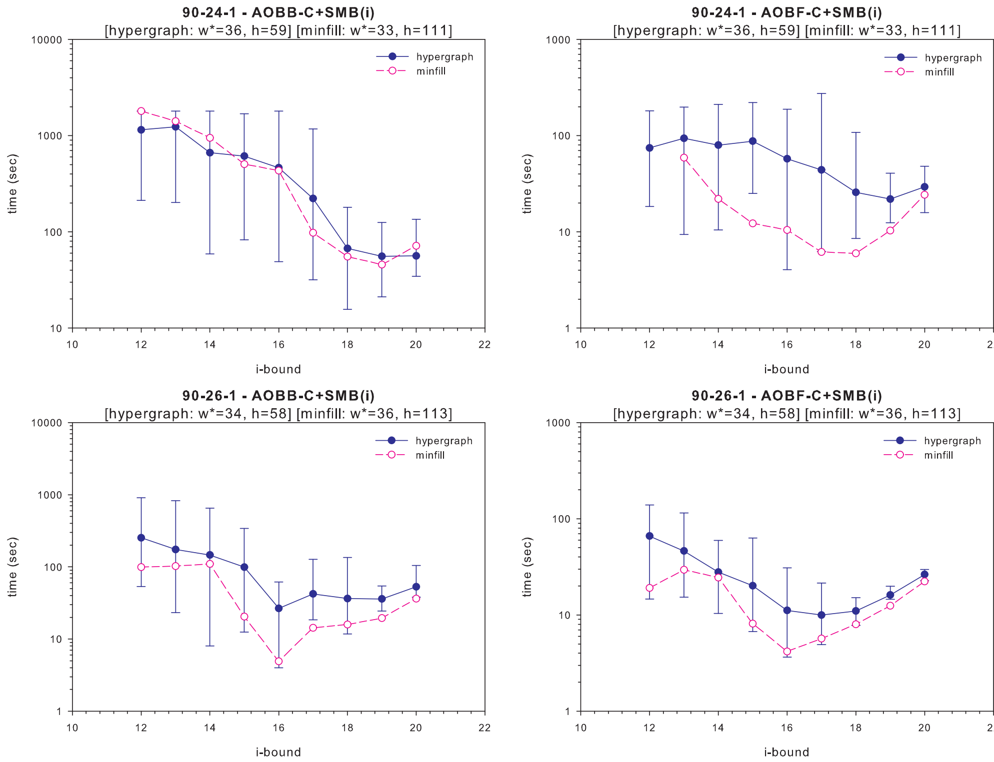


Fig. 7. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **grid networks** with `AOBB-C+SMB(`i`)` (left) and `AOBF-C+SMB(`i`)` (right). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

pre-processing time of the mini-bucket heuristic outweighs the search time. The same behavior can be observed in the case of dynamic mini-buckets as well.

Impact of the caching level. Figure 6 compares the naive (`AOBB-C+SMB(`i,j`)`) and adaptive (`AOBB-AC+SMB(`i,j`)`) caching schemes, in terms of CPU time, on two grid networks from Table 1. In each test case we chose a relatively small mini-

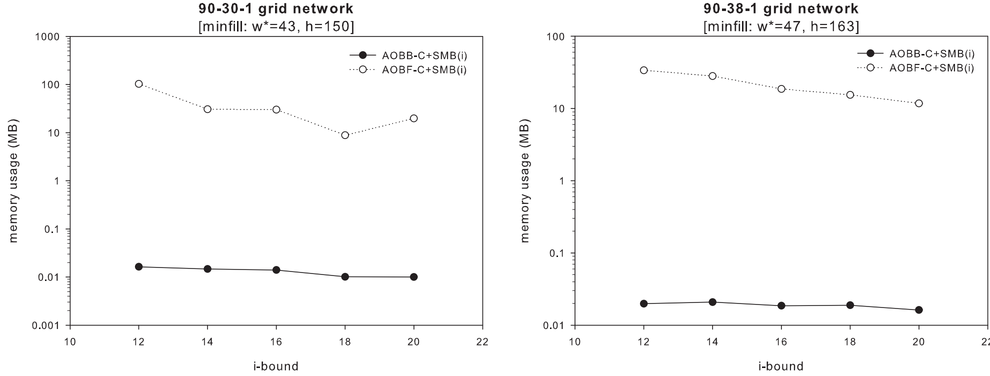


Fig. 8. Memory usage by AOBB-C+SMB(i) and AOBF-C+SMB(i) on **grid networks**.

bucket i-bound and varied the cache bound j (the X axis) from 2 to 20. We see that adaptive caching improves significantly over the naive scheme especially for relatively small j-bounds. This may be important because small j-bounds mean restricted space. For large j-bounds the two schemes are identical and approach full caching.

Impact of the pseudo tree. Since the hypergraph partitioning heuristic uses a non-deterministic algorithm, the runtime of the AND/OR search algorithms guided by the resulting pseudo trees may vary significantly from one run to the next. In Figure 7 we display the running time distribution of AOBB-C+SMB(i) (left side of the figure) and AOBF-C+SMB(i) (right side of the figure) using hypergraph based pseudo trees on grids 90-24-1 and 90-26-1, respectively. For each reported i-bound (the X axis), the corresponding data point and error bar represent the average as well as the minimum and maximum running times obtained over 20 independent runs. We also record the average induced width and depth obtained for the hypergraph pseudo trees (see the header of each plot in Figure 7). We see that the hypergraph based pseudo trees, which have far smaller depths, are sometimes able to improve the performance of AOBB-C+SMB(i), especially for relatively small i-bounds ($e.g.$, 90-24-1). For larger i-bounds, the pre-compiled mini-bucket heuristic benefits from the small induced widths obtained with the min-fill ordering. Therefore, AOBB-C+SMB(i) using min-fill based pseudo trees is generally faster (see the different Y scale). We also see that on average AOBF-C+SMB(i) is faster when it is guided by min-fill rather than hypergraph based pseudo trees. This verifies our hypothesis that memory intensive algorithms exploring the AND/OR graph are more sensitive to the context size (which is smaller for min-fill orderings), rather than the depth of the pseudo tree. These results were typical to other instances as well.

Memory usage of AND/OR graph search. Figure 8 displays the memory usage of AOBB-C+SMB(i) and AOBF-C+SMB(i) on grids 90-30-1 and 90-38-1, respectively. We see that the memory requirements of the depth-first algorithm are significantly smaller than those of best-first search. This is because AOBF-C+SMB(i) has to keep in memory the entire search space, unlike AOBB-C+SMB(i) which can

Table 3
CPU time and nodes visited for solving **genetic linkage networks** using **static mini-bucket heuristics**. Time limit 3 hours. Top part of the table shows results for i-bounds between 6 and 14, while the bottom part shows i-bounds between 10 and 18.

min-fill pseudo tree

pedigree (w*, h) (n, d)	Superlink SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=6		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.05		0.05		0.11		0.31		0.97	
ped1		-	-	-	-	1.14	7,997	0.73	3,911	1.31	2,704
(15, 61)	54.73	24.30	416,326	13.17	206,439	1.58	24,361	1.84	25,674	1.89	15,156
(299, 5)	5.44	4.19	69,751	2.17	33,908	0.39	4,576	0.65	6,306	1.36	4,494
		1.30	7,314	2.17	13,784	**0.26**	1,177	0.87	4,016	1.54	3,119
		0.12		0.45		5.38		60.97		out	
ped38		-	-	-	-	-	-	-	-		
(17, 59)	**28.36**	-	-	8120.58	85,367,022	-	-	3040.60	35,394,461		
(582, 5)	out	5946.44	34,828,046	1554.65	8,986,648	2046.95	11,868,672	272.69	1,412,976		
		out		134.41	348,723	216.94	583,401	103.17	242,429		
		0.11		0.74		5.38		37.19		out	
ped50		-	-	-	-	-	-	-	-		
(18, 58)	-	-	-	-	-	476.77	5,566,578	104.00	748,792		
(479, 5)	out	4140.29	28,201,843	2493.75	15,729,294	66.66	403,234	52.11	110,302		
		78.53	204,886	36.03	104,289	**12.75**	25,507	38.52	5,766		

pedigree (w*, h) (n, d)	Superlink SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=18	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.42		2.33		11.33		274.75		out	
ped23		-	-	-	-	76.11	339,125	270.22	74,261		
(27, 71)	9146.19	498.05	6,623,197	15.45	154,676	16.28	67,456	286.11	117,308		
(310, 5)	out	193.78	1,726,897	**10.06**	74,672	13.33	23,557	274.00	62,613		
		out		15.33	58,180	14.36	12,987	out			
		0.67		5.16		21.53		58.59		out	
ped37		-	-	-	-	-	-	-	-		
(21, 61)	64.17	273.39	3,191,218	1682.09	25,729,009	1096.79	15,598,863	128.16	953,061		
(1032, 5)	out	39.16	222,747	488.34	4,925,737	301.78	2,798,044	67.83	82,239		
		29.16	72,868	38.41	102,011	95.27	223,398	62.97	12,296		

save space by avoiding dead-caches for example. Moreover, the nodes cached by AOBB-C+SMB(i) require far less memory because they only record the optimal solution cost below them, whereas the nodes cached by AOBF-C+SMB(i) must store, in addition, the lists of their children in the search graph. For these reasons, we were able throughout the evaluation to run full caching with depth-first search.

7.2.2 Genetic Linkage Analysis

In human genetic linkage analysis [31], the *haplotype* is the sequence of alleles at different loci inherited by an individual from one parent, and the two haplo-

Table 4
CPU time in seconds and nodes visited for solving **genetic linkage networks** using **static mini-bucket** heuristics and min-fill based pseudo trees. Time limit 3 hours.

min-fill pseudo tree											
pedigree (w*, h) (n, d)	Superlink SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=18		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.51		1.42		4.59		12.87		19.30	
ped18		-	-	-	-	-	-	-	-	1515.43	1,388,791
(21, 119)	139.06	-	-	2177.81	28,651,103	270.96	2,555,078	100.61	682,175	20.27	7,689
(1184, 5)	157.05	-	-	406.88	3,567,729	52.91	397,934	23.83	118,869	20.60	2,972
		out		127.41	542,156	42.19	171,039	**19.85**	53,961	19.91	2,027
		1.42		5.11		37.53		410.96		out	
ped20		-	-	-	-	-	-	-	-		
(24, 66)	**14.72**	3793.31	54,941,659	1293.76	18,449,393	1259.05	17,810,674	1080.05	9,151,195		
(388, 5)	out	1983.00	18,615,009	635.74	6,424,477	512.16	4,814,751	681.97	2,654,646		
		out		out		out		out			
		0.34		0.89		3.20		10.46		33.42	
ped25		-	-	-	-	-	-	-	-	-	-
(34, 89)	-	-	-	-	-	9399.28	111,301,168	3607.82	34,306,937	2965.60	28,326,541
(994, 5)	out	-	-	1644.67	12,631,406	865.83	6,676,835	249.47	1,789,094	**236.88**	1,529,180
		out		out		out		out		out	
		0.42		0.83		1.78		5.75		21.30	
ped30		-	-	-	-	-	-	-	-	-	-
(23, 118)	13095.83	-	-	-	-	-	-	214.10	1,379,131	91.92	685,661
(1016, 5)	out	10212.70	93,233,570	8858.22	82,552,957	-	-	34.19	193,436	30.48	66,144
		out		out		out		30.39	72,798	**27.94**	18,795
		0.58		2.31		7.84		33.44		112.83	
ped33		-	-	-	-	-	-	-	-	-	-
(37, 165)	-	2804.61	34,229,495	737.96	9,114,411	3896.98	50,072,988	159.50	1,647,488	2956.47	35,903,215
(581, 5)	out	1426.99	11,349,475	307.39	2,504,020	1823.43	14,925,943	86.17	453,987	1373.90	10,570,695
		out		140.61	407,387	out		**74.86**	134,068	out	
		0.52		2.32		8.41		33.15		81.27	
ped39		-	-	-	-	-	-	-	-	-	-
(23, 94)	322.14	-	-	-	-	4041.56	52,804,044	386.13	2,171,470	141.23	407,280
(1272, 5)	out	-	-	-	-	968.03	7,880,928	61.20	313,496	93.19	83,714
		out		out		68.52	218,925	**41.69**	79,356	87.63	14,479
		4.20		31.33		96.28		out		out	
ped42		-	-	-	-	-	-				
(25, 76)	561.31	-	-	-	-	-	-				
(448, 5)	out	-	-	-	-	2364.67	22,595,247				
		out		out		**133.19**	93,831				

types (maternal and paternal) of an individual constitute this individual's *genotype*. When genotypes are measured by standard procedures, the result is a list of unordered pairs of alleles, one pair for each locus. The *maximum likelihood haplotype* problem consists of finding a joint haplotype configuration for all members of the pedigree which maximizes the probability of data. It can be shown that given the pedigree data, the haplotyping problem is equivalent to computing the most probable explanation of a Bayesian network that represents the pedigree [32,33].

Tables 3 and 4 display the results obtained for 12 hard linkage analysis networks [5]. We report only on search guided by static mini-bucket heuristics. The dynamic mini-bucket heuristics performed very poorly on this domain because of their prohibitively high computational overhead at large i-bounds. For comparison, we include results obtained with SUPERLINK 1.6. SUPERLINK is currently one the most efficient solvers for genetic linkage analysis, is dedicated to this domain, uses a combination of variable elimination and conditioning, and takes advantage of the determinism in the network.

Tree versus graph AOBB. We observe that AOBB-C+SMB(i) improves significantly over AOBB+SMB(i), especially for relatively small i-bounds for which the heuristic estimates are less accurate. On ped25, for example, AOBB-C+SMB(18) is 15 times faster than AOBB+SMB(18) and expands about 20 times fewer nodes. As the i-bound increases the difference between AOBB-C+SMB(i) and AOBB+SMB(i) decreases, as we saw before. Notice that the OR Branch-and-Bound with caching BB-C+SMB(i) and SAMIAM were able to solve only one instance ($e.g.$, ped18).

AOBB vs. AOBF. The overall best performing algorithm on this dataset is best-first AOBF-C+SMB(i), outperforming its competitors on 5 out of the 7 test cases. On ped42, for instance, AOBF-C+SMB(16) is 18 times faster than the depth-first Branch-and-Bound AOBB-C+SMB(16) and explores a search space 240 times smaller. In some test cases ($e.g.$, ped30) the best-first search algorithm was up to 3 orders of magnitude faster than SUPERLINK.

Impact of the pseudo tree. Figure 9 plots the running time distribution of depth-first AOBB-C+SMB(i) (left side of the figure) and best-first AOBF-C+SMB(i) (right side of the figure), guided by hypergraph based pseudo trees, over 20 independent runs on the ped1 and ped33 networks, respectively. In this case, we see that both algorithms perform much better when guided by hypergraph based pseudo trees, especially on harder instances. For instance, on the ped33 network, AOBB-C+SMB(16) using a hypergraph based pseudo tree was able to outperform AOBB-C+SMB(16) guided by a min-fill tree by almost two orders of magnitude. Similarly, AOBF-C+SMB(i) with hypergraph trees was able to solve the problem instance across all i-bounds, unlike AOBB-C+SMB(i) with a min-fill tree which succeeded only for $i \in \{14, 18\}$. Notice that the induced width of this problem along the min-fill order is very large ($w^* = 37$) which causes the mini-bucket heuristics to be relatively weak and implies a large number of dead caches. The results on other problem instances displayed a similar pattern.

Table 5 displays the results obtained for 6 additional linkage analysis networks using hypergraph partitioning based pseudo trees and the min-fill ones. We selected the hypergraph tree having the smallest depth over 100 independent runs. To the best of our knowledge, these networks were never before solved for the maximum

[5] http://bioinfo.cs.technion.ac.il/superlink/

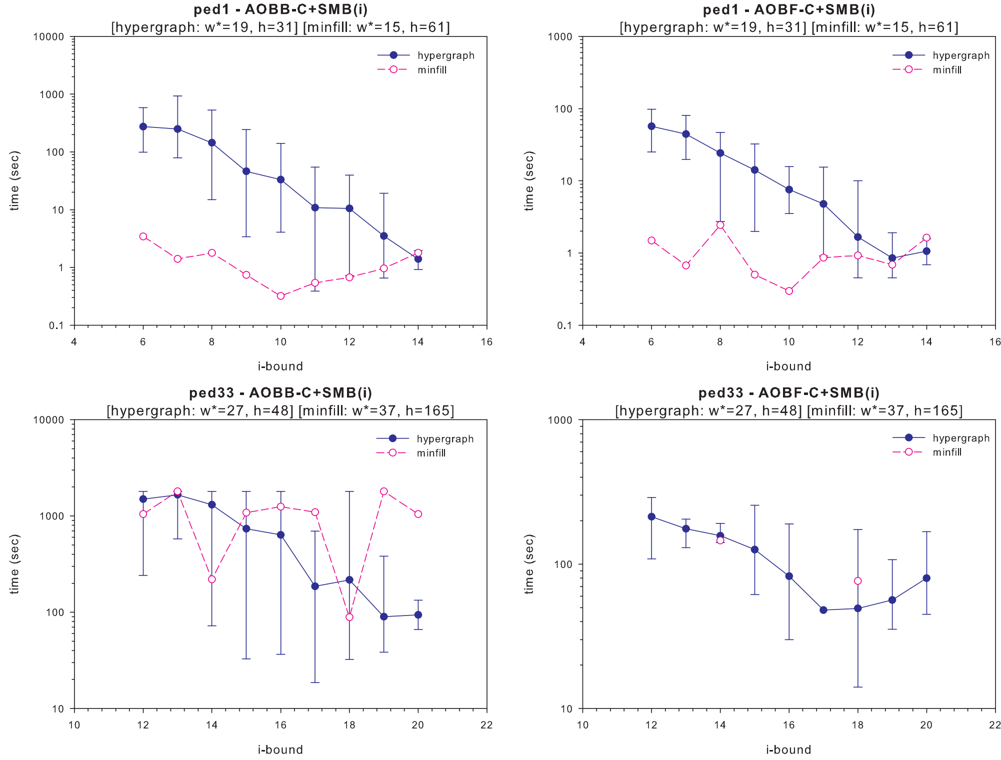


Fig. 9. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **genetic linkage networks** with `AOBB-C+SMB(`i`)` (left side) and `AOBF-C+SMB(`i`)` (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

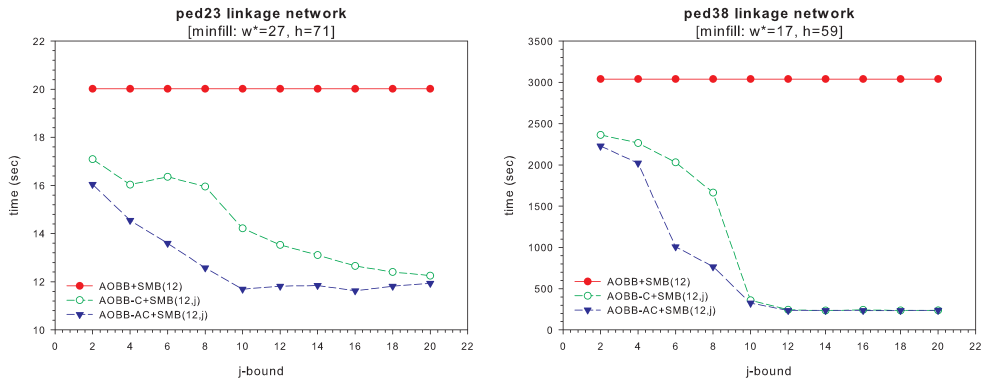


Fig. 10. Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **genetic linkage networks**. Shown is CPU time in seconds.

likelihood haplotype task. We see that the hypergraph pseudo trees offer the overall best performance as well. This can be explained by the large induced width which in this case renders most of the cache entries dead (see for instance that the difference between `AOBB+SMB(`i`)` and `AOBB-C+SMB(`i`)` is not too prominent). Therefore, the AND/OR graph explored effectively is very close to a tree and the dominant factor that impacts the search performance is then the depth of the guiding

Table 5
Impact of the pseudo tree quality on **genetic linkage networks**. Time limit 24 hours. We show results for the hypergraph partitioning heuristic (left) and the min-fill heuristic (right).

		hypergraph pseudo tree					**min-fill pseudo tree**				
pedigree (n, d)	SamIam Superlink	(w*, h)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=22		(w*, h)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=22	
			time	nodes	time	nodes		time	nodes	time	nodes
ped7			25.26		164.49			117.03		out	
(868, 4)	out		-	-	-	-		-	-		
	-	(36, 60)	88571.68	1,807,878,340	9395.17	195,845,851	(32, 133)	-	-		
			30504.84	285,084,124	**3005.66**	27,761,219		-	-		
			out		out			out			
ped9			67.93		300.06			76.31		out	
(936, 7)	out		-	-	-	-		-	-		
	-	(35, 58)	11483.89	231,301,374	3982.69	72,844,362	(27, 130)	1515.50	15,825,340		
			8922.81	117,328,162	**3292.30**	40,251,723		**1163.09**	12,444,961		
			out		out			out			
ped19			59.31		150.38			out		out	
(693, 5)	out		-	-	-	-					
	-	(35, 53)	98941.75	1,519,213,924	12530.00	174,000,317	(24, 122)				
			45075.31	466,748,365	**8321.42**	90,665,870					
			out		out						
ped34			42.21		209.51			out		out	
(923, 4)	out		-	-	-	-					
	-	(34, 60)	70504.72	1,453,705,377	13598.50	294,637,173	(32, 127)				
			67647.42	1,293,350,829	**11719.28**	220,199,927					
			out		out						
ped41			35.41		111.24			out		out	
(886, 5)	out		-	-	-	-					
	-	(36, 61)	6669.50	84,506,068	531.40	4,990,995	(33, 128)				
			3891.86	31,731,270	**380.01**	2,318,544					
			out		out						
ped44			32.92		140.81			57.88		344.68	
(644, 4)	out		-	-	-	-		-	-	-	-
	-	(31, 52)	8388.18	196,823,840	401.84	7,648,962	(26, 73)	127.42	1,114,641	385.47	668,737
			3597.12	62,385,573	**204.96**	1,355,595		**95.09**	752,970	366.18	447,514
			out		out			out		out	

pseudo tree, which is far smaller for hypergraph trees compared with min-fill based ones. Notice also that best-first search could not solve any of these networks due to memory issues. The AND/OR Branch-and-Bound algorithms with min-fill based pseudo trees could only solve two of the test instances ($e.g.$, `ped9` and `ped44`) whose induced widths were small enough. These experiments demonstrate that the selection of the pseudo tree can have an enormous impact, especially if the i-bound that can be afforded is not large enough.

Impact of the caching level. Figure 10 plots the CPU time, as a function of the cache bound j, for two linkage networks using `AOBB-C+SMB(`i,j`)` (naive caching) and `AOBB-AC+SMB(`i,j`)` (adaptive caching), respectively. In each test case we varied the cache bound j (the X axis) from 2 to 20, and fixed the mini-bucket i-bound to a relatively small value. We see again that adaptive caching is more powerful than the naive scheme especially, for relatively small j-bounds,

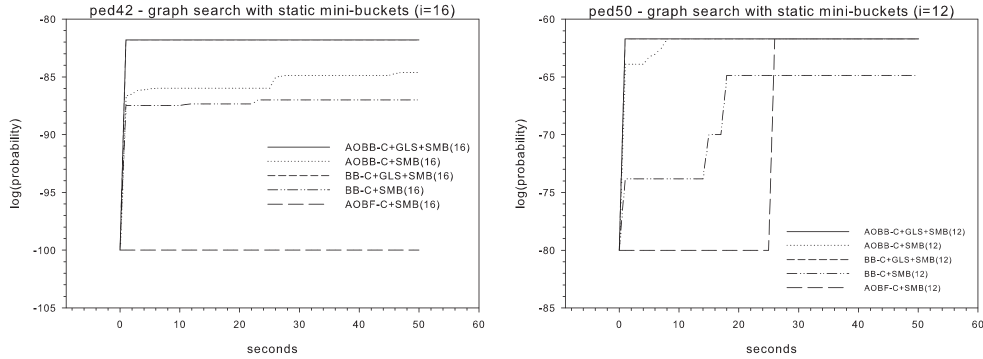


Fig. 11. Anytime behavior of `AOBB-C+SMB(`i`)` on **`ped42`** and **`ped50` linkage networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

which require restricted space. As the j-bound increases, the two schemes approach gradually full caching.

7.2.3 The Anytime Behavior of AND/OR Branch-and-Bound Search and the Impact of Good Initial Bounds

As mentioned earlier, the virtue of AND/OR Branch-and-Bound search is that, unlike best-first AND/OR search, it is an anytime algorithm. Namely, whenever interrupted, `AOBB-C` outputs the best solution found far, which yields a lower bound on the most probable explanation. On the other hand, `AOBF-C` outputs a complete solution only upon termination. In this section we evaluate the anytime behavior of `AOBB-C+SMB(`i`)`. We compare it against the state-of-the-art local search algorithm for Bayesian MPE, called *Guided Local Search* (GLS) first introduced in [34], and improved more recently by [35].

GLS [36] is a penalty-based meta-heuristic, which works by augmenting the objective function of a local search algorithm ($e.g.$ hill climbing) with penalties, to help guide them out of local minima. GLS has been shown to be successful in solving a number of practical real life problems, such as the traveling salesman problem, radio link frequency assignment problem and vehicle routing. It was also applied to the MPE task [34,35] as well as weighted MAX-SAT problems [37].

In addition to comparing against GLS, we also considered a hybrid of `AOBB` with GLS, as follows. The AND/OR Branch-and-Bound algorithms assumed a trivial initial lower bound ($i.e.$, 0), which effectively guarantees that the MPE will be computed, however it provides limited pruning. We therefore extended `AOBB-C+SMB(`i`)` to exploit a non-trivial initial lower bound computed by GLS. The algorithm is denoted by `AOBB-C+GLS+SMB(`i`)`. For comparison, we also ran the OR version of the algorithm, denoted by `BB-C+GLS+SMB(`i`)`

Figure 11 displays the search trace of the OR and AND/OR algorithms on two genetic linkage networks presented earlier in Tables 3 and 4, respectively. We chose

Table 6
CPU time and nodes visited for solving **genetic linkage analysis networks** with static mini-bucket heuristics. Number of flips for GLS was set to 250,000. Time limit 3 hours.

min-fill pseudo tree											
pedigree	SamIam	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
	Superlink	BB-C+GLS+SMB(i)		BB-C+GLS+SMB(i)		BB-C+GLS+SMB(i)		BB-C+GLS+SMB(i)		BB-C+GLS+SMB(i)	
(w*, h)	GLS	AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)	
(n, d)		AOBB-C+GLS+SMB(i)		AOBB-C+GLS+SMB(i)		AOBB-C+GLS+SMB(i)		AOBB-C+GLS+SMB(i)		AOBB-C+GLS+SMB(i)	
		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)	
		i=6		i=8		i=10		i=12		i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	1.14	7,997	0.73	3,911	1.31	2,704
ped1	5.44	8943.68	59,627,660	1367.98	9,013,771	3.84	1,798	4.05	2,524	4.75	2,077
(15, 61)	54.73	4.19	69,751	2.17	33,908	0.39	4,576	0.65	6,306	1.36	4,494
(299, 5)	0.31	3.01	46,663	2.10	29,877	**0.13**	3,138	0.33	6,092	0.92	4,350
		1.30	7,314	2.17	13,784	0.26	1,177	0.87	4,016	1.54	3,119
		-	-	-	-	-	-	-	-	out	
ped38	out	-	-	-	-	-	-	-	-		
(17, 59)	**28.36**	5946.44	34,828,046	1554.65	8,986,648	2046.95	11,868,672	272.69	1,412,976		
(582, 5)	7.05	4410.70	32,599,034	780.46	4,487,470	1650.05	9,844,485	226.44	1,366,242		
		out		134.41	348,723	216.94	583,401	103.17	242,429		
		-	-	-	-	-	-	-	-	out	
ped50	out	-	-	-	-	-	-	52.95	83,025		
(18, 58)	-	4140.29	28,201,843	2493.75	15,729,294	66.66	403,234	52.11	110,302		
(479, 5)	5.30*	3177.43	24,209,840	1610.33	13,299,343	67.85	400,698	32.67	15,865		
		78.53	204,886	36.03	104,289	**12.75**	25,507	38.52	5,766		
		i=10		i=12		i=14		i=16		i=18	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	76.11	339,125	270.22	74,261	out	
ped23	out	8556.84	39,184,112	6640.68	28,790,468	15.27	23,947	270.25	55,412		
(27, 71)	9146.19	193.78	1,726,897	10.06	74,672	13.33	23,557	274.00	62,613		
(310, 5)	3.94	196.68	1,720,633	**7.56**	73,082	10.58	20,329	274.26	60,424		
		out		15.33	58,180	14.36	12,987	out			
		-	-	-	-	-	-	-	-	out	
ped37	out	-	-	2073.12	10,612,906	-	-	3386.01	16,382,262		
(21, 61)	64.17	39.16	222,747	488.34	4,925,737	301.78	2,798,044	67.83	82,239		
(1032, 5)	8.97*	**16.36**	141,867	26.97	254,219	82.08	604,239	52.32	23,572		
		29.16	72,868	38.41	102,011	95.27	223,398	62.97	12,296		

the mini-bucket i-bound that offered the best performance and show the first 50 seconds of the search. We ran GLS for a fixed number of flips. We see that including the GLS lower bound in AND/OR Branch-and-Bound improves performance throughout. In all these test cases, the initial lower bound was in fact the optimal solution (we did not plot the GLS running time because it was less than 1 second). Therefore, AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i) were able to output the optimal solution quite early in the search, unlike AOBB-C+SMB(i) and BB-C+SMB(i). For instance, on the ped50 network, AOBB-C+GLS+SMB(12) and BB-C+GLS+SMB(12) found the optimal solution within the first second of search. AOBB-C+SMB(12), on the other hand, finds the optimal solution after 8 seconds, whereas BB-C+SMB(12) reaches a flat (suboptimal) region after 18 seconds. In this case, AOBF-C+SMB(12) finds the optimal solution after 25 seconds. The same behavior was observed on other instances as well.

Table 6 compares the OR and AND/OR search algorithms with and without an initial lower bound, as complete algorithms. Algorithms AOBB-C+GLS+SMB(i) and

Table 7
CPU time and nodes visited for solving **deterministic grid networks** with static mini-bucket heuristics. Number of flips for GLS was set to 100,000. Time limit 1 hour.

min-fill pseudo tree											
grid (w*, h) (n, e)	SamIam GLS	AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=12		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=14		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=16		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=18		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	1273.09	9,047,518	596.27	4,923,760	70.42	473,675	74.99	412,291
90-24-1	out	687.96	4,823,044	202.05	1,564,800	172.31	1,370,222	55.52	401,294	69.53	386,785
(33, 111)		-	-	66.20	425,585	20.16	93,911	11.17	7,850	28.16	27,868
(576, 20)	0.53	473.64	3,181,352	19.09	131,546	8.41	49,054	**5.45**	6,891	23.87	39,175
		out		21.94	75,637	10.59	33,770	6.06	5,144	23.80	17,291
		146.97	878,874	152.80	962,484	4.36	15,632	12.92	46,489	22.13	2,242
90-26-1	out	32.67	230,030	53.11	360,612	**3.58**	11,620	11.95	40,075	22.02	1,858
(36, 113)		36.94	252,380	87.02	559,518	4.17	14,580	7.86	6,310	22.00	1,894
(676, 40)	0.56	15.09	104,775	32.85	219,037	**3.58**	10,932	8.06	8,128	24.42	1,658
		19.06	65,271	24.39	79,619	4.27	7,190	8.05	3,777	22.44	1,435
		652.15	3,882,300	165.74	1,070,823	155.20	956,837	40.14	212,963	59.28	174,715
90-30-1	out	117.25	771,233	66.66	453,095	50.94	341,670	30.69	168,928	42.86	88,004
(43, 150)		263.32	1,498,756	74.95	446,498	68.16	376,916	23.88	95,136	53.92	148,540
(900, 60)	0.72	89.94	561,397	38.92	247,271	28.67	176,330	**15.50**	52,260	40.52	72,053
		158.97	534,385	46.73	157,187	47.27	154,496	21.06	45,201	57.97	100,800
		-	-	-	-	-	-	-	-	369.36	823,604
90-34-1	out	-	-	-	-	-	-	-	-	132.84	271,609
(45, 153)		-	-	-	-	1096.14	5,569,276	1772.51	5,516,888	294.11	630,406
(1154, 80)	1.31	-	-	-	-	550.55	2,944,055	651.04	2,614,171	124.16	238,333
		out		out		243.63	596,978	270.88	667,013	**71.19**	67,611
		969.02	2,623,971	1753.10	3,794,053	203.67	614,868	165.45	488,873	113.06	214,919
90-38-1	out	141.89	577,763	204.69	593,809	86.16	319,185	102.03	312,473	85.74	142,589
(47, 163)		854.61	2,498,702	1822.71	3,792,826	212.63	647,089	164.43	484,815	109.77	211,740
(1444, 120)	1.11	138.44	573,923	204.68	597,751	96.27	339,729	98.21	311,072	85.50	140,581
		101.69	174,786	103.80	146,237	**54.00**	95,511	53.44	78,431	73.10	59,856

BB-C+GLS+SMB(i) do not include the GLS time, because GLS can be tuned independently for each problem instance to minimize its running time, so we report its time separately (as before, GLS ran for a fixed number of flips). The "*" by the GLS running time indicates that it found the optimal solution to the respective problem instance. We see that BB-C+GLS+SMB(i) and AOBB-C+GLS+SMB(i) are sometimes able to improve significantly over BB-C+SMB(i) and AOBB-C+SMB(i), especially at relatively small i-bounds. For example, on the ped37 linkage instance, AOBB-C+GLS+SMB(12) achieves almost an order of magnitude speedup over AOBB-C+SMB(12). Similarly, BB-C+GLS+SMB(12) finds the optimal solution to ped37 in about 35 minutes, whereas BB-C+SMB(12) exceeds the 3 hour time limit.

7.2.4 *The Impact of Determinism in Bayesian Networks*

In general, when the functions of the graphical model express both hard constraints and general cost functions, it is beneficial to exploit the computational power of the

constraints explicitly via constraint propagation [38–41]. For Bayesian networks, the hard constraints are represented by the zero probability tuples of the CPTs. We note that the use of constraint propagation via directional resolution [42] or generalized arc consistency has been explored in [38,39], in the context of variable elimination algorithms where the constraints are also extracted based on the zero probabilities in the network. The approach we take for handling the determinism in Bayesian networks is based on *unit resolution* for Boolean Satisfiability (SAT). The idea of using unit resolution during search for Bayesian networks was first explored in [40]. One common way which we used for encoding hard constraints as a CNF formula is the *direct encoding* [43].

We evaluated the AND/OR Branch-and-Bound algorithm with static mini-bucket heuristics on selected classes of Bayesian networks containing zero probability tuples. The algorithm, denoted by AOBB-C+SAT+SMB(i) exploits the determinism present in the networks by applying unit resolution over the CNF encoding of the zero-probability tuples, at each node in the search tree. We used a unit resolution scheme similar to the one employed by zChaff, a state-of-the-art SAT solver introduced by [44]. We also consider the extension called AOBB-C+SAT+GLS+SMB(i) which uses GLS to compute the initial lower bound, in addition to the constraint propagation scheme.

Table 7 shows the results for 5 deterministic grid networks presented earlier. We observe that AOBB-C+SAT+SMB(i) improves significantly over AOBB-C+SMB(i), especially at relatively small i-bounds. On grid 90-30-1, AOBB-C+SAT+SMB(12) is 6 times faster than AOBB-C+SMB(12). As the i-bound increases and the search space is pruned more effectively, the difference between AOBB-C+SMB(i) and AOBB-C+SAT+SMB(i) decreases because the heuristics are strong enough to cut the search space significantly and it already does some level of constraint propagation. When focusing on the impact of the initial lower bound on AOBB-C+SAT+SMB(i) through algorithm AOBB-C+SAT+GLS+SMB(i) we see that the latter is sometimes able to improve even more. On the 90-34-1 grid, AOBB-C+SAT+GLS+SMB(16) finds the optimal solution in about 9 minutes whereas AOBB-C+SAT+SMB(16) exceeds the 1 hour time limit. We should note that best-first search does not employ a constraint propagation scheme.

7.2.5 Summary of Empirical Results on Bayesian Networks

Our extensive empirical evaluation on Bayesian networks demonstrated conclusively that the memory intensive AND/OR search algorithms guided by static mini-bucket heuristics were the best performing algorithms overall. The difference between AOBB-C+SMB(i) and the cache-less AOBB+SMB(i) was more pronounced at relatively small i-bounds which correspond to relatively weak heuristic estimates ($e.g.$, ISCAS'89 networks, grid networks, genetic linkage analysis, instances from the UAI'06 Inference Evaluation contest). For larger i-bounds, when the heuristic

estimates are stronger, the difference between graph search `AOBB-C+SMB(`i`)` and tree search `AOBB+SMB(`i`)` decreased. Best-first search `AOBF-C+SMB(`i`)` offered the best performance amongst the memory intensive AND/OR algorithms. We showed that in many cases `AOBF-C+SMB(`i`)` was able to outperform dramatically the current state-of-the-art solver for Bayesian networks such as SAMIAM and SUPERLINK (for genetic linkage analysis). However, on very large problem instances, `AOBF-C+SMB(`i`)` was outperformed by the depth-first `AOBB-C+SMB(`i`)` because of its prohibitive memory requirements. With dynamic mini-bucket heuristics both `AOBB-C+DMB(`i`)` and `AOBF-C+DMB(`i`)` proved competitive only for relatively small i-bounds, due to computational overhead. We also evaluated the impact of determinism and good initial lower bounds on depth-first AND/OR Branch-and-Bound search, over grid networks, ISCAS'89 networks, genetic linkage analysis networks and instances from the UAI'06 Inference Evaluation dataset. These empirical results, also available in [27,26], showed that applying unit resolution and starting the search with a good initial lower bound caused significant savings on those benchmark networks.

7.3 Results for Empirical Evaluation of Weighted CSPs

Let us first recap the results obtained for Weighted CSPs with our various cache-less algorithms [1]. We showed that the best performance on Weighted CSPs was obtained by the AND/OR Branch-and-Bound *tree* search algorithm with static mini-bucket heuristics, at relatively large i-bounds, especially for non-binary WCSPs with relatively small domain sizes ($e.g.$, SPOT5 networks, ISCAS'89 circuits, Mastermind game instances). The cache-less `AOBB+SMB(`i`)` dominated all its competitors, including the classic OR Branch-and-Bound `BB+SMB(`i`)` as well as the OR and AND/OR algorithms that enforce EDAC during search, namely `toolbar` and the `AOEDAC` family of algorithms, such as `AOEDAC+PVO`, `DVO+AOEDAC` and `AOEDAC+DSO`, respectively [1]. The AND/OR Branch-and-Bound with dynamic mini-bucket heuristics `AOBB+DMB(`i`)` was shown to be competitive only for relatively small i-bounds.

In this section we extend the evaluation to memory intensive depth-first and best-first search.

7.3.1 SPOT5 Benchmark

SPOT5 benchmark contains a collection of large real scheduling problems for the daily management of Earth observing satellites [45]. They can be easily formulated as WCSPs with binary and ternary constraints, as described in [1,3].

Tables 8 and 9 show detailed results on experiments with 7 SPOT5 networks using min-fill pseudo trees, as well as static and dynamic mini-bucket heuristics. The

Table 8
CPU time in seconds and number of nodes visited for solving the **SPOT5 benchmarks**, using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 3 hours.

min-fill pseudo tree												
	MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)		toolbar	
spot5	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		toolbar-BTD	
	AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOEDAC+PVO	
(w*, h)	AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		DVO+AOEDAC	
(n, k, c)	AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOEDAC+DSO	
	i=4		i=6		i=8		i=12		i=14			
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.01		0.05		0.33		21.66		150.99		4.56	218,846
29	-	-	-	-	6313.73	50,150,302	22.30	2,322	151.02	445	**0.35**	984
(14, 42)	8.77	86,058	5.05	45,509	0.66	2,738	22.02	246	151.02	481	545.43	7,837,447
(83, 4, 476)	5.53	48,995	3.66	29,702	0.56	2,267	21.67	110	149.55	265	0.81	8,698
	6.42	36,396	2.23	12,801	0.47	757	21.77	96	152.69	85	11.36	92,970
	0.11		0.17		0.56		28.83		223.58		-	-
42b	-	-	-	-	2159.26	9,598,763	145.77	684,109	224.11	3,426	9553.06	249,053,196
(18, 62)	-	-	-	-	1842.32	9,606,846	134.39	689,402	228.66	4,189	-	-
(191, 4, 1341)	-	-	-	-	1804.76	9,410,729	116.98	584,838	226.58	2,335	-	-
	35.42	118,085	29.11	106,648	**20.80**	82,611	38.91	43,127	227.55	1,475	6825.40	27,698,614
	0.02		0.03		0.11		1.24		1.24		0.31	21,939
54	664.48	5,715,457	2.06	17,787	0.38	2,289	1.27	236	1.27	236	0.18	779
(11, 33)	113.19	1,106,598	1.59	17,757	0.39	3,616	1.27	329	1.39	329	9.11	90,495
(68, 4, 283)	18.42	198,712	0.23	2,477	0.16	591	1.25	120	1.24	120	**0.06**	688
	0.41	2,714	0.11	631	0.16	312	0.69	68	1.41	68	0.75	6,614
	0.01		0.02		0.09		1.11		3.97		151.11	6,215,135
404	-	-	-	-	-	-	4336.37	32,723,215	1981.90	15,263,175	5.09	139,968
(19, 42)	430.99	3,969,398	151.99	1,373,846	14.83	144,535	1.44	3,273	4.11	367	152.81	1,984,747
(100, 4, 710)	174.09	1,396,321	51.88	529,002	2.55	23,565	1.16	598	4.11	232	12.09	88,079
	1.45	7,251	1.20	6,399	**1.02**	5,140	1.22	576	4.27	184	1.74	14,844
	0.01		0.09		0.33		8.37		35.39		-	-
408b	-	-	-	-	-	-	-	-	-	-	-	-
(24, 59)	-	-	-	-	-	-	715.35	4,784,407	128.38	567,407	-	-
(201, 4, 1847)	-	-	-	-	7507.10	54,826,929	75.08	408,619	48.00	61,986	-	-
	208.41	185,935	52.53	175,366	44.99	145,901	**16.97**	39,238	39.36	14,768	747.71	2,134,472
	0.02		0.05		0.14		0.41		0.41		-	-
503	-	-	-	-	-	-	0.50	566	0.49	566	0.65	18,800
(9, 39)	-	-	435.26	5,102,299	421.10	4,990,898	0.44	641	0.44	641	-	-
(144, 4, 639)	-	-	189.39	2,442,998	291.72	4,050,474	**0.42**	256	**0.42**	256	10005.00	44,495,545
	5.28	16,114	1.56	9,929	1.59	9,186	**0.42**	144	**0.42**	144	53.72	231,480
	0.05		0.11		0.66		47.19		365.69		-	-
505b	-	-	-	-	-	-	-	-	-	-	33.62	1,119,538
(16, 98)	-	-	-	-	-	-	-	-	395.49	143,371	-	-
(240, 1721)	-	-	-	-	-	-	1180.48	8,905,473	375.57	16,020	-	-
	51.86	149,928	42.73	144,723	**29.25**	111,223	54.09	31,692	375.52	5,758	-	-

networks 42b, 408b and 505b are sub-networks of the original ones and contain only binary constraints.

Tree vs. graph AOBB. As before, the differences in running time and number of nodes visited, between AOBB-C+SMB(i) and AOBB+SMB(i) are more prominent at relatively small i-bounds. For example, on the 408b network, AOBB-C+SMB(12) outperforms AOBB+SMB(12) by one order of magnitude. The impact of caching when using dynamic mini-bucket heuristics (Table 9) is again not that pronounced, across i-bounds. Notice that toolbar and DVO+AOEDAC (rightmost column in Table 8) are able to solve relatively efficiently only the first 3 test instances. On the

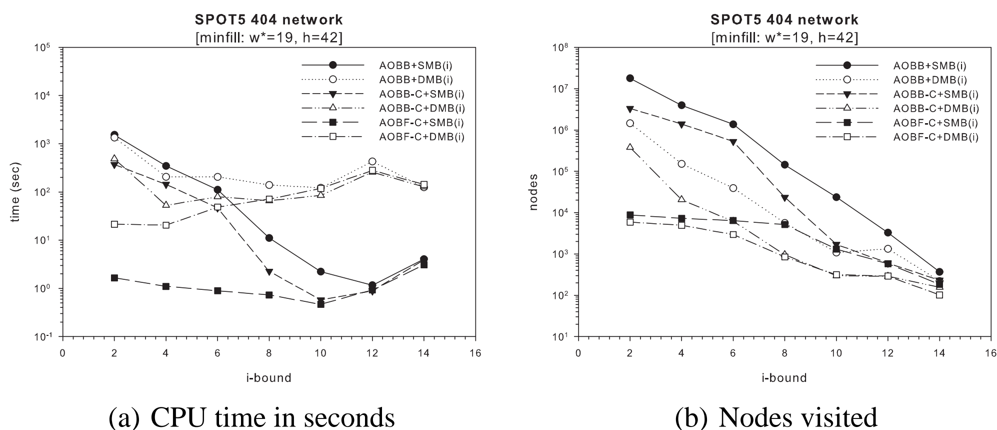
Table 9
CPU time in seconds and number of nodes visited for solving the **SPOT5 benchmarks**, using **dynamic mini-bucket heuristics** and min-fill based pseudo trees. Time limit 3 hours.

min-fill pseudo tree										
spot5	BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)	
	AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)	
(w*, h)	AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)	
(n, k, c)	AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)	
	i=4		i=6		i=8		i=12		i=14	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
29	44.24	11,637	125.72	9,417	54.86	354	627.30	320	1647.82	320
(14, 42)	65.24	14,438	52.92	11,850	121.83	364	627.29	330	1644.02	330
(83, 4, 476)	56.58	6,017	53.06	4,638	122.17	170	636.16	136	1794.60	136
	7.25	942	21.83	537	38.83	114	308.71	83	983.80	83
42b	-	-	-	-	-	-	-	-	-	-
(18, 62)	-	-	-	-	-	-	-	-	-	-
(191, 4, 1341)	-	-	-	-	-	-	-	-	-	-
	1455.62	101,453	-	-	-	-	6002.69	212	-	-
54	886.51	118,219	32.59	938	24.97	236	320.81	236	321.15	236
(11, 33)	202.14	69,362	26.73	2,188	22.19	329	271.81	329	271.55	329
(68, 4, 283)	84.27	15,214	8.80	357	10.86	120	137.39	120	137.75	120
	4.16	1,056	**3.66**	163	5.95	68	77.78	68	78.19	68
404	-	-	-	-	4895.25	78,692	3459.31	3,008	473.81	165
(19, 42)	240.36	156,338	257.20	39,144	199.67	5,612	563.02	1,327	287.53	395
(100, 4, 710)	65.52	20,457	98.83	6,152	99.78	952	320.49	286	171.02	155
	23.41	4,928	65.80	2,946	101.30	847	351.37	291	217.45	106
408b	-	-	-	-	-	-	-	-	-	-
(24, 59)	-	-	-	-	-	-	-	-	-	-
(201, 4, 1847)	-	-	-	-	-	-	-	-	-	-
	655.41	70,655	2447.91	69,434	-	-	-	-	-	-
503	-	-	-	-	-	-	246.65	566	246.65	566
(9, 39)	-	-	-	-	-	-	64.95	641	64.95	641
(144, 4, 639)	-	-	-	-	-	-	49.95	256	49.95	256
	78.69	9,143	324.09	8,175	1025.40	5,984	**25.14**	144	**25.14**	144
505b	-	-	-	-	-	-	-	-	-	-
(16, 98)	-	-	-	-	-	-	-	-	-	-
(240, 1721)	-	-	-	-	-	-	-	-	-	-
	681.40	33,969	2766.08	28,157	3653.66	12,455	-	-	-	-

other hand, toolbar-BTD fails only on the 408b instance and is overall quite competitive.

AOBB vs. AOBF. When comparing best-first against depth-first AND/OR search we see again that AOBF-C+SMB(i) improves significantly (up to several orders of magnitude), especially for relatively small i-bounds. For example, on 505b, one of the hardest instances, AOBF-C+SMB(8) finds the optimal solution in less than 30 seconds, whereas AOBB-C+SMB(8) exceeds the 3 hour time limit.

Static vs. dynamic mini-bucket heuristics. Figures 12(a) and 12(b) display the running time and number of nodes, as a function of the mini-bucket i-bound, on the 404 network ($i.e.$, corresponding to the fourth horizontal block from Tables 8 and 9, respectively). We see that the power of the dynamic mini-bucket heuristics is visible only for depth-first search and only for small i-bounds ($e.g.$, $i = 2$). At larger i-bounds, the static mini-bucket heuristics are cost effective. For instance, the difference in running time between AOBB-C+SMB(10) and AOBB-C+DMB(10)



(a) CPU time in seconds (b) Nodes visited

Fig. 12. Comparison of the impact of static and dynamic mini-bucket heuristics. Shown are the CPU time in seconds (a) and number of nodes visited (b) on the **404 SPOT5 network** from Tables 8 and 9, respectively.

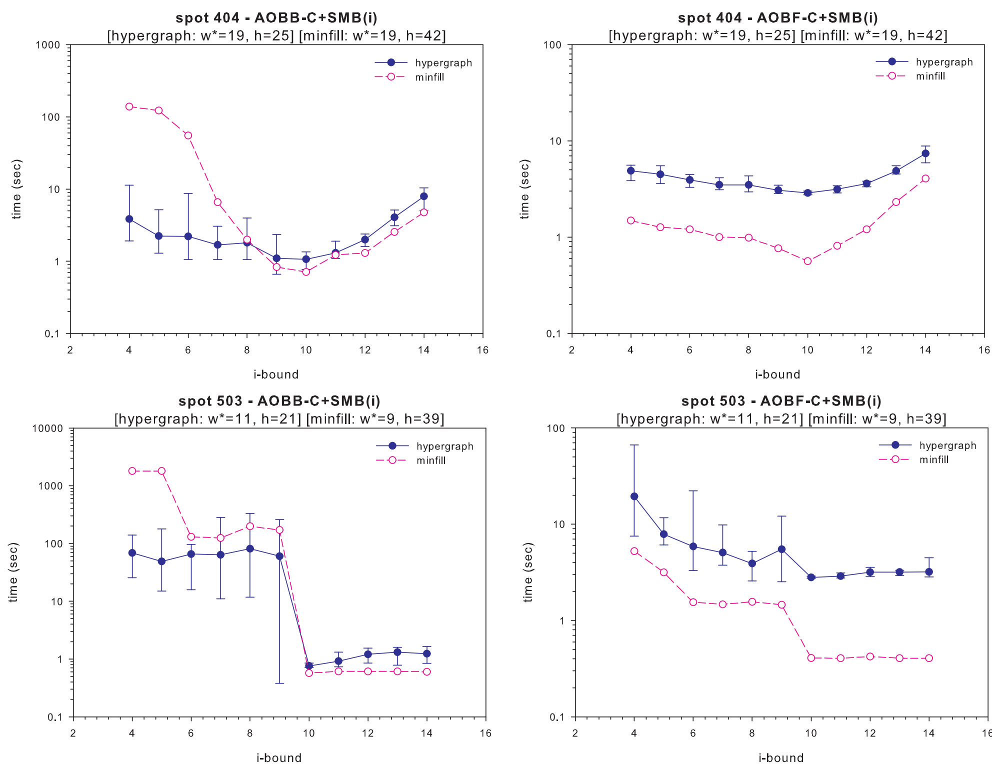


Fig. 13. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving SPOT5 networks with `AOBB-C+SMB(i)` (left side) and `AOBF-C+SMB(i)` (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

is about 2 orders of magnitude. Notice that in this case, `AOBF-C+SMB(`i`)` outperforms `AOBF-C+DMB(`i`)` across all reported i-bounds.

Impact of the pseudo tree. In Figure 13 we show the running time distribution of the algorithms using hypergraph and min-fill based pseudo trees, over 20 independent runs, for the `404` and `503` networks. We see again that the hypergraph based pseudo trees are sometimes able to improve performance, especially for relatively small i-bounds for which the heuristic estimates are less accurate. For best-first search however, the min-fill based pseudo trees offer the best performance.

7.3.2 Mastermind Game Instances

Table 10 shows the results for experiments with 6 networks corresponding to Mastermind game instances of increasing difficulty. Each of the networks is a ground instance of a relational Bayesian network that models different sizes of the popular game of Mastermind. These networks were produced by the PRIMULA System [6] and used in experimental results in [46]. For our purpose, we converted these networks into equivalent WCSP instances by taking the negative log probability of each conditional probability table entry. The table has two horizontal blocks, each showing a different range of i-bounds.

Tree vs. graph AOBB. We see again that using caching improves considerably the performance of AND/OR Branch-and-Bound search (*e.g.*, see `mm-03-08-05`). We also note that `toolbar` and `toolbar-BTD` were not able to solve any of these instances within the time limit (the results are not displayed).

AOBB vs. AOBF. We see that the best-first search algorithm `AOBF-C+SMB(`i`)` offers the overall best performance on this domain. On the `mm-03-08-05` instance, for example, `AOBF-C+SMB(18)` is about 3 times faster than `AOBB-C+SMB(18)` and about 30 times faster than `AOBB+SMB(18)`, a further demonstration of the power of caching.

Impact of the caching level. Figure 14 illustrates the CPU time, as a function of the cache bound j, on two problem instances from Table 10. We notice again the superiority of adaptive caching at relatively small j-bounds.

Impact of the pseudo tree. The running time distribution of `AOBB-C+SMB(`i`)` and `AOBF-C+SMB(`i`)` guided by hypergraph and min-fill based pseudo trees over 20 independent runs of each problem instance is displayed in Figure 15. The hypergraph trees are sometimes able to improve slightly the performance of AND/OR Branch-and-Bound, at relatively small i-bounds (*e.g.*, `mm-04-08-04`). For best-first search however, the min-fill based pseudo trees offer the best performance. The results on other instances were similar.

---

[6] ***http://www.cs.auc.dk/jaeger/Primula***

Table 10
CPU time and number of nodes visited for solving **Mastermind game instances**, using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour. `toolbar` and `toolbar-BTD` were not able to solve any of the test instances within the time limit. The top part of the table shows the results for i-bounds between 8 and 18, while the bottom part shows i-bounds between 12 and 22.

min-fill pseudo trees												
	MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)	
	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
mastermind	AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)	
	AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)	
(w*, h)	AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)	
(n, r, k)	i=8		i=10		i=12		i=14		i=16		i=18	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.30		0.34		0.44		0.80		2.00		5.31	
mm-03-08-03	59.14	49,376	19.39	9,576	51.83	41,282	8.42	3,377	9.17	3,068	12.80	2,980
(20, 57)	1.58	10,396	1.64	7,075	1.50	6,349	1.38	3,830	2.53	3,420	5.73	3,153
(1220, 3, 2)	1.05	2,770	1.22	3,299	1.14	3,010	1.22	2,273	2.39	2,114	5.56	2,031
	0.72	1,366	1.14	2,196	1.22	2,202	1.20	1,311	2.36	1,247	5.66	1,220
	0.75		0.83		1.02		1.75		4.38		15.77	
mm-03-08-04	-	-	-	-	-	-	-	-	-	-	-	-
(33, 87)	92.64	150,642	110.45	193,805	64.13	71,622	17.17	31,177	36.14	63,669	22.38	13,870
(2288, 3, 2)	21.50	20,460	34.75	28,631	15.94	14,101	9.56	8,747	16.03	11,971	19.45	5,376
	10.53	9,693	10.88	9,143	10.06	8,925	**3.89**	2,928	9.08	4,855	19.52	4,266
	0.34		0.41		0.51		0.91		2.44		7.83	
mm-04-08-03	-	-	981.26	726,162	51.42	32,948	32.53	16,633	29.19	14,151	28.11	9,881
(26, 72)	15.64	68,929	6.02	26,111	8.06	34,445	5.05	17,255	6.09	15,443	10.16	10,570
(1418, 3, 2)	3.85	7,439	1.63	3,872	2.49	5,367	2.75	4,778	4.44	4,824	9.06	3,444
	0.94	1,578	0.94	1,475	1.05	1,472	1.42	1,462	2.95	1,453	8.36	1,450

	MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)	
	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
mastermind	AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)	
	AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)	
(w*, h)	AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)	
(n,c,k)	i=12		i=14		i=16		i=18		i=20		i=22	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	1.36		2.08		4.86		16.53		65.19		246.45	
mm-04-08-04	-	-	-	-	-	-	-	-	-	-	-	-
(39, 103)	494.50	744,993	270.60	447,464	506.74	798,507	80.86	107,463	206.58	242,865	280.07	62,964
(2616, 3, 2)	114.02	82,070	66.84	61,328	93.50	79,555	30.80	13,924	91.08	28,648	253.25	11,650
	38.55	33,069	29.19	26,729	44.95	38,989	**20.64**	3,957	74.67	8,716	250.00	3,491
	2.34		8.52		8.31		24.94		84.52		out	
mm-03-08-05	-	-	-	-	-	-	-	-	-	-		
(41, 111)	-	-	-	-	-	-	1084.48	1,122,008	1283.04	1,185,327		
(3692, 3, 2)	-	-	-	-	-	-	117.39	55,033	282.35	86,588		
	out		out		473.07	199,725	**36.99**	8,297	131.88	21,950		
	1.64		3.09		7.55		21.08		77.81		out	
mm-10-08-03	-	-	-	-	-	-	-	-	-	-		
(51, 132)	161.35	290,594	99.09	326,662	89.06	151,128	84.16	127,130	144.03	133,112		
(2606, 3, 2)	19.86	14,518	19.47	14,739	22.34	13,557	29.80	9,388	89.75	12,362		
	4.80	3,705	8.16	4,501	11.17	3,622	24.67	3,619	81.52	3,573		

Memory usage of AND/OR graph search. In Figure 16 we demonstrate again the significant memory requirements of best-first AND/OR search compared with those of the depth-first AND/OR Branch-and-Bound search with full caching on two problem instances.

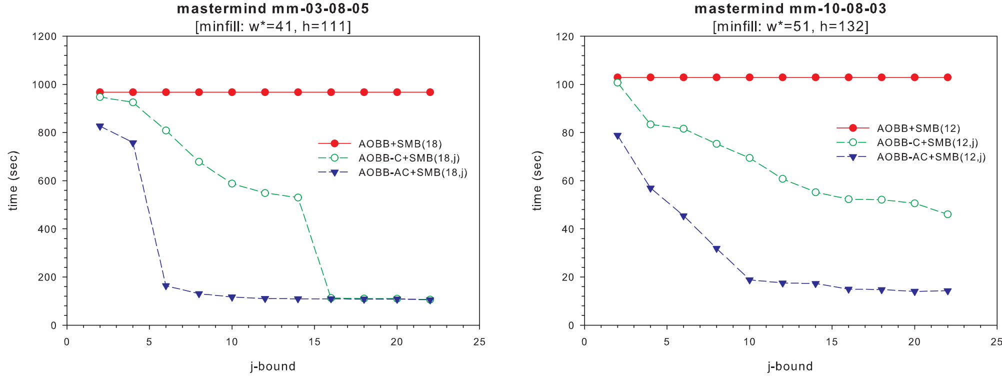


Fig. 14. Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **Mastermind networks**. Shown is CPU time in seconds.

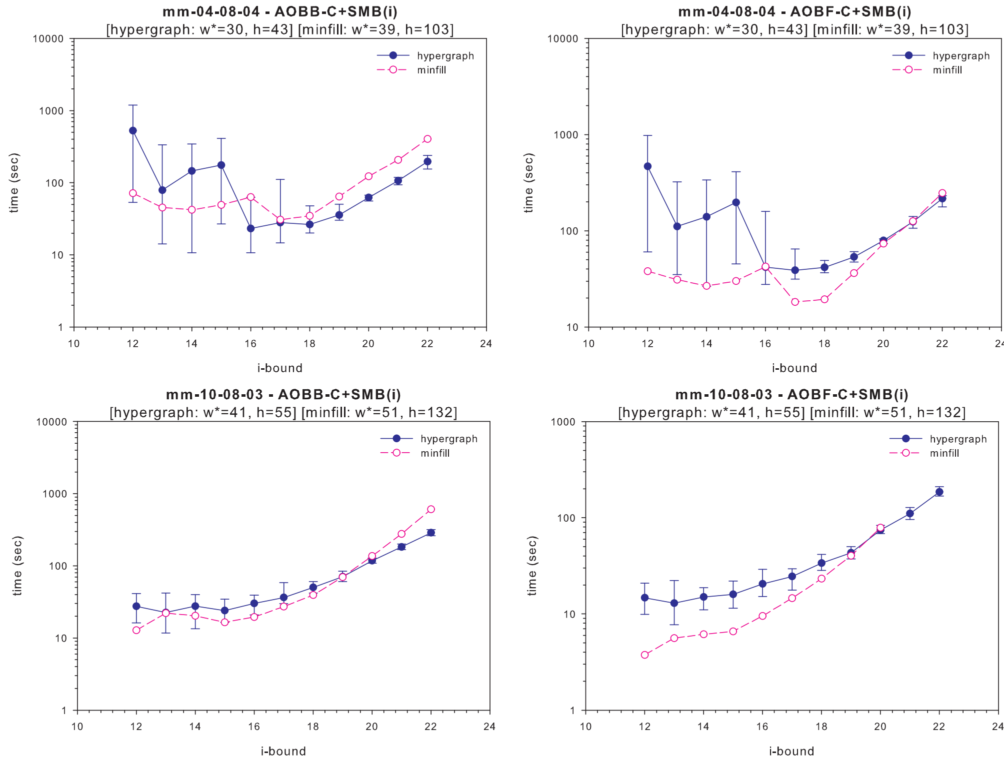


Fig. 15. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **Mastermind networks** with `AOBB-C+SMB(`i`)` (left side) and `AOBF-C+SMB(`i`)` (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

7.3.3 Summary of Empirical Results on Weighted CSPs

Our extensive empirical evaluation on WCSPs demonstrated that the best performance on this domain was obtained by best-first AND/OR search with static mini-bucket heuristics, for large i-bounds, especially on non-binary WCSPs with relatively small domain sizes ($e.g.$, Mastermind game instances, ISCAS'89 networks,

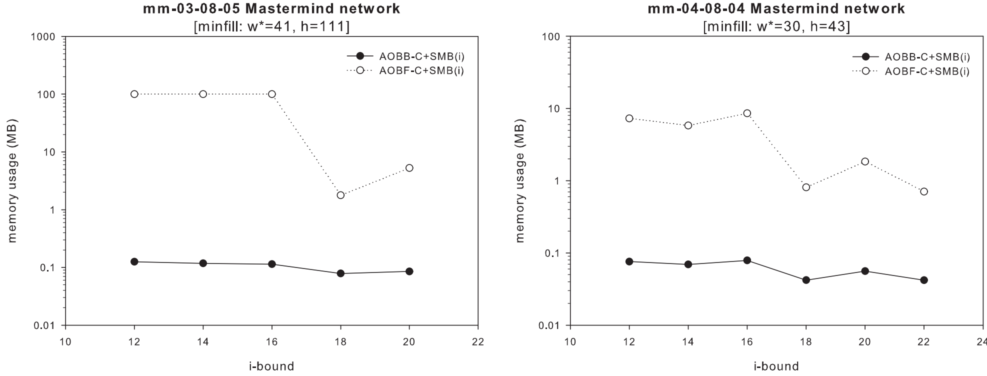


Fig. 16. Memory usage of the AOBB-C+SMB(i) and AOBF-C+SMB(i) algorithms on two **Mastermind networks** from Table 10.

instances from the SPOT5 benchmark). AOBF-C+SMB(i) dominated all its competitors, including the depth-first AOBB-C+SMB(i) as well as OR and AND/OR algorithms that enforce EDAC during search, namely toolbar, toolbar-BTD and the AOEDAC family of algorithms. Best-first AND/OR search with dynamic mini-bucket heuristics AOBF-C+DMB(i) was competitive only for relatively small i-bounds ($e.g.$, ISCAS'89 networks [26,27]). We also observed that the depth-first AND/OR Branch-and-Bound with caching and static mini-bucket heuristics AOBB-C+SMB(i) improved considerably over the cache-less version of the algorithm, namely AOBB+SMB(i). For dynamic mini-bucket heuristics, the difference between AOBB-C+DMB(i) and AOBB+DMB(i) was less prominent.

8 Summary and Conclusion

The paper extends the study of the impact of AND/OR search in graphical models from linear space search of the AND/OR tree to cache-based search of the AND/OR graph. In contrast to the traditional OR space, the AND/OR search space is sensitive to problem decomposition yielding the AND/OR search tree which can be bounded exponentially by the depth of its guiding pseudo tree. Specifically, if the graphical model has treewidth w^*, the size of the AND/OR search tree is bounded by $O(k^{w^* \log n})$ [2,18,1]. By recognizing identical subtrees, the AND/OR search tree can be extended into a graph yielding the context minimal AND/OR search graph whose size is exponential in the treewidth. The size of the context minimal OR search graph is exponential in the pathwidth. Since for some graphs the difference between treewidth and pathwidth is substantial ($e.g.$, balanced pseudo trees) the AND/OR representation implies substantial time and space savings for memory intensive algorithms traversing the AND/OR graph.

In this paper we extended the AND/OR Branch-and-Bound algorithm to traversing an AND/OR search graph by equipping it with an efficient caching mechanism. We investigated two flexible context-based caching schemes that can adapt to memory

restrictions. Since best-first search strategies are known to be superior to depth-first ones when memory is utilized, we also introduced a best-first AND/OR search algorithm that traverses the same context minimal AND/OR search graph.

All these algorithms can be guided by any heuristic function. We investigated extensively the mini-bucket heuristics introduced earlier [10] and shown to be effective in the context of the traditional OR search trees [10]. The mini-bucket heuristics can be either pre-compiled (static mini-buckets) or generated dynamically during search at each node in the search space (dynamic mini-buckets). They are parameterized by an i-bound which allows to control trade-off between heuristic strength and computational overhead.

We focused our empirical evaluation on two common optimization problems in graphical models: finding the MPE in Bayesian networks and solving combinatorial problems expressed as Weighted CSPs. Our results showed conclusively that the depth-first and best-first memory intensive AND/OR search algorithms guided by mini-bucket heuristics improve dramatically over traditional memory intensive OR search as well as over AND/OR search without caching. We summarize next the most important aspects reflecting the better performance of AND/OR graph search, such as the impact of the level of caching, the mini-bucket i-bound, constraint propagation, informed initial upper bounds and the quality of the guiding pseudo trees.

- **Impact of the caching level.** We proposed two parameterized context-based caching schemes that can adapt to the memory limitations. The naive caching records contexts with size smaller or equal to a cache bound j, while the adaptive caching saves also nodes whose context size is beyond j, based on adjusted contexts. Our results showed that for small j-bounds, adaptive caching is more powerful than the naive scheme ($e.g.$, grid networks from Figure 6, genetic linkage networks from Figure 10). As more space becomes available and as the j-bound increases, the two schemes gradually approach full caching. The savings in number of nodes due to both caching schemes are more pronounced at relatively small i-bounds of the mini-bucket heuristics. When the heuristics are strong enough to prune the search space substantially ($i.e.$, large i-bounds), the context minimal graph traversed by AND/OR Branch-and-Bound is very close to a tree and the effect of caching is reduced.
- **Impact of the mini-bucket i-bound.** Our results show conclusively that when enough memory is available the static mini-bucket heuristics with relatively large i-bounds are cost effective ($e.g.$, genetic linkage analysis networks from Tables 3 and 4, Mastermind game instances from Table 10). However, if the space is severely restricted, dynamic mini-bucket heuristics appear to be the preferred choice, especially for relatively small i-bounds. These heuristics are far more accurate for the same i-bound than the pre-compiled ones.
- **Impact of determinism.** When the graphical model contains both deterministic information (hard constraints) as well as general cost functions, we demon-

strated that it is beneficial to exploit the computational power of the constraints via constraint propagation. Our experiments on selected classes of deterministic Bayesian networks showed that enforcing unit resolution over the CNF encoding of the determinism present in the network yielded a tremendous reduction in running time ($e.g.$, deterministic grid networks from Table 7).
- **Impact of good initial upper bounds.** The AND/OR Branch-and-Bound algorithm assumed a trivial initial upper bound (resp. initial lower bound for maximization tasks). We incorporated a more informed upper bound (resp. lower bound for maximization), obtained by first solving the initial problem via local search. Our results showed a tremendous speed-up in some cases (see for example the grid network from Table 7).
- **Impact of pseudo tree quality.** The performance of the depth-first and best-first memory intensive AND/OR search algorithms is influenced significantly by the quality of the guiding pseudo tree. We investigated two heuristics for generating small induced width and/or depth pseudo trees. The min-fill based pseudo trees usually have smaller induced width but significantly larger depth, whereas the hypergraph partitioning heuristic produces much smaller depth trees but yields larger induced widths. Our experiments demonstrated that when the induced width is small enough, which is more typical for min-fill based pseudo trees, the strength of the mini-bucket heuristics compiled along these orderings determines the performance of the AND/OR search algorithms ($e.g.$, SPOT5 networks from Figure 13). However, when the graph is highly connected, the relatively large induced width causes the AND/OR algorithms to traverse a search space that is very close to a tree and, therefore, the hypergraph partitioning based pseudo trees, which tend to have smaller depths, improve performance substantially ($e.g.$, genetic linkage networks from Figure 9 and Table 5).

Our depth-first and best-first AND/OR graph search approaches leave room for future improvements, which are likely to make them more efficient in practice. The space required by `AOBB-C` and `AOBF-C` can be enormous, due to the fact that all nodes generated by the algorithms have to be stored in memory. Therefore, memory bounding strategies can be used for context minimal AND/OR graphs, as previously suggested in [19,21,47,48]. Alternatively, we can extend the AND/OR graph search algorithms to greatly expand the amount of available memory by utilizing external disk storage, as described in [49,50].

Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

[10] K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 129(1-2):91–131, 2001.

[11] R. Dechter and I. Rish. Mini-buckets: A general scheme for approximating inference. *Journal of the ACM*, 50(2):107–153, 2003.

[12] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan-Kaufmann, 1988.

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

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

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

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

[17] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying cluster-tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166(1–2):165–193, 2005.

[18] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(1):73–106, 2007.

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

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

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

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

[23] R. Mateescu and R. Dechter. AND/OR cutset conditioning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 230–235, 2005.

[24] A. Martelli and U. Montanari. Additive AND/OR graphs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1–11, 1973.

[25] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[26] R. Marinescu. *AND/OR Search Strategies for Combinatorial Optimization in Graphical Models*. PhD thesis, University of California, Irvine, 2008.

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

[28] S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *International Joint Conference in Artificial Intelligence (IJCAI)*, pages 84–89, 2005.

[29] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting tree decomposition and soft local consistency in weighted CSP. In *National Conference on Artificial Intelligence (AAAI)*, 2006.

[30] T. Sang, P. Beame, and H. Kautz. Solving Bayesian networks by weighted model counting. In *National Conference of Artificial Intelligence (AAAI)*, pages 475–482, 2005.

[31] Jurg Ott. *Analysis of Human Genetic Linkage*. The Johns Hopkins University Press, 1999.

[32] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. In *International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 189–198, 2002.

[33] M. Fishelson, N. Dovgolevsky, and D. Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 59(1):41–60, 2005.

[34] J. Park. Using weighted Max-SAT engines to solve MPE. In *National Conference of Artificial Intelligence (AAAI)*, pages 682–687, 2002.

[35] F. Hutter, H. Hoos, and T. Stutzle. Efficient stochastic local search for MPE solving. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 169–174, 2005.

[36] C. Voudouris. Guided local search for combinatorial optimization problems. Technical report, PhD Thesis. University of Essex, 1997.

[37] P. Mills and E. Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning (JAR)*, 24(1-2):205 – 223, 2000.

[38] R. Dechter and D. Larkin. Hybrid processing of beliefs and constraints. In *Uncertainty in Artificial Intelligence (UAI)*, pages 112–119, 2001.

[39] D. Larkin and R. Dechter. Bayesian inference in the presence of determinism. In *Artificial Intelligence and Statistics (AISTAT)*, 2003.

[40] D. Allen and A. Darwiche. New advances in inference using recursive conditioning. In *Uncertainty in Artificial Intelligence (UAI)*, pages 2–10, 2003.

[41] R. Dechter and R. Mateescu. Mixtures of deterministic-probabilistic networks. In *Uncertainty in Artificial Intelligence (UAI)*, pages 120–129, 2004.

[42] I. Rish and R. Dechter. Resolution vs. search: two strategies for SAT. *Journal of Automated Reasoning*, 24(1-2):225–275, 2000.

[43] T. Walsh. SAT vs CSP. In *Principles and Practice of Constraint Programming (CP)*, pages 441–456, 2000.

[44] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, 2001.

[45] E. Bensana, M. Lemaitre, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3):293–299, 1999.

[46] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1–2):4–20, 2006.

[47] P. Chakrabati, S. Ghose, A. Acharya, and S. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41(2):197–221, 1989.

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

[49] R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI-04)*, pages 683–689, 2004.

[50] R. Zhou and E. Hansen. External-memory pattern databases using structured duplicate detection. In *National Conference on Artificial Intelligence (AAAI-05)*, pages 1398–1405, 2004.