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

Radu Marinescu, Rina Dechter

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 on 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 depth-first and best-first approaches 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 including random and real-world problem instances.

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 focused on extending Branch-and-Bound algorithms to exploring the AND/OR search tree. The virtue of the AND/OR 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 the context minimal AND/OR search graph.

Email addresses: radum@ics.uci.edu (Radu Marinescu), dechter@ics.uci.edu (Rina Dechter).

Graphical models such as belief networks or constraint networks are a widely used representation framework for reasoning with probabilistic and deterministic information. These models use graphs to capture conditional independencies between variables, allowing a concise representation of the knowledge as well as efficient graph-based query processing algorithms. Combinatorial optimization problems such as finding the most likely state of a belief network or finding a solution that violates the least number of constraints can be defined within this framework and they are typically tackled with either inference or search algorithms.

Inference-based algorithms ($e.g.$, Variable Elimination, Tree Clustering) are good at exploiting the independencies captured by the underlying graphical model. They provide superior worst case time guarantees, compared to simple search, as they are time exponential in the treewidth of the problem's graph. Since those methods are also space exponential in the treewidth they are not practical for models with large treewidth. Search algorithms on the other hand, can trade off time and space in a more flexible manner.

Search-based algorithms ($e.g.$, depth-first Branch-and-Bound, best-first search) traverse the model's search space where each path represents a partial or full solution. In [1–3] we presented the AND/OR search tree for optimization tasks over graphical models and showed how this framework can exploit problem decomposition during Branch-and-Bound search [4]. As is known, the AND/OR search space can provide exponential speedups over the traditional OR search methods oblivious to problem structure: its worst-case size is exponential in the product of the graph treewidth and the logarithm of the number of variables. This yield a bound on the time complexity of any traversal algorithm, yet allows exploration using linear space. In this paper we explore the added benefit of using more space, and we show that by allowing the use of caching on top of AND/OR search, we can have a significant additional gain and we can even reach the same worst-case time and space exponential in the treewidth bounds, obeyed by inference algorithms. The advantage of search with caching, compared with inference, is in facilitating pruning due to determinism, due to the guiding heuristic function and due to context-sensitive properties. These aspects make the memory demand in search less severe than in inference, in practice.

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 used in several recent schemes such as Recursive Conditioning [5], Valued Backtracking [6] and Backtracking with Tree Decompositions [7]. Our contributions beyond those earlier schemes is in presenting these ideas in an independent manner using the notion of AND/OR search spaces, and in our extensive empirical study.

Since best-first search is known to be superior among memory intensive search al-

gorithms [8], we also 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 [8]. As we will show, these savings in number of nodes often translate into significant time savings .

Clearly, 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 [9] which is extracted automatically, from the functional specification of the graphical model using the Mini-Bucket approximation algorithm [10]. Since the accuracy of the Mini-Bucket algorithm is controlled by a bounding parameter, called i-bound, it allows heuristics having varying degrees of accuracy and results in a spectrum of search algorithms that can trade off heuristic computation and search. Following [1,2], we continue to explore empirically the efficiency of static and dynamic mini-bucket heuristics within the cache-based search spaces.

Like in [1–3], we apply the algorithms to finding the Most Probable Explanation (MPE) in belief networks [11] and to solving Weighted CSPs [12].We experiment with both random models and real-world benchmarks. 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 guiding lower bound strength, 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 quality guiding ordering schemes ($e.g.$, pseudo trees).

Following preliminary notations and definitions (Section 2), Sections 3 and 4 provide background on graphical models and the AND/OR representation of the search space. Sections 5 and 6 present the new depth-first and best-first AND/OR search algorithms exploring the context minimal AND/OR graph. Section 7 reviews the mini-bucket heuristics for AND/OR search. In Section 8 we present an extensive empirical evaluation of the proposed memory intensive search methods, while Sections 9 and 10 provide a summary of related work as well as concluding remarks and directions of future research. For completeness, Sections 2 through 4.2 repeat much of the introductory material from [1]. This paper is based in part on [13–15].

2 Preliminaries

2.1 Notations

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 $scope(f)$.

2.2 Graph Concepts

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 width)** *An* ordered graph *is a pair (G, d) where G is an undirected graph, and $d = X_1, ..., X_n$ is an ordering of the nodes. The* width of a node *is the number of the node's neighbors that precede it in the ordering. The* width of an ordering d *is the maximum width over all nodes. The* induced width of an ordered graph, *denoted by $w^*(d)$, is the width of the induced ordered graph obtained as follows: nodes are processed from last to first; when node X_i is processed, all its preceding neighbors are connected. The* induced width *of a graph, denoted by w^*, is the minimal induced width over all its orderings.*

DEFINITION **3 (hypergraph)** *A* hypergraph *is a pair $H = (\mathbf{X}, \mathbf{S})$, where $\mathbf{S} = \{S_1, ..., S_t\}$ is a set of subsets of $\mathbf{X}$, called hyperedges.*

DEFINITION **4 (tree decomposition)** *A* tree decomposition *of a hypergraph $H = (\mathbf{X}, \mathbf{S})$, is a tree $T = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is a set of nodes, also called "clusters", and $\mathbf{E}$ is a set of edges, together with a labeling function χ that associates with each vertex $v \in \mathbf{V}$ a set $\chi(v) \subseteq \mathbf{X}$ satisfying:*

(1) For each $S_i \in \mathbf{S}$ there exists a vertex $v \in \mathbf{V}$ such that $S_i \subseteq \chi(v)$;
(2) (running intersection property) For each $X_i \in \mathbf{X}$, the set $\{v \in \mathbf{V} | X_i \in \chi(v)\}$ induces a connected subtree of T.

DEFINITION **5 (treewidth, pathwidth)** *The* width *of a tree decomposition of a hypergraph is the size of the largest cluster minus 1 (i.e., $max_v|\chi(v) - 1|$). The* treewidth *of a hypergraph is the minimum width along all possible tree decompositions. The* pathwidth *is the treewidth over the restricted class of chain decompositions.*

3 Graphical Models

Graphical models include constraint networks defined by relations of allowed tuples, directed or undirected probabilistic networks and cost networks defined by cost functions. Each graphical model comes with its specific optimization queries such as finding a solution of a constraint network that violates the least number of constraints, finding the most probable assignment given some evidence, posed over probabilistic networks or finding the optimal solution for cost networks.

In general, 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.

DEFINITION **6 (graphical model)** *A* graphical model $\mathcal{R}$ *is defined by 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 the set of their respective 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: $\otimes_{i=1}^{r} f_i$.

DEFINITION **7 (cost of a full and partial assignment)** *Given a graphical model $\mathcal{R}$, the cost of a full assignment $x = (x_1, ..., x_n)$ is defined by:*

$$c(x) = \otimes_{f \in \mathbf{F}} f(x[scope(f)])$$

Given a subset of variables $\mathbf{Y} \subseteq \mathbf{X}$, the cost of a partial assignment y is the combination of all the functions whose scopes are included in $\mathbf{Y}$, namely $\mathbf{F_Y}$, evaluated at the assigned values. Namely, $c(y) = \otimes_{f \in \mathbf{F_Y}} f(y[scope(f)])$. We will often abuse notation writing $c(y) = \otimes_{f \in F_\mathbf{Y}} f(y)$ instead.

DEFINITION **8 (primal graph)** *The* primal graph *of a graphical model has the variables as its nodes and an edge connects 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 **9 (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 sometimes called an *elimination* operator because it removes the arguments from the input functions' scopes.

We next elaborate on several popular graphical models of constraint networks and belief networks which will be the primary focus of this paper (see also [1] for detailed examples of each of these graphical models).

3.1 Constraint Networks

Constraint Satisfaction is a framework for formulating real-world problems as a set of constraints between variables. The task is to find an assignment of values to variables that does not violate any constraint, or else to conclude that problem is inconsistent. Such problems are graphically represented by nodes corresponding to variables and edges corresponding to constraints between variables.

DEFINITION **10 (constraint network)** *A* constraint network *is a graphical model $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$, where $\mathbf{X} = \{X_1, ..., X_n\}$ is a set of variables, associated with discrete-valued domains $\mathbf{D} = \{D_1, ..., D_n\}$, and a set of constraints $\mathbf{C} = \{C_1, ..., C_r\}$. Each constraint C_i is a pair (S_i, R_i), where R_i is a relation $R_i \subseteq D_{S_i}$ defined on a subset of variables $S_i \subseteq \mathbf{X}$. The relation denotes all compatible tuples of D_{S_i} allowed by the constraint. The combination operator, $\bowtie$, is join. The primal graph of a constraint network is called a* constraint graph*. A solution is an assignment of values to all variables $x = (x_1, ..., x_n)$, $x_i \in D_i$, such that $\forall C_i \in \mathbf{C}$, $x_{S_i} \in R_i$. The constraint network represents its set of solutions, $\bowtie_i C_i$.*

3.2 Cost Networks

An immediate extension of constraint networks are *cost networks* where the set of functions are real-valued cost functions, the combination and elimination operators are *summation* and *minimization*, respectively, and the primary constraint optimization task is to find a solution with minimum cost.

A special class of COPs which has gained a lot of interest in recent years is the Weighted Constraint Satisfaction Problem (WCSP). WCSP extends the classical CSP formalism with *soft constraints* which are represented as integer-valued cost functions. Formally,

DEFINITION **11 (WCSP)** *A* Weighted Constraint Satisfaction Problem (WCSP) *is a cost network* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$ *where each of the cost functions* $F_i \in \mathbf{F}$ *assigns "0" (no penalty) to allowed tuples and a positive integer penalty cost to the forbidden tuples. Namely,* $f_i : D_{S_{i_1}} \times ... \times D_{S_{i_t}} \rightarrow \mathbb{N}$*, where* $S_i = \{S_{i_1}, ..., S_{i_t}\}$ *is the scope of the cost function. The optimization problem is to find a value assignment to the variables with minimum penalty cost, namely finding* $\Downarrow_{\mathbf{X}} \otimes_i f_i = min_{\mathbf{X}} \sum_i f_i$*.*

DEFINITION **12 (MAX-CSP)** *A* MAX-CSP *is a WCSP with all penalty costs equal to* 1*. Namely,* $\forall f_i \in \mathbf{F}$*,* $f_i : D_{S_{i_1}} \times ... \times D_{S_{i_t}} \rightarrow \{0, 1\}$*, where* $S_i = \{S_{i_1}, ..., S_{i_t}\}$ *is the scope of* f_i*.*

Solving a MAX-CSP can also be interpreted as finding an assignment that violates the minimum number of constraints (or maximizes the number of satisfied constraints). Many real-world problems can be formulated as MAX-CSP/WCSPs, including resource allocation problems [16], scheduling [17], bioinformatics [18,19], combinatorial auctions [20,21] or maximum satisfiability problems [22].

3.3 Belief Networks

Belief networks [11] provide a formalism for reasoning about partial beliefs under conditions of uncertainty. They are defined by a directed acyclic graph over vertices representing variables of interest ($e.g.$, the temperature of a device, the gender of a patient, a feature of an object, the occurrence of an event). The arcs signify the existence of direct causal influences between linked variables quantified by conditional probabilities that are attached to each cluster of parents-child vertices in the network.

DEFINITION **13 (belief network)** *A* belief network (BN) *is a graphical model* $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$*, where* $\mathbf{X} = \{X_1, ..., X_n\}$ *is a set of variables over multi-valued domains* $\mathbf{D} = \{D_1, ..., D_n\}$*. Given a directed acyclic graph* G *over* $\mathbf{X}$ *as nodes,* $\mathbf{P}_G = \{P_i\}$*, where* $P_i = \{P(X_i|pa(X_i))\}$ *are conditional probability tables (CPTs*

for short) associated with each variable X_i, and $pa(X_i)$ are the parents of X_i in the acyclic graph G. A belief network represents a joint probability distribution over $\mathbf{X}$, $P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i|x_{pa(X_i)})$. *An evidence set e is an instantiated subset of variables.*

When formulated as a graphical model, the functions in $\mathbf{P}_G$ denote conditional probability tables and the scopes of these functions are determined by the directed acyclic graph G: each function f_i ranges over variable X_i and its parents in G. The combination operator is multiplication, namely $\otimes_j = \prod_j$. The primal graph of a belief network is called a *moral graph*. It connects any two variables appearing in the same probability table.

DEFINITION **14 (most probable explanation)** *Given a belief network and evidence e, the* Most Probable Explanation *(MPE) task is to find a complete assignment which agrees with the evidence, and which has the highest probability among all such assignments. Namely, to find an assignment $(x_1^o, ..., x_n^o)$ such that:*

$$P(x_1^o, ..., x_n^o) = max_{x_1,...,x_n} \prod_{k=1}^{n} P(x_k, e|x_{pa_k})$$

As a reasoning problem, the MPE task is to find $\Downarrow_{\mathbf{X}} \otimes_i f_i = max_{\mathbf{X}} \prod_{i=1}^{n} P_i$.

4 AND/OR Search Spaces for Graphical Models

4.1 AND/OR Search Trees 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 [4]. It specializes the AND/OR space introduced in [23] to graphical models. The AND/OR search space is defined using a backbone *pseudo tree* [24,25].

DEFINITION **15 (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 $\mathcal{T}$. Given a pseudo tree $\mathcal{T}$ of G, the* extended graph *of G relative to $\mathcal{T}$ is defined as $G^{\mathcal{T}} = (\mathbf{V}, \mathbf{E} \cup \mathbf{E}')$.*

DEFINITION **16 (AND/OR search tree)** *Given a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$,*

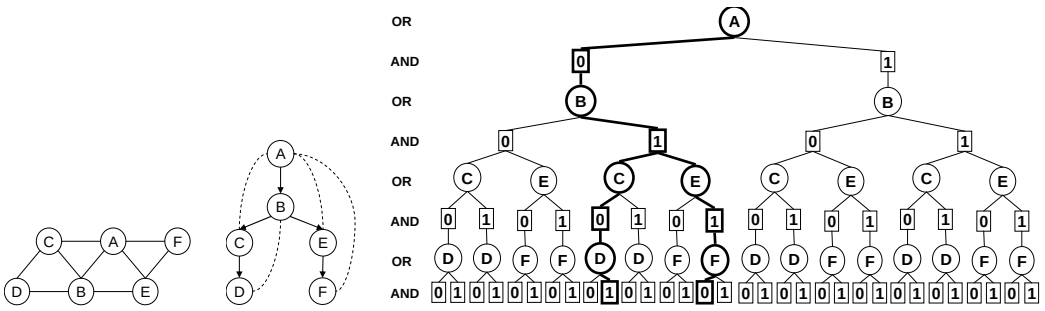


Fig. 1. AND/OR search tree for graphical models.

its primal graph G and a backbone pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree, denoted $S_{\mathcal{T}}(\mathcal{R})$, has alternating levels of AND and OR nodes. The OR nodes are labeled X_i and correspond to the variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ (or simply x_i) and correspond to value assignments in the domains of the variables. The structure of the AND/OR search tree is based on the underlying backbone pseudo tree $\mathcal{T}$. The root of the AND/OR search tree is an OR node labeled with the root of $\mathcal{T}$. A path from the root of the search tree $S_{\mathcal{T}}(\mathcal{R})$ to a node n is denoted by π_n. If n is labeled X_i or x_i the path will be denoted $\pi_n(X_i)$ or $\pi_n(x_i)$, respectively. The assignment sequence along path π_n, denoted $asgn(\pi_n)$, is the set of value assignments associated with the AND nodes along π_n:

$$asgn(\pi_n(X_i)) = \{\langle X_1, x_1 \rangle, \langle X_2, x_2 \rangle, ..., \langle X_{i-1}, x_{i-1} \rangle\}$$
$$asgn(\pi_n(x_i)) = \{\langle X_1, x_1 \rangle, \langle X_2, x_2 \rangle, ..., \langle X_i, x_i \rangle\}$$

The set of variables associated with OR nodes along the path π_n is denoted by $var(\pi_n)$: $var(\pi_n(X_i)) = \{X_1, ..., X_{i-1}\}$, $var(\pi_n(x_i)) = \{X_1, ..., X_i\}$. The parent-child relationship between nodes in the search space are defined as follows:

(1) An OR node, n, labeled by X_i has a child AND node labeled $\langle X_i, x_i \rangle$ iff $\langle X_i, x_i \rangle$ is consistent with $asgn(\pi_n)$.
(2) An AND node, n, labeled by $\langle X_i, x_i \rangle$ has a child OR node labeled Y iff Y is a child of X_i in the backbone pseudo tree $\mathcal{T}$. Each OR arc, emanating from an OR to an AND node is associated with a weight to be defined shortly.

Clearly, if a node n is labeled X_i (OR node) or x_i (AND node), $var(\pi_n)$ is the set of variables mentioned on the path from the root to X_i in the backbone pseudo tree, denoted by $path_{\mathcal{T}}(X_i)$.

Semantically, the OR states in the AND/OR search tree represent alternative ways of solving a problem, whereas the AND states represent problem decomposition into independent subproblems, all of which need to be solved.

DEFINITION **17 (solution 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 of $S_{\mathcal{T}}(\mathcal{R})$, s;
(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.

Example 1 *Figure 1(a) shows the primal graph of cost network with 6 bi-valued variables A, B, C, D, E and F, and 9 binary cost functions. Figure 1(b) displays a pseudo tree together with the back-arcs (dotted lines). Figure 1(c) shows the AND/OR search tree based on the pseudo tree. A solution subtree is highlighted. Notice that once variables A and B are instantiated, the search space below the AND node $\langle B, 0\rangle$ decomposes into two independent subproblems, one that is rooted at C and one that is rooted at E, respectively.*

The virtue of an AND/OR search tree representation is that its size may be far smaller than the traditional OR search tree.

THEOREM **1 (size of AND/OR search trees [4])** *Given a graphical model $\mathcal{R}$ and a backbone pseudo tree $\mathcal{T}$, its AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is sound and complete, and its size is $O(l \cdot k^m)$ where m is the depth of the pseudo tree, l bounds its number of leaves, and k bounds the domain size.*

Given a *tree decomposition* of the primal graph G having n nodes, whose treewidth is w^*, there exists a pseudo tree $\mathcal{T}$ of G whose depth, m, satisfies: $m \leq w^* \cdot logn$ [26,27]. Therefore,

THEOREM **2 ([4])** *A graphical model that has a treewidth w^* has an AND/OR search tree whose size is $O(n \cdot k^{w^* \cdot logn})$, where k bounds the domain size and n is the number of variables.*

The arcs in the AND/OR trees are associated with weights that are defined based on the graphical model's functions and combination operator. We next define arc weights for any graphical model using the notion of *buckets of functions*.

DEFINITION **18 (buckets relative to a pseudo tree)** *Given a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}\rangle$ and a backbone pseudo tree $\mathcal{T}$, the bucket of X_i relative to $\mathcal{T}$, denoted $B_{\mathcal{T}}(X_i)$, is the set of functions whose scopes contain X_i and are included in $path_{\mathcal{T}}(X_i)$, which is the set of variables from the root to X_i in $\mathcal{T}$. Namely,*

$$B_{\mathcal{T}}(X_i) = \{f \in \mathbf{F} | X_i \in scope(f), scope(f) \subseteq path_{\mathcal{T}}(X_i)\}$$

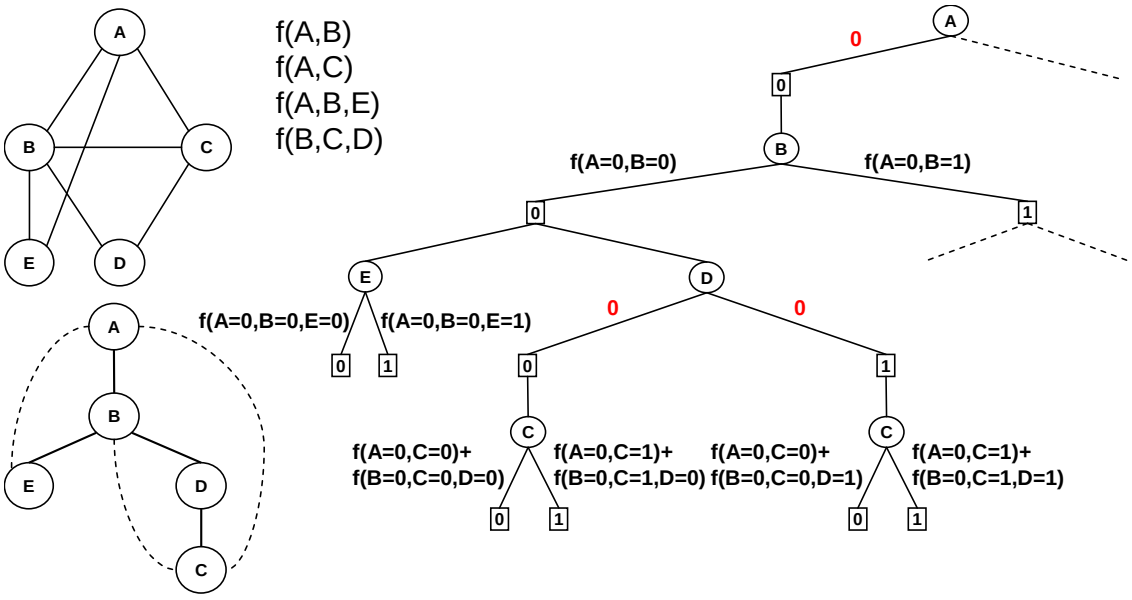


Fig. 2. Arc weights for cost networks.

For simplicity and without loss of generality we consider in the remainder a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ for which the combination and elimination operators are *summation* and *minimization*, respectively.

DEFINITION **19 (OR-to-AND weights)** *Given an AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$, of a graphical model $\mathcal{R}$, the weight $w_{(n,m)}(X_i, x_i)$ (or simply $w(X_i, x_i)$) of arc (n, m), where X_i labels n and x_i labels m, is the* combination *of all the functions in $B_{\mathcal{T}}(X_i)$ assigned by values along π_m. Formally,*

$$w(X_i, x_i) = \begin{cases} 0 & \text{, if } B_{\mathcal{T}}(X_i) = \emptyset \\ \sum_{f \in B_{\mathcal{T}}(X_i)} f(asgn(\pi_m)[scope(f)]) & \text{, otherwise} \end{cases}$$

DEFINITION **20 (cost of a solution tree)** *Given a weighted AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$, of a graphical model $\mathcal{R}$, and given a solution tree T having OR-to-AND set of arcs $arcs(T)$, the cost of T is defined by $f(T) = \sum_{e \in arcs(T)} w(e)$.*

We define $f(T_n)$ the cost of a solution tree rooted at node n. Then $f(T_n)$ can be computed recursively, as follows:

1. *If T_n consists only of a terminal AND node n, then $f(T_n) = 0$.*
2. *If n is an OR node having an AND child m in T_n, then $f(T_n) = w(n, m) + f(T_m)$, where T_m is the solution subtree of T_n that is rooted at m.*
3. *If n is an AND node having OR children $m_1, ..., m_k$ in T_n, then $f(T_n) = \sum_{i=1}^{k} f(T_{m_i})$, where T_{m_i} is the solution subtree of T_n rooted at m_i.*

Example 2 *Figure 2 shows the primal graph of a cost network, a pseudo tree that*

drives its weighted AND/OR search tree, and a portion of the AND/OR search tree with appropriate weights on the arcs expressed symbolically. In this case the bucket of E contains the function $f(A, B, E)$, the bucket of C contains two functions $f(A, C)$ and $f(B, C, D)$ and the bucket of B contains the function $f(A, B)$. We see indeed that the weights on the arcs from the OR node E to any of its AND value assignments include only the instantiated function $f(A, B, E)$, while the weights on the arcs connecting C to its AND child nodes are the sum of the two functions in its bucket instantiated appropriately. Notice that the buckets of A and D are empty and therefore the weights associated with the respective arcs are 0.

With each node n of the search tree we can associate a value $v(n)$ which stands for the answer to the particular query restricted to the subproblem below n [4].

DEFINITION **21 (value function)** *Given an optimization problem $\mathcal{P} = \langle \mathcal{R}, min \rangle$ over a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$, the* value function *of a node n in the AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is the optimal cost to the subproblem below n.*

The value of a node can be computed recursively, as follows: it is 0 for terminal AND nodes and ∞ for terminal OR nodes, respectively. The value of an internal OR node is obtained by combining (summing) *the value of each AND child node with the weight on its incoming arc and then* optimize (minimize) *over all AND children. The value of an internal AND node is the* combination (summation) *of values of its OR children. Formally, if $succ(n)$ denotes the children of the node n in the AND/OR search tree, then:*

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

If n is the root of $S_{\mathcal{T}}(\mathcal{R})$, then $v(n)$ is the minimal cost solution to the initial problem. Alternatively, the value $v(n)$ can also be interpreted as the minimum of the costs of the solution trees rooted at n. Therefore, search algorithms that traverse the AND/OR search space can compute the value of the root node yielding the answer to the problem. It can be immediately inferred from Theorems 1 and 2 that:

THEOREM **3 (complexity [4])** *A depth-first search algorithm traversing an AND/OR search tree for finding the minimal cost solution is time $O(n \cdot k^m)$, where k bounds the domain size and m is the depth of the pseudo tree, and may use linear space. If the primal graph has a tree decomposition with treewidth w^*, there there exists a pseudo tree $\mathcal{T}$ for which the time complexity is $O(n \cdot k^{w^* \cdot log n})$.*

4.2 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'. The algorithm then computes a heuristic lower bounding estimate $f(T')$ on the optimal cost extension of T' to a complete solution tree and prunes the search space below the current tip node if $f(T') \geq ub$.

The efficiency of this algorithm depends heavily on its guiding heuristic function. Subsequently, in [1,2] we investigated the power of a heuristic generation scheme based on the Mini-Bucket approximation [10], 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 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, the differences number of nodes visited as well as running time added up to several orders of magnitude.

In the following subsection we overview the notion of AND/OR search *graph* for general graphical models, which was presented in [4].

4.3 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 are merged, because identical nodes root identical search subspaces, and correspond to identical reasoning subproblems. Any two nodes that root identical weighted can be *merged*, 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* G [4] 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 **22 (induced width of a pseudo tree)** *The induced width of G relative to a pseudo tree $\mathcal{T}$, $w_{\mathcal{T}}(G)$, is the induced width along $d_{DFS}(\mathcal{T})$ ordering of the extended graph of G relative to $\mathcal{T}$, denoted $G^{\mathcal{T}}$.*

We now provide definitions which allow identifying nodes that can be merged. The idea is to find a minimal set of variable assignments from the current path that will always root the same conditioned subproblem, regardless of the assignments that are not included in this minimal set. Since the path for an OR node X_i and an AND node $\langle X_i, x_i \rangle$ differ by the assignment of X_i to x_i (Definition 16), the minimal set of assignments that we want to identify will be different for X_i and for $\langle X_i, x_i \rangle$. The following definitions distinguish between two types of context-based caching which may yield into two different schemes. The difference may seem a bit subtle. In these definitions, ancestors and descendants are with respect to the pseudo tree $\mathcal{T}$, while connection is with respect to the primal graph G.

DEFINITION **23 (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 that have connections in G to X_i or to descendants of X_i.*

DEFINITION **24 (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 have connections in G to descendants of X_i.*

It follows from these definitions that the parents of X_i, pa_i, separate in the primal 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 $G^{\mathcal{T}^*}$. The following proposition establishes the relation between pa_i and pas_i.

PROPOSITION **1 ([4])** *(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 **4 (context based merge [4])** *Given $G^{\mathcal{T}^*}$, let π_{n_1} and π_{n_2} be any two partial paths in an AND/OR search graph, ending with two nodes, n_1 and n_2.*

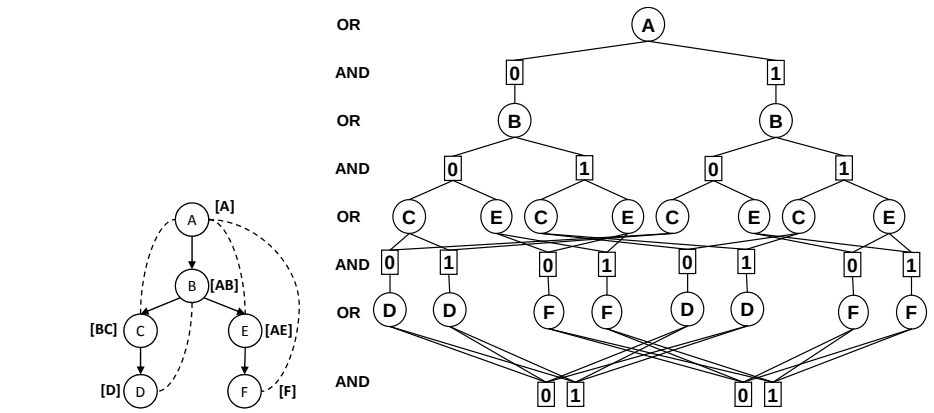
(1) If n_1 and n_2 are AND nodes annotated by $\langle X_i, x_i \rangle$ and

$$asgn(\pi_{n_1})[pas_{X_i}] = asgn(\pi_{n_2})[pas_{X_i}]$$

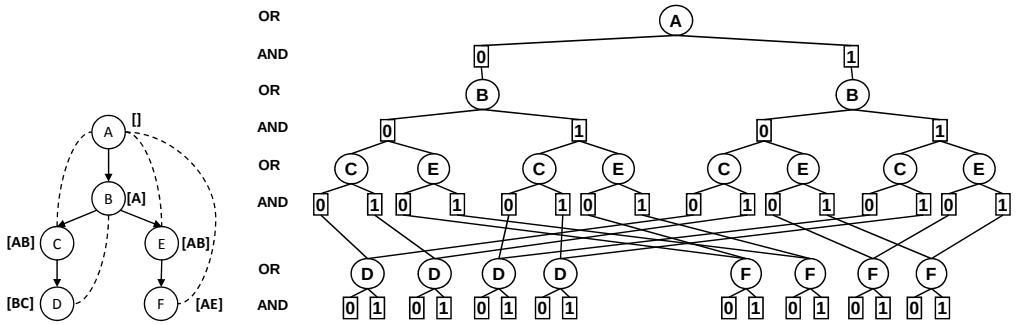
*then the AND/OR search subtrees rooted by n_1 and n_2 are identical. $asgn(\pi_{n_i})[pas_{X_i}]$ is called the **AND context** of n_i.*

(2) If n_1 and n_2 are OR nodes annotated by X_i and

$$asgn(\pi_{n_1})[pa_{X_i}] = asgn(\pi_{n_2})[pa_{X_i}]$$



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



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

Fig. 3. AND/OR search graph for graphical models

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

DEFINITION **25 (context minimal AND/OR search graph)** *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* 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, Proposition 1 shows that doing just one of them renders the other unnecessary (up to some some small constant factor). In this paper we will use AND context based merging.

THEOREM **5 (complexity [4])** *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 3 *Consider the example given in Figure 3(a). The AND contexts of each node in the pseudo tree is given in square brackets. The context minimal AND/OR search graph (based on AND merging) is given in Figure 3(b). Its size is far smaller*

than that of the AND/OR search tree from Figure 1(c) (16 vs. 54 AND nodes). Similarly, Figure 3(d) shows the context minimal AND/OR graph based on the OR contexts given in Figure 3(c). Its size is larger than that of the AND based graph (38 vs. 16 AND nodes) in this case.

4.4 Finding Good Pseudo Trees

The performance of any AND/OR search algorithm is influenced heavily by the quality of the pseudo tree. In [1–3] we described two heuristics for generating small induced width/depth pseudo trees. The *min-fill* heuristic extracts the pseudo tree by a depth-first traversal of the induced graph obtained by a min-fill elimination ordering [28]. The *hypergraph partitioning* heuristic constructs the pseudo tree by recursively decomposing the dual hypergraph associated with the graphical model [5]. We observed in [1–3] that the min-fill heuristic usually generates lower width trees, whereas the hypergraph heuristic produces much smaller depth trees. Therefore, the hypergraph based pseudo trees appear to be favorable for tree search algorithms, while the min-fill pseudo trees, which minimize the context size, may be more appropriate for graph search algorithms. Both heuristics can randomize their tie breaking rule, yielding varying qualities of the generated pseudo trees. In the experimental section we provide an extensive evaluation detailing the impact of the pseudo tree quality on the AND/OR search algorithms.

5 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 [23,29,30] in the context of general search spaces. In the following two sections we revisit the necessary definitions needed to describe the algorithms such as the notion of partial solution trees [23] to represent sets of solution trees, the exact evaluation as well as the heuristic evaluation function of a partial solution tree. 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 the algorithm presented in [1–3] for exploring AND/OR search trees to exploring AND/OR search graphs using a flexible context-based caching scheme that can adapt to the current memory limitations.

DEFINITION **26 (partial solution tree)** *A* partial solution tree T' *of a context minimal AND/OR search graph $C_{\mathcal{T}}$ is a subtree which: (1) contains the root node s of $C_{\mathcal{T}}$; (2) if n in T' is an OR node then it contains one of its AND child nodes in $C_{\mathcal{T}}$, and if n is an AND node it contains all its OR children in $C_{\mathcal{T}}$. 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 $C_{\mathcal{T}}$), or a* non-terminal *node (if it has children in $C_{\mathcal{T}}$).*

A partial solution tree can be extended (possibly in several ways) to a full solution tree. It represents $extension(T')$, the set of all full solution trees which can extend it. Clearly, a partial solution tree all of whose tip nodes are terminal in $C_{\mathcal{T}}$ is a solution tree. We next define the exact evaluation function of a partial solution tree, and will then derive the notion of a lower bound for it.

DEFINITION **27 (exact evaluation function of a partial solution tree)** *The* exact evaluation function $f^*(T')$ *of a partial solution tree T' is the minimum of the costs of all solution trees represented by T', namely:*

$$f^*(T') = min\{f(T) \mid T \in extension(T')\}$$

We define $f^*(T'_n)$ 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 $f^*(T'_n) = w(n,m) + f^*(T'_m)$, where T'_m is the partial solution subtree of T'_n that is rooted at m.*
3. *If n is an AND node having OR children $m_1, ..., m_k$ in T'_n, then $f^*(T'_n) = \sum_{i=1}^{k} f^*(T'_{m_i})$, where T'_{m_i} is the partial solution subtree of T'_n rooted at m_i.*

Clearly, we are interested to find the $f^*(T')$ of a partial solution tree T' rooted at the root s. If each non-terminal tip node n of T' is assigned a heuristic lower bound estimate $h(n)$ of $v(n)$, then it induces a heuristic lower bound evaluation function on the minimal cost of T', as follows.

DEFINITION **28 (heuristic evaluation function of a partial solution tree)** *Given a node-based heuristic function $h(m)$ which is a lower bound on the minimal cost below any node m, namely $h(m) \leq v(m)$, and given a partial solution tree T'_n rooted at node n in the context minimal AND/OR search graph $C_{\mathcal{T}}$, the* tree-based heuristic evaluation function $f(T'_n)$ *of 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 $f(T'_n) = w(n,m) + f(T'_m)$, where T'_m is the partial solution subtree of T'_n that is rooted at m.*
3. *If n is an AND node having OR children $m_1, ..., m_k$ in T'_n, then $f(T'_n) = \sum_{i=1}^{k} f(T'_{m_i})$, where T'_{m_i} is the partial solution subtree of T'_n rooted at m_i.*

During search we maintain an upper bound $ub(s)$ on the optimal solution $v(s)$ as well as the heuristic evaluation function of the current partial solution tree $f(T')$, and we can prune the search space by comparing these two measures, as is common in Branch-and-Bound search. Namely, if $f(T') \geq ub(s)$, then searching below the current tip node t of T' is guaranteed not to reduce $ub(s)$ and therefore, the search

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 $v(s) \leftarrow \infty$; $ST(s) \leftarrow \emptyset$; $OPEN \leftarrow \{s\}$ // Initialize search stack
2 Initialize cache tables with entries "NULL" // Initialize cache tables
3 **while** $OPEN \neq \emptyset$ **do**
4 $n \leftarrow top(OPEN)$; remove n from $OPEN$ // **EXPAND**
5 **if** *n is an OR node, labeled X_i* **then**
6 **foreach** $x_i \in D_i$ **do**
7 create an AND node n', labeled $\langle X_i, x_i \rangle$
8 $v(n') \leftarrow 0$; $ST(n') \leftarrow \emptyset$
9 $w(n, n') \leftarrow \sum_{f \in B_{\mathcal{T}}(X_i)} f(asgn(\pi_n))$ // Compute the OR-to-AND arc weight
10 $succ(n) \leftarrow succ(n) \cup \{n'\}$
11 **else if** *n is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
12 $cached \leftarrow false$; $deadend \leftarrow false$
13 **if** $Cache(asgn(\pi_n)[pas_i]) \neq NULL$ **then**
14 $v(n) \leftarrow Cache(asgn(\pi_n)[pas_i]).value$ // Retrieve value
15 $ST(n) \leftarrow Cache(asg(\pi_n)[pas_i]).assignment$; // Retrieve optimal assignment
16 $cached \leftarrow true$ // No need to expand below
17 **foreach** *OR ancestor m of n* **do**
18 $lb \leftarrow$ `evalPartialSolutionTree`(T'_m)
19 **if** $lb \geq v(m)$ **then**
20 $deadend \leftarrow true$
21 **break**
22 **if** $deadend == false$ ***and*** `cached` $== false$ **then**
23 **foreach** $X_j \in children_{\mathcal{T}}(X_i)$ **do**
24 create an OR node n' labeled X_j
25 $v(n') \leftarrow \infty$; $ST(n') \leftarrow \emptyset$
26 $succ(n) \leftarrow succ(n) \cup \{n'\}$
27 **else if** $deadend == true$ **then**
28 $succ(p) \leftarrow succ(p) - \{n\}$
29 Add $succ(n)$ on top of $OPEN$ // **PROPAGATE**
30 **while** $succ(n) == \emptyset$ **do**
31 **if** *n is an OR node, labeled X_i* **then**
32 **if** $X_i == X_1$ **then**
33 **return** $(v(n), ST(n))$ // Search is complete
34 $v(p) \leftarrow v(p) + v(n)$ // Update AND node value (summation)
35 $ST(p) \leftarrow ST(p) \cup ST(n)$ // Update solution tree below AND node
36 **else if** *n is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
37 $Cache(asgn(\pi_n)[pas_i]).value \leftarrow v(n)$ // Save AND node value in cache
38 $Cache(asgn(\pi_n)[pas_i]).assignment \leftarrow ST(n)$; // Save optimal assignment
39 **if** $v(p) > (w(p, n) + v(n))$ **then**
40 $v(p) \leftarrow w(p, n) + v(n)$ // Update OR node value (minimization)
41 $ST(p) \leftarrow ST(n) \cup \{(X_i, x_i)\}$ // Update solution tree below OR node
42 remove n from $succ(p)$
43 $n \leftarrow p$

space below t can be pruned.

We considered so far the case when the best solution found so far is maintained at the root node of the search tree. It is also possible to maintain the current best solutions for all the OR nodes along the active path between the tip node t of T' and s. Then, if $f(T'_m) \geq ub(m)$, where m is an OR ancestor of t in T' and T'_m is the subtree of T' rooted at m, it is also safe to prune the search space below t. This

Algorithm 2: Recursive computation of the heuristic evaluation function.

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

```
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})
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)
```

provides an efficient mechanism to discover that the search space below a node can be pruned more quickly. For illustration, see also Section 6 in [1].

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 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 13). 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 17–21). 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 $f(T'_m)$ based on Definition 28 is

described in Algorithm 2 (we give it here for completeness).

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 5–28). 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 5–10). Each OR-to-AND arc is associated with the appropriate weight (see Definition 19). 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.

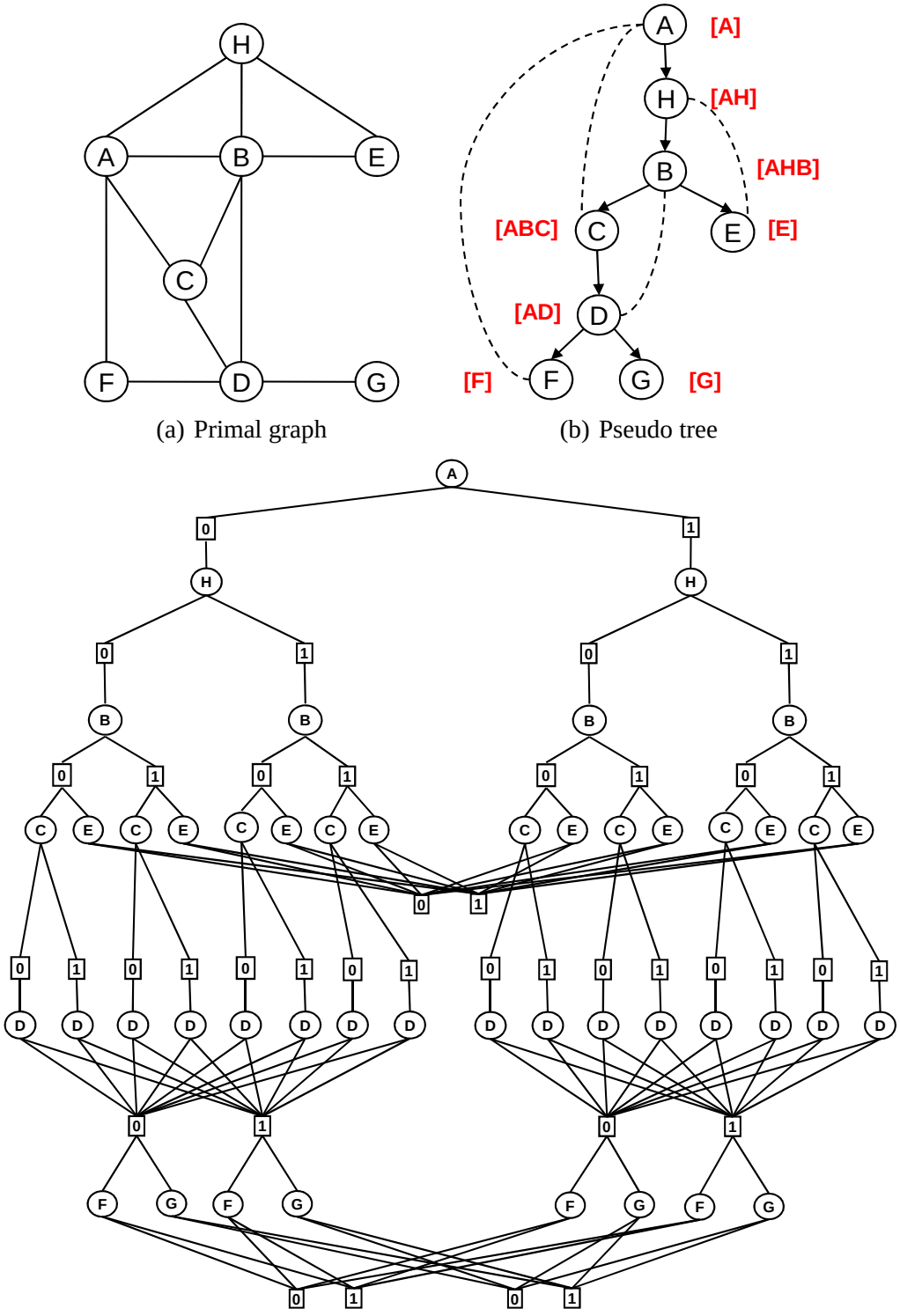
The node values are updated by the PROPAGATE step (lines 30–43). It is triggered when a node value has an empty set of descendants (note that as each successor is evaluated, it is removed from the set of successors in line 42). This means that all its children have been evaluated, and their final values are already determined. If the current node is the root, then the search terminates with its value and an optimal solution tree (line 33). 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 34). An AND node n propagates its value to its parent p in a similar way, by minimization (lines 36–41). It also saves in cache the value and optimal solution subtree below it (lines 37–38). Finally, the current node n is set to its parent p (line 43), 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 35). 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 41). Search continues either with a *propagation* step (if conditions are met) or with an *expansion* step. Clearly,

THEOREM **6 (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.*

Since 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 unifiable nodes.

5.1 Naive Caching

The first scheme, called *naive caching* and denoted hereafter 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



(a) Primal graph (b) Pseudo tree

(c) AND/OR search graph

Fig. 4. Illustration of naive caching used by `AOBB-C(2)`.

the algorithm explores the context minimal AND/OR graph via full caching.

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

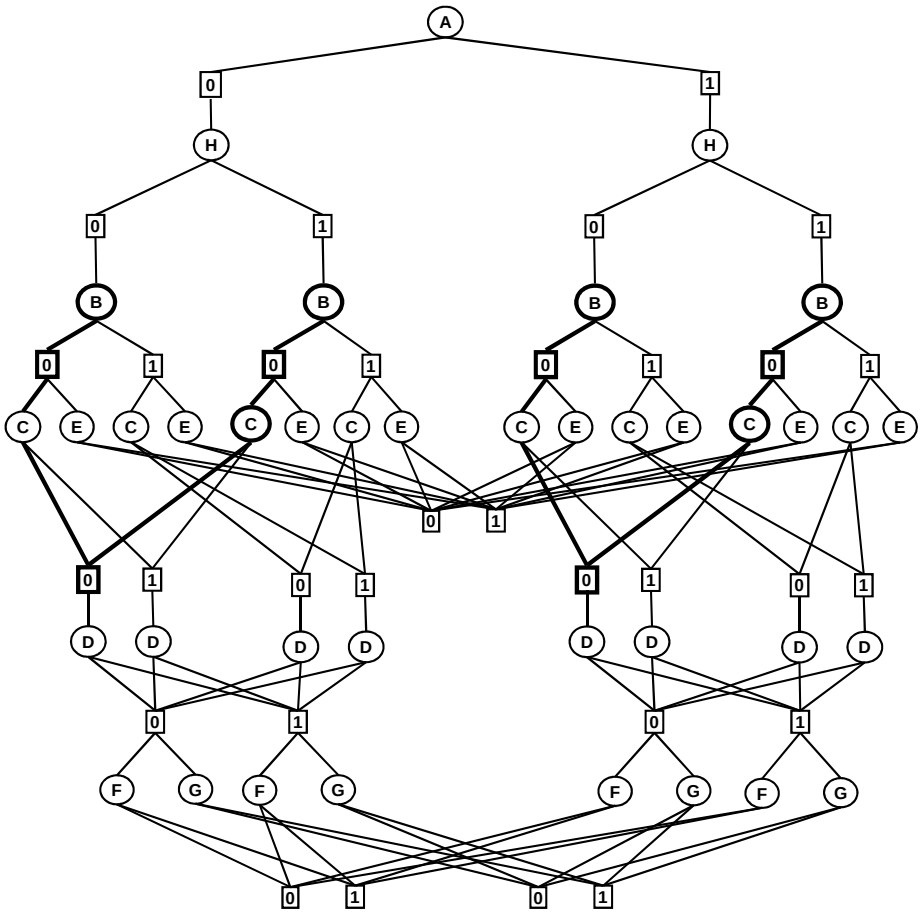


Fig. 5. Illustration of adaptive caching used by AOBB-AC(2).

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* [5,4] appear at nodes that have only one incoming arc. AOBB-C(j) needs to record only nodes that are likely to have additional incoming arcs, and these nodes can be determined by inspecting the pseudo tree. For example, if the context of a node includes that of its parent, then there is no need to store anything for that node, because it would be definitely a dead-cache.

Example 4 *Figure 4(c) displays the AND/OR search graph obtained with the naive caching scheme AOBB-C(2), relative to the pseudo tree given in Figure 4(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 $context(A) \subseteq context(H)$ and $context(H) \subseteq context(B)$, 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.

5.2 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 [31]. 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 [31].

Example 5 *Figure 5 shows the AND/OR graph traversed using the adaptive caching scheme AOBB-AC(2). In contrast to the naive scheme displayed in Figure 4, 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.*

6 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 [8].

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

The *best-first AND/OR search* algorithm, denoted by AOBF-C, that traverses the context minimal AND/OR search graph is described in Algorithm 3. It specializes Nilsson's AO* algorithm [23] 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 [23]. 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 3–9). 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 11–20). 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 19). 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 21–25). 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 $n' \in succ(n)$.

The second operation in AOBF-C is a bottom-up, cost revision, arc marking, SOLVE-labeling procedure (lines 27–39). 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 subproblem rooted at n. If we assume the monotone restriction on h, cost revisions can only be cost increases [32,23]. Therefore, not all ancestors need have cost revisions, but only those ancestors having best partial solution trees containing descendants with revised values (lines 38–39). 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 the algorithm AOBF-C will terminates in an optimal solution tree [32,23]. 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. It is possible to show that since the algorithm explores every node in the context minimal graph just once, we get:

THEOREM **7 (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 [8], but empirically this depends on how quickly AOBB-C will find an optimal solution. 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 completion only. All the above points show that the relative merit of best-first versus depth-first over context minimal AND/OR search spaces cannot be determined by theory [8] and empirical evaluation is essential.

7 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. Naturally, more accurate heuristic estimates may yield a smaller search space, however at a much higher computational cost. Therefore, the right trade-off between the computational overhead at each node and the pruning power exhibited during search may be hard to predict. The primary heuristic that we used in our experiments is the Mini-Bucket heuristic, which we presented in [1,2]. For completeness sake, we review it briefly next.

Mini-Bucket Elimination (MBE(i)) [10] is an approximation algorithm designed to avoid the high time and space complexity of *Bucket Elimination* (BE) [33], 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* [34].

Static Mini-Bucket Heuristics. In [1,2] 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 [4]), 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 correspond-

ing 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.

8 Experimental Results

In [1,2] we evaluated empirically AND/OR search algorithms for AND/OR trees only. We now extend this evaluation to algorithms exploring the context minimal AND/OR search graphs just described. As in [1,2], we have conducted a number of experiments on 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 single-core Pentium IV with 2GB of RAM, running Windows XP.

8.1 Overview and Methodology

Algorithms. We evaluated the following classes of memory intensive AND/OR search algorithms guided by mini-bucket heuristics:

- 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.

Table 1
Detailed outline of the experimental evaluation for Bayesian networks.

Benchmarks	static mini-buckets BB-C+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)	dynamic mini-buckets BB-C+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)	min-fill vs. hypergraph pseudo trees	nave vs. adaptive caching	constraint propagation	SamIam	Superlink
main							
Coding	√	√	-	-	-	√	-
Grids	√	√	√	√	√	√	-
Linkage	√	-	√	√	-	√	√
appendix							
ISCAS'89	√	√	√	√	√	√	-
UAI'06 Dataset	√	-	√	-	-	√	-

We compare these algorithms with those searching the AND/OR tree (without caching) guided by the mini-bucket heuristics, 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 questions that govern the performance of the proposed algorithms:

1 The impact of graph versus tree AND/OR Branch-and-Bound search.
2 The impact of best-first versus depth-first AND/OR search.
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.

Since the pre-compiled mini-bucket heuristics require a static variable ordering, the corresponding OR and AND/OR search algorithms used the variable ordering derived from a depth-first traversal of the guiding pseudo tree. We note however that `AOBB-C+SMB(`i`)` and `AOBB-C+DMB(`i`)` support a restricted form of dynamic variable and value ordering. Namely, there is a dynamic internal ordering of the successors of the node just expanded, before placing them onto the search stack. Specifically, in line 29 of Algorithm 1, if the current node n is AND, then the independent subproblems rooted by its OR children can be solved in decreasing order of their corresponding heuristic estimates (variable ordering). Alternatively, if n is OR, then its AND children corresponding to domain values can also be sorted in decreasing order of their heuristic estimates (value ordering).

Bayesian Networks. For the MPE task, we tested the performance of the depth-first AND/OR Branch-and-Bound and best-first AND/OR search algorithms on the

Table 2
Detailed outline of the experimental evaluation for Weighted CSPs.

Benchmarks	static mini-buckets BB-C+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)	dynamic mini-buckets BB-C+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)	min-fill vs. hypergraph pseudo trees	nave vs. adaptive caching	AOEDAC AOEDAC+PVO DVO+AOEDAC AOEDAC+DSO	toolbar toolbar-BTD
main						
SPOT5	√	√	√	√	√	√
ISCAS'89	√	√	√	√	√	√
Mastermind	√	-	√	√	√	√

following types of problems: random coding networks, grid networks, Bayesian networks derived from the ISCAS'89 digital circuits benchmark, genetic linkage analysis networks, and a subset of networks from the UAI'06 Inference Evaluation Dataset. We report here some of the results and place the rest in the Appendix.

The detailed outline of the experimental evaluation for Bayesian networks is given in Table 1. 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 [1]. SAMIAM is a public implementation of Recursive Conditioning [5] 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 [4] 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. For WCSPs we evaluated the performance of the AND/OR search algorithms on the following types of problems: scheduling problems from the SPOT5 benchmark, networks derived from the ISCAS'89 digital circuits and instances of the popular game of Mastermind. The outline of the experimental evaluation for WCSPs is detailed in Table 2.

For reference, we also report results obtained with the state-of-the-art solvers called `toolbar` [35] and `toolbar-BTD` [36] [2]. `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 [7] and computes the guiding heuristic information as well by enforcing EDAC during search. It can be shown

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

that BTD explores a context minimal AND/OR search graph, relative to a pseudo tree corresponding to the given tree decomposition [4]. In addition, we also compare with the depth-first AND/OR Branch-and-Bound tree search algorithms with EDAC heuristics and dynamic variable orderings described in [1,3]: 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.

Measures of Performance. We report the CPU time in seconds and the number of nodes visited, required for proving optimality. We specify the number of variables (n), number of evidence variables (e), maximum domain size (k), the depth of the pseudo trees (h) and the induced width of the graphs (w^*) obtained for the test instances. 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 (see Section 4.4). 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.

8.2 Results for Empirical Evaluation of Bayesian Networks

Our results reported in [1,2] demonstrateded conclusively that the AND/OR Branch-and-Bound *tree* search algorithms with static mini-bucket heuristics were the best performing algorithms on this domain. The diference 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 2 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 diference between AND/OR and OR Branch-and-Bound 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 computational overhead issues. In this section we extend the empirical evaluation to memory intensive

depth-first and best-first AND/OR search algorithms.

8.2.1 Coding Networks

We experimented with random coding networks from the class of *linear block codes*. They can be represented as 4-layer belief networks with K nodes in each layer ($i.e.$, the number of input bits). The second and third layers correspond to input information bits and parity check bits, respectively. Each parity check bit represents an XOR function of the input bits. The first and last layers correspond to transmitted information and parity check bits, respectively. Input information and parity check nodes are binary, while the output nodes are real-valued. Given a number of input bits K, number of parents P for each XOR bit, and channel noise variance σ^2, a coding network structure is generated by randomly picking parents for each XOR node. Then we simulate an input signal by assuming a uniform random distribution of information bits, compute the corresponding values of the parity check bits, and generate an assignment to the output nodes by adding Gaussian noise to each information and parity check bit. The decoding algorithm takes as input the coding network and the observed real-valued output assignment and recovers the original input bit-vector by computing an MPE assignment.

Table 3 shows the results for solving two classes of random coding networks with $K = 64$ and $K = 128$ input bits, using static and dynamic mini-bucket heuristics. The number of parents for each XOR bit was $P = 4$ and we chose the channel noise variance $\sigma^2 \in \{0.22, 0.36\}$. For each value combination of the parameters we generated 20 random instances. The guiding pseudo trees were generated using the min-fill heuristic. The top four horizontal blocks show the results for static mini-bucket heuristics, while the bottom four ones correspond to dynamic mini-buckets heuristics. The columns are indexed by the mini-bucket i-bound, which we varied between 4 and 20.

Tree vs. graph AOBB. When comparing the tree versus the graph search AND/OR Branch-and-Bound algorithms we see that `AOBB-C+SMB(`i`)` is slightly better than `AOBB+SMB(`i`)`. We observe a similar picture when using dynamic mini-buckets as well. This indicates that, on this domain, most of the cache entries were actually dead, namely the context minimal AND/OR graph was very close to a tree. Notice that SAMIAM was not able to solve any of these problem instances due to the memory limit.

AOBF vs. AOBB. When comparing the best-first versus the depth-first algorithms using static mini-bucket heuristics, we see that `AOBF-C+SMB(`i`)` is better than `AOBB-C+SMB(`i`)` for relatively small i-bounds ($i.e.$, $i \in \{4, 8\}$) which generate relatively weak heuristic estimates. For instance, on class $\langle K = 64, \sigma^2 = 0.22\rangle$, best-first search `AOBF-C+SMB(4)` is 4 orders of magnitude faster than `AOBB-C+SMB(4)`. As the i-bound increases and the heuristics become more ac-

Table 3
CPU time and nodes visited for solving **random coding networks** using **static and dynamic mini-bucket heuristics** as well as min-fill based pseudo trees. Time limit 5 minutes. The top four horizontal blocks show the results for static mini-bucket heuristics, while the bottom four blocks show the dynamic mini-bucket heuristics.

min-fill pseudo tree												
		SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)	
(K, N)	(w*, h)		i=4		i=8		i=12		i=16		i=20	
			time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
			0.02		0.02		0.07		0.68		8.33	
			-	-	16.55	174,205	0.09	148	0.72	130	8.36	130
(64, 128)	(27, 40)	out	287.10	5,052,010	6.58	119,289	0.08	152	0.68	129	8.34	129
$\sigma^2 = 0.22$			250.81	3,600,530	4.25	63,171	0.08	147	0.71	129	8.41	129
			0.04	157	**0.04**	129	0.09	128	0.72	128	8.45	128
			0.02		0.02		0.07		0.68		8.32	
			-	-	76.38	807,319	0.99	10,688	0.81	1,189	8.41	158
(64, 128)	(27, 40)	out	277.41	5,250,380	47.80	834,680	1.23	22,406	0.84	3,096	8.33	160
$\sigma^2 = 0.36$			250.32	3,907,000	35.52	518,125	0.79	12,236	0.81	1,850	8.39	148
			3.94	17,801	0.15	829	**0.12**	363	0.72	162	8.41	133
			0.05		0.06		0.18		1.80		25.65	
			-	-	256.23	1,766,930	30.57	213,184	3.30	11,073	25.88	1,656
(128, 256)	(53, 71)	out	-	-	229.02	3,227,110	16.67	206,004	3.51	22,644	25.87	3,081
$\sigma^2 = 0.22$			-	-	218.58	2,206,490	11.75	116,977	3.03	12,880	25.72	2,109
			0.14	375	**0.11**	266	0.23	262	1.90	257	25.01	258
			0.05		0.06		0.18		1.80		25.39	
			-	-	-	-	264.57	1,732,960	202.84	1,426,730	97.98	603,342
(128, 256)	(53, 71)	out	-	-	291.61	4,309,160	240.74	3,409,580	188.44	2,617,880	110.89	1,137,120
$\sigma^2 = 0.36$			-	-	290.12	2,951,230	235.08	2,312,080	178.90	1,816,940	100.32	781,438
			out		66.98	260,350	19.18	88,692	**7.23**	26,499	28.01	18,357
		SamIam	BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)	
(K, N)	(w*, h)		i=4		i=8		i=12		i=16		i=20	
			time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
			22.46	9,331	0.41	183	1.41	130	12.80	130	122.67	130
(64, 128)	(27, 40)	out	23.62	20,008	0.35	185	1.37	129	12.77	129	121.12	129
$\sigma^2 = 0.22$			21.26	13,971	0.34	176	1.36	129	12.62	129	120.81	129
			0.19	129	0.37	128	2.15	128	19.98	128	192.66	128
			46.66	18,781	5.12	1,204	5.58	432	15.47	162	123.57	144
(64, 128)	(27, 40)	out	48.71	44,734	5.17	1,864	5.53	512	15.53	164	122.90	144
$\sigma^2 = 0.36$			44.20	29,191	4.91	1,323	5.41	399	15.33	155	122.27	138
			1.96	446	**0.82**	160	2.71	132	20.50	128	191.08	128
			195.84	39,109	48.49	3,684	17.48	482	130.41	379	-	-
(128, 256)	(53, 71)	out	195.82	121,822	48.17	9,391	17.15	500	129.38	388	-	-
$\sigma^2 = 0.22$			193.30	68,571	48.06	5,241	16.88	420	128.23	355	-	-
			0.75	260	1.58	256	11.18	256	131.50	256	-	-
			288.97	62,749	229.55	19,776	234.08	4,402	276.95	804	-	-
(128, 256)	(53, 71)	out	289.09	223,938	229.91	46,768	233.96	7,947	276.31	953	-	-
$\sigma^2 = 0.36$			288.79	121,278	229.09	27,362	233.72	4,662	276.87	649	-	-
			202.41	16,041	**70.68**	2,260	163.78	709	282.36	136	-	-

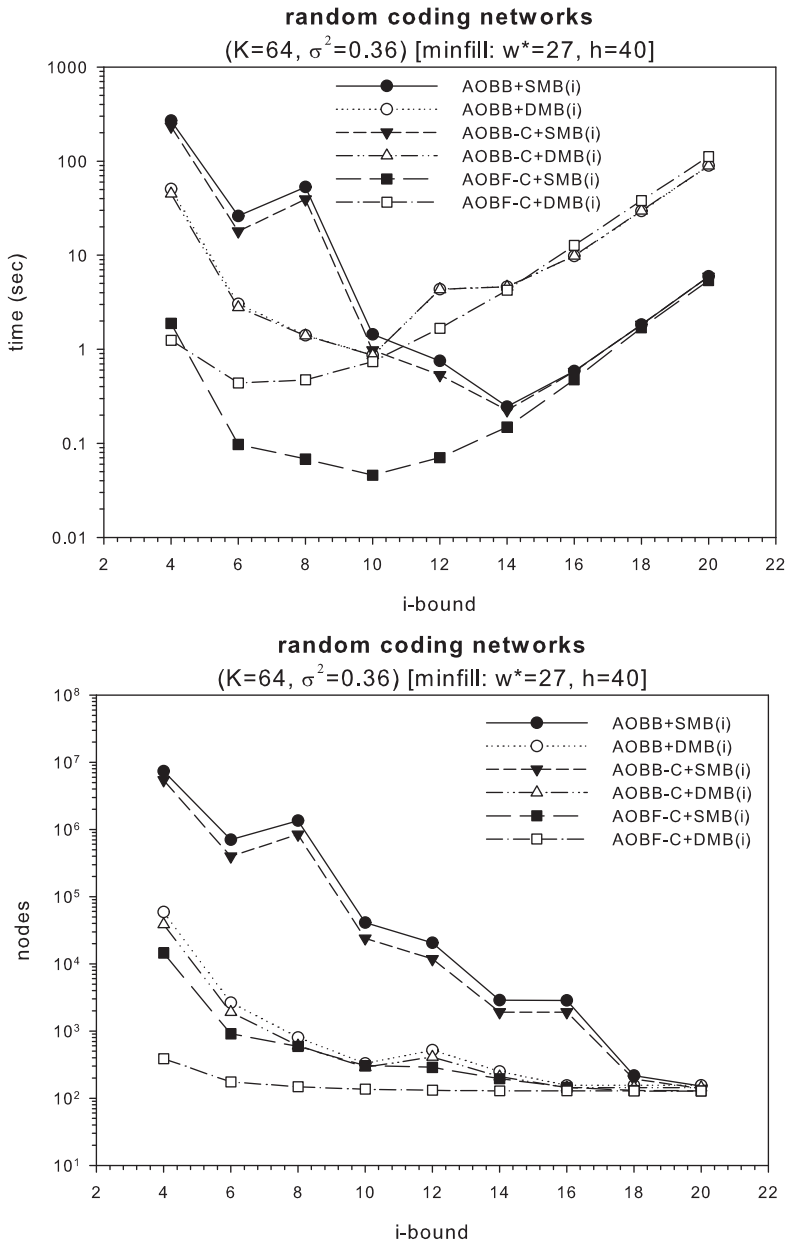


Fig. 6. Comparison of the impact of static and dynamic mini-bucket heuristics for solving the **random coding networks** with parameters $(K = 64, \sigma^2 = 0.36)$ from Table 3. We show the CPU time in seconds (top) and the number of nodes visited (bottom).

curate, the difference between Branch-and-Bound and best-first search decreases, because Branch-and-Bound finds close to optimal solutions fast, and therefore will not explore solutions whose cost is below the optimum, like best-first search. When looking at the algorithms using dynamic mini-bucket heuristics, we notice that `AOBF-C+DMB(`i`)` is again far better than `AOBB-C+DMB(`i`)` for smaller i-bounds.

Static vs. dynamic mini-bucket heuristics. When comparing the static versus dynamic mini-bucket heuristic we see that the latter is competitive only for relatively

small i-bounds ($i.e.$, $i \in \{4, 8\}$). At higher levels of the i-bound, the accuracy of the dynamic heuristic does not outweigh its computational overhead.

Figure 6 plots the average running time and number of nodes visited, as a function of the mini-bucket i-bound, on the random coding networks with parameters $(K = 64, \sigma^2 = 0.36)$ ($i.e.$, corresponding to the second and fifth horizontal blocks in Table 3). It shows explicitly how the performance of the algorithms changes with the mini-bucket strength for both heuristics. Focusing for example on best-first search, we see that i-bound of 4 is most cost effective for dynamic mini-buckets, while i-bound of 10 yields best performance for static mini-buckets. We also see clearly that the dynamic mini-bucket heuristic is more accurate yielding smaller search spaces. It also demonstrates that the dynamic mini-bucket heuristics are cost effective at relatively small i-bounds, whereas the pre-compiled version is more powerful for larger i-bounds.

We addressed so far the impact of tree versus graph AND/OR search, the impact of the mini-bucket i-bound and best-first versus depth-first search regimes. In the remainder we will also investigate the impact of the level of caching, the impact of pseudo tree quality, the impact of determinism present in the network, as well as the anytime behavior of AND/OR Branch-and-Bound and the impact of good initial bounds.

8.2.2 Random 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 [37] 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 variables are bi-valued.

Tables 4 and 5 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. We varied the mini-bucket i-bound between 8 and 16 for the first 3 grids, and between 12 and 20 for the remaining ones. 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, `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`)`. The al-

Table 4

CPU time and nodes visited for solving **grid networks** using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour. Top part of the table shows results for i-bounds between 8 and 16, while the bottom part shows i-bounds between 12 and 20.

min-fill pseudo tree											
		MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)	
grid	SamIam	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)	
(w*, h)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)	
(n, e)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)	
		i=8		i=10		i=12		i=14		i=16	
		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
		i=12		i=14		i=16		i=18		i=20	
		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 5
CPU time and nodes visited for solving **grid networks** using **dynamic mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour. Top part of the table shows results for i-bounds between 8 and 16, while the bottom part shows i-bounds between 12 and 20.

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
	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	-	-
90-30-1	-	-	-	-	-	-	-	-	-	-
(43, 150)	-	-	-	-	-	-	-	-	-	-
(900, 60)	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-
90-34-1	-	-	-	-	-	-	-	-	-	-
(45, 153)	-	-	-	-	-	-	-	-	-	-
(1154, 80)	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-
90-38-1	-	-	-	-	-	-	-	-	-	-
(47, 163)	-	-	-	-	-	-	-	-	-	-
(1444, 120)	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-

gorithm is 2 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 5) 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 2 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-

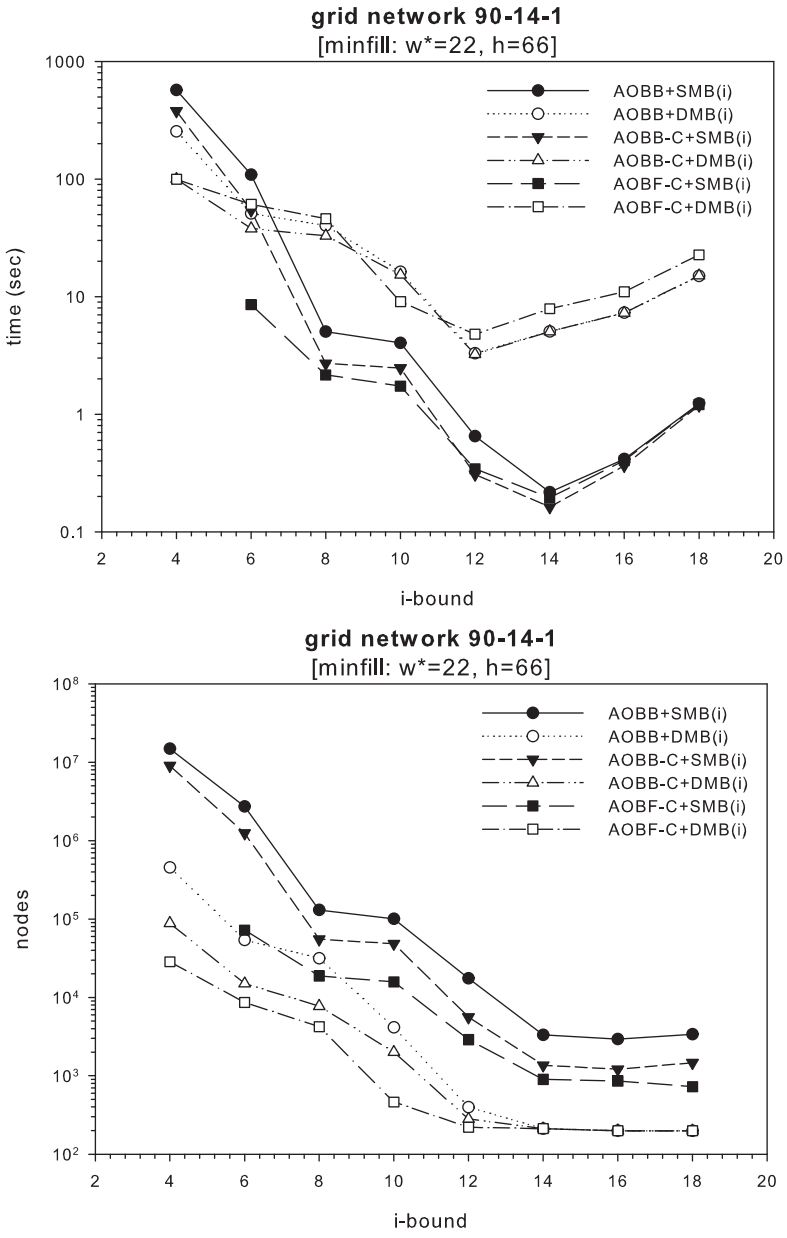


Fig. 7. Comparison of the impact of static and dynamic mini-bucket heuristics for solving the **`90-14-1` grid network** from Tables 4 and 5, respectively. We show the CPU time in seconds (top) and the number of nodes visited (bottom).

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 7). 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

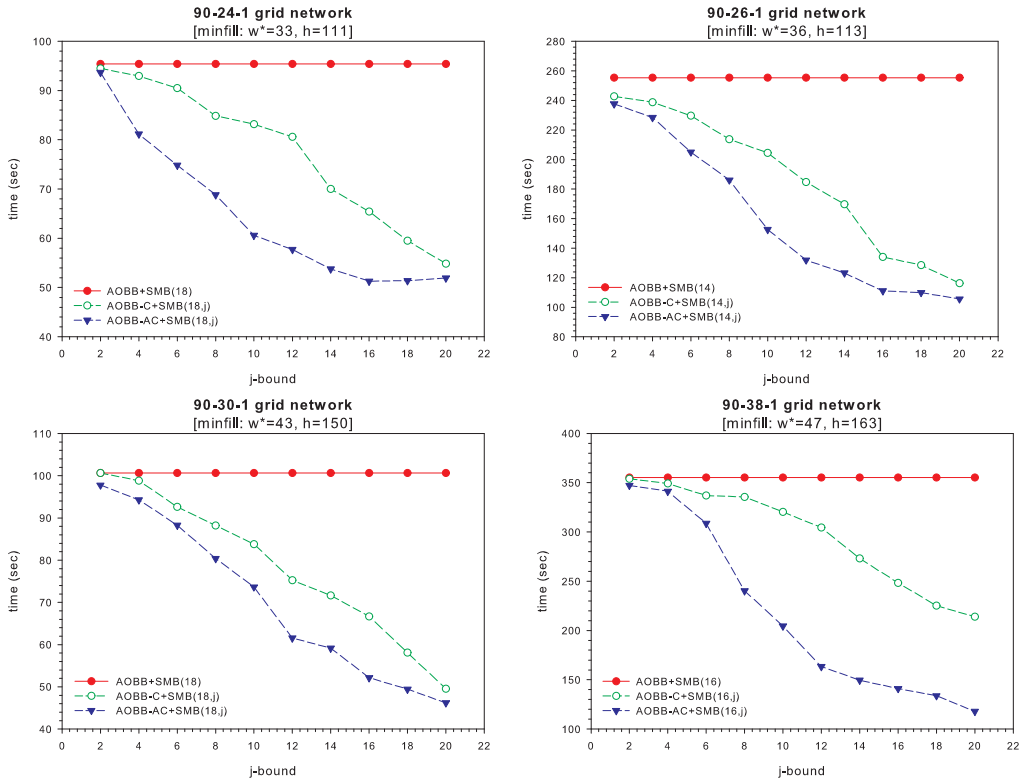


Fig. 8. Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **grid networks**. Shown is the CPU time in seconds.

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. Figure 7 shows 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 4 and 5, 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 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 level of caching. Figure 8 compares the naive (`AOBB-C+SMB(`i,j`)`) and adaptive (`AOBB-AC+SMB(`i,j`)`) caching schemes, in terms of CPU time, on 4 grid networks from Table 4. In each test case we chose a relatively small mini-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

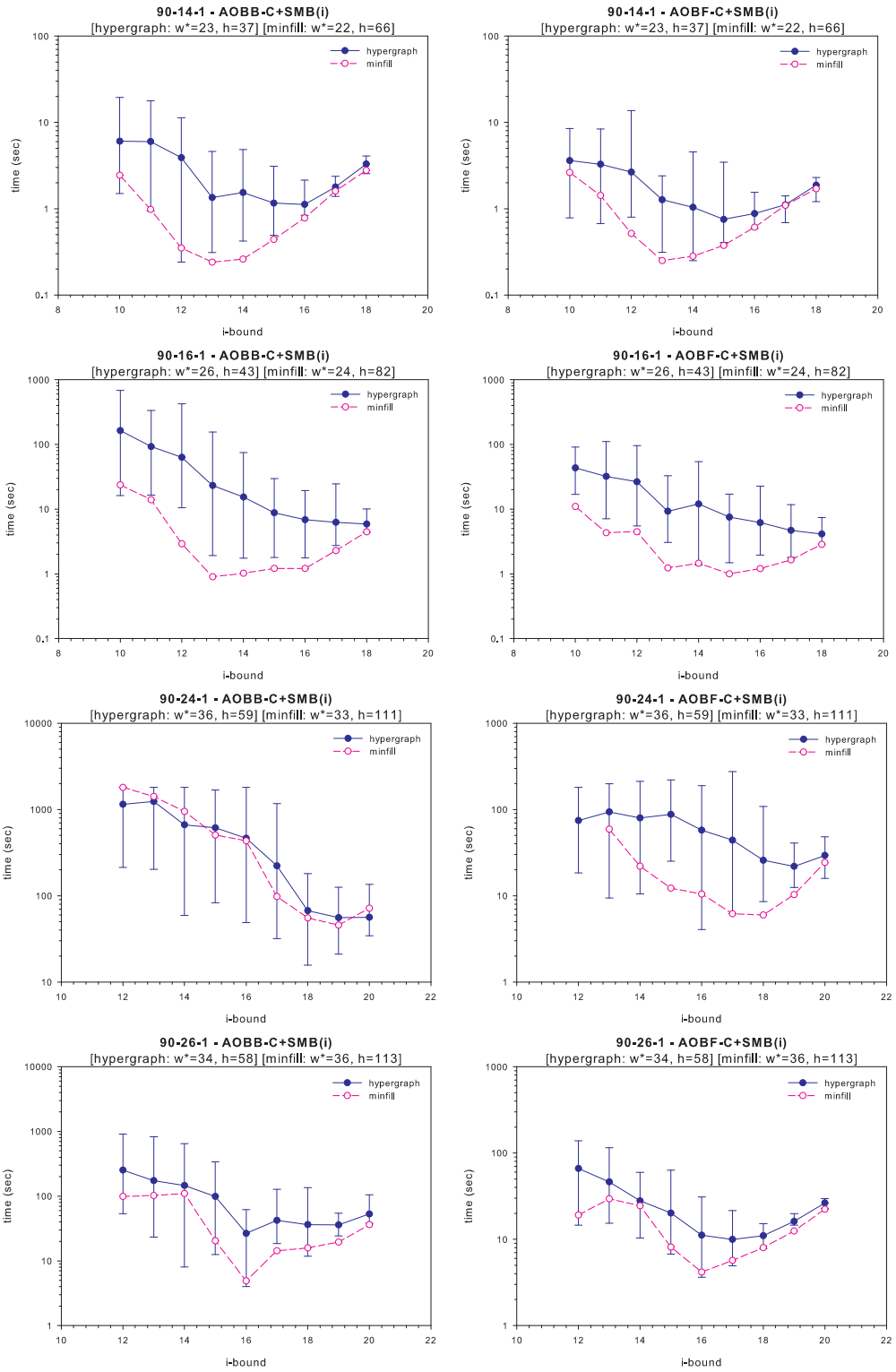


Fig. 9. Min-fill versus ypergraph partitioning heuristics. CPU time in seconds for solving **grid networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning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

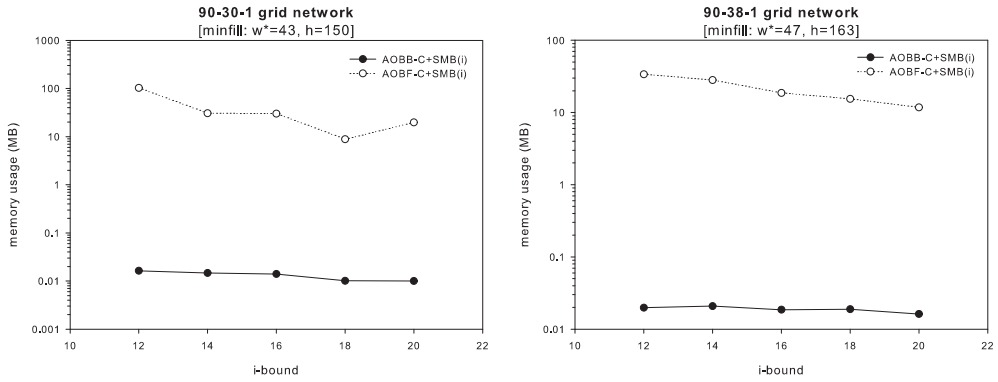


Fig. 10. Memory usage by AOBB-C+SMB(i) and AOBF-C+SMB(i) on **grid networks**.

relatively small j-bounds. This may be important because small j-bounds mean restricted space. At large j-bounds the two schemes are identical and approach the full-caching scheme.

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 9 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. For each reported i-bound, 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 9). 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. 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.

Memory usage of AND/OR graph search. Figure 10 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 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,

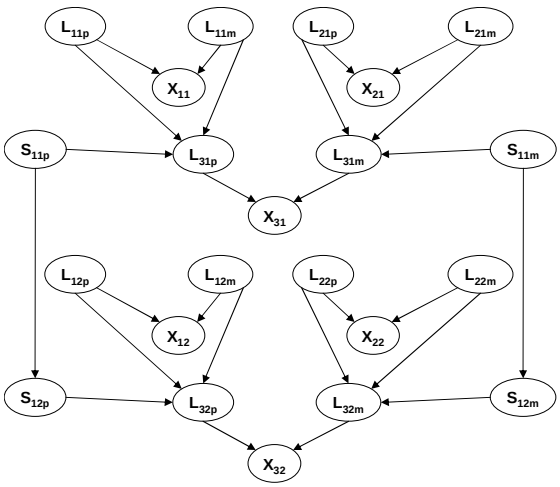


Fig. 11. A fragment of a belief network used in genetic linkage analysis.

we were able throughout the evaluation to run full caching with depth-first search.

8.2.3 Genetic Linkage Analysis

In human genetic linkage analysis [38], the *haplotype* is the sequence of alleles at different loci inherited by an individual from one parent, and the two haplotypes (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.

The pedigree data can be represented as a belief network with three types of random variables: *genetic loci* variables which represent the genotypes of the individuals in the pedigree (two genetic loci variables per individual per locus, one for the paternal allele and one for the maternal allele), *phenotype* variables, and *selector* variables which are auxiliary variables used to represent the gene flow in the pedigree. Figure 11 shows a fragment of a network that describes parents-child interactions in a simple 2-loci analysis. The genetic loci variables of individual i at locus j are denoted by $L_{i,jp}$ and $L_{i,jm}$. Variables $X_{i,j}$, $S_{i,jp}$ and $S_{i,jm}$ denote the phenotype variable, the paternal selector variable and the maternal selector variable of individual i at locus j, respectively. The conditional probability tables that correspond to the selector variables are parameterized by the *recombination ratio* θ [39]. The remaining tables contain only deterministic information. It can be shown that given the pedigree data, the haplotyping problem is equivalent to computing the Most Probable Explanation (MPE) of the corresponding belief network (for more details consult [39,40]).

Table 6
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)	SamIam	Superlink	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)	5.44	54.73	24.30	416,326	13.17	206,439	1.58	24,361	1.84	25,674	1.89	15,156
(299, 5)			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)	out	**28.36**	-	-	8120.58	85,367,022	-	-	3040.60	35,394,461		
(582, 5)			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)	out	-	-	-	-	-	476.77	5,566,578	104.00	748,792		
(479, 5)			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		
			i=10		i=12		i=14		i=16		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)	out	9146.19	498.05	6,623,197	15.45	154,676	16.28	67,456	286.11	117,308		
(310, 5)			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)	out	64.17	273.39	3,191,218	1682.09	25,729,009	1096.79	15,598,863	128.16	953,061		
(1032, 5)			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		

Tables 6 and 7 display the results obtained for 12 hard linkage analysis networks [3] (we show 5 networks in Table 6 and 7 networks in Table 7). 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 `ped37`, for example, `AOBB-C+SMB(10)`

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

Table 7
CPU time and nodes visited for solving **genetic linkage networks**. Time limit 3 hours. Shown here are 7 linkage networks in addition to the 5 networks from Table 6.

min-fill pseudo tree												
pedigree (w*, h) (n, d)	SamIam	Superlink	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)	157.05	139.06	-	-	2177.81	28,651,103	270.96	2,555,078	100.61	682,175	20.27	7,689
(1184, 5)			-	-	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)	out	**14.72**	3793.31	54,941,659	1293.76	18,449,393	1259.05	17,810,674	1080.05	9,151,195		
(388, 5)			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)	out	-	-	-	-	-	9399.28	111,301,168	3607.82	34,306,937	2965.60	28,326,541
(994, 5)			-	-	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)	out	13095.83	-	-	-	-	-	-	214.10	1,379,131	91.92	685,661
(1016, 5)			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)	out	-	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)			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)	out	322.14	-	-	-	-	4041.56	52,804,044	386.13	2,171,470	141.23	407,280
(1272, 5)			-	-	-	-	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)	out	561.31	-	-	-	-	-	-				
(448, 5)			-	-	-	-	2364.67	22,595,247				
			out		out		**133.19**	93,831				

is 7 times faster than AOBB+SMB(10) and expands about 14 times fewer nodes. As the i-bound increases the diference 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) was able to solve only 3 out of the 12 test instances ($e.g.$, ped1, ped23, ped18). The performance of SAMIAM was very poor and it was able to solve only 2 instances, namely ped1 and ped18.

AOBB vs. AOBF. As before, the overall best performing algorithm on this dataset is best-first AOBF-C+SMB(i), outperforming its competitors on 8 out of the 12 test cases. On ped42, for instance, AOBF-C+SMB(16) is 18 times faster than

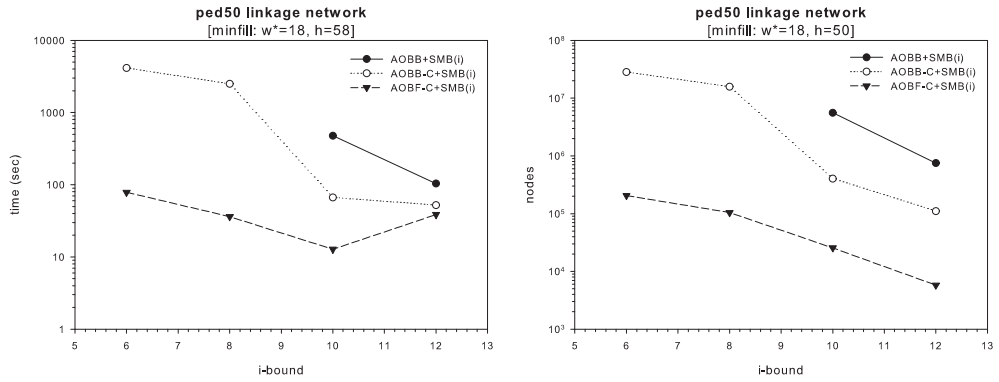


Fig. 12. CPU time and nodes visited for solving the **ped50 linkage network**.

the depth-first Branch-and-Bound AOBB-C+SMB(16) and explores a search space 240 times smaller. In some test cases ($e.g.$, ped1, ped23, ped30) the best-first search algorithm was up to 3 orders of magnitude faster than SUPERLINK. Figure 12 displays the CPU time and number of nodes explored, as a function of the mini-bucket i-bound, for the ped50 instance. In this case, AOBB+SMB(i) could not solve the problem instance for $i \in \{6, 8\}$, due to exceeding the time limit.

Impact of the pseudo tree. Figure 13 plots the running time distribution of the depth-first and best-first search algorithms AOBB-C+SMB(i) (left side of the figure) and AOBF-C+SMB(i) (right side of the figure), guided by hypergraph based pseudo trees, over 20 independent runs. 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 2 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 succeded 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 as well as a large number of dead caches.

Table 8 displays the results obtained for 6 additional linkage analysis networks using hypergraph partitioning based pseudo trees. 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 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 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 mem-

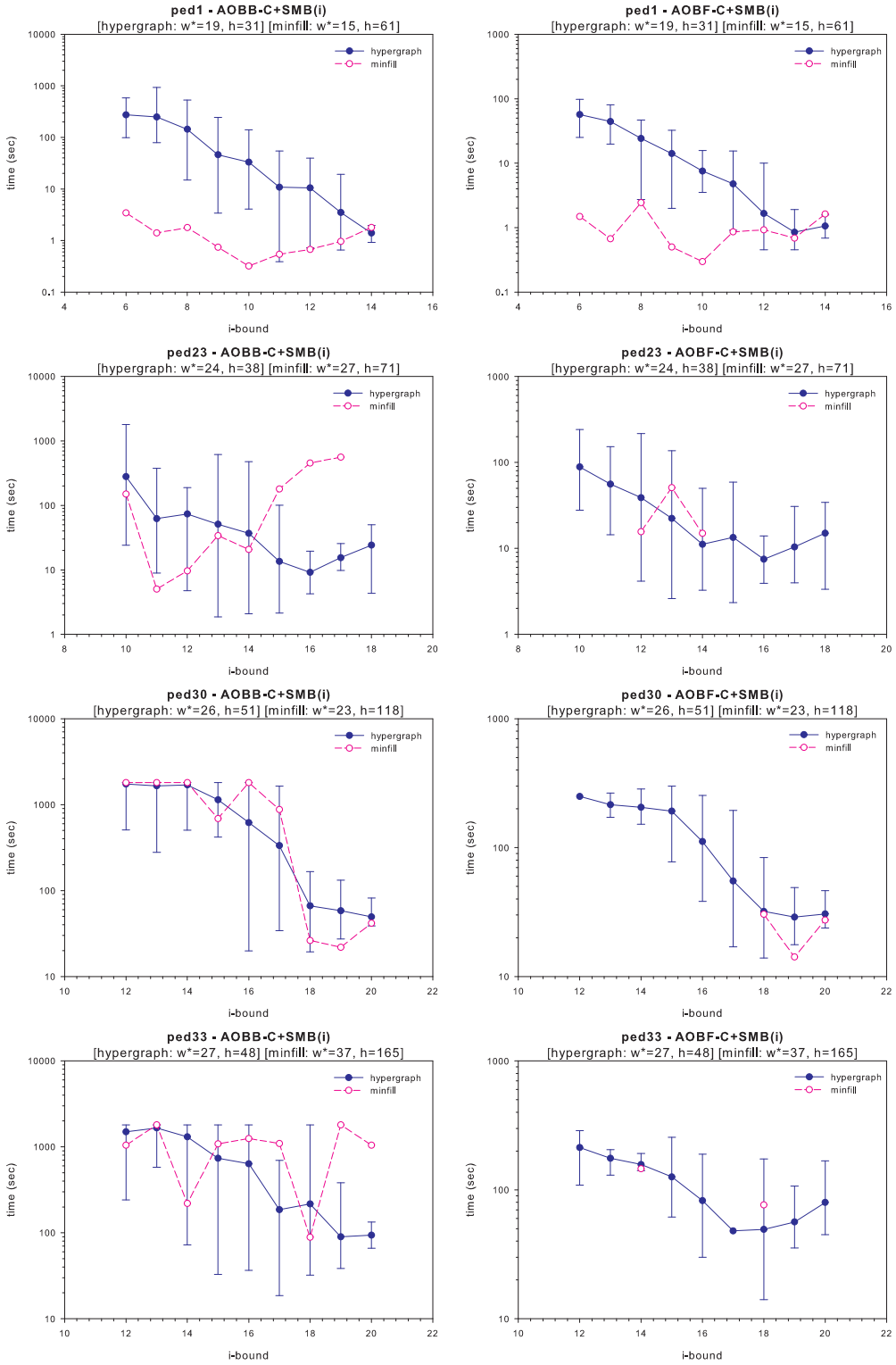


Fig. 13. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **genetic linkage networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph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

Table 8
Impact of the pseudo tree quality on **genetic linkage networks**. Time limit 24 hours. We show results for the hypergraph partitioning heuristic (left side) and the min-fill heuristic (right side).

		hypergraph pseudo tree					min-fill pseuso 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 time	nodes	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=22 time	nodes	(w*, h)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20 time	nodes	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=22 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	

ory issues. The AND/OR Branch-and-Bound algorithms with min-fill based pseudo trees could only solve 2 of the test instances ($e.g.$, `ped9` and `ped44`). This is because the induced width of these problem instances was small enough and the mini-bucket heuristics were relatively accurate to prune the search space substantially, thus overcomming the increase in pseudo tree depth. One thing that these experiments demonstrate is that the selection of the pseudo tree can have an enormous impact if the i-bound is not large enough.

Impact of the level of caching. Figure 14 displays the CPU time for solving 4 linkage analysis networks from Tables 6 and 7 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

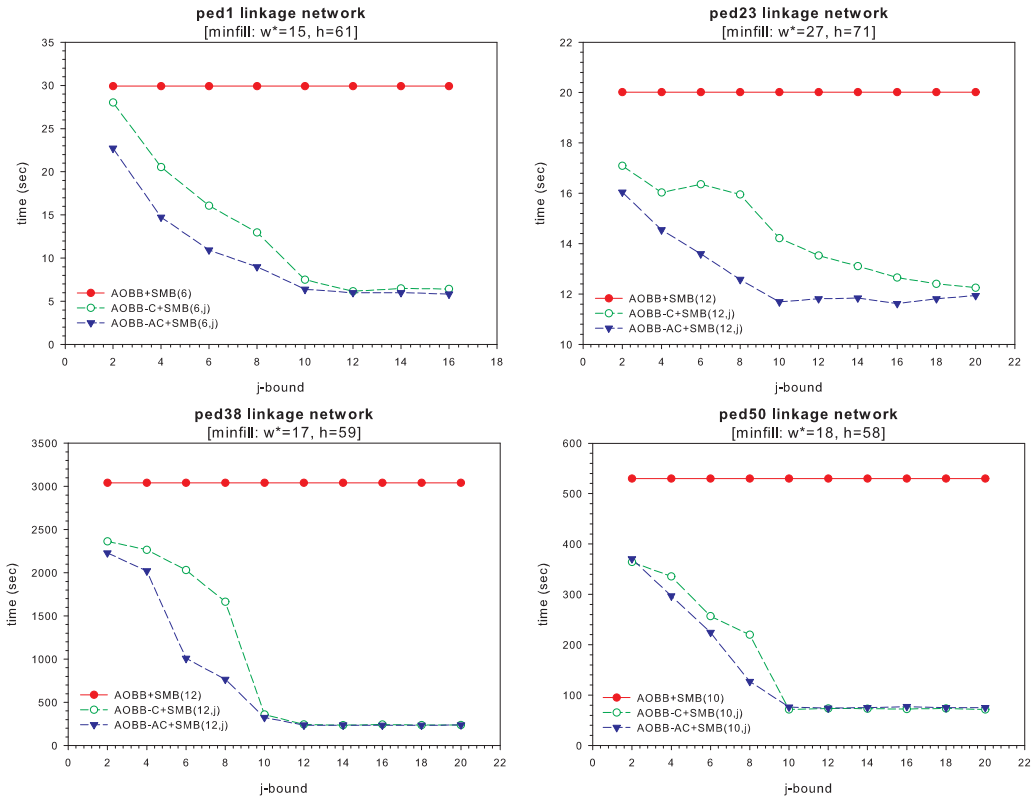


Fig. 14. Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **genetic linkage networks**. Shown is CPU time in seconds.

more powerful than the naive scheme especially, for relatively small j-bounds, which require restricted space. As the j-bound increases, the two schemes approach gradually full caching.

In the Appendix we provide additional empirical results over networks from the UAI'06 Dataset (Section A.1) and circuit diagnosis networks (Section A.2). In the following two subsections we look at the anytime behavior of the algorithms and at the impact of determinism.

8.2.4 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 [41], and improved more recently by [42].

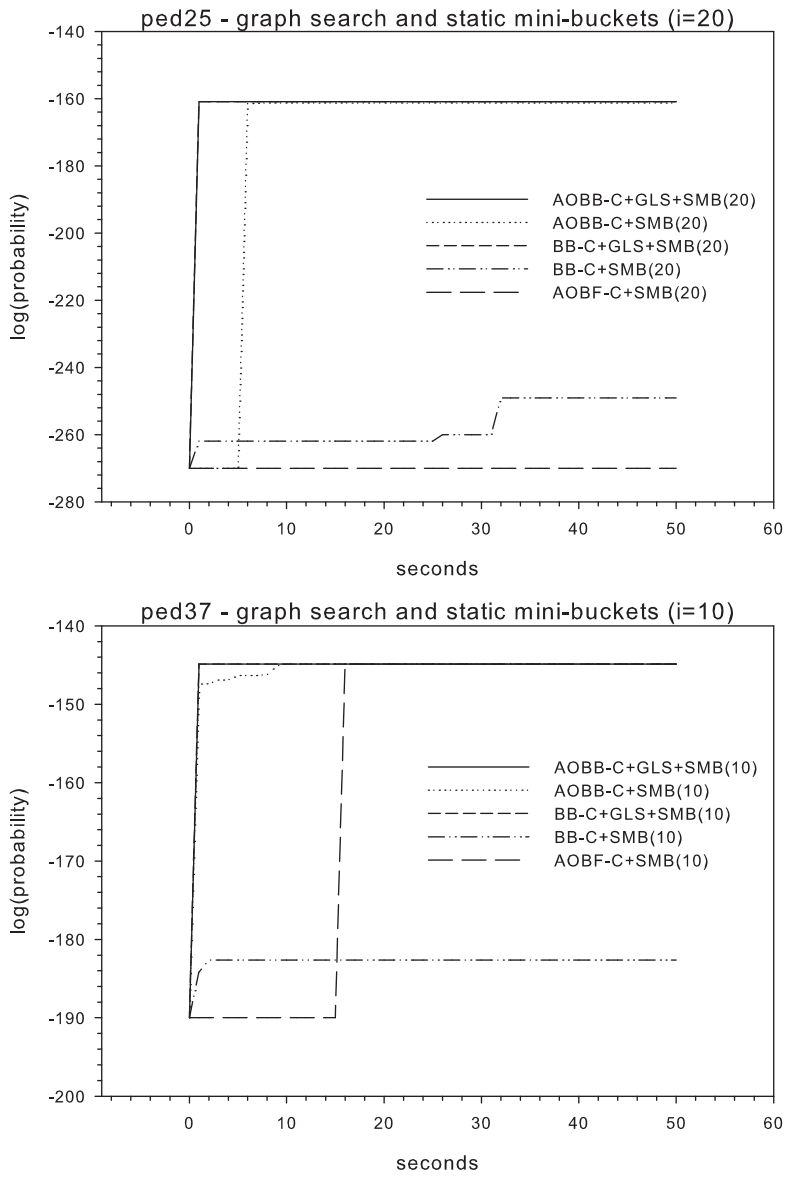


Fig. 15. Anytime behavior of AOBB-C+SMB(i) on **ped25** and **ped37 linkage networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

GLS [43] 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 solving the MPE in belief networks [41,42] as well as weighted MAX-SAT problems [44].

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

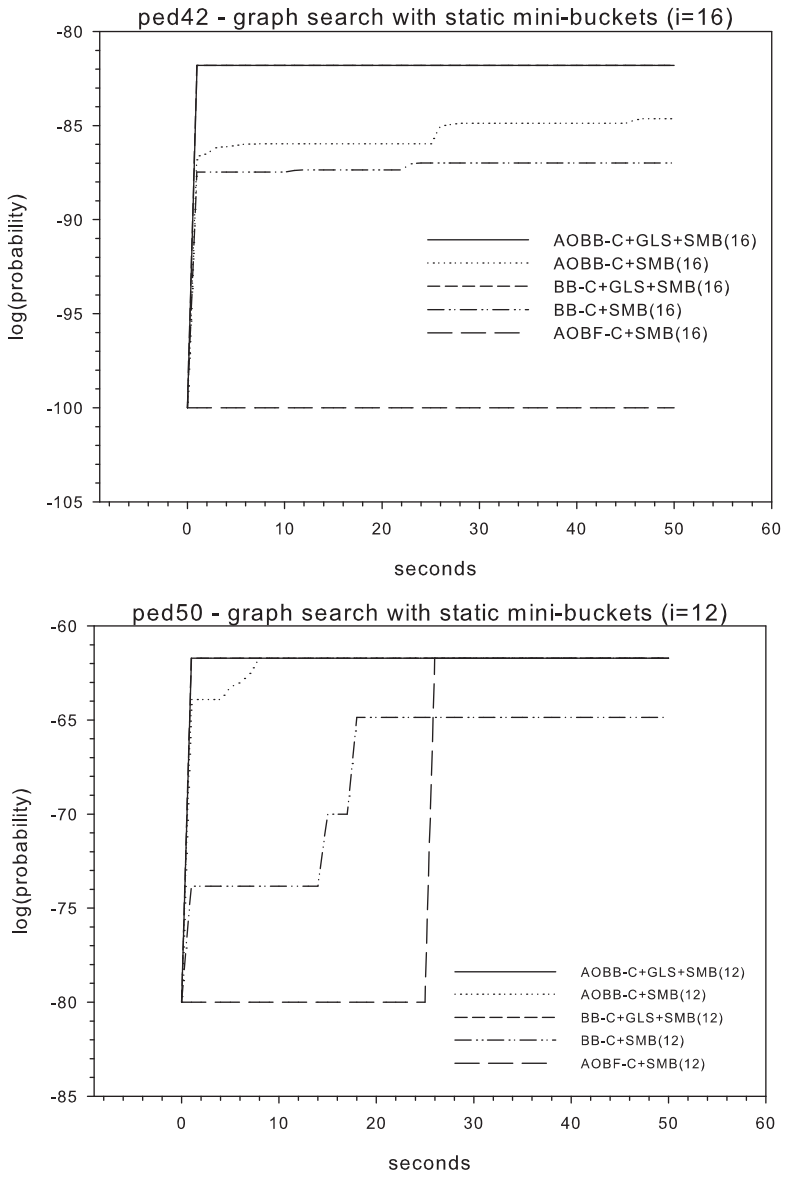


Fig. 16. Anytime behavior of AOBB-C+SMB(i) on **ped42** and **ped50 linkage networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

a non-trivial initial lower bound computed by GLS. The algorithm is denoted by AOBB-C+GLS+SMB(i). For reference, we also ran the OR version of the algorithm, denoted by BB-C+GLS+SMB(i)

Figures 15 and 16 display the search trace of the OR and AND/OR algorithms on 4 genetic linkage networks presented earlier. We chose the mini-bucket i-bound that offered the best performance in Tables 6 and 7, respectively, 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 improves performance throughout. In all these test case, 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)

Table 9
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 (w*, h) (n, d)	SamIam Superlink GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=6		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=8		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=10		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=12		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) 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		

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.

Tables 9 and 10 compare the OR and AND/OR search algorithms with and without an initial lower bound, as complete algorithms. Algorithms AOBB-C+GLS+SMB(i) and 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),

Table 10
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 (w*, h) (n, d)	SamIam Superlink GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=12		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=14		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=16		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=18		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	-	-	-	-	1515.43	1,388,791
ped18	157.05	-	-	-	-	-	-	-	-	1672.15	1,389,831
(21, 119)	139.06	-	-	406.88	3,567,729	52.91	397,934	23.83	118,869	20.60	2,972
(1184, 5)	10.16	10780.40	107,804,665	170.14	1,824,835	37.64	396,961	11.66	118,170	**10.58**	2,720
		out		127.41	542,156	42.19	171,039	19.85	53,961	19.91	2,027
		-	-	-	-	-	-	-	-	out	
ped20	out	-	-	-	-	-	-	-	-		
(24, 66)	**14.72**	1983.00	18,615,009	635.74	6,424,477	512.16	4,814,751	681.97	2,654,646		
(388, 5)	4.22	2079.43	18,611,778	667.66	6,419,317	567.20	4,812,068	682.03	2,653,400		
		out		out		out		out			
		-	-	-	-	-	-	-	-	-	-
ped25	out	-	-	-	-	-	-	-	-	-	-
(34, 89)	-	-	-	1644.67	12,631,406	865.83	6,676,835	249.47	1,789,094	**236.88**	1,529,180
(994, 5)	11.03*	-	-	1644.87	12,631,282	864.09	6,676,061	245.79	1,788,621	239.08	1,529,588
		out		out		out		out		out	
		-	-	-	-	-	-	-	-	-	-
ped30	out	-	-	-	-	-	-	-	-	-	-
(23, 118)	13095.83	10212.70	93,233,570	8858.22	82,552,957	-	-	34.19	193,436	30.48	66,144
(1016, 5)	11.00	10620.20	93,030,080	9296.01	82,552,786	-	-	32.16	193,419	**22.25**	66,128
		out		out		out		30.39	72,798	27.94	18,795
		-	-	-	-	-	-	-	-	-	-
ped33	out	-	-	-	-	-	-	-	-	-	-
(37, 165)	-	1426.99	11,349,475	307.39	2,504,020	1823.43	14,925,943	86.17	453,987	1373.90	10,570,695
(581, 5)	6.86*	1550.76	11,528,022	320.06	2,434,582	1970.72	15,124,932	80.61	453,446	1518.24	10,970,922
		out		140.61	407,387	out		**74.86**	134,068	out	
		-	-	-	-	-	-	-	-	-	-
ped39	out	-	-	-	-	-	-	-	-	-	-
(23, 94)	322.14	-	-	-	-	968.03	7,880,928	61.20	313,496	93.19	83,714
(1272, 5)	10.97*	-	-	-	-	518.04	6,473,615	59.14	313,340	81.24	61,291
		out		out		68.52	218,925	**41.69**	79,356	87.63	14,479
		-	-	-	-	-	-	out		out	
ped42	out	-	-	-	-	-	-				
(25, 76)	561.31	-	-	-	-	2364.67	22,595,247				
(448, 5)	4.25*	-	-	-	-	385.26	3,078,657				
		out		out		**133.19**	93,831				

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.

In the Appendix we provide additional empirical results on anytime behavior and impact of initial good lower bounds, over grid networks and UAI'06 networks (Section A.3).

Table 11
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

8.2.5 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 [45–48]. 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 [49] or generalized arc consistency has been explored in [45,46], 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 [47]. A detailed description of the CNF encoding based on the zero probability tuples is provided in [1].

We evaluated the AND/OR Branch-and-Bound algorithm with static mini-bucket heuristics on selected classes of Bayesian networks containing deterministic con-

ditional probability tables ($i.e.$, 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 the state-of-the-art SAT solver introduced by [50]. 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 11 shows the results for 5 deterministic grid networks from Section 8.2.2. These networks have a high degree of determinism encoded in their CPTs. Specifically, 90% of the probability tables are deterministic, containing only 0 and 1 probability tuples. 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, for example, 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. The mini-bucket heuristic already does some level of constraint propagation. When looking at the impact of the initial lower bound on AOBB-C+SAT+SMB(i) we see that AOBB-C+SAT+GLS+SMB(i) is sometimes able to improve even more. For example, on the 90-34-1 grid, AOBB-C+SAT+GLS+SMB(16) finds the optimal solution in about 9 minutes (550.55 seconds) whereas AOBB-C+SAT+SMB(16) exceeds the 1 hour time limit.

In the Appendix we provide additional empirical results on the impact of determinism over circuit diagnosis networks (Section A.4). Next we move to the class of Weighted CSPs.

8.3 Results for Empirical Evaluation of Weighted CSPs

In [1,2] we showed that the best performance on this domain was obtained by the AND/OR Branch-and-Bound *tree* search algorithm with static mini-bucket heuristics, at relatively large i-bounds, especially on non-binary WCSPs with relatively small domain sizes ($e.g.$, SPOT5 networks, ISCAS'89 circuits, Mastermind game instances). 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 (AOEDAC+PVO, DVO+AOEDAC and AOEDAC+DSO, respectively). 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.

Table 12
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												
spot5 (w*, h) (n, k, c)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=4		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=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		toolbar toolbar-BTD AOEDAC+PVO DVO+AOEDAC AOEDAC+DSO	
	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	-	-

8.3.1 Earth Observing Satellites

SPOT5 benchmark contains a collection of large real scheduling problems for the daily management of Earth observing satellites [17]. They can be easily formulated as WCSPs with binary and ternary constraints, as described in [1,3].

Tables 12 and 13 show detailed results on experiments with 7 SPOT5 networks using min-fill pseudo trees, as well as static and dynamic mini-bucket heuristics. The networks `42b`, `408b` and `505b` are sub-networks of the original ones and contain only binary constraints.

Table 13
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	-	-	-	-

Tree vs. graph AOBB. We notice again the benefit of using caching within depth-first AND/OR Branch-and-Bound search. As usual, 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 1 order of magnitude in terms of both running time and size of the search space explored. As we saw before for Bayesian networks, the impact of caching when using dynamic mini-bucket heuristics (Table 13) is not that pronounced as in the static case, across i-bounds. Notice that `toolbar` and `DVO+AOEDAC` (rightmost column in Table 12) are able to solve relatively efficiently only the first 3 test instances. On the other hand, `toolbar-BTD` fails only on the `408b` instance and is overall quite competitive.

AOBB vs. AOBF. When comparing the best-first against the depth-first AND/OR search algorithms we observe again here that `AOBF-C+SMB(`i`)` improves significantly (up to several orders of magnitude) in terms of both CPU time and number of nodes visited, especially for relatively small i-bounds. For example, on `505b`,

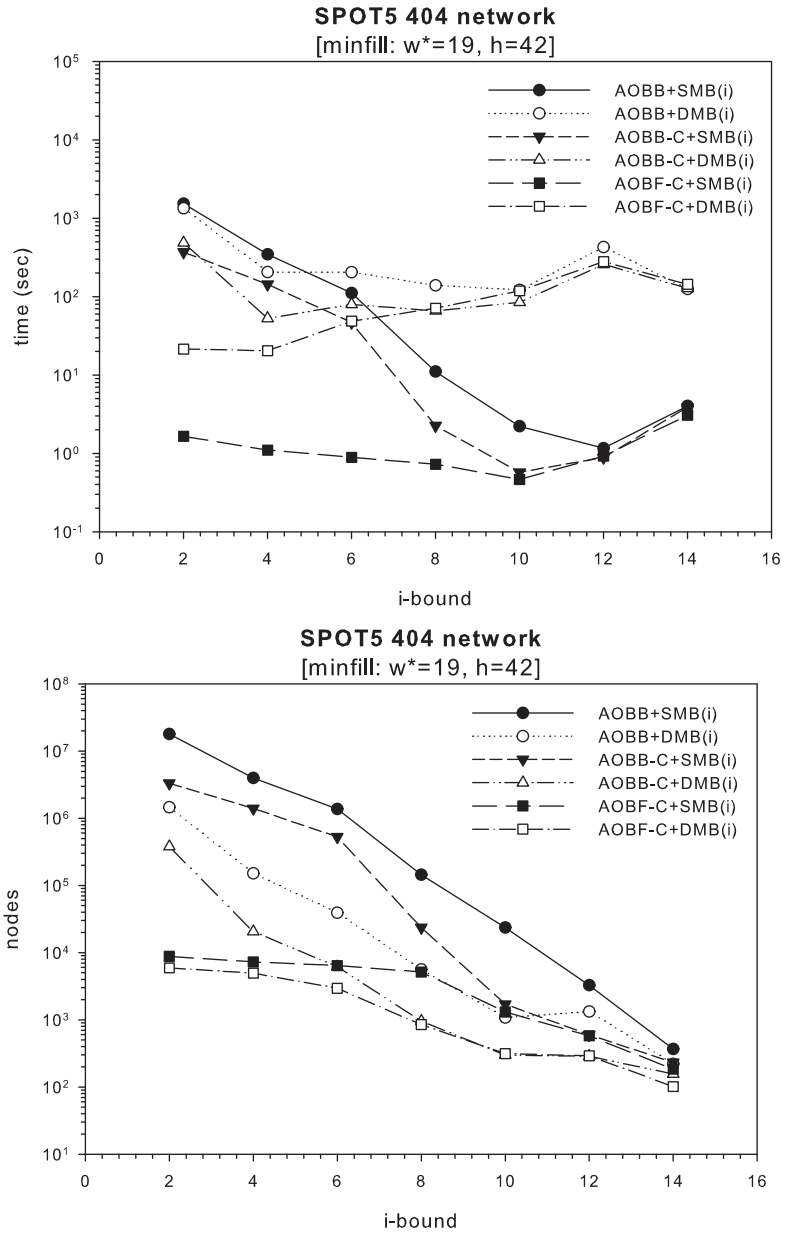


Fig. 17. Comparison of the impact of static and dynamic mini-bucket heuristics on the **404 SPOT5 network** from Tables 12 and 13. We show CPU time (top) and number of nodes (bottom).

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. Figure 17 displays 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 12 and 13, respectively). We see that the power of the dynamic mini-bucket heuristics is again more prominent for small i-bounds ($e.g.$, $i = 2$), for depth-first search. At larger

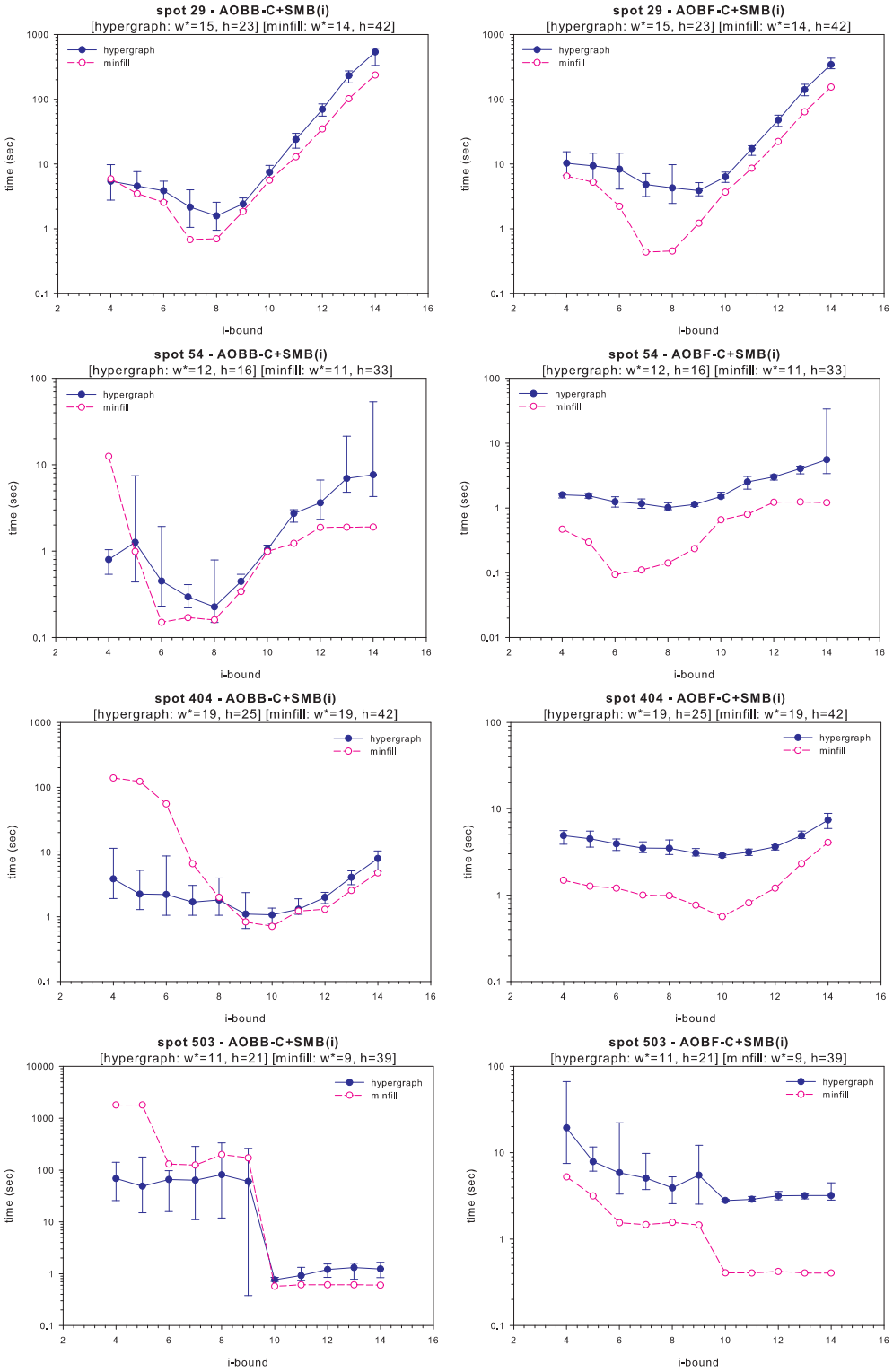


Fig. 18. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving SPOT5 networks with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

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) 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 18 we show the running time distribution of the algorithms using hypergraph based pseudo trees, over 20 independent runs. We see again that the hypergraph based pseudo trees are sometimes able to improve the performance of Branch-and-Bound search, especially for relatively small i-bounds ($e.g.$, 404, 503) for which the heuristic estimates are less accurate. For best-first search however, the min-fill pseudo trees offer the overall best performance.

8.3.2 ISCAS'89 Benchmark Circuits

ISCAS'89 circuits are a common benchmark used in formal verification and diagnosis. For our purpose, we converted each of these circuits into a non-binary WCSP instance by removing flip-flops and buffers in a standard way, creating hard constraints for gates and uniform unary cost functions for inputs. The penalty costs were distributed uniformly randomly between 1 and 10, as described in [1].

Tables 14 and 15 report the results for experiments with 10 circuits using static and dynamic mini-bucket heuristics, as well as min-fill based pseudo trees.

Tree vs. graph AOBB. When comparing the tree versus the graph AND/OR Branch-and-Bound search algorithms, we see again the same benefit of caching when using pre-compiled mini-bucket heuristics ($e.g.$, see s1238 circuit). As before, the diference between the tree and graph AND/OR algorithms is not too prominent when using dynamic mini-bucket heuristics (Table 15). The performance of toolbar that is designed specifically for the WCSP domain was very poor on this dataset and it was not able to solve the any of the problem instances within the 1 hour time limit. On the other hand, toolbar-BTD, which traverses an AND/OR search graph, is more competitive on this dataset and solves 6 out of the 10 test instances.

AOBB vs. AOBF. When comparing the depth-first versus the best-first AND/OR search algorithms (Tables 14 and 15), we see again that AOBF-C+SMB(i) outperforms significantly AOBB-C+SMB(i), especially for relatively small i-bounds. For instance, on the s1196 circuit, AOBF-C+SMB(10) is 2 orders of magnitude faster than AOBB-C+SMB(10). A similar behavior can be observed when using dynamic mini-bucket heuristics. For example, on the s1238 circuit, AOBF-C+DMB(8) outperforms AOBB-C+DMB(8) by one order of magnitude in terms of both running time and size of the search space explored. Overall, AOBF-C+SMB(i) is the best performing algorithm on this dataset.

Static vs. dynamic mini-bucket heuristics. Figure 19 plots the performance as a function of the mini-bucket i-bound, on the c880 network from Tables 14 and 15.

Table 14
CPU time in seconds and number of nodes visited for solving **ISCAS'89 circuits**, using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour.

min-fill pseudo tree												
	MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)		toolbar	
	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		toolbar-BTD	
iscas	AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)			
(w*, h)	AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)			
(n, d)	AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)			
	i=8		i=10		i=12		i=14		i=16			
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.08		0.09		0.14		0.22		0.59			
c432	-	-	-	-	9.266	52,778	9.172	52,240	1.203	1,738	-	-
(27, 45)	2010.53	23,355,897	148.39	1,713,265	5.94	76,346	5.84	75,420	0.70	1,958	-	-
(432, 2)	422.08	2,945,230	40.91	337,574	0.89	6,254	0.89	6,010	0.64	914		
	39.33	196,892	0.52	2,154	**0.31**	1,007	0.38	847	0.67	445		
	0.08		0.08		0.14		0.28		0.67			
c499	-	-	-	-	1.53	4,495	6.20	35,314	1.62	3,350	-	-
(23, 55)	96.46	1,265,425	39.65	526,517	1.42	18,851	37.26	486,656	2.16	22,065	100.96	1,203,734
(499, 2)	19.28	99,906	7.36	40,285	0.47	2,401	5.83	34,708	1.10	3,260		
	3.91	14,049	2.45	8,816	**0.34**	1,032	2.52	8,755	1.11	1,936		
	0.16		0.19		0.22		0.45		1.05			
c880	-	-	-	-	-	-	-	-	1173.93	4,792,550	-	-
(27, 67)	1698.08	19,992,512	1316.73	15,247,946	505.75	5,835,825	1134.61	13,568,696	245.06	2,837,010	-	-
(881, 2)	100.66	516,056	91.66	446,893	31.06	169,138	59.35	316,124	14.78	78,268		
	1.36	4,454	0.91	2,792	**0.81**	2,231	1.19	2,862	1.44	1,589		
	0.02		0.03		0.06		0.14		0.31			
s386	0.33	2,015	0.33	2,281	0.30	1,734	0.31	1,191	0.47	1,191	-	-
(19, 44)	0.14	2,073	0.33	4,867	0.22	2,699	0.22	1,420	0.37	1,420	0.19	738
(172, 2)	0.06	592	0.17	1,334	0.12	755	0.16	446	0.33	446		
	0.05	187	0.08	304	0.08	203	0.16	172	0.33	172		
	0.13		0.17		0.30		0.73		2.20			
s935	-	-	-	-	-	-	-	-	-	-	-	-
(66, 101)	2559.30	21,438,706	342.80	3,074,516	-	-	41.34	348,699	7.86	51,441	1.51	11,368
(441, 2)	1285.07	6,623,608	143.53	763,933	-	-	22.28	128,372	4.80	15,010		
	6.16	25,493	1.22	4,087	**1.19**	3,319	1.22	2,216	2.42	883		
	0.16		0.19		0.38		0.94		2.99			
s1196	-	-	-	-	-	-	-	-	-	-	-	-
(54, 97)	-	-	1347.95	12,392,442	-	-	1949.37	15,775,180	384.20	3,318,953	376.35	1,276,514
(562, 2)	3347.38	13,554,137	503.30	2,425,152	2299.72	11,488,366	734.66	3,524,780	149.81	793,417		
	22.67	72,075	**2.89**	9,336	13.02	40,210	7.27	21,989	3.56	2,090		
	0.16		0.22		0.38		0.92		3.20			
s1238	-	-	-	-	-	-	-	-	-	-	-	-
(59, 94)	-	-	-	-	1722.53	18,302,873	1394.86	14,213,319	38.08	360,788	-	-
(541, 2)	1897.37	8,386,634	1682.99	7,431,223	281.05	1,350,933	248.27	1,220,658	12.64	59,635		
	34.09	137,960	29.41	111,205	12.31	53,095	6.64	26,101	**4.63**	7,142		
	0.12		0.14		0.17		0.31		0.69			
s1423	-	-	-	-	-	-	-	-	4.58	7,382	-	-
(19, 44)	71.63	648,520	25.58	228,634	7.56	68,102	7.92	70,043	10.03	87,483	-	-
(749, 2)	7.61	37,244	2.75	11,423	1.48	7,164	1.39	5,868	1.34	3,787		
	1.16	3,873	0.70	2,193	**0.53**	1,683	0.69	1,663	1.00	1,317		
	0.16		0.24		0.41		1.05		3.45			
s1488	-	-	-	-	10.75	23,620	13.75	25,420	13.64	16,834	-	-
(47, 67)	6.67	50,613	46.83	430,141	4.00	29,729	5.19	33,827	5.20	17,904	1.80	9,315
(667, 2)	3.33	15,998	13.14	45,560	2.22	9,337	3.11	10,640	4.00	3,378		
	0.36	778	0.41	724	0.56	688	1.22	710	3.61	710		
	0.19		0.25		0.45		1.16		3.58			
s1494	-	-	191.36	366,822	52.47	140,792	19.86	44,190	20.23	38,034	-	-
(48, 69)	132.62	833,720	17.70	455,131	376.65	3,207,718	15.49	83,318	18.47	124,765	2.41	12,122
(661, 2)	62.87	127,934	5.64	17,279	27.64	80,895	6.92	23,131	9.02	20,004		
	1.44	5,694	**0.59**	1,472	0.95	2,311	1.50	1,476	3.81	985		

Table 15
CPU time in seconds and number of nodes visited for solving **ISCAS'89 circuits**, using **dynamic mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour.

min-fill pseudo tree										
	BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)	
iscas	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, d)	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
c432	403.44	33,506	191.69	14,303	13.56	1,026	8.86	627	25.53	1,014
(27, 45)	45.59	34,904	25.83	16,482	6.94	1,070	4.55	692	18.44	1,067
(432, 2)	35.19	3,861	20.69	2,302	6.69	860	4.53	627	18.38	773
	1.53	448	2.28	444	3.02	434	4.92	432	16.64	440
c499	40.99	3,502	31.85	3,102	9.91	987	42.66	2,848	64.72	1,664
(23, 55)	26.13	13,529	14.44	6,101	4.33	1,002	25.91	3,353	55.08	1,736
(499, 2)	24.44	2,485	13.42	1,726	4.28	742	25.25	1,251	56.14	963
	1.39	931	2.25	579	3.73	541	9.08	499	26.76	509
c880	-	-	-	-	547.35	18,112	648.52	19,546	851.63	17,125
(27, 67)	1078.04	796,699	762.16	569,471	85.64	32,748	170.55	36,187	411.42	20,357
(881, 2)	786.49	31,788	560.80	16,546	68.36	2,486	153.36	2,736	391.47	2,405
	8.77	1,378	9.94	1,304	**7.28**	956	16.83	958	43.94	894
s386	2.58	1,191	2.91	1,191	3.41	1,191	4.28	1,191	5.97	1,191
(19, 44)	0.81	1,420	1.14	1,420	1.61	1,420	2.52	1,420	4.19	1,420
(172, 2)	0.69	446	1.02	446	1.53	446	2.44	446	4.05	446
	0.30	172	0.50	172	0.86	172	1.53	172	2.89	172
s935	49.27	6,217	264.99	9,028	301.39	7,842	957.57	8,080	685.65	6,389
(66, 101)	18.27	7,400	234.47	10,250	267.02	9,164	915.57	11,164	653.32	8,377
(441, 2)	16.55	1,568	228.71	3,682	263.58	2,279	903.12	2,528	637.05	1,527
	5.47	479	23.87	553	27.19	454	140.51	490	243.98	441
s1196	233.39	18,040	335.50	15,525	670.04	13,677	1362.32	11,939	2938.12	10,988
(54, 97)	61.64	21,849	114.16	17,524	246.02	15,443	921.08	13,687	2556.58	12,419
(562, 2)	50.80	3,787	97.53	3,160	217.97	2,888	857.35	2,772	2393.16	2,413
	6.80	688	11.58	586	32.11	635	102.45	632	320.50	584
s1238	784.04	34,905	521.27	15,685	1395.39	17,852	2021.31	11,264	-	-
(59, 94)	266.45	39,493	188.83	21,252	566.96	20,945	913.24	13,857	-	-
(541, 2)	242.16	8,792	174.80	4,265	544.35	4,511	887.65	3,078	-	-
	18.69	827	22.47	666	57.59	591	192.10	632	1109.43	706
s1423	-	-	71.39	3,629	134.36	8,132	62.39	3,045	87.06	3,815
(19, 44)	38.36	26,772	35.02	17,801	36.19	19,719	22.27	3,513	36.83	4,323
(749, 2)	28.97	3,078	28.64	2,492	30.31	2,361	22.08	1,477	36.19	1,456
	5.97	1,191	6.25	1,141	9.48	1,126	12.39	762	23.30	754
s1488	146.03	14,365	139.83	12,475	181.58	12,748	306.35	12,748	730.54	12,748
(47, 67)	20.64	15,064	31.34	13,279	67.78	13,762	193.88	13,762	617.33	13,762
(667, 2)	18.33	2,824	29.20	2,634	65.34	2,576	190.94	2,576	614.10	2,576
	2.86	670	5.61	668	13.80	667	41.81	667	141.00	667
s1494	276.49	23,931	267.91	21,032	246.30	14,898	228.83	9,465	841.61	9,498
(48, 69)	71.52	25,104	84.92	22,082	112.49	15,698	151.00	9,706	761.02	9,913
(661, 2)	66.25	4,794	78.97	4,018	110.36	3,059	149.30	2,386	753.68	1,959
	10.42	758	**9.88**	679	20.38	667	58.75	666	189.33	665

Focusing for example on AOBF-C+SMB(i) we notice again the U-shaped curve formed by the running time.

Impact of the level of caching. Figure 20 displays the CPU time, as a function of the cache bound j, on 4 ISCAS'89 networks from Tables 14 using AOBB-C+SMB(i, j) (naive caching) and AOBB-AC+SMB(i,j) (adaptive caching), respectively. The spectrum of results is similar to what we observed before. Namely, adaptive

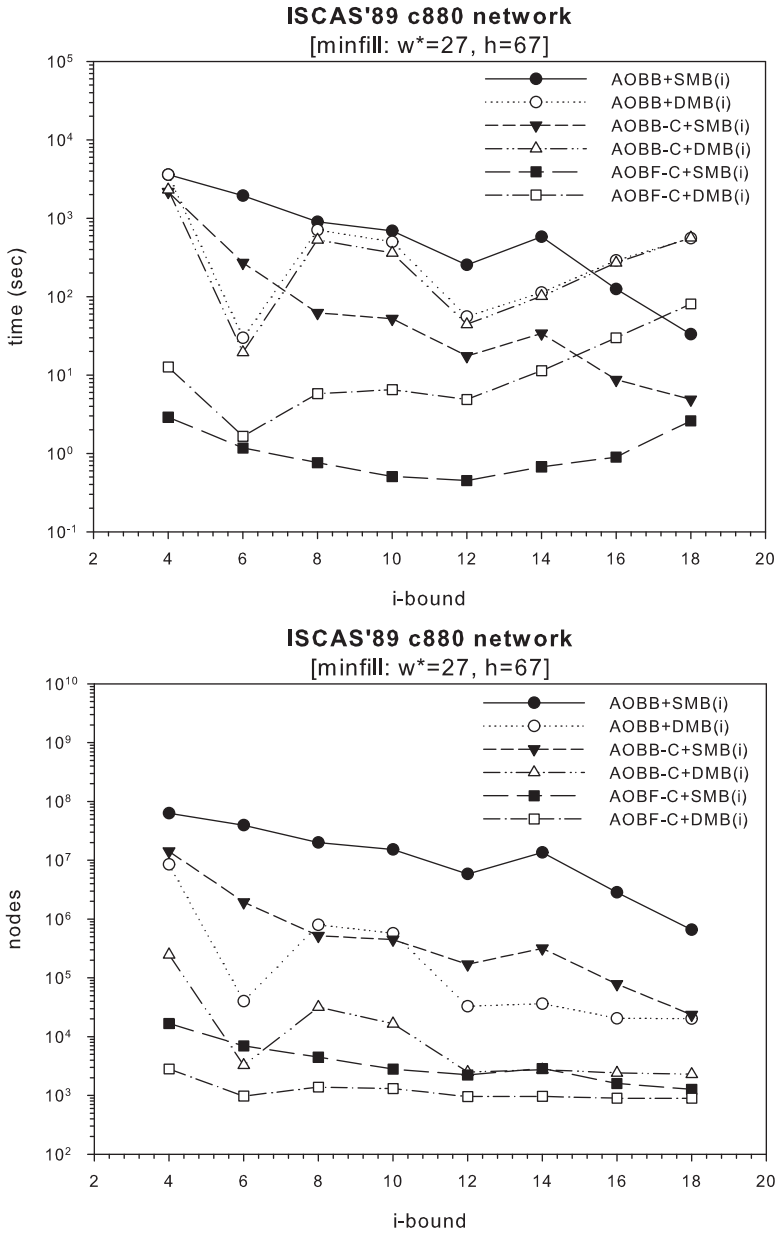


Fig. 19. Comparison of the impact of static and dynamic mini-bucket heuristics on the **c880 ISCAS'89 network** from Tables 14 and 15. We show CPU time (top) and number of nodes (bottom).

caching is more powerful than naive caching at smaller j bounds. As the cache bound increases, the two schemes approach gradually full caching. Notice that instances 1196 and 1488 have induced widths far larger than the maximum reported j-bound, and therefore the caching schemes will become identical when j is closer to the induced width.

Impact of the pseudo tree. The running time distribution over 20 independent runs of AOBB-C+SMB(i) and AOBF-C+SMB(i), using hypergraph based pseudo

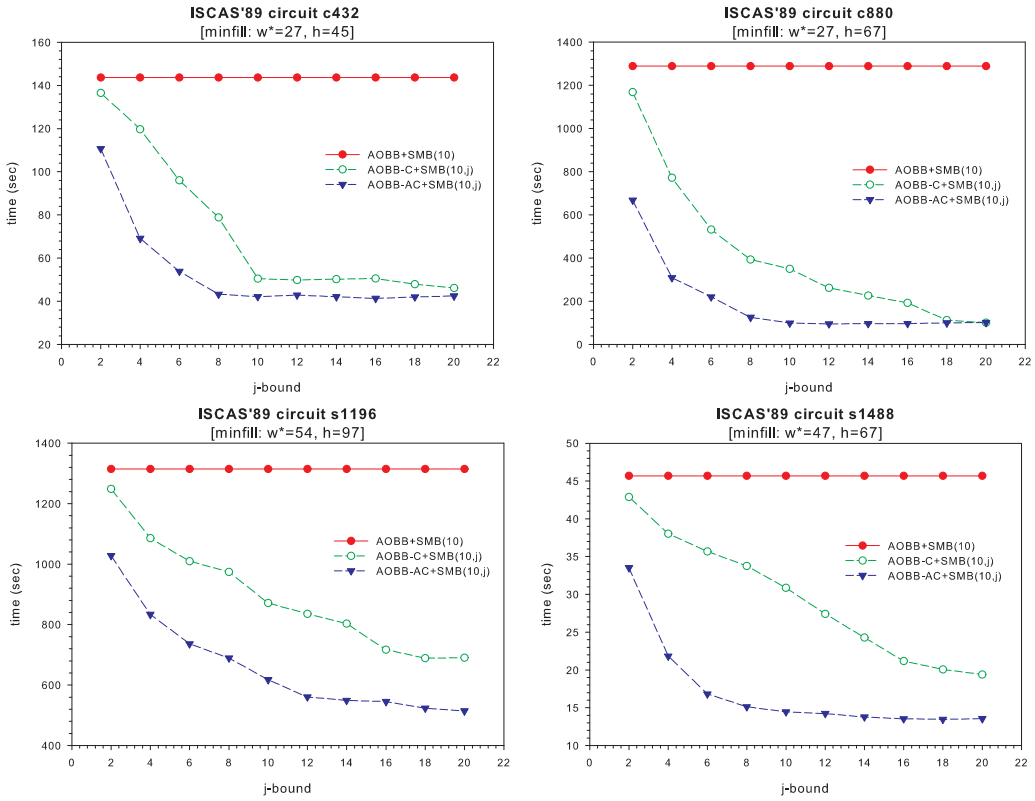


Fig. 20. Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **ISCAS'89 circuits**. Shown is CPU time in seconds.

trees, is displayed in Figure 21. We observe again that, in some cases, the hypergraph trees are able to improve significantly the performance of Branch-and-Bound as well as best-first search ($e.g.$, `c880`, `s1238`).

8.3.3 Mastermind Game Instances

Each of the Mastermind networks is a ground instance of a relational Bayesian network that models differing sizes of the popular game of Mastermind. These networks were produced by the PRIMULA System [4] and used in experimental results from [51]. For our purpose, we converted these networks into equivalent WCSP instances by taking the negative log probability of each conditional probability table entry. The resulting WCSP instances are quite large with the number of bi-valued variables n ranging between 1220 and 3692, and containing n unary and ternary constraints.

Table 16 shows the results for experiments with 6 game instances of increasing difficulty. The performance of the AND/OR algorithms with dynamic mini-buckets was quite poor in this case due to prohibitively high computational overhead at large i-bounds and is therefore not shown.

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

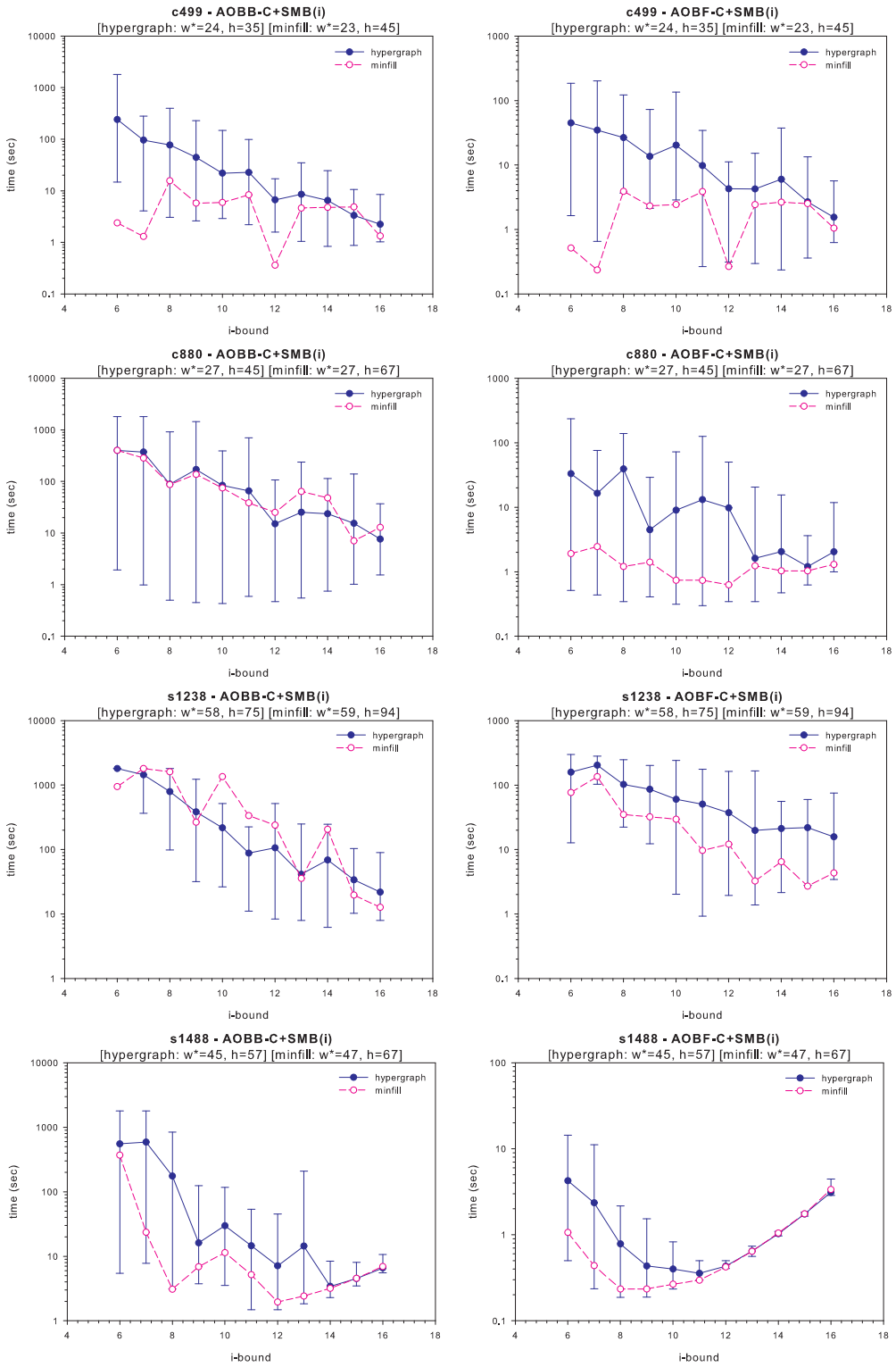


Fig. 21. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **ISCAS'89 networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

Table 16
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
	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		

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.

AOBB vs. AOBF. When comparing best-first against depth-first AND/OR search, we see that `AOBF-C+SMB(`i`)` offers the overall best performance on this domain as well. 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

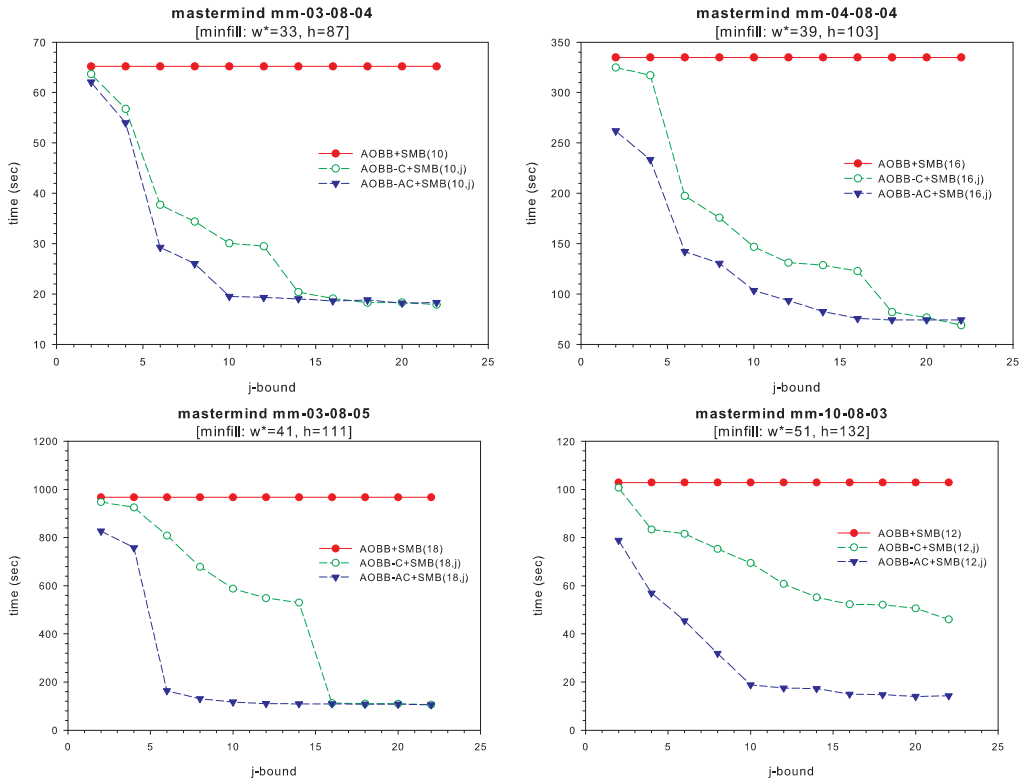


Fig. 22. Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **Mastermind networks**. Shown is CPU time in seconds.

`AOBB+SMB(18)`.

Impact of the level of caching. Figure 22 illustrates the CPU time, as a function of the cache bound j, on 4 problem instances from Table 16. 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 pseudo trees over 20 independent runs is displayed in Figure 23. 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.

Memory usage of AND/OR graph search. In Figure 24 we show 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.

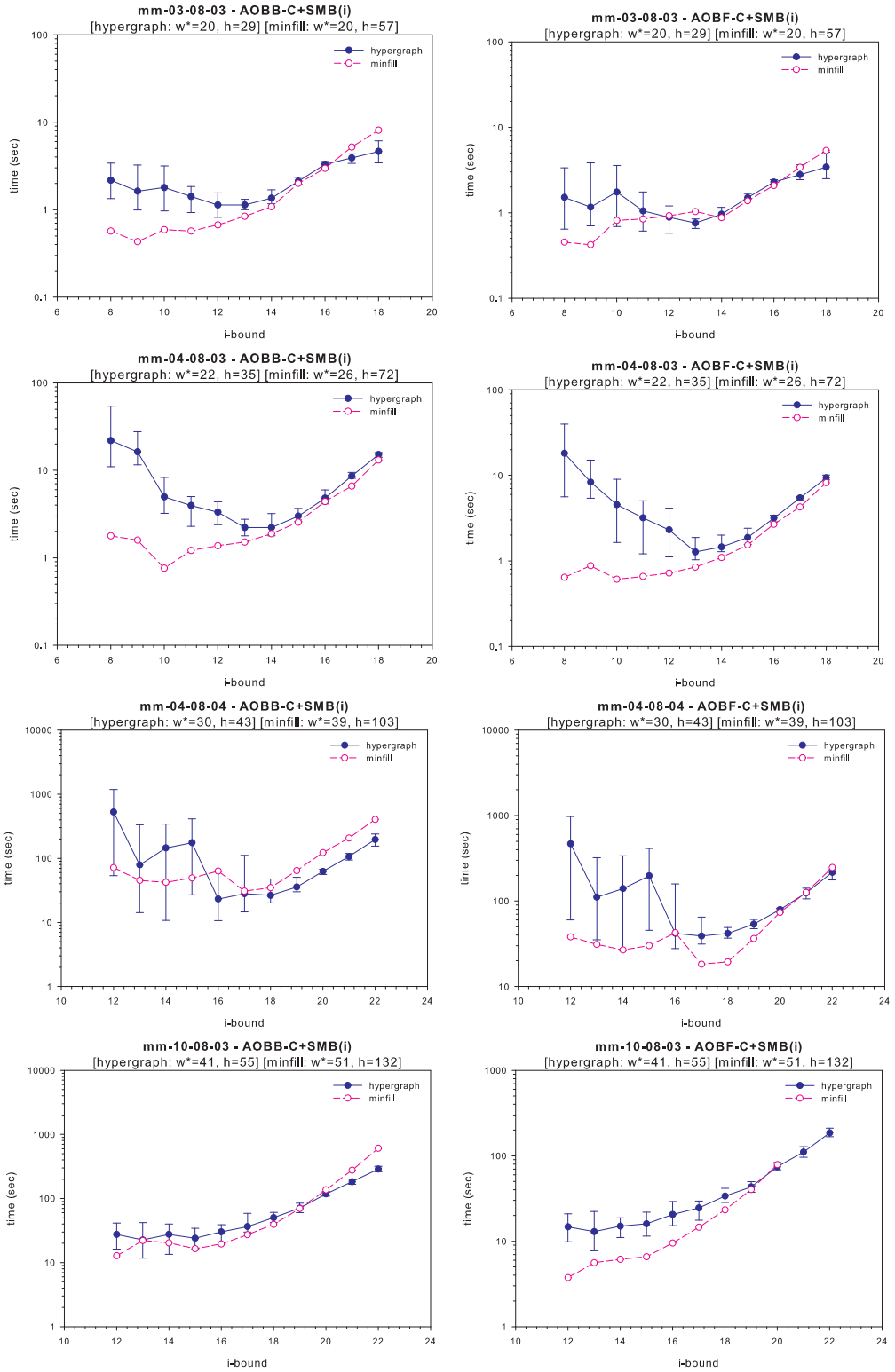


Fig. 23. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **Mastermind networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

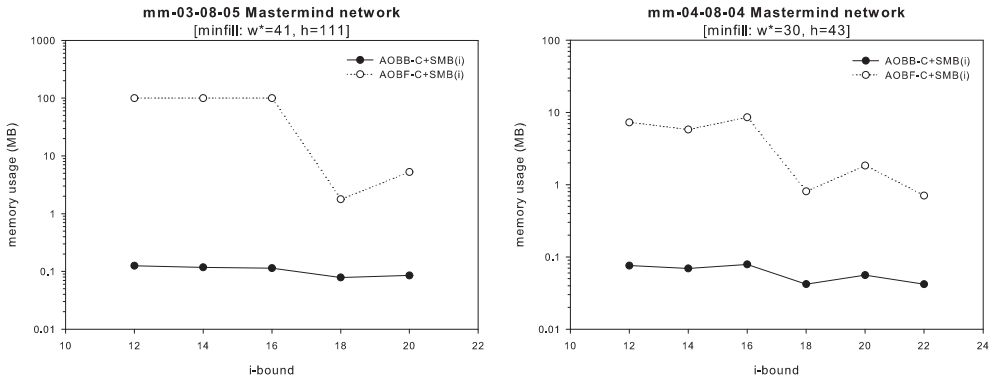


Fig. 24. Memory usage of the AOBB-C+SMB(i) and AOBF-C+SMB(i) algorithms on the **Mastermind networks** from Table 16.

9 Related Work

The idea of exploiting structural properties of the problem in order to enhance the performance of search algorithms in constraint satisfaction is not new. Freuder and Quinn [24] introduced the concept of pseudo tree arrangement of a constraint graph as a way of capturing independencies between subsets of variables. Subsequently, *pseudo tree search* is conducted over a pseudo tree arrangement of the problem which allows the detection of independent subproblems that are solved separately.

More recently, [52] extended pseudo tree search [24] to optimization tasks in order to boost the Russian Doll search [53] for solving Weighted CSPs. Dechter's graph-based back-jumping algorithm [54] uses a depth-first (DFS) spanning tree to extract knowledge about dependencies in the graph. The notion of DFS-based search was also used by [55] for a distributed constraint satisfaction algorithm. Bayardo and Miranker [25] reformulated the pseudo tree search algorithm in terms of back-jumping and showed that the depth of a pseudo-tree arrangement is always within a logarithmic factor off the induced width of the graph.

Recursive Conditioning (RC) [5] is based on the divide and conquer paradigm. Rather than instantiating variables to obtain a tree structured network like the cycle cutset scheme, RC instantiates variables with the purpose of breaking the network into independent subproblems, on which it can recurse using the same technique. The computation is driven by a data-structure called *dtree*, which is a full binary tree, the leaves of which correspond to the network CPTs. It can be shown that RC explores an AND/OR space [4]. A pseudo tree can be generated from the static ordering of RC dictated by the dtree. This ensures that whenever RC splits the problem into independent subproblems, the same happens in the AND/OR space.

Backtracking with Tree-Decomposition (BTD) [7] is a memory intensive method for solving constraint satisfaction and optimization problems which combines search techniques with the notion of tree decomposition. This mixed approach can be

viewed as searching an AND/OR search space whose backbone pseudo tree is defined by and structured along the tree decomposition. What is defined in [7] as structural goods, that is parts of the search space that would not be visited again as soon as their consistency (or optimal value) is known, corresponds precisely to the decomposition of the AND/OR space at the level of AND nodes, which root independent subproblems.

Value Elimination [6] is a recently developed algorithm for Bayesian inference. It was already explained in [6] that, under static variable ordering, there is a strong relation between Value Elimination and Variable Elimination. Given a static ordering d for Value Elimination, it can be shown that it actually traverses an AND/OR space [4]. The pseudo tree underlying the AND/OR search graph traversal by Value Elimination can be constructed as the bucket tree in reversed d. However, the traversal of the AND/OR space will be controlled by d, advancing the frontier in a hybrid depth or breadth first manner.

10 Summary and Conclusion

The paper continues to investigate the impact of the AND/OR search spaces perspective to solving general constraint optimization problems in graphical models. In contrast to the traditional OR space, the AND/OR search space is sensitive to problem decomposition. The size of the AND/OR search tree can be bounded exponentially by the depth of its guiding pseudo tree. This implies exponential time savings for any linear space search algorithms traversing the AND/OR search tree, in particular AND/OR Branch-and-Bound search, as we showed in [1–3]. Specifically, if the graphical model has treewidth w^*, the depth of the pseudo tree is $O(w^* \cdot logn)$. The AND/OR search tree can be extended into a graph by merging identical subtrees using graph information only. The size of the context minimal AND/OR search graph is exponential in the treewidth while 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. Searching the AND/OR search graph can be implemented by goods caching during search.

We therefore extended the AND/OR Branch-and-Bound algorithm to traversing an AND/OR search graph rather than an AND/OR search tree by equipping it with an efficient caching mechanism. We investigated two flexible context-based caching schemes that can adapt to the current 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 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 [9] and shown to be effective in the context of OR search trees [9]. 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 the Mini-Bucket i-bound which allows for a controllable 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 WCSPs. Our results demonstrated 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 Branch-and-Bound algorithms 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 cachin, the mini-bucket i-bound, constraint propagation, informed initial upper bounds and the quality of the guiding pseudo trees.

- **Impact of the level of caching.** 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 the cache bound j. 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 powerfull than the naive scheme ($e.g.$, grid networks from Figure 8, genetic linkage networks from Figure 14, ISCAS'89 circuits from Figure 20). As more space becomes available and the j-bound increases, the two schemes gradually approach full caching. The savings in number of nodes due to caching 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 diminished.
- **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 6 and 7, Mastermind game instances from Table 16). However, if the space is severly restricted, the dynamic mini-bucket heuristics appear to be the preferred choice, especially for relatively small i-bounds ($e.g.$, ISCAS'89 networks from Tables 14). This is because these heuristics are far more accurate for the same i-bound than the pre-compiled version.
- **Impact of determinism.** When the graphical model contains both deterministic information (hard constraints) as well as general cost functions, we demonstrated 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 was able in some cases to render the

search space almost backtrack-free ($e.g.$, ISCAS'89 networks from Table A.8). This caused in some cases a tremendous reduction in running time for the corresponding AND/OR Branch-and-Bound algorithms ($e.g.$, see for example the `s953` network from Table A.8).

- **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 that in some cases it causes a tremendeous speed-up over the initial approach (see for example the grid network from Table A.5, and the ISCAS'89 networks from Table A.8).
- **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/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 with larger induced widths. Our experiments demonstrated that when the induced width is small enough, which is typically the case 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 18). 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 have far smaller depths than the min-fill based ones, improve performance substantially ($e.g.$, genetic linkage networks from Figure 13 and Table 8). This is because for tree search the depth of the pseudo tree matters more than the induced width.

Our best-first and depth-first AND/OR graph search approaches leave room for future improvements, which are likely to make it more efficient in practice. `AOBF-C` may be improved in a variety of ways to render it more practical in special situations. First, rather than recompute a new estimated best partial solution tree after every node expansion, it is possible instead to expand one or more leaf nodes and some number of their descendants all at once, and then recompute an estimated best partial solution tree. This strategy can reduce the computational overhead of frequent bottom-up operations but incurs the risk that some node expansions may not be on the best solution tree.

As mentioned earlier, the space required by `AOBF-C` can be enormous, due to the fact that all nodes generated by the algorithm have to be saved prior to termination. Therefore, a memory bounding strategy may also be used for context minimal AND/OR graphs, as previously suggested in [23,29,56,57]. To employ it, the algorithm periodically reclaims needed storage space by discarding some portions of the explicated AND/OR search graph. For example, it is possible to determine a

few of those partial solution trees within the entire search graph having the *largest* estimated costs. These can be discarded periodically, with the risk of discarding one that might turn out to be the top of an optimal solution tree.

Acknowledgments

This work was partially supported by the NSF grants IIS-0086529 and IIS-0412854, the MURI ONR award N00014-00-1-0617 and the NIH grant R01-HG004175-02.

References

[1] R. Marinescu and R. Dechter. And/or branch-and-bound search for combinatorial optimization in graphical. *Technical Report. University of California, Irvine (submitted)*, 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 and R. Mateescu. And/or search spaces for graphical models. *Artificial Intelligence*, 171(1):73–106, 2007.

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

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

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

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

[9] 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.

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

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

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

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

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

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

[16] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.

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

[18] S. de Givry, I. Palhiere, Z. Vitezica, and T. Schiex. Mendelian error detection in complex pedigree using weighted constraint satisfaction techniques. In *ICLP Workshop on Constraint Based Methods for Bioinformatics*, 2005.

[19] P. Thbault, S. de Givry, T. Schiex, and C. Gaspin. Combining constraint processing and pattern matching to describe and locate structured motifs in genomic sequences. In *Fifth IJCAI-05 Workshop on Modelling and Solving Problems with Constraints*, 2005.

[20] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 542–547, 1999.

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

[22] S. de Givry, J. Larrosa, and T. Schiex. Solving max-sat as weighted csp. In *Principles and Practice of Constraint Programming (CP)*, pages 363–376, 2003.

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

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

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

[26] H. Bodlaender and J. Gilbert. Approximating treewidth, pathwidth and minimum elimination tree-height. *Technical Report, Utrecht University*, 1991.

[27] R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In *National Conference on Artificial Intelligence (AAAI)*, pages 298–304, 1996.

[28] U. Kjæaerulff. Triangulation of graph-based algorithms giving small total space. *Technical Report, University of Aalborg, Denmark*, 1990.

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

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

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

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

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

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

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

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

[41] J. Park. Using weighted max-sat engines to solve mpe. In *National Conference of Artificial Intelligence (AAAI)*, pages 682–687, 2002.

[42] 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.

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

[44] 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.

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

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

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

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

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

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

[51] 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.

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

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

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

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

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

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

Table A.1
CPU time and nodes visited for solving **UAI'06 networks**. Time limit 30 minutes.

minfill pseudo tree											
		MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)	
bn	SamIam	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)		AOBB+SMB(i)	
(w*, h)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)	
(n, k)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)	
		i=17		i=18		i=19		i=20		i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		5.53		10.31		17.45		38.36		62.11	
BN_31		-	-	-	-	-	-	-	-	-	-
(46, 160)	out	1026.73	4,741,037	1394.90	7,895,304	664.27	3,988,933	680.61	4,293,760	131.17	380,470
(1156, 2)		411.33	1,445,200	486.47	2,131,977	209.80	831,431	210.81	889,782	81.61	94,507
		140.41	293,445	126.23	292,293	85.69	142,650	86.00	114,046	**73.14**	25,392
		7.39		13.34		24.38		46.08		81.72	
BN_33		-	-	-	-	-	-	-	-	-	-
(43, 163)	-	1404.15	3,540,778	293.85	685,246	618.55	1,441,245	410.08	1,018,353	197.08	360,880
(1444, 2)		429.02	982,130	125.78	210,552	236.42	408,855	160.61	256,191	120.33	89,308
		75.92	142,932	**41.14**	41,865	58.14	61,064	73.20	49,760	95.16	22,256
		7.61		12.86		24.50		40.33		64.63	
BN_35		-	-	-	-	-	-	-	-	-	-
(41, 168)	-	464.44	1,755,561	548.11	1,954,720	316.78	1,108,708	199.67	663,784	226.10	622,551
(1444, 2)		42.95	126,215	107.17	243,533	81.59	151,632	56.11	65,657	78.27	58,973
		29.77	29,837	36.58	34,987	43.28	28,088	51.28	15,953	76.28	18,048
		7.25		13.58		22.61		44.14		87.30	
BN_37		-	-	-	-	-	-	-	-	-	-
(45, 159)	-	126.85	428,643	97.03	298,477	79.75	183,016	65.74	89,948	121.39	168,957
(1444, 2)		26.42	55,571	20.19	33,475	25.45	14,703	45.61	8,815	94.55	16,400
		15.83	15,399	19.47	11,046	26.55	6,621	46.84	4,315	90.66	5,610
		6.86		13.13		25.58		44.06		75.49	
BN_39		-	-	-	-	-	-	-	-	-	-
(48, 162)	-	-	-	-	-	-	-	-	-	1202.01	3,366,427
(1444, 2)		1161.65	2,615,679	1370.21	3,448,072	507.18	1,499,020	403.07	1,043,378	220.74	518,011
		117.03	340,362	247.08	725,738	131.44	316,862	112.27	213,676	**111.20**	127,872
		6.97		11.98		21.09		36.44		65.75	
BN_41		-	-	-	-	-	-	-	-	-	-
(49, 164)	-	188.60	486,844	151.80	364,363	83.39	168,340	109.92	195,506	123.58	162,274
(1444, 2)		56.72	119,737	47.30	77,653	33.81	32,774	50.81	38,467	76.42	31,763
		23.50	42,795	**22.05**	20,485	27.22	12,030	43.38	16,549	71.61	11,648

A Experiments - Bayesian Networks

A.1 UAI'06 Evaluation Dataset

The UAI 2006 Evaluation Dataset [5] contains a collection of random as well as real-world belief networks that were used during the first UAI 2006 Inference Evaluation contest.

Tables A.1 and A.2 show the results for experiments with 15 networks from the repository. Instances `BN_31` through `BN_41` are random grid networks with deterministic CPTs, while instances `BN_126` through `BN_134` represent random cod-

[5] http://ssli.ee.washington.edu/bilmes/uai06InferenceEvaluation

Table A.2
CPU time and nodes visited for solving **UAI'06 networks**. Time limit 30 minutes.

minfill pseudo tree											
bn (w*, h) (n, k)	SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=17		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=19		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=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		3.27		6.69		11.63		23.42		47.84	
BN_126		301.56	2,085,673	823.32	6,662,948	512.27	3,189,855	55.16	257,866	54.39	70,027
(54, 70)	-	363.05	4,459,174	953.71	10,991,861	118.58	1,333,266	52.24	386,490	57.74	150,391
(512, 2)		255.27	2,324,776	816.71	8,423,064	92.85	829,994	47.67	244,943	55.27	99,056
		16.22	64,202	25.64	85,148	26.88	76,645	30.95	37,666	54.59	30,197
		3.42		6.66		14.59		26.66		47.66	
BN_127		-	-	-	-	-	-	-	-	130.27	631,093
(57, 74)	out	-	-	-	-	-	-	-	-	155.09	1,384,957
(512, 2)		-	-	-	-	-	-	-	-	128.94	860,026
		51.80	223,327	58.63	251,134	62.75	215,796	64.28	166,741	66.35	84,007
		3.81		7.58		13.64		28.30		49.02	
BN_128		4.14	2,558	7.59	1,266	14.84	10,244	28.31	471	49.13	3,147
(48, 73)	out	4.13	5,587	7.47	1,712	14.89	18,734	29.05	625	49.39	5,823
(512, 2)		4.11	3,636	7.48	1,411	15.00	12,034	28.38	552	49.33	4,203
		3.97	883	7.75	925	13.78	808	28.39	478	49.13	575
		3.56		5.58		12.67		27.81		50.60	
BN_129		-	-	-	-	176.24	1,603,304	1337.90	11,794,805	257.42	1,855,134
(52, 68)	out	865.99	11,469,012	-	-	194.91	1,999,591	-	-	259.83	2,542,057
(512, 2)		573.74	5,730,592	-	-	167.14	1,688,675	1388.01	13,437,762	219.09	1,747,613
		out		194.56	922,831	out		**132.45**	537,371	246.39	910,769
		3.03		6.50		10.95		26.31		46.44	
BN_130		**21.56**	182,120	-	-	869.44	7,310,190	-	-	57.06	109,669
(54, 67)	out	28.67	348,660	-	-	1015.05	10,905,151	-	-	60.91	205,010
(512, 2)		22.49	239,771	-	-	863.15	8,414,475	-	-	58.94	147,085
		27.72	115,091	68.53	273,987	76.53	299,172	60.55	158,650	69.63	107,771
		3.44		6.59		11.20		21.88		39.70	
BN_131		**17.06**	137,631	39.02	323,431	1149.74	10,230,128	47.25	228,703	-	-
(48, 72)	out	24.36	296,576	55.20	677,149	-	-	66.63	673,358	-	-
(512, 2)		18.69	176,456	41.63	396,234	1254.88	12,395,143	50.42	303,818	-	-
		29.03	116,166	50.13	209,748	28.47	79,689	36.89	73,163	65.74	120,153
		2.95		5.59		10.50		25.56		45.77	
BN_132		-	-	-	-	-	-	756.69	6,584,446	**578.99**	4,819,402
(49, 71)	out	-	-	-	-	-	-	912.40	10,251,600	823.40	10,207,347
(512, 2)		-	-	-	-	-	-	778.22	7,456,812	643.96	6,037,908
		out		out		out		out		out	
		3.61		7.03		13.20		27.50		52.69	
BN_133		-	-	**16.84**	104,521	31.28	171,645	127.32	929,016	55.33	30,699
(54, 71)	out	-	-	19.38	169,574	35.58	272,258	168.17	1,859,117	56.22	71,195
(512, 2)		-	-	17.19	133,794	31.64	202,954	135.60	1,184,600	55.22	40,483
		59.61	258,891	27.44	98,148	32.91	93,613	45.09	90,337	55.31	13,491
		3.38		6.34		12.09		27.08		54.35	
BN_134		-	-	-	-	-	-	-	-	-	-
(52, 70)	out	-	-	-	-	-	-	-	-	-	-
(512, 2)		-	-	-	-	-	-	-	-	-	-
		out		**85.77**	373,081	out		96.19	377,064	97.59	214,591

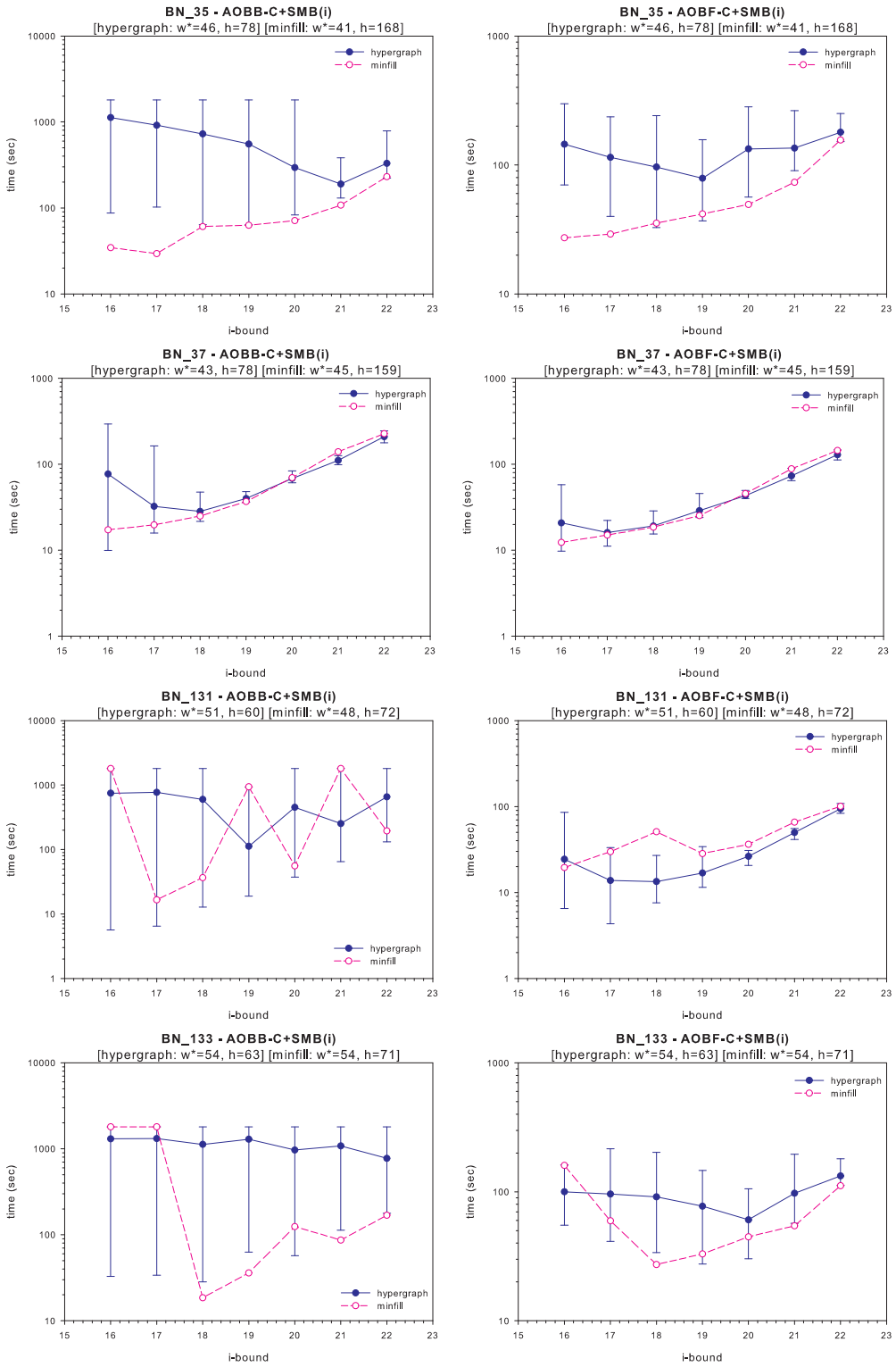


Fig. A.1. Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **UAI'06 networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

ing networks with 128 input bits, 4 parents per XOR bit and channel variance $\sigma^2 = 0.40$. We report only on the Branch-and-Bound and best-first search algorithms using static mini-bucket heuristics. The dynamic mini-bucket heuristics were not competitive due to their much higher computational overhead at relatively large i-bounds. The guiding pseudo trees were generated in this case using the min-fill heuristic.

We notice again the superiority of AOBB-C+SMB(i) compared with the tree version of the algorithm, AOBB+SMB(i), at relatively small i-bounds where both algorithms rely primarily on search rather than on pruning, and especially on the first set of grid networks ($e.g.$, BN_31, ..., BN_41). For instance, on the BN_35 network, AOBB-C+SMB(16) finds the most probable explanation 10 times faster than AOBB+SMB(16) exploring a search space 14 times smaller. This is in contrast to what we observe on the second set of coding networks ($e.g.$, BN_126, ..., BN_133), where AOBB-C+SMB(i) is only slightly better than AOBB+SMB(i) across the reported i-bounds. This is because the AND/OR graph explored effectively was very close to a tree due to the substantial pruning caused by the mini-bucket heuristics.

Overall, best-first AND/OR search offers the best performance on this domain and the difference in running time as well as size of the search space explored is up to several orders of magnitude, compared to the Branch-and-Bound algorithms. For example, on the BN_131 network, AOBF-C+SMB(16) finds the optimal solution in less than 20 seconds, whereas both AOBB+SMB(16) and AOBB-C+SMB(16) exceed the 30 minute time bound.

Figure A.1 plots the running time distribution of AOBB-C+SMB(i) and AOBF-C+SMB(i) using hypergraph partitioning based pseudo trees, over 20 independent runs. We see that the hypergraph trees are sometimes able to improve the performance of AOBB-C+SMB(i), especially at small i-bounds ($e.g.$, BN_133). For best-first search, the min-fill trees usually offer the best performance (except on BN_131, where the hypergraph trees are superior across i-bounds).

A.2 ISCAS'89 Benchmark Circuits

ISCAS'89 circuits [6] are a common benchmark used in formal verification and diagnosis. For our purpose, we converted each of these circuits into a belief network by removing flip-flops and buffers in a standard way, creating a deterministic conditional probabilistic tables for each gate and putting uniform distributions on the input signals.

Tables A.3 and A.4 show the results for experiments with 10 circuits, using min-fill based pseudo trees as well as static and dynamic mini-bucket heuristics. As usual,

[6] Available at http://www.fm.vslib.cz/kes/asic/iscas/

Table A.3
CPU time and nodes visited for solving belief networks derived from **ISCAS'89 circuits** with static mini-bucket heuristics and min-fill pseudo trees. Time limit 30 minutes.

minfill pseudo tree													
		MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)		MBE(i)	
	SamIam	BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
iscas89		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, d)		i=6		i=8		i=10		i=12		i=14		i=16	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.06		0.08		0.09		0.14		0.20		0.56	
c432		-	-	-	-	-	-	0.35	432	0.45	432	0.78	432
(27, 45)	out	-	-	-	-	1154.96	20,751,699	**0.16**	432	0.24	432	0.59	432
(432, 2)		-	-	-	-	182.53	2,316,024	**0.16**	432	0.24	432	0.58	432
		out		out		106.27	488,462	0.20	432	0.28	432	0.63	432
		0.08		0.09		0.09		0.16		0.30		0.66	
c499		0.38	499	0.42	499	0.42	499	0.48	499	0.59	499	1.03	499
(23, 55)	139.89	**0.11**	499	0.13	499	0.13	499	0.19	499	0.33	499	0.69	499
(499, 2)		0.13	499	0.11	499	0.14	499	0.19	499	0.31	499	0.69	499
		0.17	499	0.17	499	0.17	499	0.25	499	0.39	499	0.73	499
		0.17		0.16		0.19		0.22		0.44		1.05	
c880		-	-	1.56	881	1.80	881	1.70	881	1.84	881	2.58	881
(27, 67)	out	0.23	884	**0.22**	881	0.25	881	0.28	881	0.50	881	1.14	881
(880, 2)		**0.22**	884	0.24	881	0.25	881	0.28	881	0.48	881	1.11	881
		0.33	884	0.34	881	0.36	881	0.39	881	0.61	881	1.20	881
		0.03		0.03		0.05		0.08		0.16		0.31	
s386		0.17	1,358	0.11	677	0.06	172	0.09	172	0.17	172	0.33	172
(19, 44)	3.66	**0.05**	257	**0.05**	257	**0.05**	172	0.08	172	0.16	172	0.33	172
(172, 2)		**0.05**	207	**0.05**	207	**0.05**	172	0.08	172	0.16	172	0.39	172
		0.05	194	**0.05**	194	0.06	172	0.08	172	0.16	172	0.30	172
		0.13		0.14		0.17		0.28		0.70		2.14	
s953		-	-	-	-	-	-	-	-	1170.80	4,031,967	841.72	3,075,116
(66, 101)	out	1054.79	9,919,295	23.67	238,780	58.00	549,181	36.06	434,481	2.72	21,499	3.77	19,117
(440, 2)		899.63	7,715,133	17.99	155,865	48.13	417,924	17.00	132,139	**2.19**	13,039	3.03	8,007
		out		41.03	150,598	110.45	408,828	36.50	113,322	4.06	12,256	4.19	7,143
		0.14		0.16		0.19		0.34		0.91		2.94	
s1196		-	-	-	-	-	-	-	-	-	-	-	-
(54, 97)	out	31.55	316,875	332.14	3,682,077	7.44	77,205	31.39	320,205	26.24	289,873	11.77	99,935
(560, 2)		18.05	104,316	124.53	686,069	**3.69**	26,847	14.23	94,985	9.47	62,883	6.05	25,262
		26.16	77,019	158.19	372,129	7.22	23,348	26.97	80,264	17.64	48,114	9.17	20,307
		0.14		0.16		0.20		0.36		0.86		2.98	
s1238		-	-	-	-	398.13	2,078,885	208.45	1,094,713	931.71	4,305,175	51.86	253,706
(59, 94)	out	4.45	57,355	14.77	187,499	3.70	47,340	2.28	25,538	2.45	20,689	3.94	13,032
(540, 2)		1.77	12,623	4.95	34,056	1.30	8,476	**1.00**	5,418	1.42	4,780	3.38	3,364
		2.30	5,921	6.61	17,757	1.70	4,298	1.31	2,730	1.69	2,415	3.56	1,673
		0.13		0.12		0.14		0.16		0.31		0.66	
s1423		-	-	-	-	-	-	0.98	762	1.19	749	1.55	749
(24, 54)	107.48	0.27	1,986	0.50	5,171	0.53	5,078	**0.22**	866	0.36	749	0.70	749
(748, 2)		**0.22**	1,246	**0.22**	1,256	**0.22**	1,235	**0.22**	818	0.36	749	0.70	749
		0.31	959	0.31	921	0.31	913	0.31	774	0.44	749	0.80	749
		0.14		0.17		0.22		0.39		1.00		3.30	
s1488		15.38	92,764	1.69	6,460	3.20	17,410	1.77	6,511	1.94	4,083	3.95	830
(47, 67)	out	16.58	135,563	2.20	17,150	3.39	28,420	1.63	12,285	1.64	6,670	3.38	964
(667, 2)		13.22	82,294	**1.02**	5,920	2.50	15,621	1.19	6,024	1.47	3,516	3.38	784
		21.75	74,658	1.67	5,499	4.22	14,445	1.84	5,372	1.80	3,124	3.48	749
		0.14		0.17		0.22		0.42		1.06		3.36	
s1494		10.86	64,629	978.87	3,412,403	222.28	815,708	5.94	36,804	73.35	268,814	4.08	1,874
(48, 69)	out	14.75	158,070	47.41	479,498	11.69	118,754	18.74	202,343	3.06	21,530	3.56	2,431
(661, 2)		7.30	41,798	19.69	108,768	4.81	27,711	7.00	41,977	**2.06**	8,104	3.50	1,750
		9.67	24,849	27.28	65,859	7.86	19,678	11.48	28,793	3.03	6,484	3.72	1,625

Table A.4
CPU time and nodes visited for solving belief networks derived from **ISCAS'89 circuits** with dynamic mini-bucket heuristics and min-fill pseudo trees. Time limit 30 minutes.

	minfill pseudo tree											
	BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)		BB-C+DMB(i)	
iscas89	AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)		AOBB+DMB(i)	
	AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)		AOBB-C+DMB(i)	
(w*, h)	AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)		AOBF-C+DMB(i)	
(n, d)	i=6		i=8		i=10		i=12		i=14		i=16	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
c432	-	-	159.56	21,215	2.50	432	3.20	432	4.61	432	9.05	432
(27, 45)	-	-	32.00	39,711	1.02	432	1.69	432	3.06	432	7.64	432
(432, 2)	1161.25	323,359	23.02	4,951	**1.00**	432	1.73	432	3.09	432	7.59	432
	1019.19	86,460	26.05	2,342	1.58	432	2.70	432	4.70	432	11.19	432
c499	1.95	499	2.11	499	2.52	499	3.77	499	6.67	499	18.00	499
(23, 55)	**0.58**	499	0.73	499	1.14	499	2.41	499	5.25	499	16.48	499
(499, 2)	**0.58**	499	0.74	499	1.14	499	2.41	499	5.27	499	16.59	499
	0.83	499	1.11	499	1.88	499	3.75	499	8.03	499	24.72	499
c880	8.30	881	10.64	881	10.19	881	13.33	881	18.56	881	31.95	881
(27, 67)	1.25	881	1.47	881	2.16	881	3.92	881	9.11	881	22.06	881
(880, 2)	**1.20**	881	1.42	881	2.11	881	3.94	881	9.03	881	22.70	881
	1.74	881	2.20	881	3.41	881	6.14	881	13.81	881	32.58	881
s386	0.22	172	0.28	172	0.39	172	0.59	172	1.05	172	2.00	172
(19, 44)	0.13	172	0.17	172	0.28	172	0.52	172	0.97	172	1.89	172
(172, 2)	**0.11**	172	0.17	172	0.30	172	0.52	172	0.97	172	1.87	172
	0.18	172	0.30	172	0.50	172	0.83	172	1.51	172	2.86	172
s953	33.02	2,737	16.75	912	46.28	1,009	17.20	467	137.08	577	128.41	447
(66, 101)	32.08	2,738	**15.95**	913	45.80	1,010	16.17	468	135.61	578	127.72	447
(440, 2)	32.23	2,738	15.98	913	45.92	1,010	16.14	468	136.09	578	127.83	447
	54.72	2,738	25.22	913	73.86	1,010	26.45	468	213.59	578	208.19	447
s1196	3.75	580	4.81	568	37.45	924	88.91	863	386.75	1,008	876.84	817
(54, 97)	1.56	660	2.45	568	33.30	924	77.02	863	362.32	1,008	881.15	817
(560, 2)	**1.55**	620	2.44	568	33.52	924	79.05	863	355.10	1,008	852.14	817
	2.53	604	4.03	568	63.70	924	154.17	857	676.68	1,008	1653.96	817
s1238	43.56	5,841	6.77	601	302.53	17,278	36.39	651	76.70	558	215.21	551
(59, 94)	2.61	1,089	3.70	795	13.16	1,824	26.39	849	59.20	744	188.27	737
(540, 2)	**2.52**	704	3.63	619	12.97	996	26.22	667	59.09	571	188.31	564
	4.00	635	6.17	610	21.30	769	44.23	657	97.00	564	306.08	557
s1423	5.05	751	5.27	749	5.67	749	6.66	749	9.09	749	15.83	749
(24, 54)	0.88	751	0.97	749	1.36	749	2.27	749	4.75	749	11.55	749
(748, 2)	**0.83**	751	0.95	749	1.34	749	2.22	749	4.73	749	11.45	749
	1.24	751	1.56	749	2.28	749	3.69	749	7.45	749	17.23	749
s1488	4.34	670	4.39	670	5.81	668	10.64	667	27.50	667	86.81	667
(47, 67)	**1.13**	670	1.67	670	3.11	668	7.70	667	24.19	667	83.58	667
(667, 2)	**1.13**	670	1.64	670	3.06	668	7.67	667	24.25	667	83.86	667
	1.89	670	2.95	670	5.62	668	13.58	667	41.12	667	139.93	667
s1494	7.80	814	5.61	679	15.16	719	25.03	686	70.19	686	149.49	667
(48, 69)	7.53	898	**2.95**	679	12.59	719	22.44	686	68.11	686	146.44	667
(661, 2)	5.06	814	2.97	679	12.66	719	22.98	686	69.81	686	149.05	667
	8.00	814	4.50	679	17.39	719	30.20	686	88.50	686	195.53	667

for each test instance we generated a single MPE query without any evidence. We see that AOBB-C+SMB(i) improves over AOBB+SMB(i), especially at relatively small i-bounds. For instance, on the s1196 circuit, AOBB-C+SMB(8) is about 3 times faster than AOBB+SMB(i). This is in contrast to what we see when using dynamic mini-bucket heuristics. Here, there is no noticable difference between the tree and graph AND/OR Branch-and-Bound, because the pruning power of the heuristics rendered the search space almost backtrack free, across i-bounds. Over-

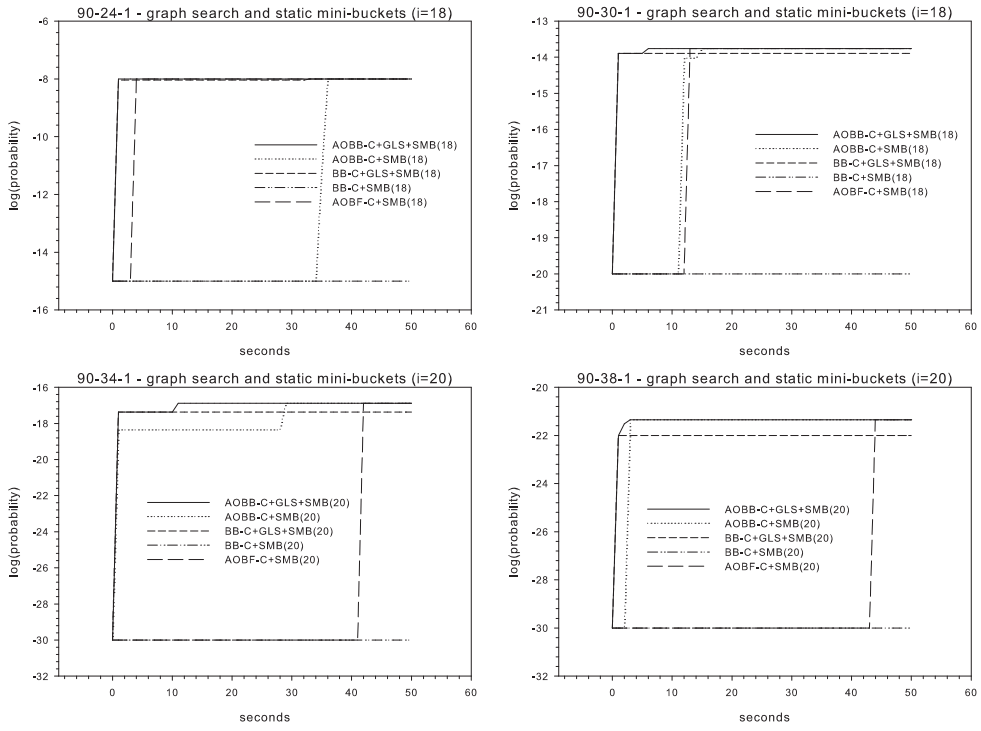


Fig. A.2. Anytime behavior of AOBB-C+SMB(i) on **grid networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

all, the dynamic mini-bucket heuristics were inferior to the corresponding static ones for large i-bounds, however, smaller i-bound dynamic mini-buckets were sometimes cost-effective (*e.g.*, s953). Notice that SAMIAM is able to solve only 2 out of 10 test instances. Moreover, AOBF-C+SMB(i) (resp. AOBF-C+DMB(i)) was overall inferior to AOBB-C+SMB(i) (resp. AOBB-C+DMB(i)) because of its computational overhead.

A.3 The Anytime Behavior of AND/OR Branch-and-Bound Search and the Impact of Initial Bounds

Figures A.2 and A.3 show the search trace of the AND/OR Branch-and-Bound algorithms for solving selected instances of grid networks and UAI'06 Dataset, respectively. We see again that AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i) take advantage of the quality of the initial lower bound produced by GLS, and find close to optimal solutions much earlier than AOBB-C+SMB(i) and BB-C+SMB(i), respectively.

Tables A.5, A.6, and A.7 report detailed results for AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i) on grid networks and UAI'06 Dataset networks, respectively. We see that the lower bound computed by GLS was in many cases equal to the optimal solution and therefore AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i)

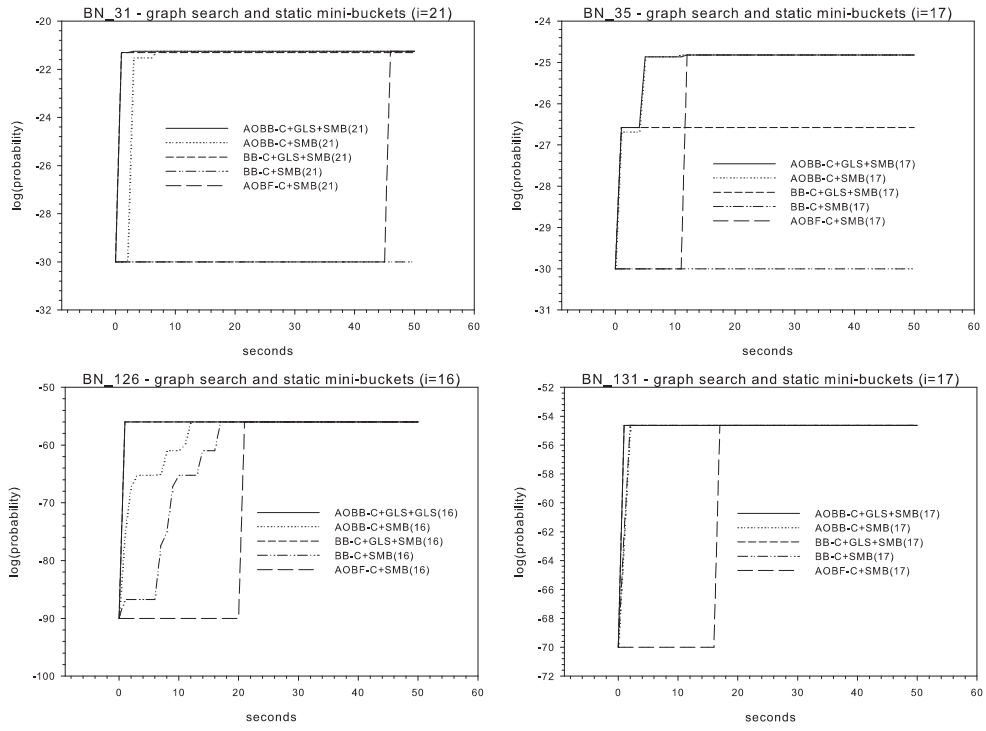


Fig. A.3. Anytime behavior of AOBB-C+SMB(i) on **UAI'06 networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

improved considerably over AOBB-C+SMB(i) and BB-C+SMB(i), respectively.

A.4 The Impact of Determinism in Bayesian Networks

Table A.8 shows the results for experiments with 5 belief networks derived from ISCAS'89 circuits. We see that constraint propagation via unit resolution plays a dramatic role on this domain, rendering the search space almost backtrack-free across i-bounds. For instance, on the s953, AOBB-C+SAT+SMB(6) is 3 orders of magnitude faster than AOBB-C+SMB(6), while AOBF-C+SMB(6) exceeded the memory limit. When looking at the AND/OR Branch-and-Bound algorithms that exploit the local search based initial lower bound, namely AOBB-C+GLS+SMB(i) and AOBB-C+SAT+GLS+SMB(i), we see that they did not expand any nodes. This is because the lower bound obtained by GLS, which was the optimal solution in this case, was equal to the mini-bucket upper bound computed at the root node. The best performance on this domain were achieved by AOBB-C+SAT+SMB(i) and AOBB-C+SAT+GLS+SMB(i), respectively, for the smallest reported i-bound (namely $i = 6$). Notice also the poor performance of SAMIAM which ran out of memory on all tests.

Table A.5

CPU time and nodes visited for solving **grid networks** with static mini-bucket heuristics. Time limit 1 hour. Number of flips for GLS is 50,000.

minfill pseudo tree											
		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
grid	SamIam	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)	
	GLS	AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)	
(w*, h)		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)	
(n, e)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)	
		i=8		i=10		i=12		i=14		i=16	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.23	3,297	0.06	373	**0.05**	102	0.06	102	0.06	102
90-10-1	0.13	0.38	3,272	0.19	289	0.19	0	0.19	0	0.20	0
(16, 26)	0.25*	0.14	2,638	0.06	819	0.05	101	0.06	101	0.06	101
(100, 0)		0.28	2,580	0.22	789	0.19	0	0.20	0	0.19	0
		0.27	2,012	0.11	661	**0.05**	100	0.06	100	0.06	100
		126.69	1,233,891	121.00	1,317,992	1.52	16,547	0.42	2,770	0.61	1,450
90-14-1	11.97	21.02	217,185	31.64	339,762	0.88	5,892	0.50	1,122	0.78	1,178
(23, 37)	0.43*	4.22	55,120	3.66	48,513	0.45	5,585	**0.23**	1,361	0.53	1,210
(196, 0)		3.59	45,023	2.77	32,454	0.66	3,684	0.45	1,067	0.78	1,062
		3.20	18,796	2.70	15,764	0.55	2,899	0.30	898	0.63	857
		-	-	-	-	40.05	345,255	2.38	16,942	1.23	5,327
90-16-1	147.19	-	-	1163.43	9,106,361	35.72	306,583	1.97	12,104	1.42	4,614
(26, 42)	0.49*	209.60	2,695,249	35.45	441,364	4.23	50,481	1.19	11,029	**0.95**	4,810
(256, 0)		37.28	453,073	8.14	96,962	4.17	46,138	1.44	10,702	1.23	4,552
		25.70	126,861	10.59	54,796	4.47	22,993	1.42	6,015	1.22	3,067
		i=12		i=14		i=16		i=18		i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	-	-	-	-	-	-
90-24-1	out	-	-	1773.64	6,065,308	609.65	2,008,431	111.58	263,250	632.68	1,705,699
(36, 61)	0.53	-	-	1273.09	9,047,518	596.27	4,923,760	70.42	473,675	74.99	412,291
(576, 20)		3594.60	24,363,798	66.20	425,585	20.16	93,911	11.17	7,850	28.16	27,868
		out		21.94	75,637	10.59	33,770	**6.06**	5,144	23.80	17,291
		-	-	-	-	395.67	1,635,447	-	-	67.09	277,685
90-26-1	out	-	-	-	-	235.36	922,243	65.39	282,394	41.70	73,616
(35, 64)	0.56	146.97	878,874	152.80	962,484	4.36	15,632	12.92	46,489	22.13	2,242
(676, 40)		43.64	248,603	85.72	495,039	10.83	14,580	14.47	6,226	28.38	1,466
		19.06	65,271	24.39	79,619	**4.27**	7,190	8.05	3,777	22.44	1,435
		-	-	-	-	-	-	-	-	-	-
90-30-1	out	-	-	-	-	-	-	-	-	-	-
(38, 68)	0.72	652.15	3,882,300	165.74	1,070,823	155.20	956,837	40.14	212,963	59.28	174,715
(900, 60)		276.00	1,491,880	84.39	442,754	78.81	376,916	31.69	89,045	64.23	148,540
		158.97	534,385	46.73	157,187	47.27	154,496	**21.06**	45,201	57.97	100,800
		-	-	-	-	-	-	-	-	-	-
90-34-1	out	-	-	-	-	-	-	-	-	-	-
(43, 79)	1.31	-	-	-	-	-	-	-	-	369.36	823,604
(1154, 80)		-	-	-	-	980.51	4,943,817	1751.86	5,516,888	315.38	630,406
		out		out		243.63	596,978	270.88	667,013	**71.19**	67,611
		-	-	-	-	-	-	-	-	-	-
90-38-1	out	-	-	-	-	-	-	-	-	-	-
(47, 86)	1.11	969.02	2,623,971	1753.10	3,794,053	203.67	614,868	165.45	488,873	113.06	214,919
(1444, 120)		819.16	2,450,643	1806.57	3,804,190	224.80	607,453	187.63	482,946	138.64	211,562
		101.69	174,786	103.80	146,237	54.00	95,511	**53.44**	78,431	73.10	59,856

Table A.6
CPU time and nodes visited for solving **UAI’06 networks** with static mini-bucket heuristics. Time limit 30 minutes. Number of flips for GLS is 500,000.

minfill pseudo tree											
		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)		BB-C+SMB(i)	
bn	SamIam	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)	
	GLS	AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)		AOBB-C+SMB(i)	
(w*, h)		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)	
(n, d)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)		AOBF-C+SMB(i)	
		i=17		i=18		i=19		i=20		i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	-	-	-	-	-	-
BN_31		-	-	-	-	-	-	-	-	-	-
(46, 160)	out	411.33	1,445,200	486.47	2,131,977	209.80	831,431	210.81	889,782	81.61	94,507
(1156, 2)	9.86*	357.86	1,172,122	375.05	1,573,677	202.66	775,258	187.34	752,284	79.01	56,409
		140.41	293,445	126.23	292,293	85.69	142,650	86.00	114,046	**73.14**	25,392
		-	-	-	-	-	-	-	-	-	-
BN_33		-	-	-	-	-	-	-	-	-	-
(43, 163)	-	429.02	982,130	125.78	210,552	236.42	408,855	160.61	256,191	120.33	89,308
(1444, 2)	12.30*	434.97	980,701	134.47	207,658	244.72	399,206	167.39	245,144	129.35	85,745
		75.92	142,932	**41.14**	41,865	58.14	61,064	73.20	49,760	95.16	22,256
		-	-	-	-	-	-	-	-	-	-
BN_35		-	-	-	-	-	-	-	-	-	-
(41, 168)	-	42.95	126,215	107.17	243,533	81.59	151,632	56.11	65,657	78.27	58,973
(1444, 2)	12.38	49.97	120,205	112.42	224,908	89.85	151,619	66.16	74,585	89.31	71,614
		29.77	29,837	36.58	34,987	43.28	28,088	51.28	15,953	76.28	18,048
		-	-	-	-	-	-	-	-	-	-
BN_37		-	-	-	-	-	-	-	-	-	-
(45, 159)	-	26.42	55,571	20.19	33,475	25.45	14,703	45.61	8,815	94.55	16,400
(1444, 2)	12.70	29.77	48,211	26.17	31,674	32.11	13,808	49.63	7,774	99.00	19,871
		15.83	15,399	19.47	11,046	26.55	6,621	46.84	4,315	90.66	5,610
		-	-	-	-	-	-	-	-	-	-
BN_39		-	-	-	-	-	-	-	-	-	-
(48, 164)	-	1161.65	2,615,679	1370.21	3,448,072	507.18	1,499,020	403.07	1,043,378	220.74	518,011
(1444, 2)	12.88	472.36	1,076,698	782.69	2,026,535	276.27	778,118	190.16	436,932	113.67	168,410
		117.03	340,362	247.08	725,738	131.44	316,862	112.27	213,676	**111.20**	127,872
		-	-	-	-	-	-	-	-	-	-
BN_41		-	-	-	-	-	-	-	-	-	-
(49, 164)	-	56.72	119,737	47.30	77,653	33.81	32,774	50.81	38,467	76.42	31,763
(1444, 2)	12.29*	63.16	117,948	52.52	73,947	40.45	30,930	58.53	37,018	86.72	30,487
		23.50	42,795	**22.05**	20,485	27.22	12,030	43.38	16,549	71.61	11,648

Table A.7
CPU time and nodes visited for solving **UAI'06 networks** with static mini-bucket heuristics. Time limit 30 minutes. Number of flips for GLS is 500,000.

minfill pseudo tree											
bn (w*, h) (n, d)	SamIam GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=17		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=18		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=19		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=20		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		301.56	2,085,673	823.32	6,662,948	512.27	3,189,855	55.16	257,866	54.39	70,027
BN_126		**9.83**	63,674	15.78	85,215	19.31	76,346	27.69	37,226	51.38	30,317
(54, 70)	-	255.27	2,324,776	816.71	8,423,064	92.85	829,994	47.67	244,943	55.27	99,056
(512, 2)	6.08*	10.91	83,227	17.74	117,859	20.66	99,518	28.66	49,175	54.28	42,873
		16.22	64,202	25.64	85,148	26.88	76,645	30.95	37,666	54.59	30,197
		-	-	-	-	-	-	-	-	130.27	631,093
BN_127		**26.44**	238,020	31.02	250,746	36.19	215,054	44.34	166,176	57.52	83,380
(57, 74)	out	-	-	-	-	-	-	-	-	128.94	860,026
(512, 2)	5.75*	27.59	282,349	31.11	295,100	38.67	280,166	46.03	214,590	57.47	113,743
		51.80	223,327	58.63	251,134	62.75	215,796	64.28	166,741	66.35	84,007
		4.14	2,558	7.59	1,266	14.84	10,244	28.31	471	49.13	3,147
BN_128		4.50	854	8.05	694	14.17	778	29.44	461	48.75	551
(48, 73)	out	4.11	3,636	7.48	1,411	15.00	12,034	28.38	552	49.33	4,203
(512, 2)	5.95*	4.14	1,022	7.91	974	13.92	991	28.75	547	49.64	674
		3.97	883	7.75	925	13.78	808	28.39	478	49.13	575
		-	-	-	-	176.24	1,603,304	1337.90	11,794,805	257.42	1,855,134
BN_129		244.08	2,419,418	150.30	1,408,350	150.56	1,352,916	119.70	923,635	142.14	914,833
(52, 68)	out	573.74	5,730,592	-	-	167.14	1,688,675	1388.01	13,437,762	219.09	1,747,613
(512, 2)	5.89*	245.08	2,443,843	95.64	961,434	142.55	1,412,079	**76.16**	564,895	138.53	979,046
		out		194.56	922,831	out		132.45	537,371	246.39	910,769
		21.56	182,120	-	-	869.44	7,310,190	-	-	57.06	109,669
BN_130		**14.55**	114,610	87.28	751,400	41.73	299,845	42.86	158,612	58.53	107,880
(54, 67)	out	22.49	239,771	-	-	863.15	8,414,475	-	-	58.94	147,085
(512, 2)	5.87*	15.36	158,150	36.24	364,352	43.25	392,961	43.19	211,380	57.91	144,741
		27.72	115,091	68.53	273,987	76.53	299,172	60.55	158,650	69.63	107,771
		17.06	137,631	39.02	323,431	1149.74	10,230,128	47.25	228,703	-	-
BN_131		**15.42**	118,238	26.77	212,338	19.56	82,414	28.69	73,552	51.69	122,085
(48, 72)	out	18.69	176,456	41.63	396,234	1254.88	12,395,143	50.42	303,818	-	-
(512, 2)	5.87*	16.70	150,341	28.22	256,361	20.34	101,662	29.16	91,103	54.12	156,925
		29.03	116,166	50.13	209,748	28.47	79,689	36.89	73,163	65.74	120,153
		-	-	-	-	-	-	756.69	6,584,446	578.99	4,819,402
BN_132		683.65	5,987,145	429.96	3,750,177	838.83	7,484,051	627.50	5,584,689	392.78	3,296,711
(49, 71)	out	-	-	-	-	-	-	778.22	7,456,812	643.96	6,037,908
(512, 2)	5.89*	686.08	6,499,878	439.89	4,252,274	718.66	6,905,710	453.25	4,319,442	**387.02**	3,557,198
		out		out		out		out		out	
		-	-	**16.84**	104,521	31.28	171,645	127.32	929,016	55.33	30,699
BN_133		29.13	258,988	17.09	102,193	22.77	93,433	36.28	90,006	53.97	17,865
(49, 71)	out	-	-	17.19	133,794	31.64	202,954	135.60	1,184,600	55.22	40,483
(512, 2)	5.79*	30.50	329,146	16.50	125,945	22.66	116,553	36.17	112,317	53.92	17,069
		59.61	258,891	27.44	98,148	32.91	93,613	45.09	90,337	55.31	13,491
		-	-	-	-	-	-	-	-	-	-
BN_134		105.61	1,029,072	**43.16**	373,641	115.67	1,065,258	60.94	376,402	75.16	213,954
(52, 70)	out	-	-	-	-	-	-	-	-	-	-
(512, 2)	5.83*	109.97	1,170,028	44.33	439,065	123.91	1,253,376	60.72	401,521	76.38	241,382
		out		85.77	373,081	out		96.19	377,064	97.59	214,591

Table A.8
CPU time and nodes visited for solving belief networks derived from **ISCAS'89 circuits** using static mini-bucket heuristics. Time limit 30 minutes.

minfill pseudo tree											
iscas89 (w*, h) (n, d)	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=6		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=8		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=10		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	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	182.53	2,316,024	0.16	432	0.24	432
c432	out	374.29	4,336,403	189.13	2,043,475	1.02	9,512	0.16	432	0.25	432
(27, 45)		**0.05**	0	0.06	0	0.09	0	0.13	0	0.19	0
(432, 2)	0.08*	0.06	0	0.08	0	0.09	0	0.13	0	0.20	0
		out		out		106.27	488,462	0.20	432	0.28	432
		899.63	7,715,133	17.99	155,865	48.13	417,924	17.00	132,139	2.19	13,039
s953	out	0.19	829	0.16	667	0.20	685	0.31	623	0.74	623
(66, 101)		**0.12**	0	0.13	0	0.17	0	0.28	0	0.69	0
(440, 2)	0.05*	0.13	0	0.13	0	0.17	0	0.30	0	0.70	0
		out		41.03	150,598	110.45	408,828	36.50	113,322	4.06	12,256
		18.05	104,316	124.53	686,069	3.69	26,847	14.23	94,985	9.47	62,883
s1196	out	0.19	565	0.19	565	0.23	565	0.38	565	0.92	565
(54, 97)		0.14	0	0.16	0	0.20	0	0.34	0	0.89	0
(560, 2)	0.08*	**0.13**	0	0.14	0	0.20	0	0.34	0	0.87	0
		26.16	77,019	158.19	372,129	7.22	23,348	26.97	80,264	17.64	48,114
		13.22	82,294	1.02	5,920	2.50	15,621	1.19	6,024	1.47	3,516
s1488	out	0.20	708	0.20	667	0.25	667	0.44	667	1.06	667
(47, 67)		0.14	0	0.16	0	0.22	0	0.44	0	0.99	0
(667, 2)	0.13*	**0.13**	0	0.16	0	0.20	0	0.47	0	0.99	0
		21.75	74,658	1.67	5,499	4.22	14,445	1.84	5,372	1.80	3,124
		7.30	41,798	19.69	108,768	4.81	27,711	7.00	41,977	2.06	8,104
s1494	out	0.20	665	0.22	665	0.27	665	0.45	665	1.11	665
(48, 69)		**0.16**	0	0.17	0	0.22	0	0.41	0	1.09	0
(661, 2)	0.11*	**0.16**	0	0.17	0	0.22	0	0.42	0	1.22	0
		9.67	24,849	27.28	65,859	7.86	19,678	11.48	28,793	3.03	6,484