Artificial Intelligence 173 (2009) 1457–1491

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

AND/OR Branch-and-Bound search for combinatorial optimization in graphical models

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

[a] Cork Constraint Computation Centre, University College Cork, Ireland
[b] Donald Bren School of Information and Computer Science, University of California, Irvine, CA 92697, USA

ARTICLE INFO

Article history:
Received 12 April 2008
Received in revised form 1 July 2009
Accepted 10 July 2009
Available online 21 July 2009

Keywords:
Search
AND/OR search
Decomposition
Graphical models
Bayesian networks
Constraint networks
Constraint optimization

ABSTRACT

This is the first of two papers presenting and evaluating the power of a new framework for combinatorial optimization in graphical models, based on AND/OR search spaces. We introduce a new generation of depth-first Branch-and-Bound algorithms that explore the AND/OR search tree using static and dynamic variable orderings. The virtue of the AND/OR representation of the search space is that its size may be far smaller than that of a traditional OR representation, which can translate into significant time savings for search algorithms. The focus of this paper is on linear space search which explores the AND/OR search tree. In the second paper we explore memory intensive AND/OR search algorithms. In conjunction with the AND/OR search space we investigate the power of the mini-bucket heuristics in both static and dynamic setups. We focus on two most common optimization problems in graphical models: finding the Most Probable Explanation in Bayesian networks and solving Weighted CSPs. In extensive empirical evaluations we demonstrate that the new AND/OR Branch-and-Bound approach improves considerably over the traditional OR search strategy and show how various variable ordering schemes impact the performance of the AND/OR search scheme.

1. Introduction

Graphical models such as Bayesian 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. Optimization problems such as finding the most likely state of a Bayesian 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) were always known to be good at exploiting the independencies captured by the underlying graphical model. They provide worst case time guarantees exponential in the treewidth of the underlying graph. Unfortunately, any method that is time-exponential in the treewidth is also space exponential in the treewidth or separator width, therefore not practical for models with large treewidth.

Search-based algorithms (*e.g.*, depth-first Branch-and-Bound search) traverse the model's search space where each path represents a partial or full solution. The linear structure of such traditional search spaces does not retain the independencies represented in the underlying graphical models and, therefore, search-based algorithms may not be nearly as effective

* Corresponding author.
E-mail addresses: r.marinescu@4c.ucc.ie (R. Marinescu), dechter@ics.uci.edu (R. Dechter).
[1] This work was done while at the University of California, Irvine.

0004-3702/$ – see front matter
doi:10.1016/j.artint.2009.07.003

as inference-based algorithms in using this information. Moreover, these methods do not accommodate informative performance guarantees. This situation has changed in the past few years with the introduction of AND/OR search algorithms for graphical models. In addition, search methods require only an implicit, generative, specification of the functional relationships (that may be given in a procedural or functional form) while inference schemes often rely on an explicit tabular representation over the (discrete) variables. For these reasons, search-based algorithms are usually the preferred choice for models with large treewidth and with implicit representation.

The AND/OR search space for graphical models [1] is a new framework that is sensitive to the independencies in the model, often resulting in exponentially reduced complexities. It is guided by a *pseudo tree* [2,3] that captures independencies in the graphical model, resulting in a search space exponential in the depth of the pseudo tree, rather than in the number of variables.

In this paper we present a new generation of AND/OR Branch-and-Bound algorithms (AOBB) that explore the AND/OR search tree in a depth-first manner for solving optimization problems in graphical models. As in traditional Branch-and-Bound search, the efficiency of these algorithms depends heavily also on their guiding heuristic function. A class of partitioning-based heuristic functions, based on the Mini-Bucket approximation [4] and known as *static mini-bucket heuristics* was shown to be powerful for optimization problems [5] in the context of the traditional OR search spaces. The Mini-Bucket algorithm provides a scheme for extracting heuristic information from the functional specification of the graphical model and is applicable to any graphical model. The accuracy of the Mini-Bucket algorithm is controlled by a bounding parameter, called i-bound, which allows varying degrees of heuristics accuracy and results in a spectrum of search algorithms that can trade off heuristic strength and search [5]. We show here how the pre-computed mini-bucket heuristic as well as any other heuristic information can be incorporated into AND/OR search. We also introduce *dynamic mini-bucket heuristics*, which are computed dynamically at each node of the search tree.

Since variable orderings can influence dramatically the search performance, we also introduce a collection of *dynamic* AND/OR Branch-and-Bound algorithms that combine AND/OR decomposition with dynamic variable orderings.

We apply the depth-first AND/OR Branch-and-Bound approach to two common optimization problems in graphical models: finding the Most Probable Explanation (MPE) in Bayesian networks [6] and solving Weighted Constraint Satisfaction Problems (WCSP) [7]. Our results show conclusively on various benchmark problems that the new depth-first AND/OR Branch-and-Bound algorithms improve dramatically over traditional ones exploring the OR search space, especially when the heuristic estimates are inaccurate and the algorithms rely primarily on search.

Following preliminary notations and definitions (Section 2), Sections 3, 4 and 5 provide background on graphical models, on the classic OR Branch-and-Bound approach, and on the AND/OR representation of the search space. Section 6 presents our new depth-first AND/OR Branch-and-Bound algorithm. Section 7 presents several general purpose heuristic functions that can guide the search focusing on the mini-bucket heuristics. Section 8 describes its extension with dynamic variable ordering heuristics. Section 9 shows the empirical evaluation, Section 10 overviews related work and Section 11 provides a summary and concluding remarks.

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.*, $X, Y, Z, \ldots$), sets of variables by bold faced uppercase letters (*e.g.*, $\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \ldots$) and values of variables by lower case letters (*e.g.*, $x, y, z, \ldots$). An assignment $(X_1 = x_1, \ldots, X_n = x_n)$ can be abbreviated as $x = (\langle X_1, x_1\rangle, \ldots, \langle X_n, x_n\rangle)$ or $x = (x_1, \ldots, x_n)$. 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 $x = (x_1, \ldots, x_n)$ 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 $\mathbf{V} = \{X_1, \ldots, X_n\}$ is a set of vertices (nodes), and $\mathbf{E} = \{(X_i, X_j) \mid X_i, X_j \in V\}$ 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 $d = X_1, \ldots, X_n$ 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 $\mathbf{S} = \{S_1, \ldots, S_t\}$ 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) For each $X_i \in \mathbf{X}$, the set $\{v \in \mathbf{V} \mid X_i \in \chi(v)\}$ induces a connected subtree of T (running intersection property).

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.

2.3. AND/OR search spaces

An AND/OR state space representation of a problem is a 4-tuple $\langle S, O, S_g, s_0 \rangle$ [8]. S is a set of states which can be either OR or AND states (the OR states represent alternative ways for solving the problem while the AND states often represent problem decomposition into subproblems, all of which need to be solved). O is a set of operators. An OR operator transforms an OR state into another state, and an AND operator transforms an AND state into a set of states. There is a set of goal states $S_g \subseteq S$ and a start node $s_0 \in S$.

The AND/OR state space model induces an explicit AND/OR search *graph*. Each state is a node and child nodes are obtained by applicable AND or OR operators. The search graph includes a *start* node. The terminal nodes (having no children) are labeled as SOLVED or UNSOLVED.

A *solution tree* of an AND/OR search graph G is a subtree which: (1) contains the start node s_0; (2) if n in the tree is an OR node then it contains one of its child nodes in G, and if n is an AND node it contains all its children in G; (3) all its terminal nodes are SOLVED.

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) $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a set of variables;
(2) $\mathbf{D} = \{D_1, \ldots, D_n\}$ is the set of their respective finite domains of values;
(3) $\mathbf{F} = \{f_1, \ldots, f_r\}$ is a set of real-valued functions, each defined over a subset of variables $S_i \subseteq \mathbf{X}$ (i.e., the scope);
(4) $\bigotimes_i f_i \in \{\prod_i f_i, \sum_i f_i\}$ is a combination operator.

The graphical model represents the combination of all its functions: $\bigotimes_{i=1}^{r} f_i$.

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

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

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, $c(y) = \bigotimes_{f \in \mathbf{F_Y}} f(y[scope(f)])$. We will often abuse notation writing $c(y) = \bigotimes_{f \in F_\mathbf{Y}} f(y)$ instead.

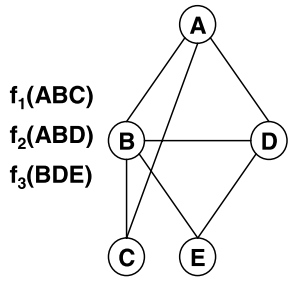
Definition 8 *(primal graph).* The *primal graph* of a graphical model has the variables as its nodes and an edge connects any two variables that appear in the scope of the same function.

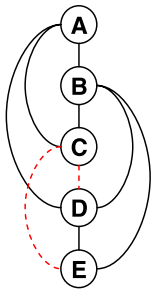
There are various queries (tasks) that can be posed over graphical models. We refer to all as *automated reasoning problems*. In general, an optimization task is a reasoning problem defined as a function from a graphical model to a set of elements, most commonly, the real numbers.

A	B	C	f_1(ABC)
0	0	0	∞
0	0	1	∞
0	1	0	∞
0	1	1	2
1	0	0	∞
1	0	1	2
1	1	0	∞
1	1	1	2

A	B	D	f_2(ABD)
0	0	0	1
0	0	1	∞
0	1	0	0
0	1	1	2
1	0	0	6
1	0	1	5
1	1	0	6
1	1	1	5

B	D	E	f_3(BDE)
0	0	0	∞
0	0	1	3
0	1	0	∞
0	1	1	4
1	0	0	∞
1	0	1	3
1	1	0	∞
1	1	1	4





(a) Functions (b) Primal graph (c) Induced graph

Fig. 1. A WCSP instance with cost functions $f_1(A, B, C)$, $f_2(A, B, D)$ and $f_3(B, D, E)$.

Definition 9 *(constraint optimization problem).* A *constraint optimization problem* (*COP*) 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 $\Downarrow_S f \in \{\max_S f, \min_S f\}$ and the optimization problem is to compute $\Downarrow_{\mathbf{X}} \bigotimes_{i=1}^{r} f_i$.

The min/max ($\Downarrow$) operator is sometimes called an *elimination* operator because it removes the arguments in S from the input functions' scopes.

We next overview briefly two popular graphical models of constraint networks and belief networks, which will be the primary focus of this paper. For a detailed description of these models we refer the reader to [1,9].

A *constraint network* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ has a set of constraints $\mathbf{C} = \{C_1, \ldots, C_r\}$ as its functions. Each constraint is a pair $C_i = (S_i, R_i)$, where $S_i \subseteq \mathbf{X}$ is the scope of the relation R_i defined over S_i, denoting the allowed combinations of values. The primal graph of a constraint network is called a *constraint graph*. The Constraint Satisfaction Problem (CSP) seeks to determine if a constraint network has a solution, and if so, to find one.

An immediate extension of constraint networks are *cost networks* where the set of functions are real-valued functions, the combination and elimination operators are *summation* and *minimization*, respectively, and the primary constraint optimization task is to find a solution having minimum cost. A special class of constraint optimization problems that has gained attention 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. In a WCSP $\mathcal{W} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ each function $f_i \in \mathbf{F}$ assigns "0" (no penalty) to allowed tuples and a positive integer penalty to forbidden tuples. The optimization problem is to find a value assignment to the variables with minimum penalty. As a reasoning problem, solving a WCSP is to find $\Downarrow_{\mathbf{X}} \bigotimes_{i=1}^{r} = \min_{\mathbf{X}} \sum_{i=1}^{r} f_i$.

Example 1. Fig. 1 shows an example of a WCSP instance with bi-valued variables. The cost functions are given in Fig. 1(a). The value ∞ indicates an inconsistent tuple. Figs. 1(b) and 1(c) depict the primal and the induced graph along the ordering $d = (A, B, C, D, E, F)$, respectively. The induced graph is obtained by adding the dotted-arcs. It can be shown that the minimal cost solution is 5 and corresponds to the assignment $(A = 0, B = 1, C = 1, D = 0, E = 1)$.

A *belief network* $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P} \rangle$ is defined over a directed acyclic graph $G = (\mathbf{X}, \mathbf{E})$ and its functions $P_i \in \mathbf{P}$ denote conditional probability tables (CPTs), $P_i = P(X_i | pa_i)$, where pa_i is the set of *parent* nodes pointing to X_i in G. A belief network represents a joint probability distribution over $\mathbf{X}$, $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | pa_i)$. When formulated as a graphical model, the scopes of the functions in $\mathbf{P}$ 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 $\bigotimes_j = \prod_j$. The primal graph of a belief network is called a *moral graph*. It connects any two variables appearing in the same probability table.

A common optimization task is the *most probable explanation* (MPE) task. It calls for finding a complete assignment which agrees with the evidence e in the network, where e an instantiated subset of variables, and which has the highest probability among such assignments, namely to find an assignment $(x_1^o, \ldots, x_n^o)$ such that: $P(x_1^o, \ldots, x_n^o) = \max_{x_1, \ldots, x_n} \prod_{i=1}^{n} P(x_i, e | x_{pa_i})$. As a reasoning problem, the MPE task is to find $\Downarrow_{\mathbf{X}} \bigotimes_{i=1}^{r} f_i = \max_{\mathbf{X}} \prod_{i=1}^{n} P_i$.

Overview of previous work on WCSP and MPE. We will mention related work separately for WCSP and MPE. Clearly, both tasks are NP-hard. A number of complete and incomplete algorithms have been developed for WCSP. Stochastic Local Search (SLS) algorithms, such as GSAT [10,11], developed for Boolean Satisfiability and Constraint Satisfaction can be directly applied to WCSP [12]. SLS algorithms cannot guarantee an optimal solution, but they have been successful in practice on many classes of SAT and CSP problems. A number of search-based complete algorithms, using partial forward checking [13] for heuristic computation, have been developed [14,15]. The Branch-and-Bound algorithm proposed by [5] uses bounded mini-bucket inference to compute the guiding heuristic function. More recently, [16–18] introduced a family of depth-first Branch-and-Bound algorithms that maintain various levels of directional soft arc-consistency.

Complete algorithms for MPE used in the past either the cycle cutset technique (also called conditioning) [6], the join-tree clustering technique [19,20], or the bucket elimination scheme [21]. These methods work well only if the network is sparse enough. The algorithms based on cutset conditioning have time complexity exponential in the cutset size but require only linear space, whereas join-tree clustering and bucket elimination algorithms are both time and space exponential in the cluster size that equals the induced width (or treewidth) of the network's moral graph. Following Pearl's stochastic simulation algorithms [6], the suitability of Stochastic Local Search (SLS) algorithms for MPE was studied in the context of medical diagnosis applications [22] and more recently in [23–25]. Best-First search algorithms were proposed [26] as well as algorithms based on linear programming [27]. Some extensions are also available for the task of finding the k most-likely explanations [28,29]. We recently introduced in [5,30] a collection of depth-first Branch-and-Bound algorithms that use bounded inference, in particular the Mini-Bucket approximation [4], for computing the guiding heuristic function.

In the next section we present inference and search approaches on which we build in this paper.

4. Search and inference for combinatorial optimization

4.1. Bucket and mini-bucket elimination

Bucket Elimination (BE) is a unifying framework for inference (*e.g.*, dynamic programming) applicable to probabilistic and deterministic reasoning [21]. Given an optimization problem, namely a collection of cost functions, and given a variable ordering d, the algorithm partitions the functions into buckets, each associated with a single variable. A function is placed in the bucket of its argument that appears latest in the ordering. The algorithm has two phases. During the first, top-down phase, it processes each bucket, from last to first by a variable elimination procedure that computes a new function which is placed in a lower bucket. The variable elimination procedure computes the combination of all functions and eliminates the bucket's variable. During the second, bottom-up phase, the algorithm constructs a solution by assigning a value to each variable along the ordering, consulting the functions created during the top-down phase. The complexity of the algorithm is time and space $O(\exp(w^*))$, where w^* is the induced width of the primal graph along the ordering d [21].

BE can be viewed as message passing from leaves to root along a bucket tree [9]. Let $\{B(X_1), \ldots, B(X_n)\}$ denote a set of buckets, one for each variable, along an ordering $d = (X_1, \ldots, X_n)$. A *bucket tree* has buckets as its nodes. Bucket $B(X)$ is connected to bucket $B(Y)$ if the function generated in bucket $B(X)$ by BE is placed in $B(Y)$. The variables of $B(X)$, are those appearing in the scopes of any of its new and old functions.

Mini-Bucket Elimination (MBE) is an approximation of bucket elimination. It is designed to avoid the space and time problem of full bucket elimination 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 [4]. The algorithm outputs not only a lower bound (resp. an upper bound for maximization problems) on the cost of the optimal solution and an assignment, but also the collection of the *augmented buckets* which contain both the original as well as the intermediate functions generated by the algorithm. The complexity of the algorithm, which is parameterized by the i-bound, is time and space $O(\exp(i))$ where $i < n$ [4]. It can be viewed as solving by bucket elimination a simplified problem that is sparser [5,31]. When the i-bound is large enough (*i.e.*, $i \geqslant w^*$), the Mini-Bucket algorithm coincides with full BE on the original problem.

4.2. Branch-and-Bound search with mini-bucket heuristics

Most exact search algorithms for solving optimization problems in graphical models follow a *Branch-and-Bound* schema [32]. This algorithm performs a depth-first traversal of the search tree defined by the problem, where internal nodes represent partial assignments and leaf nodes stand for complete ones. Throughout the search, the algorithm maintains a global bound on the cost of the optimal solution, which corresponds to the cost of the best full variable instantiation found thus far. At each node, the algorithm computes a heuristic estimate of the best solution extending the current partial assignment and prunes the respective subtree if the heuristic estimate is not better than the current global bound (that is – not greater for maximization problems, not smaller for minimization problems). The algorithm requires only a limited amount of memory and can be used as an anytime scheme, namely whenever interrupted, Branch-and-Bound outputs the best solution found so far.

The effectiveness of Branch-and-Bound depends on the quality of the heuristic function. We next describe briefly a general scheme for generating heuristic estimates based on the Mini-Bucket approximation. This scheme is parameterized by the Mini-Bucket i-bound, thus allowing for a controllable trade-off between pre-processing (for heuristics generation) and search [5].

Definition 10 *(mini-bucket heuristic evaluation function [5]).* Given an ordered set of augmented buckets $\{B(X_1), \ldots, B(X_p), \ldots, B(X_n)\}$ generated by the Mini-Bucket algorithm MBE(i) along the ordering $d = (X_1, \ldots, X_p, \ldots, X_n)$, and given a partial assignment $\bar{x}^p = (x_1, \ldots, x_p)$, the heuristic evaluation function $f(\bar{x}^p) = g(\bar{x}^p) + h(\bar{x}^p)$ is defined follows:

(1) $g(\bar{x}^p) = (\sum_{f_i \in B(X_1 \ldots X_p)} f_i)(\bar{x}^p)$ is the *combination* of all the input functions that are fully instantiated along the current path, where $B(X_1 \ldots X_p)$ denotes the buckets $B(X_1)$ through $B(X_p)$ in the ordering d;

(2) The *mini-bucket heuristic* function $h(\bar{x}^p)$ is defined as the *combination* of all the intermediate functions h_j^k, $h(\bar{x}^p) = (\sum_{h_j^k \in B(X_1 \ldots X_p)} h_j^k)(\bar{x}^p)$, that satisfy the following properties:
- They are generated in buckets $B(X_{p+1})$ through $B(X_n)$,
- They reside in buckets $B(X_1)$ through $B(X_p)$.

Kask and Dechter showed [5] that for any partial assignment $\bar{x}^p = (x_1, \ldots, x_p)$ of the first p variables in the ordering, the evaluation function $f(\bar{x}^p) = g(\bar{x}^p) + h(\bar{x}^p)$ is *admissible* and *monotonic* [8].

Branch-and-Bound guided by the *Mini-Bucket heuristics* is denoted by BBMB(i). The algorithm was introduced for a static variable ordering and has a space complexity dominated by the pre-processing step which is exponential in the i-bound [5]. BBMB(i) was evaluated extensively for probabilistic and deterministic optimization tasks. The results showed conclusively that the scheme overcomes partially the memory explosion of bucket elimination allowing a gradual trade-off of space for time, and of time for accuracy when used as an anytime scheme.

Subsequently, [30,33] explored the feasibility of generating partition-based heuristics during search, rather than in a pre-processing manner. This allows dynamic variable and value ordering, a feature that can have tremendous impact on search. The dynamic generation of these heuristics is facilitated by Mini-Bucket-Tree Elimination, MBTE(i), a partition-based approximation defined over cluster-trees [33]. MBTE(i) outputs multiple (lower or upper) bounds for each possible variable and value extension at once, which is much faster than running MBE(i) n times, once for each variable.

The resulting *Branch-and-Bound with Mini-Bucket-Tree heuristics* [30,33], called BBBT(i), applies the MBTE(i) heuristic computation at each node of the search tree. Clearly, the algorithm has a higher time overhead compared with BBMB(i) for the same i-bound, which computes the mini-buckets once. It is exponential in the i-bound multiplied by the number of nodes visited, but it can prune the search space much more effectively. Experimental results on probabilistic and deterministic graphical models showed that the power of BBBT(i) is more pronounced over BBMB(i) only at relatively small i-bounds. This quality is important because small i-bounds imply restricted space.

5. AND/OR search trees for graphical models

In this section we overview the AND/OR search space for graphical models [1,8], which forms the core of our work in this paper. For simplicity and without loss of generality we consider in the remainder of the paper an optimization problem $\mathcal{P} = \langle \mathcal{R}, \min \rangle$ over a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$ for which the combination and elimination operators are *summation* and *minimization*, respectively.

As noted in Section 4, the usual way to do search in graphical models is to instantiate variables in turn, following a static/dynamic variable ordering. In the simplest case this process defines a search tree (called here OR search tree), whose state nodes represent partial variable assignments. In order to capture the independence structure of the underlying graphical model it was recently extended by AND nodes, yielding the AND/OR search space for graphical models [1]. The AND/OR search space is defined using a *pseudo tree* [2,3].

Definition 11 *(pseudo tree, extended graph).* Given an undirected graph $G = (\mathbf{V}, \mathbf{E})$, a directed rooted tree $\mathcal{T} = (\mathbf{V}, \mathbf{E}')$ defined on all its nodes is called *pseudo tree* if any arc of G which is not included in $\mathbf{E}'$ is a back-arc, namely it connects a node to an ancestor in $\mathcal{T}$. The arcs in $\mathbf{E}'$ may not all be included in $\mathbf{E}$. Given a pseudo tree $\mathcal{T}$ of G, the *extended graph* of G relative to $\mathcal{T}$ is defined as $G^{\mathcal{T}} = (\mathbf{V}, \mathbf{E} \cup \mathbf{E}')$.

We next define the notion of AND/OR search tree for a graphical model.

Definition 12 *(AND/OR search tree [1]).* Given a graphical model $\mathcal{R}$, 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 $\mathcal{T}$. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in $\mathcal{T}$. 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.

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, conditioned on the assignment above them, all of which need to be solved.

Following the general definition of a solution tree for AND/OR search spaces [8] we have here that:

Definition 13 *(solution tree).* A *solution tree* of an AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is an AND/OR subtree T such that: (i) it contains the root of $S_{\mathcal{T}}(\mathcal{R})$, s; (ii) if a non-terminal AND node $n \in S_{\mathcal{T}}(\mathcal{R})$ is in T then all of its children are in T; (iii) if

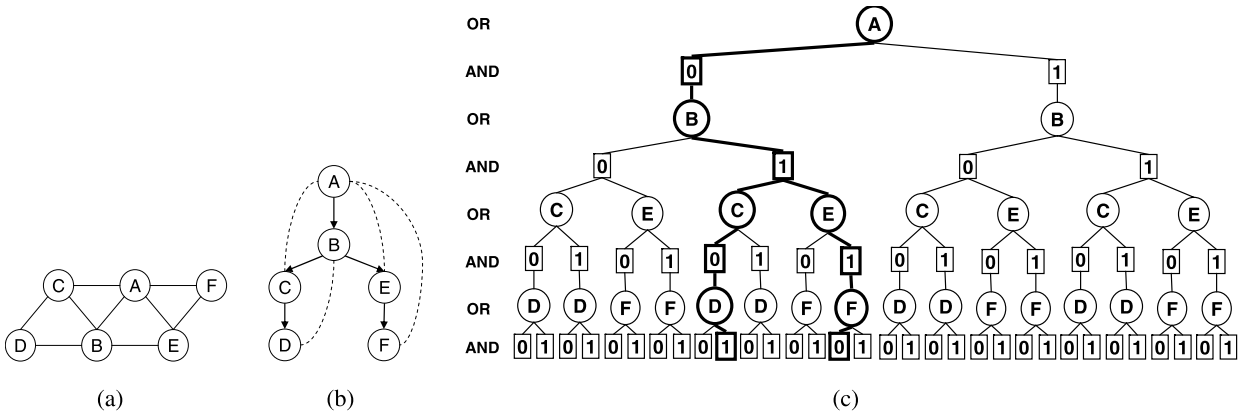


Fig. 2. AND/OR search spaces for graphical models.

a non-terminal OR node $n \in S_{\mathcal{T}}(\mathcal{R})$ is in T then exactly one of its children is in T; (iv) all its leaf (terminal) nodes are consistent.

Example 2. Fig. 2(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. Fig. 2(b) displays a pseudo tree together with the back-arcs (dotted lines). Fig. 2(c) shows the AND/OR search tree based on the pseudo tree. A solution tree is highlighted. Notice that once variables A and B are instantiated, the search space below the AND node labeled $\langle B, 0\rangle$ 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. It was shown that the AND/OR search tree represents all solutions and is therefore sound. Its size is controlled by some graph parameters, as follows:

Theorem 1 *(size of AND/OR search trees [1]). Given a graphical model $\mathcal{R}$ and a backbone pseudo tree $\mathcal{T}$, the size of its AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ 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. Moreover, if $\mathcal{R}$ has treewidth w^*, then there is a pseudo tree whose associated AND/OR search tree is $O(n \cdot k^{w^* \cdot \log n})$.*

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

Definition 14 *(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,

$$B_{\mathcal{T}}(X_i) = \left\{ f \in \mathbf{F} \mid X_i \in scope(f),\ scope(f) \subseteq path_{\mathcal{T}}(X_i) \right\}.$$

Definition 15 *(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 (i.e., sum)* of all the functions in $B_{\mathcal{T}}(X_i)$ assigned by values along π_m. Formally,

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

Definition 16 *(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 $f(T) = \sum_{e \in arcs(T)} w(e)$.

Let T_n be the subtree of T rooted at node n in T. The cost $f(T)$ 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 T_n is rooted at an OR node having an AND child m in T_n, then $f(T_n) = w(n, m) + f(T_m)$.
(3) If T_n is rooted at an AND node having OR children $m_1, \ldots, m_k$ in T_n, then $f(T_n) = \sum_{i=1}^{k} f(T_{m_i})$.

Example 3. Fig. 3 shows the primal graph of a cost network with functions $f_1(A, B)$, $f_2(A, C)$, $f_3(A, B, E)$ and $f_4(B, C, D)$, a pseudo tree that drives its weighted AND/OR search tree, and a portion of the AND/OR search tree with appropriate

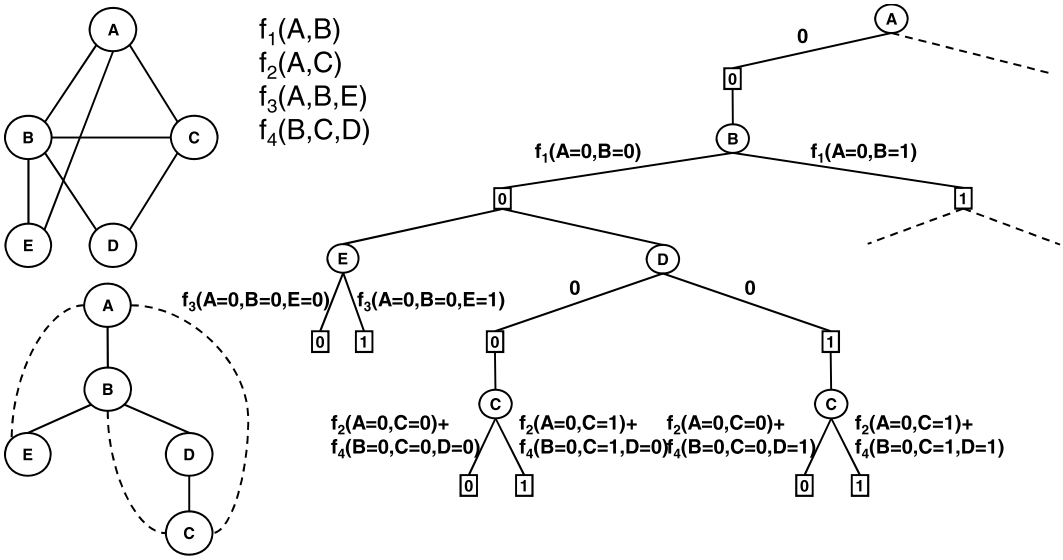


Fig. 3. Arc weights for a cost network with 5 variables and 4 cost functions.

weights on the arcs expressed symbolically. In this case the bucket of E contains the function $f_3(A, B, E)$, the bucket of C contains two functions $f_2(A, C)$ and $f_4(B, C, D)$ and the bucket of B contains the function $f_1(A, B)$. 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 $f_3(A, B, E)$, 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 [1].

Definition 17 *(node value).* 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* of a node n in the AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is the optimal cost to the subproblem below n, namely the subproblem conditioned on the assignments along the path π_n.

As was shown in [1], specializing combination and elimination to summation and minimization, respectively, we can show that 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 *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 *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. Search algorithms that traverse the AND/OR search space can compute the value of the root node yielding the answer to the problem. In [1] a generic depth-first AND/OR search algorithm, called AO, is described. It can be immediately inferred from Theorem 1 that:

Theorem 2 *(complexity [1]). 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 exists a pseudo tree $\mathcal{T}$ for which the time complexity is $O(n \cdot k^{w^* \cdot \log n})$.*

6. AND/OR Branch-and-Bound search

This section introduces the main contribution of the paper which is a Branch-and-Bound algorithm for AND/OR search spaces of graphical models. Traversing AND/OR search spaces by best-first or depth-first Branch-and-Bound algorithms were

described as early as [8,34,35]. Here we adapt these algorithms to graphical models. We will revisit next the notion of partial solution trees [8] to represent sets of solution trees which will be used in our description.

Definition 18 *(partial solution tree).* A *partial solution tree* T' of an AND/OR search tree $S_{\mathcal{T}}$ is a subtree which: (i) contains the root node s of $S_{\mathcal{T}}$; (ii) if n in T' is an OR node then it contains at most one of its AND child nodes in $S_{\mathcal{T}}$, and if n is an AND node then it contains all its OR children in $S_{\mathcal{T}}$ or it has no child nodes. 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 $S_{\mathcal{T}}$), or a *non-terminal* node (if it has children in $S_{\mathcal{T}}$).

A partial solution tree may 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 $S_{\mathcal{T}}$ is a solution tree.

Brute-force depth-first AND/OR tree search. A simple depth-first search algorithm, called AO, that traverses the AND/OR search tree was described in [1]. The algorithm maintains the partial solution being explored and computes the value of each node in a depth-first manner. It interleaves a forward expansion of the current partial solution tree with a cost revision step that updates the node values. In the expansion step, the algorithm selects a tip node of the current partial solution tree and expands it by generating its successors. It also associates each OR-to-AND arc with the appropriate weight. The node values are updated by the propagation step, in the usual way: OR nodes by minimization, while AND nodes by summation. The search terminates when the root node is evaluated and the algorithm returns both the optimal cost and an optimal solution tree. For more details see [1].

Heuristic lower bounds on partial solution trees. Search algorithms for optimization tasks often use a guiding heuristic evaluation function. We will now show how to extend the brute-force AO algorithm into a Branch-and-Bound scheme, guided by a lower bound heuristic evaluation function. For that, we first define the exact evaluation function of a partial solution tree, and will then derive the notion of a lower bound. Like in OR search, we assume a given heuristic evaluation function $h(n)$ associated with each node n in the AND/OR search tree such that $h(n) \leqslant h^*(n)$, where $h^*(n)$ is the best cost extension of the conditioned subproblem below n (*i.e.*, $h^*(n) = v(n)$). We call $h(n)$ a *node-based heuristic function*.

Definition 19 *(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: $f^*(T') = \min\{f(T) \mid T \in \mathit{extension}(T')\}$.

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 $m_1, \ldots, m_k$ 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.

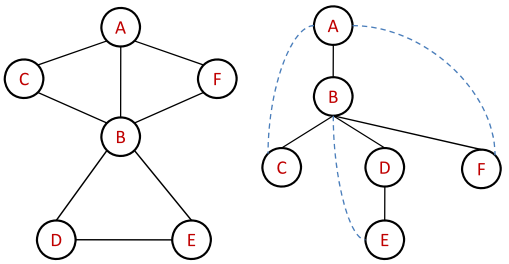
Clearly, we are interested to find the $f^*(T')$ of a partial solution tree T' rooted at the root s. If each non-terminal tip node n of T' is assigned a heuristic lower bound estimate $h(n)$ of $v(n)$, then it induces a heuristic evaluation function on the minimal cost extension of T', as follows.

Definition 20 *(heuristic evaluation function of a partial solution tree).* Given a node-based heuristic function $h(m)$ which is a lower bound on the optimal cost below any node m, namely $h(m) \leqslant v(m)$, and given a partial solution tree T'_n rooted at node n in the AND/OR search tree $S_{\mathcal{T}}$, a *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 $m_1, \ldots, m_k$ 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.

Proposition 1. *Clearly, by definition,* $f(T'_n) \leqslant f^*(T'_n)$. *If* n *is the root of the AND/OR search tree, then* $f(T') \leqslant f^*(T')$.

Example 4. Consider the cost network with bi-valued variables A, B, C, D, E and F in Fig. 4(a). The cost functions $f_1(A,B,C)$, $f_2(A,B,F)$ and $f_3(B,D,E)$ are given in Fig. 4(b). A partially explored AND/OR search tree relative to the pseudo tree from Fig. 4(a) is displayed in Fig. 4(c). The current partial solution tree T' is highlighted. It contains the nodes: A, $\langle A,0\rangle$, B, $\langle B,1\rangle$, C, $\langle C,0\rangle$, D, $\langle D,0\rangle$ and F. The nodes labeled by $\langle D,0\rangle$ and by F are non-terminal tip nodes and their corresponding heuristic estimates are $h(\langle D,0\rangle) = 4$ and $h(F) = 5$, respectively. The node labeled by $\langle C,0\rangle$ is a terminal tip node of T'.



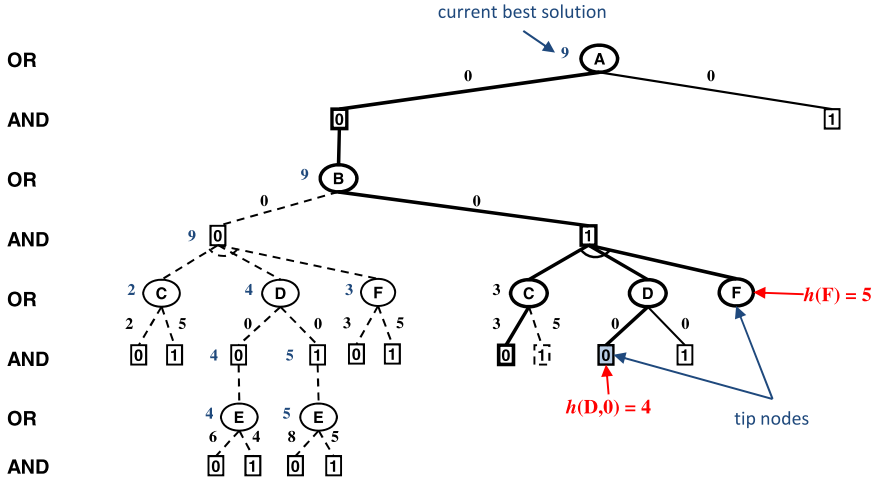
(a) Cost network and pseudo tree

A	B	C	f_1(ABC)
0	0	0	2
0	0	1	5
0	1	0	3
0	1	1	5
1	0	0	9
1	0	1	3
1	1	0	7
1	1	1	2

A	B	F	f_2(ABF)
0	0	0	3
0	0	1	5
0	1	0	1
0	1	1	4
1	0	0	6
1	0	1	5
1	1	0	6
1	1	1	5

B	D	E	f_3(BDE)
0	0	0	6
0	0	1	4
0	1	0	8
0	1	1	5
1	0	0	9
1	0	1	3
1	1	0	7
1	1	1	4

(b) Cost functions



(c) Partial solution tree

Fig. 4. Cost of a partial solution tree.

The subtree rooted at $\langle B, 0\rangle$ along the path $(A, \langle A, 0\rangle, B, \langle B, 0\rangle)$ is fully explored, yielding the current best solution cost found so far equal to 9. We assume that the search is currently at the tip node labeled by $\langle D, 0\rangle$ of T'. The heuristic evaluation function of T' is computed recursively as follows:

$$
\begin{aligned}
f(T') &= w(A,0) + f\left(T'_{\langle A,0\rangle}\right) \\
&= w(A,0) + f\left(T'_{B}\right) \\
&= w(A,0) + w(B,1) + f\left(T'_{\langle B,1\rangle}\right) \\
&= w(A,0) + w(B,1) + f\left(T'_{C}\right) + f\left(T'_{D}\right) + f\left(T'_{F}\right) \\
&= w(A,0) + w(B,1) + w(C,0) + f\left(T'_{\langle C,0\rangle}\right) + w(D,0) + f\left(T'_{\langle D,0\rangle}\right) + h(F) \\
&= w(A,0) + w(B,1) + w(C,0) + 0 + w(D,0) + h\left(\langle D,0\rangle\right) + h(F) \\
&= 0 + 0 + 3 + 0 + 0 + 4 + 5 \\
&= 12.
\end{aligned}
$$

Notice that if the pseudo tree $\mathcal{T}$ is a chain, then a partial tree T' is also a chain and corresponds to the partial assignment $\bar{x}^p = (x_1, \ldots, x_p)$. In this case, $f(T')$ is equivalent to the classical definition of the heuristic evaluation function of $\bar{x}^p$. Namely, $f(T')$ is the sum of the cost of the partial solution $\bar{x}^p$, $g(\bar{x}^p)$, and the heuristic estimate of the optimal cost extension of $\bar{x}^p$ to a complete solution.

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 $f(T') \geqslant ub(s)$, then searching below the current tip node t of T' is guaranteed not to reduce $ub(s)$ and therefore, the search space below t can be pruned.

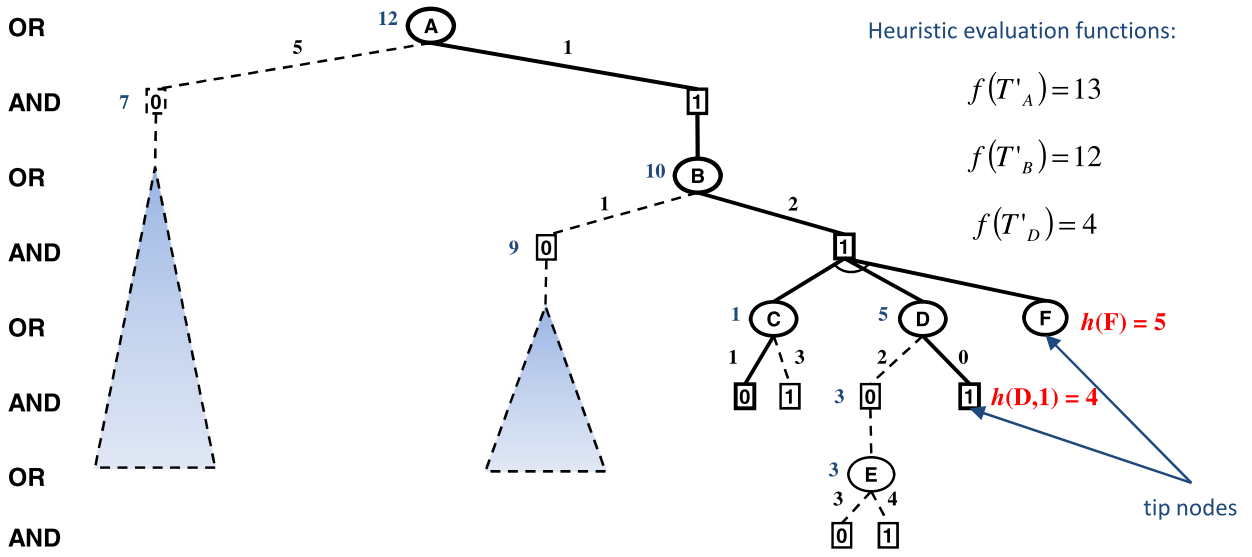


Fig. 5. Illustration of the pruning mechanism.

Example 5. For illustration, consider again the partially explored AND/OR search tree from Example 4 (see Fig. 4(c)). In this case, the current best solution found after exploring the subtree below $\langle B, 0\rangle$, which ends the path $(A, \langle A, 0\rangle, B, \langle B, 0\rangle)$, is 9. Since we computed $f(T') = 12$ for the current partial solution tree highlighted in Fig. 4(c), then exploring the subtree rooted at $\langle D, 0\rangle$, which is the current tip node, cannot yield a better solution and search can be pruned.

Up until now we considered 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 $f(T'_m) \geqslant ub(m)$, 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 tree below t. This provides a faster mechanism to discover that the search space below a node can be pruned.

Example 6. Consider the partially explored weighted AND/OR search tree in Fig. 5, relative to the pseudo tree from Fig. 4(a). The current partial solution tree T' is highlighted. It contains the nodes: A, $\langle A, 1\rangle$, B, $\langle B, 1\rangle$, C, $\langle C, 0\rangle$, D, $\langle D, 1\rangle$ and F. The nodes labeled by $\langle D, 1\rangle$ and by F are non-terminal tip nodes and their corresponding heuristic estimates are $h(\langle D, 1\rangle) = 4$ and $h(F) = 5$, respectively. The subtrees rooted at the AND nodes labeled $\langle A, 0\rangle$, $\langle B, 0\rangle$ and $\langle D, 0\rangle$ are fully evaluated, and therefore the current upper bounds of the OR nodes labeled A, B and D, along the active path, are $ub(A) = 12$, $ub(B) = 10$ and $ub(D) = 5$, respectively. Moreover, the heuristic evaluation functions of the partial solution subtrees rooted at the OR nodes along the current path can be computed recursively based on Definition 20, namely $f(T'_A) = 13$, $f(T'_B) = 12$ and $f(T'_D) = 4$, respectively. Notice that while we could prune the subtree below $\langle D, 1\rangle$ because $f(T'_A) > ub(A)$, we could discover this pruning earlier by looking at node B only, because $f(T'_B) > ub(B)$. Therefore, the partial solution tree T'_A need not be consulted in this case.

Depth-first AND/OR Branch-and-Bound tree search. The *AND/OR Branch-and-Bound* algorithm, AOBB, for searching AND/OR trees for graphical models, is described by Algorithm 1. It interleaves a forward expansion (EXPAND) of the current partial solution tree with a backward propagation step (PROPAGATE) that updates the nodes upper-bounds of values. The fringe of the search is maintained by a stack called OPEN, the current node is n, its parent p, and the current path π_n. A data structure $ST(n)$ maintains the actual best solution found in the subtree below 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.

EXPAND selects a tip node n of the current partial solution tree and expands it by generating its successors. If n is an OR node, labeled X_i, then its successors are AND nodes represented by the values x_i in variable X_i's domain (lines 6–11). Each OR-to-AND arc is associated with the appropriate weight (see Definition 15). 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 20–23). There are no weights associated with AND-to-OR arcs.

Before expanding the current AND node n, labeled $\langle X_i, x_i\rangle$, the algorithm computes the heuristic evaluation function for every partial solution subtree rooted at the OR ancestors of n along the path from the root (lines 12–18). The search below n is terminated if, for some OR ancestor m, $f(T'_m) \geqslant v(m)$, where $v(m)$ is the current best upper bound on the optimal cost below m. The recursive computation of $f(T'_m)$ based on Definition 20 is described in Algorithm 2. Notice also that for any OR node n, labeled X_i in the search tree, $v(n)$ is trivially initialized to ∞ and is updated in line 36.

Algorithm 1: AOBB: Depth-first AND/OR Branch-and-Bound 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, heuristic function $h(n)$.
Output: Minimal cost solution to $\mathcal{P}$ and an optimal solution tree.

```
1  create an OR node s labeled X_1                                   // Create and initialize the root node
2  v(s) ← ∞; ST(s) ← ∅; OPEN ← {s}
3  while OPEN ≠ ∅ do
4      n ← top(OPEN); remove n from OPEN                             // EXPAND
5      succ(n) ← ∅
6      if n is an OR node, labeled X_i then
7          foreach x_i ∈ D_i do
8              create an AND node n' labeled by ⟨X_i, x_i⟩
9              v(n') ← 0; ST(n') ← ∅
10             w(n, n') ← Σ_{f∈B_T(X_i)} f(asgn(π_n))              // Compute the OR-to-AND arc weight
11             succ(n) ← succ(n) ∪ {n'}
12     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
13         deadend ← false
14         foreach OR ancestor m of n do
15             f(T'_m) ← evalPartialSolutionTree(T'_m, h(m))
16             if f(T'_m) ⩾ v(m) then
17                 deadend ← true                                  // Pruning the subtree below the current tip node
18                 break
19         if deadend == false then
20             foreach X_j ∈ children_T(X_i) do
21                 create an OR node n' labeled by X_j
22                 v(n') ← ∞; ST(n') ← ∅
23                 succ(n) ← succ(n) ∪ {n'}
24         else
25             p ← parent(n)
26             succ(p) ← succ(p) − {n}
27     Add succ(n) on top of OPEN
28     while succ(n) == ∅ do
29         let p be the parent of n                                // PROPAGATE
30         if n is an OR node, labeled X_i then
31             if X_i == X_1 then
32                 return (v(n), ST(n))                            // Search terminates
33             v(p) ← v(p) + v(n)                                  // Update AND value
34             ST(p) ← ST(p) ∪ ST(n)                               // Update solution tree below AND node
35         if n is an AND node, labeled ⟨X_i, x_i⟩ then
36             if v(p) > (w(p, n) + v(n)) then
37                 v(p) ← w(p, n) + v(n)                           // Update OR value
38                 ST(p) ← ST(n) ∪ {(X_i, x_i)}                    // Update solution tree below OR node
39         remove n from succ(p)
40         n ← p
```

Algorithm 2: Recursive computation of the heuristic evaluation function.

function: evalPartialSolutionTree$(T'_n, h(n))$
Input: Partial solution subtree T'_n rooted at node n, heuristic function $h(n)$.
Output: Return heuristic evaluation function $f(T'_n)$.

```
1  if succ(n) == ∅ then
2      if n is an AND node then
3          return 0
4      else
5          return h(n)
6  else
7      if n is an AND node then
8          let m_1, ..., m_k be the OR children of n
9          return Σ_{i=1}^{k} evalPartialSolutionTree(T'_{m_i}, h(m_i))
10     else if n is an OR node then
11         let m be the AND child of m
12         return w(n, m) + evalPartialSolutionTree(T'_m, h(m))
```

PROPAGATE propagates node values bottom up in the search tree. It is triggered when a node has an empty set of descendants (note that as each successor is evaluated, it is removed from the set of successors in line 39). 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 32). 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 33). An AND node n propagates its value to its parent p in a similar way, by minimization (lines 35–38). Finally, the current node n is set to its parent p (line 40), because n was completely evaluated. Each node in the search tree also records the current best assignment to the variables of the subproblem below it and when the algorithm terminates it contains an optimal solution tree. Specifically, if n is an AND node, then $ST(n)$ is the union of the optimal solution trees propagated from n's OR children (line 34). If n is an OR node and n' is its AND child such that $n' = \arg\min_{m \in succ(n)}(w(n,m) + v(m))$, then $ST(n)$ is obtained from the label of n' combined with the optimal solution tree below n' (line 38). Search continues either with a *propagation* step (if conditions are met) or with an *expansion* step.

Theorem 3. *The time complexity of the depth-first AND/OR Branch-and-Bound algorithm (AOBB) is $O(n \cdot k^m)$, where m is the depth of the pseudo tree, k bounds the domain size and n is the number of variables, and it can use linear space. If the underlying primal graph has treewidth w^*, then AOBB is time $O(n \cdot k^{w^* \cdot \log n})$.*

Proof. The time complexity follows immediately from the size of the AND/OR search tree explored (see Theorems 1 and 2). Since only the current partial solution tree needs to be stored in memory, the algorithm can operate in linear space. □

AOBB can naturally accommodate minimization tasks such as solving Weighted CSPs. For maximization problems, such as the MPE task in Bayesian networks, one need only replace summation by *multiplication* (for AND nodes) and minimization by *maximization* (for OR nodes), respectively. In this case, the current values maintained by OR nodes are lower bounds on the exact values, while the heuristic evaluation function of the current partial solution tree yields an upper-bound on the optimal cost. Moreover, the node values must be initialized with 1 for AND nodes and 0 for OR nodes, respectively.

7. Lower bound heuristics for AND/OR search

The effectiveness of any Branch-and-Bound search strategy greatly depends on the quality of the heuristic evaluation function. Naturally, more accurate heuristic estimates may yield a smaller search space, possibly at a much higher computational cost. The right trade-off between the computational overhead and the pruning power exhibited during search may be hard to predict. One of the primary heuristics we used is the Mini-Bucket heuristic introduced in [5] for OR search spaces. In the following subsections we discuss its extension to AND/OR search spaces. We also extend the local consistency based lower bound developed in [16–18] to AND/OR search spaces. Both of these heuristic functions were used in our experiments.

7.1. Static mini-bucket heuristics

Consider the cost network and pseudo tree shown in Figs. 6(a) and 6(b), respectively, and consider also the variable ordering $d = (A, B, C, D, E, F, G)$ and the bucket and mini-buckets configuration in the output as displayed in Figs. 6(a) and 6(d), respectively (see Sections 4.1 and 4.2 for more details). For clarity, we display the execution of the bucket and mini-bucket elimination along the bucket tree corresponding to the given elimination ordering. The bucket tree is also a pseudo tree [1]. The functions denoted on the arcs are those messages sent from a bucket node to its parent in the tree.

Let us assume, without loss of generality, that variables A and B have been instantiated during search. Let $h^*(a,b,c)$ be the minimal cost solution of the subproblem rooted at node C in the pseudo tree, conditioned on $(A = a, B = b, C = c)$. In the AND/OR search tree, this is represented by the subproblem rooted at the AND node labeled $\langle C, c\rangle$, ending the path $\{A, \langle A, a\rangle, B, \langle B, b\rangle, C, \langle C, c\rangle\}$. By definition,

$$h^*(a,b,c) = \min_{d,e}\big(f_7(c,e) + f_6(b,e) + f_3(a,d) + f_5(c,d) + f_4(b,d)\big). \tag{2}$$

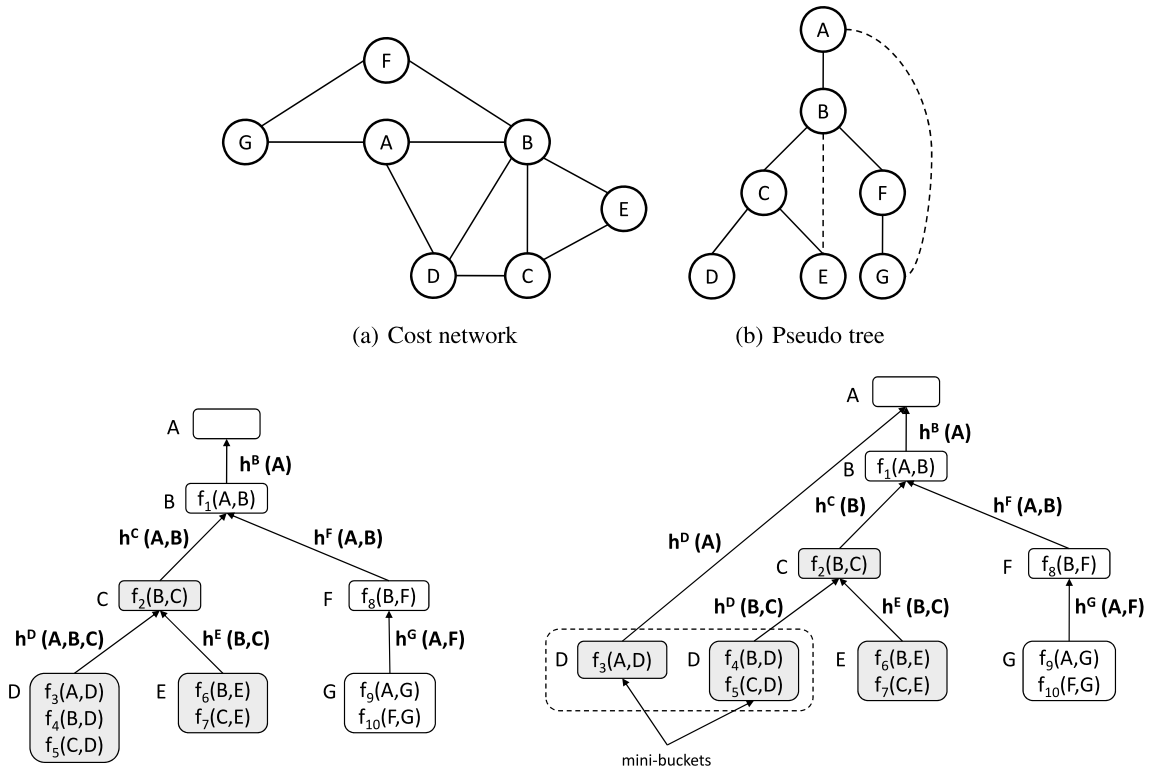
Notice that we restrict ourselves to the subproblem over variables D and E only. Therefore, we obtain:

$$\begin{aligned} h^*(a,b,c) &= \min_d\big(f_3(a,d) + f_5(c,d) + f_4(b,d) + \min_e\big(f_7(c,e) + f_6(b,e)\big)\big) \\ &= \min_d\big(f_3(a,d) + f_5(c,d) + f_4(b,d)\big) + \min_e\big(f_7(c,e) + f_6(b,e)\big) \\ &= h^D(a,b,c) + h^E(b,c) \end{aligned}$$

where

$$\begin{aligned} h^D(a,b,c) &= \min_d\big(f_3(a,d) + f_5(c,d) + f_4(b,d)\big), \\ h^E(b,c) &= \min_e\big(f_7(c,e) + f_6(b,e)\big). \end{aligned}$$

Notice that the functions $h^D(a,b,c)$ and $h^E(b,c)$ are produced by the bucket elimination algorithm shown in Fig. 6(c). Specifically, the function $h^D(a,b,c)$, generated in bucket of D by bucket elimination, is the result of a minimization operation over variable D. In practice, however, this function may be too hard to compute as it requires processing a function on four



(a) Cost network (b) Pseudo tree

(c) Bucket Elimination (d) Mini-Bucket Elimination MBE(3)

Fig. 6. Static mini-bucket heuristics for $i = 3$.

variables. It can be replaced by a partition-based approximation (*e.g.*, the minimization is split into two parts). This yields a lower bound approximation, denoted by $h(a, b, c)$, namely:

$$\begin{aligned} h^*(a,b,c) &= \min_d \big(f_3(a,d) + f_5(c,d) + f_4(b,d)\big) + h^E(b,c) \\ &\geqslant \min_d f_3(a,d) + \min_d \big(f_5(c,d) + f_4(b,d)\big) + h^E(b,c) \\ &= h^D(a) + h^D(b,c) + h^E(b,c) \\ &\triangleq h(a,b,c) \end{aligned}$$

where

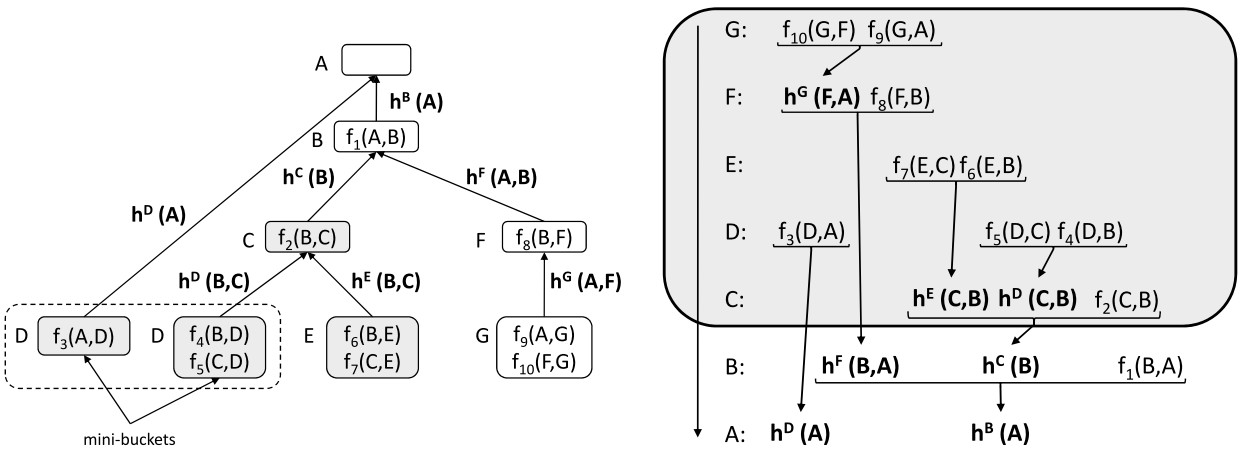
$$\begin{aligned} h^D(a) &= \min_d f_3(a,d), \\ h^D(c,b) &= \min_d \big(f_5(c,d) + f_4(b,d)\big). \end{aligned}$$

The functions $h^D(a)$ and $h^D(b,c)$ are the ones computed by the Mini-Bucket algorithm MBE(3), shown in Fig. 6(d). Therefore, the function $h(a,b,c)$ can be constructed during search from the pre-compiled mini-buckets, yielding a lower bound on the minimal cost of the respective subproblem.

For OR nodes, such as n, labeled by C, ending the path $\{A, \langle A,a\rangle, B, \langle B,b\rangle, C\}$, $h(n)$ can be obtained by minimizing over the values $c \in D_C$ the sum between the weight $w(n,m)$ and the heuristic estimate $h(m)$ below the AND child m of n. Namely, $h(n) = \min_m(w(n,m) + h(m))$.

In summary, similarly to [5], the mini-bucket heuristic associated with any node in the AND/OR search tree can be obtained from the pre-compiled mini-bucket functions. As was shown in earlier work [5], the mini-bucket heuristic function $h(n)$ associated with a node n in the search tree yields a lower bound on the minimum cost of the conditioned subproblem below n (see [5] for additional details).

Definition 21 *(static mini-bucket heuristic).* Given an ordered set of augmented buckets $\{B(X_1), \ldots, B(X_n)\}$ generated by the Mini-Bucket algorithm MBE(i) along the bucket tree $\mathcal{T}$, and given a node n in the AND/OR search tree, the *static mini-bucket heuristic* function $h(n)$ is computed as follows:



(a) AND/OR static mini-buckets (b) OR static mini-buckets

Fig. 7. AND/OR versus OR static mini-bucket heuristics for $i = 3$.

(1) If n is an AND node, labeled by $\langle X_p, x_p \rangle$, then:

$$h(n) = \sum_{h^k \in \{B(X_p) \cup B(X_p^1 \ldots X_p^q)\}} h^k.$$

Namely, it is the sum of the intermediate functions h^k that satisfy the following two properties:
- They are generated in buckets $B(X_k)$, where X_k is any descendant of X_p in the bucket tree $\mathcal{T}$,
- They reside in bucket $B(X_p)$ or the buckets $B(X_p^1 \ldots X_p^q) = \{B(X_p^1), \ldots, B(X_p^q)\}$ that correspond to the ancestors $\{X_p^1, \ldots, X_p^q\}$ of X_p in $\mathcal{T}$.

(2) If n is an OR node, labeled by X_p, then:

$$h(n) = \min_m \big(w(n,m) + h(m)\big)$$

where m is the AND child of n labeled with value x_p of X_p.

Example 7. Fig. 6(d) shows the bucket tree for the cost network in Fig. 6(a) together with the intermediate functions generated by MBE(3) along the ordering $d = (A, B, C, D, E, F, G)$. The static mini-bucket function $h(a, b, c)$ associated with the AND node labeled $\langle C, c \rangle$ ending the path $(A = a, B = b, C = c)$ in the AND/OR search tree is by definition $h(a, b, c) = h^D(a) + h^D(c, b) + h^E(b, c)$. The intermediate functions $h^D(c, b)$ and $h^E(b, c)$ are generated in buckets D and E, respectively, and reside in bucket C. The function $h^D(a)$ is also generated in bucket D, but it resides in bucket A, which is an ancestor of C in the bucket tree.

We see that the computation of the static mini-bucket heuristic of a node n in the AND/OR search tree is identical to the OR case (see Definition 10), except that it only considers the intermediate functions generated by the buckets corresponding to the current conditioned subproblem rooted at n.

Example 8. For example, consider again the cost network in Fig. 6(a). Figs. 7(a) (which repeats Fig. 6(d)) and 7(b) show the compiled bucket structure obtained by MBE(3) along the given elimination order $d = (A, B, C, D, E, F, G)$, for the AND/OR and OR spaces, respectively. The static mini-bucket heuristic function underestimating the minimal cost extension of the partial assignment $(A = a, B = b, C = c)$ in the OR search space is $h(a, b, c) = h^D(a) + h^D(c, b) + h^E(b, c) + h^F(b, a)$. Namely, it involves the extra function $h^F(b, a)$ which was generated in bucket F and resides in bucket B, as shown in Fig. 7(b). This is because, in the OR space, variables F and G are part of the subproblem rooted at C, unlike the AND/OR search space.

7.2. Dynamic mini-bucket heuristics

It is also possible to generate the mini-bucket heuristic information dynamically during search, as we show next. The idea is to compute MBE(i) conditioned on the current partial assignment.

Definition 22 *(dynamic mini-bucket heuristics).* Given a bucket tree $\mathcal{T}$ with buckets $\{B(X_1), \ldots, B(X_n)\}$, 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:

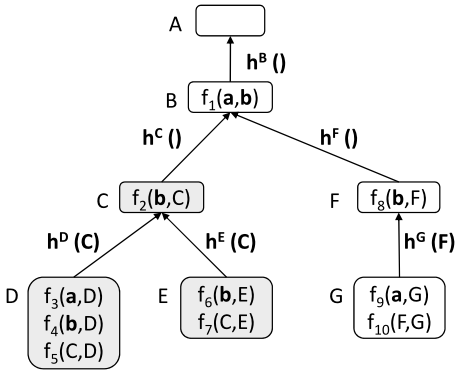


Fig. 8. Dynamic mini-bucket heuristics for $i = 3$.

(1) If n is an AND node labeled by $\langle X_p, x_p \rangle$, then:

$$h(n) = \sum_{h^k \in B(X_p)} h^k.$$

Namely, it is the sum of the intermediate functions h^k that reside in bucket $B(X_p)$ and were generated by MBE(i), conditioned on $asgn(\pi_n)$, in buckets $B(X_k)$, where X_k is any descendant of X_p in $\mathcal{T}$.

(2) If n is an OR node labeled by X_p, then:

$$h(n) = \min_m \big(w(n,m) + h(m)\big)$$

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 effort 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. More importantly, the dynamic mini-bucket heuristic can be used with dynamic variable ordering heuristics, unlike the pre-compiled one, which restricts search to be conducted in an order that respects a static pseudo tree structure. However, when using dynamic mini-bucket heuristics with a static variable ordering, rather than recomputing a new ordering and bucket structure at each node in the search tree, we use the initial variable ordering and partitioning into buckets restricted to the current subproblem.

Example 9. Fig. 8 shows the bucket tree structure corresponding to the binary cost network instance displayed in Fig. 6(a), along the elimination ordering (A, B, C, D, E, F, G). The dynamic mini-bucket heuristic estimate $h(a, b, c)$ of the AND node labeled $\langle C, c \rangle$ ending the path $\{A, \langle A, a \rangle, B, \langle B, b \rangle, C, \langle C, c \rangle\}$ is computed by MBE(3) on the subproblem represented by the buckets D and E, conditioned on the partial assignment $(A = a, B = b, C = c)$. Namely, MBE(3) processes buckets D and E by eliminating the respective variables, and generates two new functions: $h^D(c)$ and $h^E(c)$, as illustrated in Fig. 8. These new functions are in fact constants since variables A, B and C are assigned in the scopes of the input functions that constitute the conditioned subproblem: $f_3(a, D)$, $f_4(b, D)$, $f_5(c, D)$, $f_6(b, E)$ and $f_7(c, E)$, respectively. Therefore $h(a, b, c) = h^D(c) + h^E(c)$ and it equals the exact $h^*(a, b, c)$ in this case.

7.3. Local consistency based heuristics for AND/OR search

Another class of heuristic lower bounds developed for guiding Branch-and-Bound search for solving binary Weighted CSPs is based on exploiting local consistency algorithms for cost functions. In the next section we overview the basic principles behind these types of heuristics and discuss their extension to AND/OR trees.

7.3.1. Review of local consistency for weighted CSPs

As in the classical CSP, enforcing soft local consistency on the initial problem provides in polynomial time an *equivalent* problem defining the same cost distribution on complete assignments, with possible smaller domains [16–18].

Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a binary WCSP, where $\mathbf{X} = \{X_1, \ldots, X_n\}$ and $\mathbf{D} = \{D_1, \ldots, D_n\}$ are the variables and their corresponding domains. $\mathbf{C}$ is the set of binary and unary soft constraints. A binary soft constraint $C_{ij}(X_i, X_j) \in \mathbf{C}$ (or C_{ij} in short) is $C_{ij}(X_i, X_j) : D_i \times D_j \to \mathbb{N}$. A unary soft constraint $C_i(X_i) \in \mathbf{C}$ (or C_i in short) is $C_i(X_i) : D_i \to \mathbb{N}$. We assume the existence of a unary constraint C_i for every variable X_i, and a zero-arity constraint, denoted by $C_\emptyset$. If no such constraints are defined,

we can always define dummy ones, as $C_i(x_i) = 0$, $\forall x_i \in D_i$ or $C_\emptyset = 0$. We denote by $\top$, the maximum allowed cost (*e.g.*, $\top = \infty$). The cost of a tuple $x = (x_1, \ldots, x_n)$, denoted by $cost(x)$, is defined by:

$$cost(x) = \sum_{C_{ij} \in \mathbf{C}} C_{ij}\big(x[i, j]\big) + \sum_{C_i \in \mathbf{C}} C_i\big(x[i]\big) + C_\emptyset.$$

For completeness, we define next some local consistencies in WCSP, in particular *node*, *arc* and *directional arc consistency*, as in [16,17]. We assume that the set of variables $\mathbf{X}$ is totally ordered. We note that there are several stronger local consistencies which were defined in recent years, such as *full directional arc consistency* (FDAC) [16,17] or *existential directional arc consistency* (EDAC) [18].

Definition 23 *(soft node consistency [16,17]).* Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a binary WCSP. (X_i, x_i) is star node consistent (NC*) if $C_\emptyset + C_i(x_i) < \top$. Variable X_i is NC* if: (i) all its values are NC* and (ii) there exists a value $x_i \in D_i$ such that $C_i(x_i) = 0$. Value x_i is called a *support* for variable X_i. $\mathcal{R}$ is NC* if every variable is NC*.

Definition 24 *(soft arc consistency [16,17]).* Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a binary WCSP. (X_i, x_i) is arc consistent (AC) with respect to constraint C_{ij} if there exists a value $x_j \in D_j$ such that $C_{ij}(x_i, x_j) = 0$. Value x_j is called a *support* for the value x_i. Variable X_i is AC if all its values are AC wrt. every binary constraint affecting X_i. $\mathcal{R}$ is AC* if every variable is AC and NC*.

Definition 25 *(soft directional arc consistency [16,17]).* Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a binary WCSP. (X_i, x_i) is directional arc consistent (DAC) with respect to constraint C_{ij}, $i < j$, if there exists a value $x_j \in D_j$ such that $C_{ij}(x_i, x_j) + C_j(x_j) = 0$. Value x_j is called a *full support* of x_j. Variable X_i is DAC is all its values are DAC wrt. every C_{ij}, $i < j$. $\mathcal{R}$ is DAC* if every variable is DAC and NC*.

For our purpose, we point out that enforcing such local consistencies is done by the repeated application of atomic operations called *arc equivalence preserving transformations* [36]. This process may increase the value of $C_\emptyset$ and the unary costs $C_i(x_i)$ associated with domain values. The zero-arity cost function $C_\emptyset$ defines a *strong lower bound* which can be exploited by Branch-and-Bound algorithms while the updated $C_i(x_i)$ can inform variable and value orderings [16–18].

If we consider two cost functions C_{ij}, defined over variables X_i and X_j, and C_i, defined over variable X_i, a value $x_i \in D_i$ and a cost α, we can add α to $C_i(x_i)$ and subtract α from every $C_{ij}(x_i, x_j)$ for all $x_j \in D_j$. Simple arithmetic shows that the global cost distribution is unchanged while costs may have moved from the binary to the unary level (if $\alpha > 0$, this is called a *projection*) or from the unary to the binary level (if $\alpha < 0$, this is called an *extension*). In these operations, any cost above $\top$, the maximum allowed cost, can be considered as infinite and is thus unaffected by subtraction. If no negative cost appears and if all costs above $\top$ are set to $\top$, the remaining problem is always a valid and equivalent WCSP. The same mechanism, at the unary level, can be used to move costs from the C_i to $C_\emptyset$. Finally, any value a such that $C_i(x_i) + C_\emptyset$ is equal to $\top$ can be deleted. For a detailed description of these operations, we refer the reader to [16–18].

7.3.2. Extension of local consistency to AND/OR search spaces

As mentioned earlier, the zero-arity constraint $C_\emptyset$ which is obtained by enforcing local consistency, can be used as a heuristic function to guide Branch-and-Bound search. The extension of this heuristic to AND/OR search spaces is fairly straightforward and is similar to the extension of the mini-bucket heuristics from OR to AND/OR spaces. Consider P_n, the subproblem rooted at the AND node n, labeled $\langle X_i, x_i \rangle$, in the AND/OR search tree defined by a pseudo tree $\mathcal{T}$. The heuristic function $h(n)$ underestimating $v(n)$ is the zero-arity cost function $C_\emptyset^n$ resulted from enforcing soft arc consistency over P_n only, subject to the current partial instantiation of the variables along the path from the root of the search tree. Note that P_n is defined by the variables and cost functions corresponding to the subtree rooted at X_i in $\mathcal{T}$. If n is an OR node labeled X_i then $h(n)$ is computed in the usual way, namely $h(n) = \min_m(w(n, m) + h(m))$, where m is the AND child of n, labeled with value x_i of X_i. Notice that in this case the weights associated with the OR-to-AND arcs are computed now relative to the equivalent subproblem resulted from enforcing arc consistency.

There is a strong relation between directional arc consistency and mini-buckets. It was shown in [16] that given a WCSP with $\top = \infty$, and a variable ordering, the lower bound induced by mini-buckets involving at most 2 variables is the same as the lower bound induced by $C_\emptyset$ after the problem is made directional arc consistent. However, the mini-bucket computation provides only a lower bound while DAC enforcing provides both a lower bound and a directional arc consistent equivalent problem. All the work done to compute the lower bound is captured in this problem which offers the opportunity to perform incremental updates of the lower bound.

8. Dynamic variable orderings

The depth-first AND/OR Branch-and-Bound algorithm introduced in Section 6 assumed a static variable ordering determined by the underlying pseudo tree of the primal graph. In classical CSPs, dynamic variable ordering is known to have a significant impact on the size of the search space explored [37]. Well known variable ordering heuristics, such as *min-domain* [38], *min-dom/ddeg* [39], *brelaz* [40] and *min-dom/wdeg* [41,42] were shown to improve dramatically the performance

of systematic search algorithms. In this section we discuss some strategies that allow dynamic variable orderings in AND/OR search.

We distinguish two classes of variable ordering heuristics:

(1) *Graph*-based heuristics (*e.g.*, pseudo tree) that try to maximize problem decomposition, and
(2) *Semantic*-based heuristics (*e.g.*, min-domain) that aim at shrinking the search space, based on context and current value assignment.

These two approaches are orthogonal, namely we can use one as the primary guide and break ties based on the other. We present three schemes of combining these heuristics. For simplicity and without loss of generality we consider the *min-domain* as our semantic variable ordering heuristic. It selects the next variable to instantiate as the one having the smallest current domain among the uninstantiated (future) variables. Clearly, it can be replaced by any other heuristic.

8.1. Partial variable ordering (PVO)

The first approach, called *AND/OR Branch-and-Bound with Partial Variable Ordering* and denoted by AOBB+PVO uses the static graph-based decomposition given by a pseudo tree with a dynamic semantic ordering heuristic applied over chain portions of the pseudo tree. It is an adaptation of the ordering heuristics developed in [43,44] which were used for solving large-scale SAT problem instances.

Consider the pseudo tree from Fig. 2(a) inducing the following variable groups (or chains): $\{A, B\}$, $\{C, D\}$ and $\{E, F\}$, respectively. This implies that variables $\{A, B\}$ should be considered before $\{C, D\}$ and $\{E, F\}$. The variables in each group can be dynamically ordered based on a second, independent heuristic. Notice that once variables $\{A, B\}$ are instantiated, the problem decomposes into independent components that can be solved separately.

AOBB+PVO can be derived from Algorithm 1 with some simple modifications. As usual, the algorithm traverses an AND/OR search tree in a depth-first manner, guided by a pre-computed pseudo tree $\mathcal{T}$. When the current AND node n, labeled $\langle X_i, x_i \rangle$ is expanded in the forward step (line 9), the algorithm generates its OR successor, labeled by X_j, based on the semantic variable ordering heuristic (line 12). Specifically, the OR node m, labeled X_j corresponds to the uninstantiated variable with the smallest current domain in the current pseudo tree chain. If there are no uninstantiated variables left in the current chain, namely variable X_i was instantiated last, then the OR successors of n are labeled by the variables with the smallest domain from the variable chains rooted by X_i in $\mathcal{T}$.

8.2. Full Dynamic Variable Ordering (DVO)

A second, orthogonal approach to partial variable orderings, called *AND/OR Branch-and-Bound with Full Dynamic Variable Ordering* and denoted by DVO+AOBB, gives priority to the dynamic semantic variable ordering heuristic and applies static problem decomposition as a secondary principle during search. This idea was also explored in [45] for model counting, and more recently in [46] for weighted model counting.

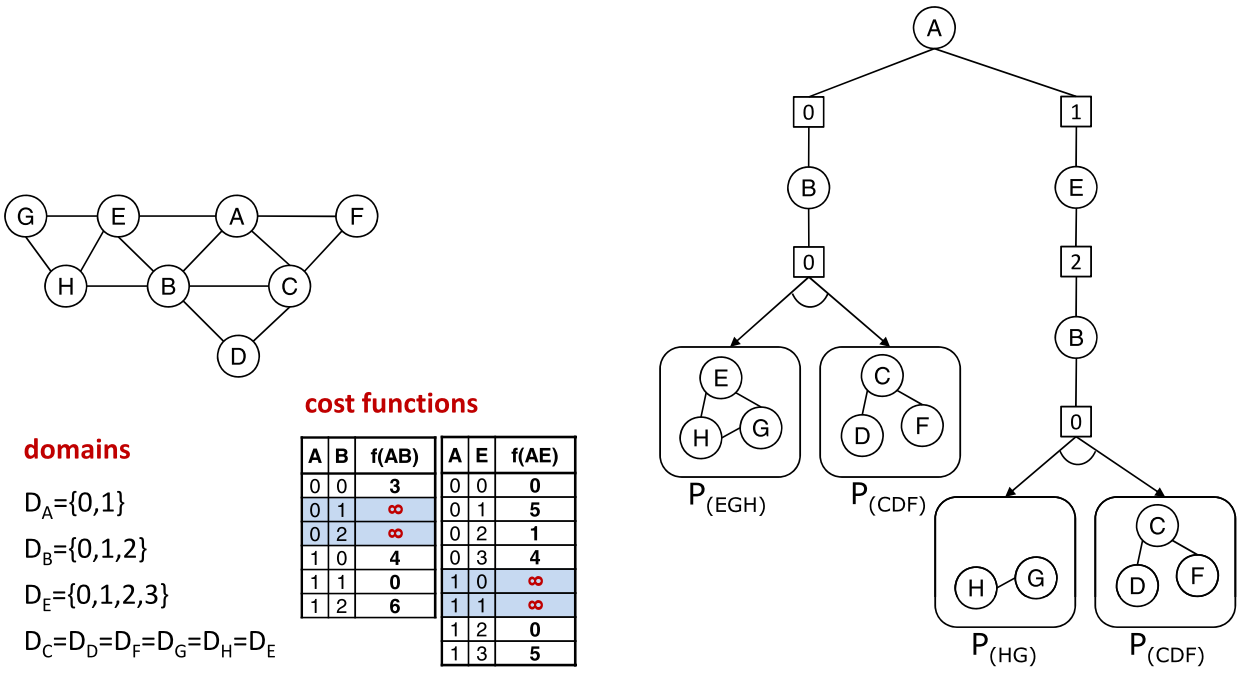
For illustration, consider the cost network with 8 variables $\{A, B, C, D, E, F, G, H\}$, 13 binary cost functions, and the domains given in Fig. 9(a), as follows: $D_A = \{0, 1\}$, $D_B = \{0, 1, 2\}$, and $D_C = D_D = D_E = D_F = D_G = D_H = \{0, 1, 2, 3\}$, respectively. Each of the cost functions $f(A, B)$ and $f(A, E)$ assigns an ∞ cost to two of their corresponding tuples, whereas the remaining 11 functions do not contain such tuples.

During search, variables are instantiated in min-domain order. However, after each variable assignment we test for problem decomposition and solve the remaining subproblems independently. Fig. 9(b) shows the partial AND/OR search tree obtained after several variable instantiations based on the min-degree ordering heuristic. Notice that, depending on the order in which the variables are instantiated, the primal graph may decompose into independent components *higher* or *deeper* in the search tree. For instance, after instantiating A to 0, the values $\{1, 2\}$ can be removed from the domain of B, because the corresponding tuples have cost ∞ in the cost function $f(A, B)$ (see Fig. 9(a)). Therefore, B is the next variable to be instantiated, at which point the problem decomposes into independent components, as shown in Fig. 9(b). Similarly, when A is instantiated to 1, values $\{0, 1\}$ can also be removed from the domain of E, because of the cost function $f(A, E)$. Then, variable E, having 2 values left in its domain, is selected next in the min-domain order, followed by B with domain size 3.

DVO+AOBB can be expressed by modifying Algorithm 1 as follows. It instantiates the variables dynamically using the min-domain ordering heuristic while maintaining the current graph structure. Specifically, after the current AND node n, labeled $\langle X_i, x_i \rangle$, is expanded, DVO+AOBB tentatively removes from the primal graph all nodes corresponding to the instantiated variables together with their incoming arcs. If disconnected components are detected, their corresponding subproblems are then solved separately and the results combined in an AND/OR manner. In this case a variable selection may yield a significant impact on tightening the search space, yet, it may not yield a good decomposition for the remaining problem.

8.3. Dynamic separator ordering (DSO)

The third approach, *AND/OR Branch-and-Bound with Dynamic Separator Ordering* (AOBB+DSO), exploits constraint propagation which can be used for dynamic graph-based decomposition with a dynamic semantic variable ordering, giving priority



(a) Cost network (b) AND/OR search tree

Fig. 9. Full dynamic variable ordering for AND/OR Branch-and-Bound search.

to the first. At each AND node we apply a lookahead procedure hoping to detect singleton variables (*i.e.*, with only one feasible value left in their domains). When the value of a variable is known, it can be removed from the corresponding subproblem, yielding a stronger decomposition of the simplified primal graph.

AOBB+DSO defined on top of Algorithm 1 creates and maintains a separator S of the current primal graph. A graph separator can be computed using the hypergraph partitioning method presented in [44]. The next variable is chosen dynamically from S by the min-domain ordering heuristic until S is fully instantiated and the current problem decomposes into several independent subproblems, which are then solved separately. The separator of each component is created from a simplified subgraph resulted from previous constraint propagation steps and it may differ for different value assignments. Clearly, if no singleton variables are discovered by the lookahead steps this approach is computationally identical to AOBB+PVO, although it may have a higher overhead due to the dynamic generation of the separators.

9. Experimental results

We have conducted a number of experiments on two common optimization problem classes in graphical models: finding the Most Probable Explanation in Bayesian networks and solving Weighted CSPs. We implemented[2] our algorithms in C++ and carried out all experiments on a 1.8 GHz dual-core Athlon64 with 2 GB of RAM running Ubuntu Linux 7.04.

9.1. Overview and methodology

9.1.1. MPE task for Bayesian networks

We tested the performance of the AND/OR Branch-and-Bound algorithms on the following types of problems[3]: random Bayesian networks, random coding networks, grid networks, Bayesian networks derived from the ISCAS'89 digital circuits benchmark, genetic linkage analysis networks, networks from the Bayesian Networks Repository, and Bayesian networks used in the UAI'06 Inference Evaluation contest. We report here in detail the results obtained for grid networks and genetic linkage analysis networks only, but we summarize the results over the entire set of benchmarks, and refer the reader to [47,48] for further details.

We evaluated the two classes of depth-first AND/OR Branch-and-Bound search algorithms, guided by the static and dynamic mini-bucket heuristics, denoted by AOBB+SMB(i) and AOBB+DMB(i), respectively. We compare these algorithms against traditional depth-first OR Branch-and-Bound algorithms with static and dynamic mini-bucket heuristics introduced

[2] The code is available online at: http://graphmod.ics.uci.edu/group/Software.
[3] Available online at: http://graphmod.ics.uci.edu/group/Repository.

in [5,30], denoted by BB+SMB(i) and BB+DMB(i), respectively, which were among the best-performing complete search algorithms for this domain at the time. The parameter i represents the mini-bucket i-bound and controls the accuracy of the heuristic. The pseudo trees that guide AND/OR search algorithms were generated using the min-fill and hypergraph partitioning heuristics, described later in this section. We also consider an extension of the AND/OR Branch-and-Bound that exploits the determinism present in the Bayesian network by constraint propagation.

Since the pre-compiled mini-bucket heuristics require a static variable ordering, the corresponding OR and AND/OR search algorithms used the static variable ordering derived from a depth-first traversal of the guiding pseudo tree as well. When we applied dynamic variable orderings with dynamic mini-bucket heuristics we observed that the computational overhead was prohibitively large compared with the static variable ordering setup. We therefore do not report these results. Note however that the AOBB+SMB(i) and AOBB+DMB(i) algorithms apply 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 27 of Algorithm 1, if the current node n is an AND node, then the independent subproblems rooted by its OR children can be solved in decreasing order of their corresponding heuristic estimates (yielding local 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 (thus yielding a value ordering).

We compared our algorithms with the SamIam version 2.3.2 software package.[4] SamIam contains an implementation of Recursive Conditioning [49] which can also be viewed as an AND/OR search algorithm. The algorithm uses a context-based caching mechanism that records the optimal solution of the solved subproblems and retrieves the saved values when the same subproblems are encountered later during search. This version of recursive conditioning traverses a context minimal AND/OR search graph [1], rather than a tree, and 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 for its storing. Our search regime however does not consume any additional space.

9.1.2. Weighted CSPs

We evaluated the performance of the AND/OR Branch-and-Bound algorithms on: random binary WCSPs, scheduling problems from the SPOT5 benchmark, networks derived from the ISCAS'89 digital circuits benchmark, radio link frequency assignment problems and instances of the Mastermind game. We report here detailed results for Mastermind game instances and SPOT5 problem instances only. We also provide a summary of the results obtained on the other types of problems, and refer the reader to [47,48] for the full results.

In addition to the mini-bucket heuristics, we also consider a heuristic evaluation function that is computed by maintaining Existential Directional Arc-Consistency (EDAC) [18]. AOBB with this heuristic is called AOEDAC. We also used the extension of AOEDAC that incorporates dynamic variable orderings heuristics described earlier yielding: AOEDAC+PVO (partial variable ordering – Section 8.1), DVO+AOEDAC (full dynamic variable ordering – Section 8.2) and AOEDAC+DSO (dynamic separator ordering – Section 8.3). For comparison, we report results obtained with our implementation of the classic OR Branch-and-Bound with EDAC, denoted here by BBEDAC.

For comparison, we ran toolbar,[5] which contains an OR Branch-and-Bound implementation that maintains EDAC during search and uses dynamic variable orderings. toolbar was introduced in [18] and is currently one of the best performing solvers for binary WCSPs.

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

Measures of performance In all our experiments we report the CPU time in seconds and the number of nodes visited for solving the problems. We also specify the problems' parameters such as the number of variables (n), number of evidence variables (e), maximum domain size (k), number of functions (c), maximum arity of the functions (r), the depth of the pseudo tree (h) and the induced width of the graph (w^*). 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 best performance points are highlighted. In each table, "–" denotes that the respective algorithm exceeded the time limit. Similarly, "out" stands for exceeding the 2 GB memory limit.

9.2. Finding good pseudo trees

The performance of the AND/OR Branch-and-Bound search algorithms is influenced by the quality of the guiding pseudo tree. Finding the minimal depth/induced width pseudo tree is a hard problem [2,3,50]. We describe next two heuristics for generating pseudo trees with relatively small depths/induced widths which we used in our experiments.

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

9.2.1. Min-Fill heuristic

Min-Fill [51] is one of the best and most widely used heuristics for creating small induced width elimination orders. An ordering is generated by placing the variable with the smallest *fill set* (*i.e.*, number of induced edges that need be added to fully connect the neighbors of a node) at the end of the ordering, connecting all of its neighbors and then removing the variable from the graph. The process continues until all variables have been eliminated.

Once an elimination order is given, the pseudo tree can be extracted as a depth-first traversal of the min-fill induced graph, starting with the variable that initiated the ordering, always preferring as successor of a node the earliest adjacent node in the induced graph. An ordering uniquely determines a pseudo tree. This approach was first used by [3].

To improve orderings, we can run the min-fill ordering several times by randomizing the tie breaking rule. In our experiments, we ran the min-fill heuristic just once and broke the ties lexicographically.

9.2.2. Hypergraph decomposition heuristic

An alternative heuristic for generating a low height balanced pseudo tree is based on the recursive decomposition of the dual hypergraph associated with the graphical model.

Definition 26 *(dual hypergraph).* The *dual hypergraph* of a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, is a pair $\mathcal{H}(\mathcal{R}) = (\mathbf{V}, \mathbf{E})$, where each function in $\mathbf{F}$ is a vertex $v_i \in \mathbf{V}$ and each variable in $\mathbf{X}$ is an edge $e_j \in \mathbf{E}$ connecting all the functions (vertices) in which it appears.

Definition 27 *(hypergraph separators).* Given a dual hypergraph $\mathcal{H} = (\mathbf{V}, \mathbf{E})$ of a graphical model, a *hypergraph separator decomposition* is a triple $\langle \mathcal{H}, S, \alpha \rangle$ where: (i) $\mathcal{S} \subset \mathbf{E}$, and the removal of $\mathcal{S}$ separates $\mathcal{H}$ into k disconnected components (subgraphs); and (ii) α is a relation over the size of the disjoint subgraphs (i.e., balance factor).

It is well known that the problem of finding the minimal size hypergraph separator is hard. However heuristic approaches were developed over the years. A good approach is packaged in `hMeTiS`.[6]

We will use this software as a basis for our pseudo tree generation. Following [49], generating a pseudo tree $\mathcal{T}$ for $\mathcal{R}$ using `hMeTiS` is fairly straightforward. The vertices of the hypergraph are partitioned into two balanced (roughly equal-sized) parts, denoted by $\mathcal{H}_{\text{left}}$ and $\mathcal{H}_{\text{right}}$ respectively, while minimizing the number of hyperedges across. A small number of crossing edges translates into a small number of variables shared between the two sets of functions. $\mathcal{H}_{\text{left}}$ and $\mathcal{H}_{\text{right}}$ are then each recursively partitioned in the same fashion, until they contain a single vertex. The result of this process is a tree of hypergraph separators which can be shown to also be a pseudo tree of the original model where each separator corresponds to a subset of variables chained together.

Since the hypergraph partitioning heuristic uses a non-deterministic algorithm (*i.e.*, `hMeTiS`), the depth and induced width of the resulting pseudo tree may vary significantly from one run to the next. In our experiments we picked the pseudo tree with the smallest depth out of 10 independent runs.

From the experiments presented in the remainder of this section, we observed that the min-fill heuristic generates lower induced width pseudo trees, while the hypergraph heuristic produces much smaller depth pseudo trees. Therefore, perhaps the hypergraph based pseudo trees appear to be favorable for tree search algorithms guided by heuristics that are not sensitive to the treewidth (*e.g.*, local consistency based heuristics).

9.3. Results for empirical evaluation of Bayesian networks

In this section we show, using grid networks and linkage analysis networks, the impact of (1) AND/OR versus OR search, (2) static versus dynamic mini-bucket heuristics as well as (3) the impact of exploiting determinism.

9.3.1. Grid networks

In random grid networks, the nodes are arranged in an $N \times N$ square and each CPT is generated uniformly at random. We experimented with problem instances having bi-valued variables that were initially developed in [52] 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 (having only 0 and 1 probability entries).

Table 1 displays the results for experiments with 8 grids of increasing difficulty, using min-fill based pseudo trees. The columns are indexed by the mini-bucket i-bound. The table is organized into two horizontal blocks, each corresponding to a different range of i-bound values. For each test instance we ran a single MPE query with e evidence variables picked randomly. We observe that `AOBB+SMB(`i`)` is better than `BB+SMB(`i`)` at relatively small i-bounds (*i.e.*, $i \in \{8, 10, 12\}$) when the heuristic is weak. This demonstrates the benefit of AND/OR over classical OR search when the heuristic estimates are relatively weak and the algorithms rely primarily on search rather than on pruning via the heuristic evaluation function. As the i-bound increases and the heuristic estimates become strong enough to cut the search space substantially, the difference between AND/OR and OR Branch-and-Bound decreases, especially on the first 3 easier instances. On the harder instances,

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

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

minfill pseudo tree													
grid (w^*, h) (n, e)	Samlam	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=8$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=10$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=12$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=14$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=16$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=18$	
	time	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-10-1	0.13	0.01		0.02		0.04		0.07		0.07		0.08	
(13, 39)		0.12	3348	**0.04**	424	0.05	153	0.07	153	0.08	153	0.09	153
(100, 0)		0.17	8080	0.06	2052	0.05	101	0.07	101	0.08	101	0.08	101
		0.87	543	0.57	250	0.48	153	0.54	153	0.54	153	0.54	153
		0.34	344	0.33	241	0.32	101	0.39	101	0.39	101	0.39	101
90-14-1	11.97	0.02		0.04		0.11		0.22		0.72		2.71	
(22, 66)		75.71	1,235,366	71.98	1,320,090	1.07	18,852	0.54	5035	0.90	2826	2.78	1075
(196, 0)		4.27	130,619	3.44	100,696	0.61	17,479	**0.32**	3321	0.81	2938	2.80	3386
		149.44	16,415	52.34	2894	12.46	537	13.71	211	19.22	199	38.05	198
		65.74	31,476	33.57	4137	7.50	397	12.00	211	17.65	199	36.87	198
90-16-1	147.19	0.03		0.05		0.14		0.46		1.01		4.36	
(24, 82)		–	–	–	–	23.74	347,479	1.85	18,855	**1.44**	6098	4.53	1894
(256, 0)		362.66	10,104,350	91.03	2,600,690	7.53	193,440	1.89	39,825	1.78	23,421	4.55	5842
		771.73	43,366	553.08	13,363	172.14	2011	166.61	1169	65.15	414	181.71	414
		1114.19	462,180	410.87	47,121	109.11	3227	80.57	719	40.68	260	109.76	260

grid (w^*, h) (n, e)	Samlam	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=10$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=12$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=14$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=16$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=18$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i=20$	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-24-1	–	0.14		0.33		0.89		2.69		7.61		31.26	
(33, 111)		–	–	–	–	–	–	–	–	–	–	–	–
(576, 20)		–	–	–	–	1500.66	24,117,151	921.96	18,238,983	**93.73**	1,413,764	111.46	1,308,009
		–	–	–	–	–	–	–	–	–	–	–	–
		–	–	–	–	–	–	1367.38	2739	1979.42	1228	2637.71	598
90-26-1	–	0.16		0.37		1.02		3.39		11.74		36.16	
(36, 113)		–	–	–	–	–	–	324.30	2,234,558	–	–	70.53	327,859
(676, 40)		1533.11	17,899,574	206.93	2,903,489	242.37	3,205,257	**7.43**	59,055	21.48	165,182	36.49	5777
		–	–	–	–	–	–	–	–	–	–	–	–
		1852.27	177,661	–	–	–	–	1514.18	2545	2889.49	1191	–	–
90-30-1	–	0.25		0.53		1.35		4.36		13.34		50.53	
(43, 150)		–	–	–	–	–	–	–	–	–	–	–	–
(900, 60)		–	–	742.51	9,445,224	239.08	3,324,942	215.56	3,039,966	101.10	1,358,569	**87.68**	485,300
		–	–	–	–	–	–	–	–	–	–	–	–
		–	–	–	–	–	–	–	–	–	–	–	–
90-34-1	–	0.33		0.66		1.60		5.35		18.42		62.17	
(45, 153)		–	–	–	–	–	–	–	–	–	–	–	–
(1154, 80)		–	–	–	–	–	–	–	–	–	–	**257.14**	1,549,829
		–	–	–	–	–	–	–	–	–	–	–	–
		–	–	–	–	–	–	–	–	–	–	–	–
90-38-1	–	0.41		0.82		2.16		6.43		20.46		72.10	
(47, 163)		–	–	–	–	–	–	–	–	–	–	–	–
(1444, 120)		–	–	936.65	6,835,745	1858.99	12,321,175	341.05	2,850,393	252.67	2,079,146	**199.44**	1,038,065
		–	–	–	–	–	–	–	–	–	–	–	–
		–	–	–	–	–	–	–	–	–	–	–	–

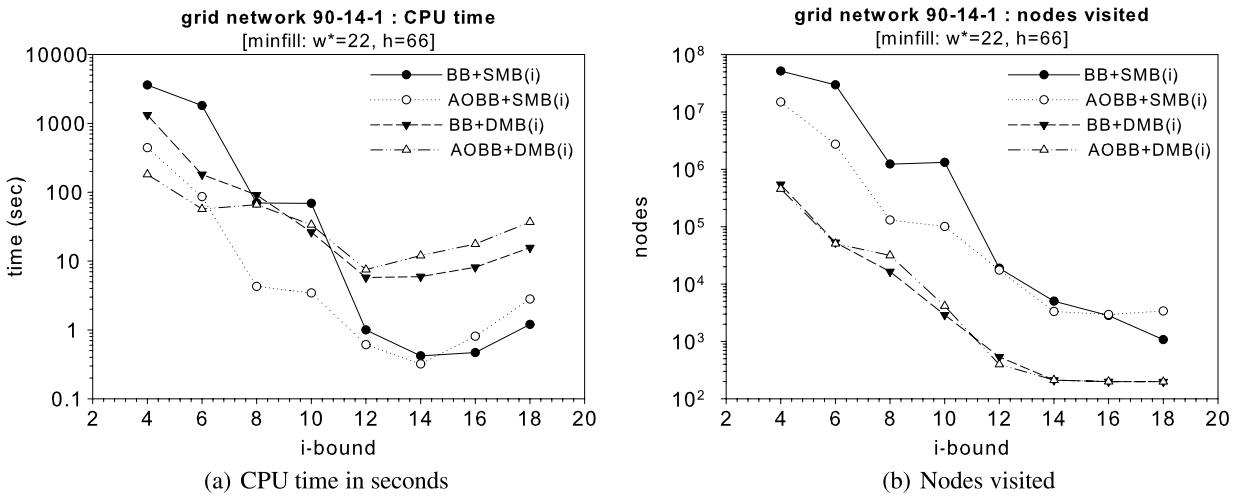


Fig. 10. Comparison of the impact of static and dynamic mini-bucket heuristics. Shown are the CPU time in seconds (a) and the number of nodes visited (b) on the **`90-14-1` grid network** from Table 1.

however, `AOBB+SMB(`i`)` with the largest reported i-bounds offers the best performance. For example, on the `90-30-1` grid, `AOBB+SMB(20)` found the MPE in about 87 seconds, whereas `BB+SMB(20)` exceeded the 1 hour time limit.

When focusing on dynamic mini-bucket heuristics, we see that `AOBB+DMB(`i`)` is better than `BB+DMB(`i`)` at relatively small i-bounds, but the difference is not that prominent as in the static case. This is probably because these heuristics are far more accurate compared with the pre-compiled version and the savings in number of nodes caused by traversing the AND/OR search tree do not translate into additional time savings. When comparing the static and dynamic mini-bucket heuristics, we see that the latter are competitive only for relatively small i-bounds, because of their higher computational overhead. This may be significant because small i-bounds usually require restricted space. At higher levels of the i-bound, the accuracy of the dynamic mini-bucket heuristic does not outweigh its overhead.

In some exceptional cases the OR Branch-and-Bound explored fewer nodes than the AND/OR counterpart. For example, on the `90-16-1` grid, the search space explored by `AOBB+SMB(16)` was almost 4 times larger that explored by `BB+SMB(16)`. This can be explained by the internal dynamic ordering used by AND/OR Branch-and-Bound to solve independent subproblems rooted at the AND nodes in the search tree.

Figs. 10(a) and 10(b) plot the running time and number of nodes visited by `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)` (resp. `BB+SMB(`i`)` and `BB+DMB(`i`)`), on the `90-14-1` grid network from Table 1. Focusing on `AOBB+SMB(`i`)` (resp. `BB+SMB(`i`)`) in Fig. 10(a) 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 $i = 10$ the time is 3.44 for `AOBB+SMB(10)` and 71.98 for `BB+SMB(10)`, respectively), but then as i increases further the time starts to increase again. The same behavior can be observed in the case of `AOBB+DMB(`i`)` (resp. `BB+DMB(`i`)`) as well. When looking at the size of the search space explored as a function of the i-bound (shown in Fig. 10(b)) we can see that as the i-bound increases, the strength of the heuristic estimates increases as well, therefore pruning the search space more effectively.

Fig. 11 displays the running time distribution of `AOBB+SMB(`i`)` using hypergraph based pseudo trees for the `90-24-1` (left) and `90-26-1` (right) grid networks, respectively. For each reported i-bound (the X axis), the corresponding data point and error bar show the average as well as the minimum and maximum running time obtained over 20 independent runs of the algorithm with a 30 minute time limit. We also record the average induced width and pseudo tree depth obtained with the hypergraph partitioning heuristic (shown in the header of each plot in Fig. 11). For comparison, we also display the results obtained with the min-fill pseudo trees from Table 1. We see that the hypergraph based pseudo trees are significantly shallower compared with the min-fill ones, and in some cases they are able to improve performance dramatically, especially at relatively small i-bounds. For example, on the grid `90-24-1`, `AOBB+SMB(14)` guided by a hypergraph pseudo tree is about 2 orders of magnitude faster than `AOBB+SMB(14)` using a min-fill pseudo tree. At larger i-bounds, the pre-compiled mini-bucket heuristic benefits from the small induced width which normally is obtained with the min-fill ordering. Therefore `AOBB+SMB(`i`)` using min-fill based trees is generally faster than `AOBB+SMB(`i`)` guided by hypergraph based trees (*e.g.*, `90-26-1`).

9.3.2. Genetic linkage analysis

In human genetic linkage analysis [53], 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

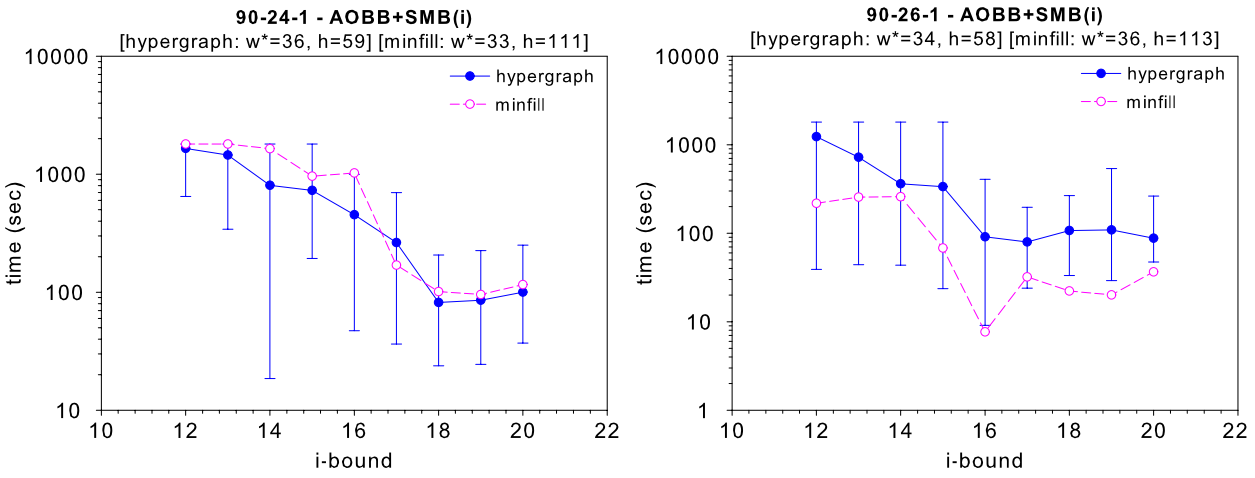


Fig. 11. Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. Distribution of CPU time for solving the **90-24-1** (left) and **90-26-1** (right) **grid networks** with AOBB+SMB(i).

pedigree which maximizes the probability of data. It can be shown that given the pedigree data, the haplotyping problem is equivalent to computing the most probable explanation of a Bayesian network that represents the pedigree [54,55].

Table 2 shows the results for experiments with 12 genetic linkage networks[7] using AND/OR Branch-and-Bound search guided by static mini-bucket heuristics. The columns are indexed by the mini-bucket i-bound. The table is organized into three horizontal blocks, each corresponding to a different range of i-bound values. For comparison, we include results obtained with SUPERLINK 1.6. SUPERLINK [54,55] which is currently one of the most efficient solvers for genetic linkage analysis, uses a combination of variable elimination and conditioning, and takes advantage of the determinism in the network. We did not run AOBB+DMB(i) (resp. BB+DMB(i)) on this domain because of its prohibitively high computational overhead associated with relatively large i-bounds.

We observe that AOBB+SMB(i) is the overall best performing algorithm, outperforming its competitors on 8 out of the 12 test networks. For example, on the ped23 instance, AOBB+SMB(12) is 2 orders of magnitude faster than SUPERLINK, whereas SAMIAM and BB+SMB(i) exceed the 2GB memory bound and the 3 hour time limit, respectively. Similarly, on the ped30 instance, AOBB+SMB(20) outperforms SUPERLINK with about 2 orders of magnitude, while neither SAMIAM nor BB+SMB(20) are able to solve the problem instance. Notice however that the ped42 instance is solved only by SUPERLINK. When looking at the impact of the mini-bucket i-bound, we see again that the performance of Branch-and-Bound changes with the mini-bucket strength.

Fig. 12 displays the running time distribution of AOBB+SMB(i) with hypergraph based pseudo trees for 2 linkage instances from Table 2. We see that the hypergraph based pseudo trees are significantly shallower compared with the min-fill based ones, and in some cases they are able to improve the performance dramatically for relatively small i-bounds.

9.3.3. 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 [56–59]. 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 [60] or generalized arc consistency has been explored in [56,57], in the context of variable elimination algorithms where the constraints are also extracted based on the zero probabilities in the Bayesian network. The approach we take for handling the determinism in belief networks is based on the known technique of *unit resolution* for Boolean Satisfiability (SAT). The idea of using unit resolution during search for Bayesian networks was first explored in [58]. One common way which we used for encoding hard constraints as a CNF formula is the *direct encoding* [61].

We evaluated the AND/OR Branch-and-Bound algorithms with static and dynamic mini-bucket heuristics on selected classes of Bayesian networks containing deterministic conditional probability tables (*i.e.*, zero probability tuples). The algorithms exploit 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. They are denoted by AOBB+SAT+SMB(i) and AOBB+SAT+DMB(i), respectively. We used a unit resolution scheme similar to the one employed by zChaff, a state-of-the-art SAT solver introduced by [62]. These experiments were performed on a 2.4 GHz Pentium IV with 2 GB of RAM running Windows XP, and therefore the CPU times reported here may be slower than those in the previous sections.

[7] Available at http://bioinfo.cs.technion.ac.il/superlink/. The corresponding belief network of the pedigree data was extracted using the export feature of the SUPERLINK 1.6 program.

Table 2
CPU time in seconds and nodes visited for solving **genetic linkage networks** using static mini-bucket heuristics and min-fill based pseudo trees. Time limit 3 hours. The three horizontal blocks of the table show different ranges of the mini-bucket i-bounds.

min-fill pseudo tree											
pedigree (n, k) (w^*, h)	Superlink Samlam	MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 6$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 8$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 10$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 12$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 14$	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped1	54.73	0.05		0.05		0.11		0.31		0.97	
(299, 5)	5.44	–	–	–	–	6.34	37,657	7.33	42,447	8.30	41,134
(15, 61)		24.30	416,326	13.17	206,439	**1.58**	24,361	1.84	25,674	1.89	15,156
ped38		0.12		0.45		2.20		60.97		out	
(582, 5)	**28.36**	–	–	–	–	–	–	–	–		
(17, 59)	out	–	–	8120.58	85,367,022	–	–	3040.60	35,394,461		
ped50		0.11		0.74		5.38		37.19		out	
(479, 5)	–	–	–	–	–	–	–	–	–		
(18, 58)	out	–	–	–	–	476.77	5,566,578	**104.00**	748,792		
pedigree (n, k) (w^*, h)	Superlink Samlam	MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 10$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 12$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 14$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 16$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 18$	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped23	9146.19	0.42		2.33		11.33		274.75		out	
(310, 5)	out	–	–	–	–	3176.72	14,044,797	343.52	358,604		
(27, 71)		498.05	6,623,197	**15.45**	154,676	16.28	67,456	286.11	117,308		
ped37	**64.17**	0.67		5.16		21.53		58.59		out	
(1032, 5)	out	–	–	–	–	–	–	–	–		
(21, 61)		273.39	3,191,218	1682.09	25,729,009	1096.79	15,598,863	128.16	953,061		
pedigree (n, k) (w^*, h)	Superlink Samlam	MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 12$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 14$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 16$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 18$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i = 20$	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped18	139.06	0.51		1.42		4.59		12.87		19.30	
(1184, 5)	157.05	–	–	–	–	–	–	–	–	–	–
(21, 119)		–	–	2177.81	28,651,103	270.96	2,555,078	100.61	682,175	**20.27**	7689
ped20	**14.72**	1.42		5.11		37.53		410.96		out	
(388, 5)	out	–	–	–	–	–	–	–	–		
(24, 66)		3793.31	54,941,659	1293.76	18,449,393	1259.05	17,810,674	1080.05	9,151,195		
ped25	–	0.34		0.72		2.27		6.56		29.30	
(994, 5)	out	–	–	–	–	–	–	–	–	–	–
(34, 89)		–	–	–	–	9399.28	111,301,168	3607.82	34,306,937	**2965.60**	28,326,541
ped30	13095.83	0.42		0.83		1.78		5.75		21.30	
(1016, 5)	out	–	–	–	–	–	–	–	–	–	–
(23, 118)		–	–	–	–	–	–	214.10	1,379,131	**91.92**	685,661
ped33	–	0.58		2.31		7.84		33.44		112.83	
(581, 4)	out	–	–	–	–	–	–	–	–	–	–
(37, 165)		2804.61	34,229,495	737.96	9,114,411	3896.98	50,072,988	**159.50**	1,647,488	2956.47	35,903,215
ped39	322.14	0.52		2.32		8.41		33.15		81.27	
(1272, 5)	out	–	–	–	–	–	–	–	–	–	–
(23, 94)		–	–	–	–	4041.56	52,804,044	386.13	2,171,470	**141.23**	407,280
ped42	**561.31**	4.20		31.33		206.40		out		out	
(448, 5)	out	–	–	–	–	–	–				
(25, 76)		–	–	–	–	–	–				

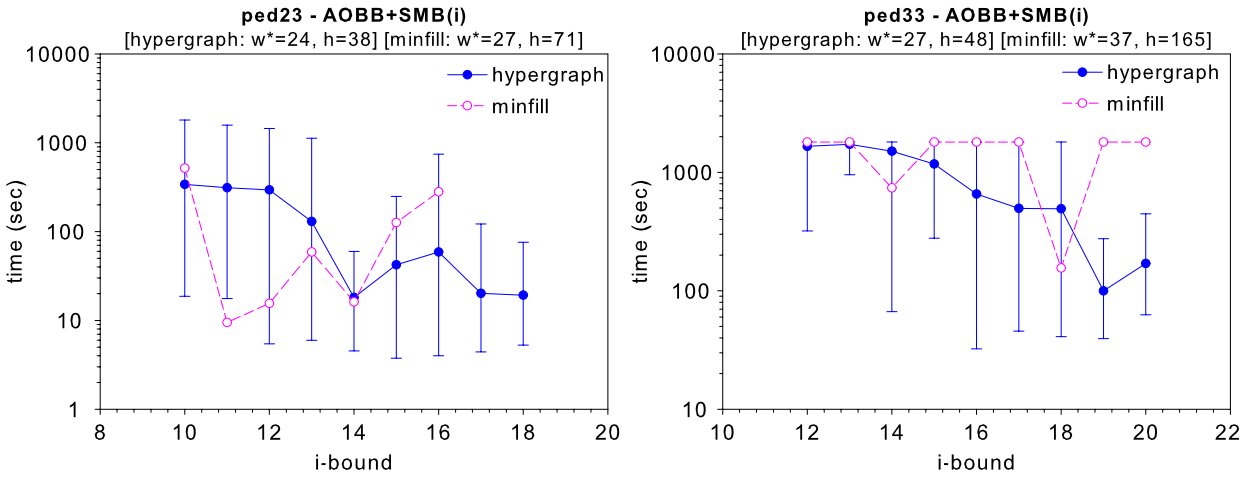


Fig. 12. Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. Distribution of CPU time for solving the **ped23** (left) and **ped33** (right) **genetic linkage networks** with AOBB+SMB(i).

Table 3 shows the results for experiments with the grid networks from Section 9.3.1. As mentioned earlier, 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 entries.

We observe that AOBB+SAT+SMB(i) improves significantly over AOBB+SMB(i), especially at relatively small i-bounds. For example, on the 90-26-1 grid instance, AOBB+SAT+SMB(10) is 9 times faster than AOBB+SMB(10). As the i-bound increases and the search space is pruned more effectively, the difference between AOBB+SMB(i) and AOBB+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 comparing the AND/OR search algorithms with dynamic mini-bucket heuristics, we see that the difference between AOBB+DMB(i) and AOBB+SAT+DMB(i) is again more pronounced at relatively small i-bounds.

9.3.4. Summary of empirical results on Bayesian networks

Our extensive empirical evaluation on Bayesian networks demonstrated conclusively that the AND/OR Branch-and-Bound *tree* search algorithms guided by static mini-bucket heuristics were the best performing algorithms overall. The difference between AOBB+SMB(i) and the OR tree search counterpart BB+SMB(i) was more pronounced at relatively small i-bounds (corresponding to relatively weak heuristic estimates) and amounted to almost 2 orders of magnitude in terms of both running time and size of the search space explored (*e.g.*, ISCAS'89 networks, grid networks, instances from the UAI'06 Inference Evaluation contest, genetic linkage analysis). For larger i-bounds, when the heuristic estimates are strong enough to prune the search space substantially, the difference between AND/OR and OR Branch-and-Bound decreased. We also showed that AOBB+SMB(i) was in many cases able to outperform dramatically the current state-of-the-art solvers for Bayesian networks such as SamIam and Superlink (for genetic linkage analysis). With dynamic mini-bucket heuristics AOBB+DMB(i) AND/OR Branch-and-Bound proved competitive only for relatively small i-bounds due to computational overhead issues (*e.g.*, ISCAS'89 networks, instances from the Bayesian Networks Repository). This suggests that the dynamic mini-bucket heuristics can be considered when space is limited. We also evaluated the impact of determinism over ISCAS'89 networks and genetic linkage analysis networks. These empirical results, also available in [47,48], showed that while applying unit resolution caused significant time savings on the ISCAS'89 networks, it was not cost effective for linkage networks.

9.4. Results for empirical evaluation of Weighted CSPs

In this section we focus on Weighted CSP problems. We evaluate both mini-bucket and EDAC heuristics when the problems are solved in a static variable ordering. We also evaluate the impact of dynamic variable orderings when using directional arc-consistency (EDAC) based heuristics.

9.4.1. SPOT5 benchmark

SPOT5 benchmark contains a collection of large real scheduling problems for the daily management of Earth observing satellites [63]. The problem of scheduling an Earth observing satellite is to select from a set of candidate photographs, the best subset such that a set of imperative constraints are satisfied and the total importance of the selected photographs is maximized. These problem instances can be naturally casted as WCSPs with binary and ternary cost functions, as described in [63].

Table 4 reports the results obtained for experiments with 7 SPOT5 networks, using min-fill pseudo trees. We see that AOBB+SMB(i) is the best performing algorithm on this dataset. The overhead of the dynamic mini-bucket heuristics out-

Table 3
CPU time and nodes visited for solving **deterministic grid networks** using mini-bucket heuristics, min-fill based pseudo trees and constraint propagation. Time limit 1 hour. The two horizontal blocks of the table correspond to different ranges of the mini-bucket i-bound.

min-fill pseudo tree												
grid (w^*, h) (n, e)	AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=8$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=10$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=12$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=14$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=16$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=18$	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-10-1	0.31	8080	0.11	2052	**0.02**	101	0.05	101	0.05	101	0.06	101
(13, 39)	0.28	7909	0.09	2050	0.05	101	0.06	101	0.06	101	0.06	101
(100, 0)	0.31	344	0.30	241	0.24	101	0.30	101	0.30	101	0.28	101
	0.52	344	0.47	241	0.39	101	0.47	101	0.47	101	0.47	101
90-14-1	7.84	130,619	6.42	100,696	1.03	17,479	0.34	3321	0.61	2938	1.81	3386
(22, 66)	2.36	45,870	2.52	46,064	0.66	11,914	**0.31**	3286	0.61	2922	1.78	3359
(196, 0)	62.17	31,476	25.22	4137	5.05	397	7.61	211	10.67	199	21.23	198
	33.03	10,135	16.08	3270	4.92	396	7.72	211	10.88	199	21.64	198
90-16-1	646.83	10,104,350	164.02	2,600,690	13.14	193,440	2.92	39,825	2.08	23,421	2.92	5842
(24, 82)	121.24	2,209,097	78.97	1,416,247	6.99	121,595	2.25	35,376	**1.84**	22,986	2.84	5609
(256, 0)	1030.41	462,180	316.77	47,121	75.13	3227	52.16	719	25.63	260	65.05	260
	841.32	452,923	248.38	37,670	55.86	2264	49.99	719	25.03	260	64.99	260
grid (w^*, h) (n, e)	AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=10$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=12$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=14$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=16$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=18$		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) $i=20$	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-24-1	–	–	–	–	2214.12	24,117,151	1479.15	18,238,983	132.35	1,413,764	135.72	1,308,009
(33, 111)	1529.21	18,103,859	2605.56	30,929,553	689.47	9,868,626	738.17	11,100,088	**106.00**	1,282,902	121.67	1,273,738
(576, 20)	–	–	–	–	–	–	884.41	2739	1223.18	1228	1634.57	598
	–	–	–	–	–	–	843.79	2739	1173.48	1228	1611.74	598
90-26-1	2217.15	17,899,574	314.88	2,903,489	382.22	3,205,257	8.42	59,055	23.14	165,182	22.22	5777
(36, 113)	233.94	2,527,496	103.56	1,264,309	167.27	1,805,787	**6.20**	43,798	19.36	150,345	22.11	4935
(676, 40)	1420.21	177,661	–	–	–	–	938.98	2545	1701.64	1191	2638.95	691
	1099.87	171,961	1592.53	108,694	1034.26	12,819	862.38	2545	1583.37	1191	2478.19	691
90-30-1	–	–	1125.40	9,445,224	379.14	3,324,942	339.66	3,039,966	147.99	1,358,569	93.63	485,300
(43, 150)	754.427	7,050,411	367.41	3,723,781	190.38	2,002,447	164.39	1,734,294	107.95	1,150,182	**70.14**	387,242
(900, 60)	–	–	–	–	–	–	–	–	–	–	–	–
	–	–	–	–	–	–	–	–	–	–	–	–
90-34-1	–	–	–	–	–	–	–	–	–	–	462.41	1,549,829
(45, 153)	–	–	–	–	–	–	–	–	–	–	**255.08**	981,831
(1154, 80)	–	–	–	–	–	–	–	–	–	–	–	–
	–	–	–	–	–	–	–	–	–	–	–	–
90-38-1	–	–	2007.47	6,835,745	3589.43	12,321,175	800.72	2,850,393	566.11	2,079,146	368.60	1,038,065
(47, 163)	1128.56	5,121,466	410.94	1,972,430	578.54	2,339,244	270.05	1,349,223	278.11	1,249,270	**204.56**	702,806
(1444, 120)	–	–	–	–	–	–	–	–	–	–	–	–
	–	–	–	–	–	–	–	–	–	–	–	–

weighs search pruning here. We also see, once again, the impact of the AND/OR versus the OR search space. For instance, on the `404` network, the difference between `AOBB+SMB(12)` and `BB+SMB(12)`, in terms of running time and size of the search space explored, is up to 3 orders of magnitude. The best performances on this domain are obtained by `AOBB+SMB(`i`)` at relatively large i-bounds which generate very accurate heuristic estimates. For example, `AOBB+SMB(14)` is the only algorithm able to solve the `505b` network. `AOEDAC` and `toolbar` were able to solve relatively efficiently only 3 out of the 7 test instances (*e.g.*, `29`, `54` and `404`).

In Figs. 13(a) and 13(b) we plot the running time and number of nodes visited by `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)` (resp. `BB+SMB(`i`)` and `BB+DMB(`i`)`), as a function of the i-bound, on the `29` SPOT5 network from Table 4. We see that `AOBB+SMB(`i`)` achieves the best performance at $i=8$, whereas `AOBB+DMB(`i`)` performs best only at the smallest reported i-bound, namely $i=4$. This suggests, again, that dynamic mini-bucket heuristics can be considered when space is limited.

Table 4
CPU time and nodes visited for solving **SPOT5 networks** using mini-bucket heuristics and min-fill based pseudo trees. Time limit 2 hours.

minfill pseudo tree												
spot5 (w^*, h) (n, k, c)	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i = 4$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i = 6$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i = 8$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i = 12$		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) $i = 14$		AOEDAC toolbar	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
29	0.01		0.03		0.34		21.72		147.66		613.79	8,997,894
(14, 42)	–	–	–	–	–	–	25.69	5095	148.27	632	4.56	218,846
(83, 4, 476)	8.44	86,058	4.83	45,509	**0.64**	2738	21.74	246	147.69	481		
	44.42	12,007	131.64	9713	57.22	541	678.22	507	1758.78	507		
	28.27	14,438	65.91	11,850	53.72	364	630.09	330	1675.74	330		
42b	0.01		0.11		0.50		28.81		223.14		–	–
	–	–	–	–	2154.64	9,655,444	148.11	712,685	228.17	12,255	–	–
(18, 62)	–	–	–	–	1790.76	9,606,846	**131.34**	689,402	223.64	4189		
(191, 4, 1341)	–	–	–	–	–	–	–	–	–	–		
	–	–	–	–	–	–	–	–	–	–		
54	0.01		0.02		0.09		1.25		1.23		31.34	823,326
(11, 33)	668.77	6,352,998	2.98	27,383	0.59	4996	1.28	921	1.52	921	**0.31**	21,939
(68, 4, 283)	105.99	1,106,598	1.50	17,757	0.34	3616	1.28	329	1.27	329		
	1150.54	163,993	52.44	2469	38.63	921	464.58	921	465.35	921		
	204.11	69,362	27.27	2188	21.91	329	266.55	329	265.89	329		
404	0.01		0.02		0.09		1.09		4.03		255.83	3,260,610
(19, 42)	–	–	–	–	–	–	4009.57	32,763,223	1827.05	15,265,025	151.11	6,215,135
(100, 4, 710)	413.18	3,969,398	146.05	1,373,846	14.08	144,535	**1.39**	3273	4.06	367		
	–	–	–	–	–	–	–	–	1964.20	2015		
	238.97	156,338	272.46	39,144	215.17	5612	565.06	1327	167.90	220		
408b	0.02		0.08		0.31		8.30		35.22		–	–
(24, 59)	–	–	–	–	–	–	–	–	–	–	–	–
(201, 4, 1847)	–	–	–	–	–	–	682.12	4,784,407	**124.67**	567,407		
	–	–	–	–	–	–	–	–	–	–		
	–	–	–	–	–	–	–	–	–	–		
503	0.01		0.03		0.14		0.39		0.39		–	–
(9, 39)	–	–	–	–	–	–	1.22	5229	1.22	5229	–	–
(144, 4, 639)	–	–	412.63	5,102,299	397.77	4,990,898	**0.44**	641	**0.44**	641		
	–	–	–	–	–	–	690.44	5229	694.86	5229		
	–	–	–	–	–	–	64.02	641	64.52	641		
505b	0.01		0.01		0.12		48.20		372.27		–	–
(16, 98)	–	–	–	–	–	–	–	–	–	–	–	–
(240, 4, 1721)	–	–	–	–	–	–	–	–	**392.08**	143,371		
	–	–	–	–	–	–	–	–	–	–		
	–	–	–	–	–	–	–	–	–	–		

Fig. 14 displays the running time distribution of AOBB+SMB(i) guided by hypergraph based pseudo trees, over 20 independent runs, for two SPOT5 instances from Table 4. The hypergraph based trees have far smaller depths than the min-fill ones, and therefore are again able to improve the running time over min-fill based ones only at relatively small i-bounds (*e.g.*, 404). On average, however, the min-fill pseudo trees generally yield a more robust performance, especially for larger i-bounds of the mini-bucket heuristics (*e.g.*, 503).

9.4.2. Mastermind game instances

Table 5 shows the results for experiments with 6 networks corresponding to Mastermind game instances of increasing difficulty. Each of these networks is a ground instance of a relational Bayesian network that models differing sizes of the popular game of Mastermind. They were produced by the PRIMULA System[8] and used in experimental results from [64]. For our purpose, we converted these networks into equivalent WCSP instances by taking the negative log probability of each conditional probability table entry, multiplying it with 1000 and rounding it to the nearest integer. The resulting WCSP

[8] Available at: http://www.cs.auc.dk/jaeger/Primula.

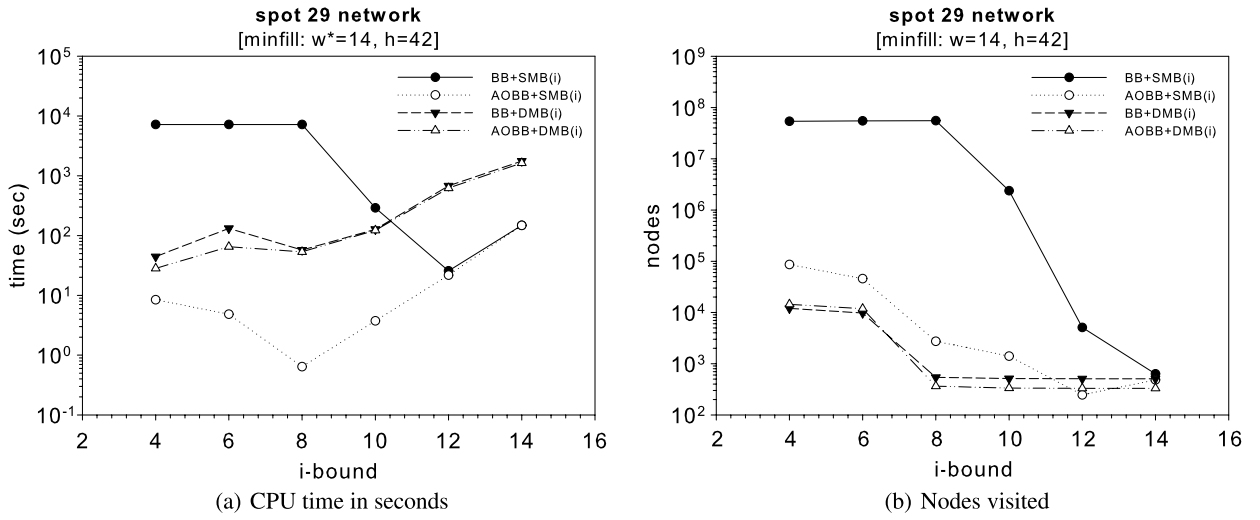


Fig. 13. Comparison of the impact of static and dynamic mini-bucket heuristics. CPU time (a) and number of nodes visited (b) on the **29 SPOT5 instance** from Table 4.

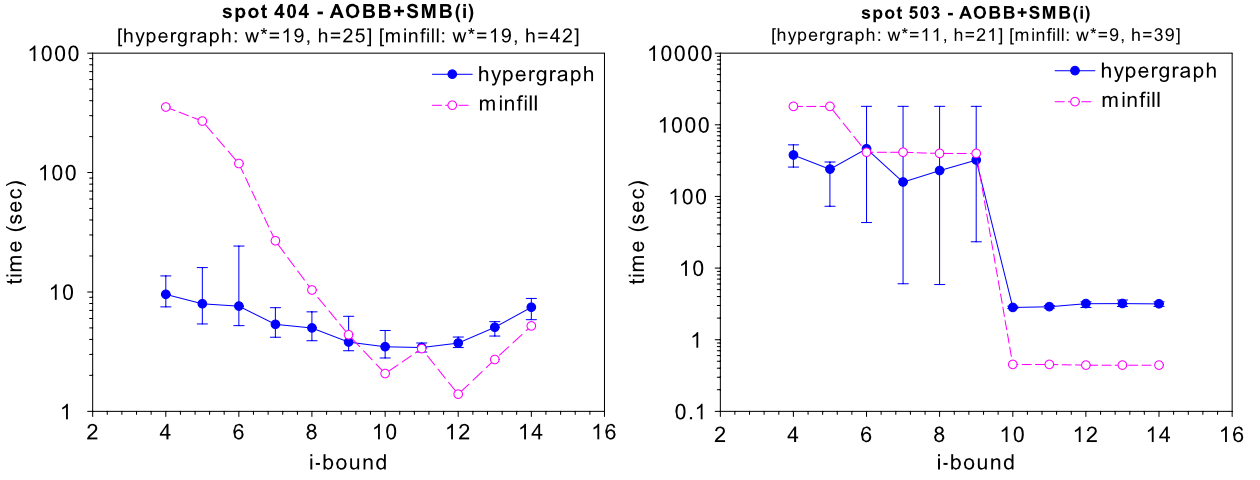


Fig. 14. Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. Distribution of CPU time for solving the **404** (left) and **503** (right) **SPOT5 networks** with AOBB+SMB(i).

instances are quite large with the number of bi-valued variables n ranging between 1220 and 3692, and containing n unary and ternary cost functions. The table has two horizontal blocks each showing a different range of i-bounds.

We see again that AOBB+SMB(i) offers the overall best performance. For example, AOBB+SMB(10) solves the mm-04-08-03 instance in about 3 seconds, whereas BB+SMB(10) exceeds the 1 hour time limit. We did not report results with dynamic mini-bucket heuristics because of the prohibitively large computational overhead associated with relatively large i-bounds. We also note that the EDAC based algorithms were not able to solve any of these instances within the alloted time bound (not shown in the table).

In Fig. 15 we display the running time distribution of AOBB+SMB(i) guided by hypergraph based pseudo trees over 20 independent runs, for two game instances from Table 5. The spectrum of results is similar to what we observed earlier.

9.4.3. The impact of dynamic variable orderings

In this section we evaluate the impact of dynamic variable orderings on AND/OR Branch-and-Bound search guided by local consistency (EDAC) heuristics.

Table 6 shows the results for experiments with the SPOT5 networks from Section 9.4.1. For reference, the last column of the table shows the best performances obtained with AOBB+SMB(i) (the value of the mini-bucket i-bound is given in parenthesis). We see that variable ordering can have a tremendous impact on performance. Indeed, AOEDAC+DSO is the best performing among the EDAC based algorithms, and is able to solve 6 out of 7 test instances. The second best algorithm in this category is DVO+AOEDAC which solves relatively efficiently 3 test networks. This demonstrates the benefit of using dynamic variable ordering heuristics within the AND/OR Branch-and-Bound search. We also observe that the best

Table 5
CPU time in seconds and nodes visited for solving **Mastermind game instances** using static mini-bucket heuristics and min-fill based pseudo trees. Time limit 1 hour. AOEDAC and toolbar did not solve any of the test instances within the time limit.

minfill pseudo tree												
mastermind (w^*, h) (n, r, k)	MBE(i) BB+SMB(i) AOBB+SMB(i) $i=8$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=10$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=12$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=14$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=16$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=18$	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
mm-03-08-03	0.17		0.22		0.35		0.91		2.83		7.99	
(20, 57)	–	–	–	–	–	–	897.87	873,606	946.84	915,095	738.13	720,764
(1220, 3, 2)	1.16	10,369	**0.88**	7075	0.93	6349	1.23	3830	3.11	3420	8.25	3153
mm-03-08-04	0.48		0.60		0.89		2.08		6.45		25.15	
(33, 87)	–	–	–	–	–	–	–	–	–	–	–	–
(2288, 3, 2)	72.37	150,642	66.69	193,805	36.22	71,622	**10.15**	31,177	25.16	63,669	29.27	13,870
mm-04-08-03	0.21		0.27		0.48		1.06		3.54		12.52	
(26, 72)	–	–	–	–	1609.86	1,315,415	1603.71	1,175,430	1157.09	901,309	1924.02	1,451,854
(1418, 3, 2)	8.20	68,929	**3.05**	26,111	4.23	34,445	3.10	17,255	5.29	15,443	13.71	10,570
mastermind (w^*, h) (n, r, k)	MBE(i) BB+SMB(i) AOBB+SMB(i) $i=12$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=14$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=16$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=18$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=20$		MBE(i) BB+SMB(i) AOBB+SMB(i) $i=22$	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
mm-04-08-04	1.19		2.35		6.85		26.47		106.37		395.57	
(39, 103)	–	–	–	–	–	–	–	–	–	–	–	–
(2616, 3, 2)	324.06	744,993	166.67	447,464	310.06	798,507	**64.72**	107,463	192.39	242,865	414.54	62,964
mm-03-08-05	2.14		4.54		11.82		39.01		134.46		497.45	
(41, 111)	–	–	–	–	–	–	–	–	–	–	–	–
(3692, 3, 2)	–	–	–	–	–	–	**835.90**	1,122,008	1162.22	1,185,327	1200.65	1,372,324
mm-10-08-03	1.48		3.78		11.39		34.53		127.55		593.25	
(51, 132)	–	–	–	–	–	–	–	–	–	–	–	–
(2606, 3, 2)	109.50	290,594	128.29	326,662	**64.31**	151,128	74.14	127,130	169.84	133,112	623.83	79,724

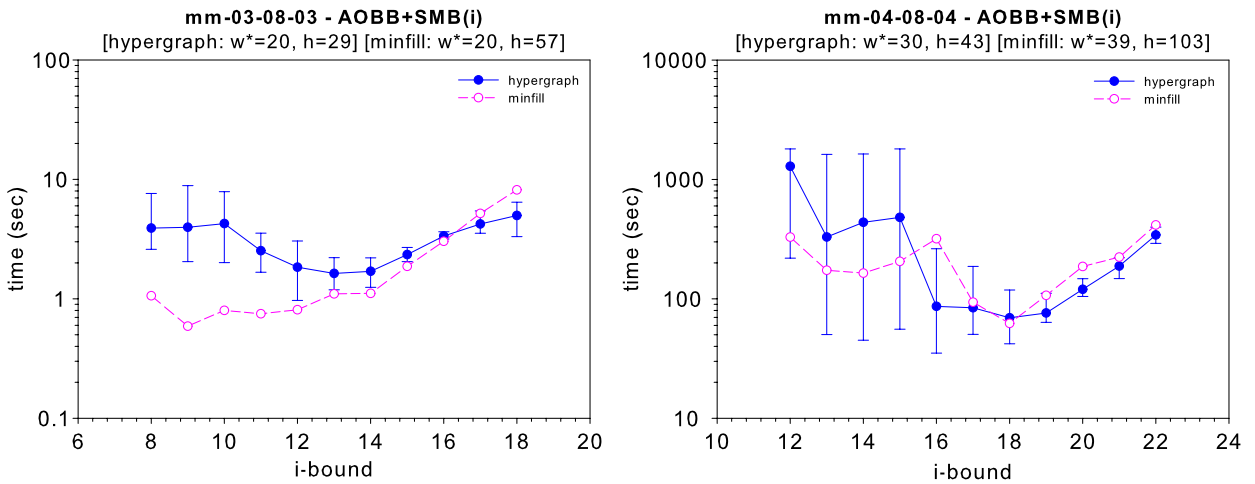


Fig. 15. Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving the **mm-03-08-03** (left) and **mm-04-08-04** (right) **Mastermind networks** with AOBB+SMB(i).

performance points highlighted in Table 6 are inferior to those corresponding to AOBB+SMB(i). For example, on the 42b network, the difference in running time and size of the search space explored between AOBB+SMB(12) and AOEDAC+DSO is up to one order of magnitude in favor of the former. Similarly, the 505b network could not be solved by any of the EDAC based algorithms, whereas AOBB+SMB(14) finds the optimal solution in about 6 minutes. Notice that toolbar is much better than BBEDAC in all test cases. This can be explained by its more careful and optimized implementation of EDAC within toolbar.

Table 6
CPU time in seconds and nodes visited for solving **SPOT5 networks** using EDAC heuristics, dynamic variable orderings and min-fill based pseudo trees. Time limit 2 hours.

minfill pseudo tree										
spot5	n c	w^* h		toolbar	BBEDAC	AOEDAC	AOEDAC+PVO	DVO+AOEDAC	AOEDAC+DSO	AOBB+SMB(i)
29	16	7	time	4.56	109.66	613.79	545.43	**0.83**	11.36	**0.64** ($i=8$)
	57	8	nodes	218,846	710,122	8,997,894	7,837,447	8698	92,970	2738
42b	14	9	time	–	–	–	–	–	**6825.4**	**131.34** ($i=12$)
	75	9	nodes	–	–	–	–	–	27,698,614	689,402
54	14	9	time	0.31	0.97	31.34	9.11	**0.06**	0.75	**0.34** ($i=8$)
	75	9	nodes	21,939	8270	823,326	90,495	688	6614	3616
404	16	10	time	151.11	2232.89	255.83	152.81	12.09	**1.74**	**1.39** ($i=12$)
	89	12	nodes	6,215,135	7,598,995	3,260,610	1,984,747	88,079	14,844	3273
408b	18	10	time	–	–	–	–	–	**747.71**	**124.67** ($i=14$)
	106	13	nodes	–	–	–	–	–	2,134,472	567,407
503	22	11	time	–	–	–	–	–	**53.72**	**0.44** ($i=12$)
	131	15	nodes	–	–	–	–	–	231,480	641
505b	16	9	time	–	–	–	–	–	–	**392.08** ($i=14$)
	70	10	nodes	–	–	–	–	–	–	143,271

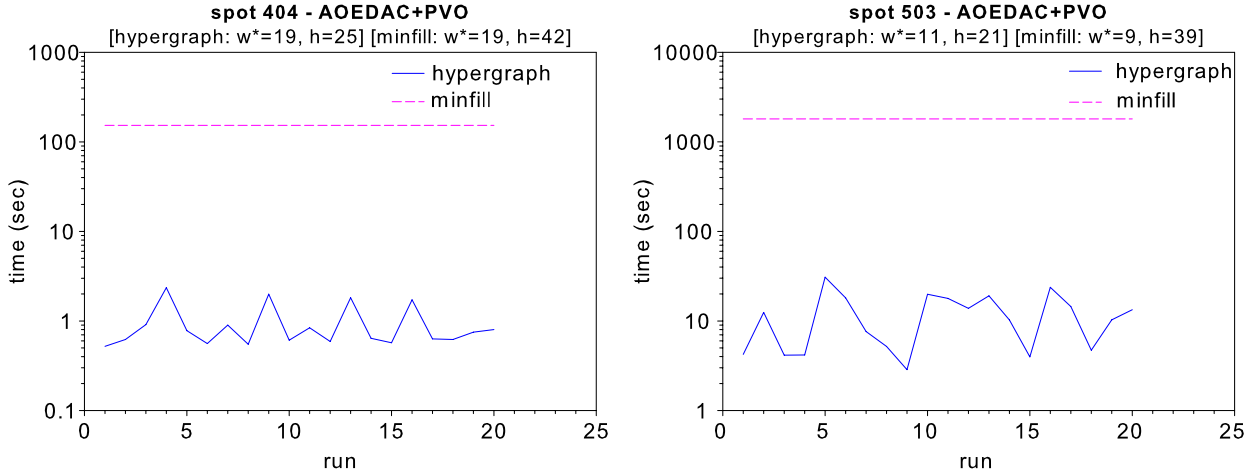


Fig. 16. Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. Distribution of CPU time for solving the **404** (left) and **503** (right) **SPOT5 networks** with AOEDAC+PVO.

In Fig. 16 we show the running time distribution of AOEDAC+PVO with hypergraph pseudo trees, on 20 independent runs, for two networks from Table 6. In this case, the difference between the min-fill and the hypergraph case is dramatic, resulting in up to three orders of magnitude in favor of the latter.

We also evaluated the impact of dynamic variable orderings on radio link frequency assignment problems (detailed results for these experiments are available online in [47,48]). The AOEDAC algorithms with dynamic variable orderings were again superior to the OR BBEDAC as well as the AOEDAC using a static variable ordering. However, their performance was quite inferior to that of toolbar. We suspect that this was mainly due to implementation issues.

9.4.4. Summary of empirical results on WCSPs

Our extensive empirical evaluation on WCSPs demonstrated again that the best performance on this domain was obtained by the AND/OR Branch-and-Bound *tree* search algorithm with static mini-bucket heuristics, for large i-bounds, especially on non-binary WCSPs with relatively small domain sizes (*e.g.*, Mastermind game instances, ISCAS'89 networks, instances from the SPOT5 benchmark). AOBB+SMB(i) dominated all its competitors, including the classic 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. The AND/OR Branch-and-Bound with dynamic mini-bucket heuristics AOBB+DMB(i) was competitive only for relatively small i-bounds (*e.g.*, ISCAS'89 networks [47,48]). We also observed that on binary problems having large domain sizes, the mini-

bucket heuristics were far inferior to those based on enforcing local consistency (*e.g.*, radio link frequency assignment problems [47,48]).

10. Related work

The idea of exploiting structural properties of the problem in order to enhance the performance of search algorithms is not new. It was first introduced in constraint satisfaction, then moved to satisfiability. Later, it was recognized in the probabilistic community via the cycle-cutset [6] and recursive conditioning [49] algorithms and followed up by value elimination [65]. It was extended to optimization at about the same time. We next elaborate more on these various contributions.

In constraints, Freuder and Quinn [2] 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* [2] is conducted over a pseudo tree arrangement of the problem which allows the detection of independent subproblems that are solved separately. Bayardo and Miranker [3] 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. More recently, [66] extended pseudo tree search [2] to optimization tasks in order to boost the Russian Doll search [14] for solving Weighted CSPs. Our AND/OR Branch-and-Bound algorithm is also related to the Branch-and-Bound method proposed by [35] for acyclic AND/OR graphs and game trees. The difference is that we specialize the AND/OR search over graphical models. Here, the decomposition is graph-based.

Dechter's graph-based back-jumping algorithm [67] 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 [68] for a distributed constraint satisfaction algorithm. More recently, distributed constraint optimization problems in which multiple agents are involved, are solved using a pseudo tree arrangement in a best-first or depth-first manner using linear space of each agent [69–71]. A distributed variable elimination algorithm that uses a pseudo tree arrangements of the agents was also proposed in [72].

In probabilistic reasoning, *Recursive Conditioning* (RC) [49] is a search method based on the divide and conquer paradigm. Like AND/OR search, 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* [49]. It was shown in [1] that RC explores an AND/OR space whose guiding pseudo tree can be generated from the static ordering dictated by the dtree. *Value Elimination* (VE) [65] is a recently developed algorithm for Bayesian inference. Given a static ordering d for VE, it was shown that it traverses an AND/OR space [1]. The pseudo tree underlying the AND/OR search graph traversal by VE can be constructed as the bucket tree in reversed order of d. The traversal of the AND/OR space will be controlled by d, advancing the frontier in a hybrid depth or breadth first manner.

In optimization, *Backtracking with Tree-Decomposition* (BTD) [73] is a memory intensive method for solving constraint optimization problems which combines search techniques with the notion of tree decomposition. This mixed approach can in fact be viewed as searching an AND/OR search space whose backbone pseudo tree is defined by and structured along the tree decomposition [1].

We note however that Recursive Conditioning, Backtracking with Tree Decomposition and Value Elimination, unlike our AND/OR Branch-and-Bound search, are not restricted to be linear space search methods. They can be parameterized and use various levels of caching, which can yield in the worst case an exponential space complexity. In a subsequent article we will extend the AND/OR algorithms to use substantial memory by exploring an AND/OR search *graph*, rather than the tree.

11. Summary and conclusion

The paper investigates the impact of graph-based AND/OR search spaces on solving general constraint optimization problems in graphical models focusing on search trees that do not facilitate caching. In contrast to the traditional OR search, the new AND/OR search is sensitive the problem's structure. The linear space AND/OR tree search algorithms can be exponentially better (and never worse) than the linear space OR tree search algorithms. Specifically, the size of their search tree is exponential in the depth of the guiding pseudo tree rather than the number of variables, as in the OR case.

The AND/OR Branch-and-Bound algorithm that we introduced explores the AND/OR search tree in a depth-first manner and can be guided by any heuristic function. We investigated extensively the mini-bucket heuristic and showed empirically that it can prune the search space very effectively. The mini-bucket heuristics can be either pre-compiled (static mini-buckets) or generated dynamically at each node in the search tree (dynamic mini-buckets). They are parameterized by an i-bound which allows for a controllable trade-off between heuristic strength and its computational overhead. We also explored the effectiveness of a class of heuristic functions derived from local consistency algorithms, in the context of WCSPs. Since variable ordering can influence dramatically search performance, we also introduced and investigated empirically several ordering schemes that combine the AND/OR decomposition principle with dynamic variable ordering heuristics.

We focused our empirical evaluation on finding the MPE in Bayesian networks and solving WCSPs. Our results demonstrated conclusively that in many cases the depth-first AND/OR Branch-and-Bound algorithms improve dramatically over traditional OR Branch-and-Bound search, especially for relatively weak guiding heuristic estimates when space is really restricted. We summarize next the most important additional factors that when augmented on top of AND/OR search help improve its performance. This includes the mini-bucket i-bound, dynamic variable orderings, constraint propagation and the quality of the guiding pseudo tree.

- **Impact of the mini-bucket i-bound on AND/OR search.** Our results show conclusively that when enough memory is available static mini-bucket heuristics with relatively large i-bounds are cost effective (*e.g.*, genetic linkage analysis networks from Table 2, Mastermind networks from Table 5). However, if space is restricted, dynamic mini-bucket heuristics, which exploit the partial assignment along the search path, appear to be superior. This occurs for small i-bounds when the dynamic heuristics are more accurate then the static ones.
- **Impact of dynamic variable ordering.** Our dynamic AND/OR search approach was shown to be powerful in conjunction with local consistency based heuristics. The AND/OR Branch-and-Bound algorithms with EDAC heuristics and dynamic variable orderings were sometimes by two orders of magnitude better than their static counterparts (*e.g.*, the 503 SPOT5 network from Table 6).
- **Impact of determinism.** When the graphical model contains both deterministic information (hard constraints) as well as general cost functions or probabilities, we demonstrated that it is beneficial to exploit the computational power of the constraints explicitly, via constraint propagation methods. Our experiments on selected classes of deterministic Bayesian networks showed that enforcing unit resolution over the CNF encoding of the determinism present in the network yielded a tremendous reduction in running time for the corresponding AND/OR algorithms (*e.g.*, deterministic grid networks from Table 3).
- **Impact of static variable ordering.** The performance of the AND/OR search algorithms is highly influenced 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 yield small induced width but significantly larger depth, whereas the hypergraph partitioning heuristic produces much smaller depth trees but with larger induced widths. Our experiments demonstrated indeed that the AND/OR algorithms using mini-bucket heuristics benefit, on average, from the min-fill based pseudo trees because the guiding mini-bucket heuristic is sensitive to the induced width size. In some exceptional cases however, the hypergraph partitioning based pseudo trees were able to improve significantly the search performance, especially for relatively small i-bounds, because in those cases the smaller depth guarantees a smaller AND/OR search tree. The picture is reversed for local consistency based heuristics which are not sensitive to the induced width. Here, the hypergraph based trees were able to improve performance by up to 3 orders of magnitude over the min-fill based trees (*e.g.*, SPOT5 networks from Fig. 16).

Clearly, there are various ways for improvements. For instance, one could incorporate good initial upper bound techniques (using incomplete schemes), apply additional schemes for exploiting determinism or use improved mini-bucket schemes. For example, the recent improvement of the Mini-Bucket algorithm, called *Depth-First Mini-Bucket Elimination* [74], could be explored further in the context of AND/OR search.

Acknowledgements

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

References

[1] R. Dechter, R. Mateescu, AND/OR search spaces for graphical models, Artificial Intelligence 171 (1) (2007) 73–106.
[2] E. Freuder, M. Quinn, Taking advantage of stable sets of variables in constraint satisfaction problems, in: International Joint Conference on Artificial Intelligence (IJCAI-1985), 1985, pp. 1076–1078.
[3] R. Bayardo, D. Miranker, On the space–time trade-off in solving constraint satisfaction problems, in: International Joint Conference on Artificial Intelligence (IJCAI-1995), 1995, pp. 558–562.
[4] R. Dechter, I. Rish, Mini-buckets: A general scheme for approximating inference, Journal of ACM 2 (50) (2003) 107–153.
[5] K. Kask, R. Dechter, A general scheme for automatic generation of search heuristics from specification dependencies, Artificial Intelligence 129 (1–2) (2001) 91–131.
[6] J. Pearl, Probabilistic Reasoning in Intelligent Systems, Morgan Kaufmann, 1988.
[7] S. Bistarelli, U. Montanari, F. Rossi, Semiring based constraint solving and optimization, Journal of ACM 44 (2) (1997) 309–315.
[8] Nils J. Nilsson, Principles of Artificial Intelligence, Tioga, 1980.
[9] K. Kask, R. Dechter, J. Larrosa, A. Dechter, Unifying cluster-tree decompositions for reasoning in graphical models, Artificial Intelligence 166 (1–2) (2005) 225–275.
[10] S. Minton, M.D. Johnston, A.B. Philips, P. Laired, Solving large scale constraint satisfaction and scheduling problems using heuristic repair methods, in: National Conference on Artificial Intelligence (AAAI-1990), 1990, pp. 17–24.
[11] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: National Conference on Artificial Intelligence (AAAI-1992), 1992, pp. 440–446.
[12] R. Wallace, Analysis of heuristic methods for partial constraint satisfaction problems, in: Principles and Practice of Constraint Programming (CP-1996), 1996, pp. 482–496.
[13] E. Freuder, R. Wallace, Partial constraint satisfaction, Artificial Intelligence 58 (1–3) (1992) 21–70.
[14] G. Verfaillie, M. Lemaitre, T. Schiex, Russian doll search for solving constraint optimization problems, in: National Conference on Artificial Intelligence (AAAI), 1996, pp. 298–304.
[15] J. Larrosa, P. Meseguer, Partition-based lower bound for MAX-CSP, in: Principles and Practice of Constraint Programming (CP-1999), 1999, pp. 303–315.
[16] J. Larrosa, T. Schiex, In the quest of the best form of local consistency for weighted CSPs, in: National Conference of Artificial Intelligence (AAAI-2003), 2003, pp. 631–637.
[17] J. Larrosa, T. Schiex, Solving weighted CSP by maintaining arc consistency, Artificial Intelligence 159 (1–2) (2004) 1–26.

[18] S. de Givry, F. Heras, J. Larrosa, M. Zytnicki, Existential arc consistency: Getting closer to full arc consistency in weighted CSPs, in: International Joint Conference in Artificial Intelligence (IJCAI-2005), 2005, pp. 84–89.
[19] P. Shenoy, G. Shafer, Propagating belief functions with local computations, IEEE Expert 4 (1) (1986) 43–52.
[20] F.V. Jensen, S. Lauritzen, K. Olesen, Bayesian updating in recursive graphical models by local computation, Computational Statistics Quarterly 4 (1) (1990) 269–282.
[21] R. Dechter, Bucket elimination: A unifying framework for reasoning, Artificial Intelligence 113 (1–2) (1999) 41–85.
[22] Y. Peng, J.A. Reggia, A connectionist model for diagnostic problem solving, IEEE Transactions on Systems, Man and Cybernetics (1989).
[23] K. Kask, R. Dechter, Stochastic local search for Bayesian networks, in: Workshop on AI and Statistics (AI-STAT-1999), 1999, pp. 113–122.
[24] J. Park, Using weighted MAX-SAT engines to solve MPE, in: National Conference of Artificial Intelligence (AAAI-2002), 2002, pp. 682–687.
[25] F. Hutter, H. Hoos, T. Stutzle, Efficient stochastic local search for MPE solving, in: International Joint Conference on Artificial Intelligence (IJCAI-2005), 2005, pp. 169–174.
[26] S.E. Shimony, E. Charniak, A new algorithm for finding MAP assignments to belief networks, in: Uncertainty in Artificial Intelligence (UAI-1991), 1991, pp. 185–193.
[27] E. Santos, On the generation of alternative explanations with implications for belief revision, in: Uncertainty in Artificial Intelligence (UAI-1991), 1991, pp. 339–347.
[28] Z. Li, B. DAmbrosio, An efficient approach for finding the MPE in belief networks, in: Uncertainty in Artificial Intelligence (UAI-1993), 1993, pp. 342–349.
[29] B.K. Sy, Reasoning MPE to multiply connected belief networks using message-passing, in: National Conference of Artificial Intelligence (AAAI-1992), 1992, pp. 570–576.
[30] R. Marinescu, K. Kask, R. Dechter, Systematic vs non-systematic algorithms for solving the MPE task, in: Uncertainty in Artificial Intelligence (UAI-2003), 2003, pp. 394–402.
[31] A. Choi, M. Chavira, A. Darwiche, Node splitting: A scheme for generating upper bounds in Bayesian networks, in: Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI), 2007, pp. 57–66.
[32] E. Lawler, D. Wood, Branch-and-bound methods: A survey, Operations Research 14 (4) (1966) 699–719.
[33] R. Dechter, K. Kask, J. Larrosa, A general scheme for multiple lower bound computation in constraint optimization, in: Principles and Practice of Constraint Programming (CP), 2001, pp. 346–360.
[34] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison–Wesley, 1984.
[35] L. Kanal, V. Kumar, Search in Artificial Intelligence, Springer-Verlag, 1988.
[36] M. Cooper, T. Schiex, Arc consistency for soft constraints, Artificial Intelligence 154 (1–2) (2003) 199–227.
[37] R. Dechter, Constraint Processing, MIT Press, 2003.
[38] R. Haralick, G. Elliot, Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence 14 (3) (1980) 263–313.
[39] C. Bessiere, J.-C. Regin, MAC and combined heuristics: two reasons to forsake FC (and CBJ) on hard problems, in: Principles and Practice of Constraint Programming (CP-1996), 1996, pp. 61–75.
[40] D. Brelaz, New method to color the vertices of a graph, Communications of the ACM 4 (22) (1979) 251–256.
[41] C. Lecoutre, F. Boussemart, F. Hemery, L. Sais, Boosting systematic search by weighting constraints, in: European Conference on Artificial Intelligence (ECAI-2004), 2004, pp. 146–150.
[42] C. Lecoutre, F. Boussemart, F. Hemery, Backjump-based techniques versus conflict directed heuristics, in: International Conference on Tools with Artificial Intelligence (ICTAI-2004), 2004, pp. 549–557.
[43] J. Huang, A. Darwiche, A structure-based variable ordering heuristic, in: International Joint Conference on Artificial Intelligence (IJCAI-2003), 2003, pp. 1167–1172.
[44] W. Li, P. van Beek, Guiding real-world SAT solving with dynamic hypergraph separator decomposition, in: International Conference on Tools with Artificial Intelligence (ICTAI-2004), 2004, pp. 542–548.
[45] R. Bayardo, J.D. Pehoushek, Counting models using connected components, in: National Conference of Artificial Intelligence (AAAI-2000), 2000, pp. 157–162.
[46] T. Sang, P. Beame, H. Kautz, A dynamic approach to MPE and weighted MAX-SAT, in: International Joint Conference on Artificial Intelligence (IJCAI-2007), 2007, pp. 549–557.
[47] R. Marinescu, AND/OR search strategies for combinatorial optimization in graphical models, PhD thesis, University of California, Irvine, 2008.
[48] R. Marinescu, R. Dechter, AND/OR branch-and-bound search for combinatorial optimization in graphical models, Technical report, University of California, Irvine, 2008.
[49] A. Darwiche, Recursive conditioning, Artificial Intelligence 126 (1–2) (2001) 5–41.
[50] H. Bodlaender, J. Gilbert, Approximating treewidth, pathwidth and minimum elimination tree-height, Technical report, Utrecht University, 1991.
[51] U. Kjæaerulff, Triangulation of graph-based algorithms giving small total space, Technical report, University of Aalborg, Denmark, 1990.
[52] T. Sang, P. Beame, H. Kautz, Solving Bayesian networks by weighted model counting, in: National Conference of Artificial Intelligence (AAAI-2005), 2005, pp. 475–482.
[53] J. Ott, Analysis of Human Genetic Linkage, The Johns Hopkins University Press, 1999.
[54] M. Fishelson, D. Geiger, Exact genetic linkage computations for general pedigrees, in: International Conference on Intelligent Systems for Molecular Biology, 2002, pp. 189–198.
[55] M. Fishelson, N. Dovgolevsky, D. Geiger, Maximum likelihood haplotyping for general pedigrees, Human Heredity 59 (1) (2005) 41–60.
[56] R. Dechter, D. Larkin, Hybrid processing of beliefs and constraints, in: Uncertainty in Artificial Intelligence (UAI-2001), 2001, pp. 112–119.
[57] D. Larkin, R. Dechter, Bayesian inference in the presence of determinism, in: Artificial Intelligence and Statistics (AISTAT-2003), 2003.
[58] D. Allen, A. Darwiche, New advances in inference using recursive conditioning, in: Uncertainty in Artificial Intelligence (UAI-2003), 2003, pp. 2–10.
[59] R. Dechter, R. Mateescu, Mixtures of deterministic-probabilistic networks, in: Uncertainty in Artificial Intelligence (UAI), 2004, pp. 120–129.
[60] I. Rish, R. Dechter, Resolution vs. search: Two strategies for SAT, Journal of Automated Reasoning 24 (1–2) (2000) 225–275.
[61] T. Walsh, SAT vs CSP, in: Principles and Practice of Constraint Programming (CP), 2000, pp. 441–456.
[62] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: Design Automation Conference (DAC-2001), 2001.
[63] E. Bensana, M. Lemaitre, G. Verfaillie, Earth observation satellite management, Constraints 4 (3) (1999) 293–299.
[64] M. Chavira, A. Darwiche, M. Jaeger, Compiling relational Bayesian networks for exact inference, International Journal of Approximate Reasoning 42 (1–2) (2006) 4–20.
[65] F. Bacchus, S. Dalmao, T. Pittasi, Value elimination: Bayesian inference via backtracking search, in: Uncertainty in Artificial Intelligence (UAI-2003), 2003, pp. 20–28.
[66] J. Larrosa, P. Meseguer, M. Sanchez, Pseudo-tree search with soft constraints, in: European Conference on Artificial Intelligence (ECAI-2002), 2002, pp. 131–135.
[67] R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition, Artificial Intelligence 41 (3) (1990) 273–312.

[68] Z. Collin, R. Dechter, S. Katz, On the feasibility of distributed constraint satisfaction, in: International Joint Conference on Artificial Intelligence (IJCAI-1991), 1991, pp. 318–324.
[69] P. Modi, W. Shen, M. Tambe, M. Yokoo, An asynchronous complete method for distributed constraint optimization, in: International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003), 2003, pp. 161–168.
[70] P. Modi, W. Shen, M. Tambe, M. Yokoo, ADOPT: Asynchronous distributed constraint optimization with quality guarantees, Artificial Intelligence 161 (1–2) (2005) 149–180.
[71] W. Yeoh, A. Felner, S. Koening, BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm, in: International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2008), 2008, pp. 591–598.
[72] A. Petcu, B. Faltings, DPOP: A scalable method for multiagent constraint optimization, in: International Joint Conference on Artificial Intelligence (IJCAI-2005), 2005, pp. 266–271.
[73] P. Jegou, C. Terrioux, Decomposition and good recording for solving MAX-CSPs, in: European Conference on Artificial Intelligence (ECAI-2004), 2004, pp. 196–200.
[74] E. Rollon, J. Larrosa, Depth-first mini-bucket elimination, in: Principles and Practice of Constraint Programming (CP-2005), 2005, pp. 563–577.