Journal of Artificial Intelligence Research 55 (2016) 889-952 Submitted 9/15; published 4/16

# Searching for the M Best Solutions in Graphical Models

**Natalia Flerova** NFLEROVA@UCI.EDU
*University of California, Irvine*
*Irvine, CA 92697, USA*

**Radu Marinescu** RADU.MARINESCU@IE.IBM.COM
*IBM Research – Ireland*

**Rina Dechter** DECHTER@UCI.EDU
*University of California, Irvine*
*Irvine, CA 92697, USA*

## Abstract

The paper focuses on finding the $m$ best solutions to combinatorial optimization problems using best-first or depth-first branch and bound search. Specifically, we present a new algorithm m-A*, extending the well-known A* to the $m$-best task, and for the first time prove that all its desirable properties, including soundness, completeness and optimal efficiency, are maintained. Since best-first algorithms require extensive memory, we also extend the memory-efficient depth-first branch and bound to the $m$-best task.

We adapt both algorithms to optimization tasks over graphical models (e.g., Weighted CSP and MPE in Bayesian networks), provide complexity analysis and an empirical evaluation. Our experiments confirm theory that the best-first approach is largely superior when memory is available, but depth-first branch and bound is more robust. We also show that our algorithms are competitive with related schemes recently developed for the $m$-best task.

## 1. Introduction

The usual aim of combinatorial optimization is to find an optimal solution, minimum or maximum, of an objective function. However, in many applications it is desirable to obtain not just a single optimal solution, but a set of the first $m$ best solutions for some integer $m$. We are motivated by many real-life domains, in which such task arises. For instance, a problem of finding the most likely haplotype in a pedigree can be presented as finding the most probable assignment in a Bayesian network that encodes the genetic information (Fishelson, Dovgolevsky, & Geiger, 2005). In practice the data is often corrupted or missing, which makes the single optimal solution unreliable. It is possible to increase the confidence in the answer by finding a set of the $m$ best solutions and then choosing the final solution with an expert help or by obtaining additional genetic data. More examples of the $m$-best tasks arise in procurement auction problems and in probabilistic expert systems, where certain constraints often cannot be directly incorporated into the model, either because they make the problem infeasibly complex or they are too vague to formalize (e.g. idiosyncratic preferences of a human user). Thus in such domains it may be more practical to first find several good solutions to a relaxed problem and then pick the one that satisfies all additional constraints in a post-processing manner. Additionally, sometimes a set of diverse assignments with approximately the same cost is required, as in reliable communication network design. Finally, in the context of summation problem over graphical models, such as probability of evidence or the partition function, an approximation can be derived by summing over the $m$ most likely tuples.

The problem of finding the $m$ best solutions has been well studied. One of the earliest and most influential works belongs to Lawler (1972). He provided a general scheme that extends any optimization algorithm to the $m$-best task. The idea is to compute the next best solution successively by finding a single optimal solution for a slightly different reformulation of the original problem that excludes the solutions generated so far. This approach has been extended and improved over the years and is still one of the primary strategies for finding the $m$ best solutions. Other approaches are more direct, trying to avoid the repeated computation inherent to Lawler's scheme. Two earlier works that are most relevant and provide the highest challenge to our work are by Nilsson (1998) and Aljazzar and Leue (2011).

- Nilsson proposed a junction-tree based message-passing scheme that iteratively finds the $m$ best solutions. He claimed that it has the best runtime complexity among $m$-best schemes for graphical models. Our analysis (Section 6) shows that indeed Nilsson's scheme has the second best worst case time complexity after our algorithm BE+m-BF (Section 5.3). However, in practice this scheme is not feasible for problems having a large induced width.

- In their recent work Aljazzar and Leue proposed an algorithm called K*, an A* search-style scheme for finding the $k$ shortest paths that is interleaved with breadth-first search. They used a specialized data structure and it is unclear if this approach can be straightforwardly extended to graphical models, a point that we will leave to future work.

One of the popular approximate approaches to solving optimization problems is based on the LP-relaxation of the problem (Wainwright & Jordan, 2003). The $m$-best extension of this approach (Fromer & Globerson, 2009) does not guarantee exact solutions, but is quite efficient in practice. We will discuss these and other previous works further in Section 6.

Our main focus lies in optimization in the context of graphical models, such as Bayesian networks, Markov networks and constraint networks. However, some of the algorithms developed can be used for more general purpose tasks, such as finding $m$ shortest paths in a graph. Various graph-exploiting algorithms for solving optimization tasks over graphical models were developed in the past few decades. Such algorithms are often characterized as being either of *inference* type (e.g., message-passing schemes, variable elimination) or of *search* type (e.g., AND/OR search or recursive-conditioning). In our earlier works, (e.g., Flerova, Dechter, & Rollon, 2011), we extended inference schemes as represented by the bucket elimination algorithm (BE) (Dechter, 1999, 2013) to the task of finding the $m$ best solutions. However, due to their large memory requirements, variable elimination algorithms, including bucket elimination, cannot be used in practice for finding exact solutions to combinatorial optimization tasks when the problem's graph is dense. Depth-first branch and bound (DFBnB) and best-first search (BFS) are more flexible and can trade space for time. Our work explores the question of solving the $m$ best solutions task using the heuristic search schemes.

Our contribution lies in extending the heuristic algorithms to the m best solutions task. We describe general purpose $m$-best variants of both depth-first branch and bound and best-first search, more specifically A*, yielding algorithms m-BB and m-A* respectively, and analyze their properties. We show that m-A* inherits all A*'s desirable properties (Dechter & Pearl, 1985), most significantly it is optimally efficient compared to any alternative exact search-based scheme. We also discuss the size of the search space explored by m-BB. We then extend our new $m$-best algorithms to graphical models by exploring the AND/OR search space.

We evaluate the resulting algorithms on 6 benchmarks having more than 300 instances in total, and examine the impact of the number of solutions $m$ on the algorithms' behaviour. In particular, we observe that the runtime of the most of the schemes (except for the depth-first branch and bound exploring an AND/OR tree) scales much better with $m$ than what worst case theoretical analysis suggests.

We also show that a m-A* search using the exact bucket elimination heuristic (a scheme we call BE+m-BF) is highly efficient on easier problems but suffers severely from memory issues over denser graphs, far more than the A*-based schemes using approximate mini-bucket heuristics. Finally, we compare our schemes with some of the most efficient algorithms based on the LP-relaxation (Fromer & Globerson, 2009; Batra, 2012), showing competitiveness and even superiority for large values of $m$ ($m \geq 10$), while providing optimality guarantees.

The paper is organized as follows. In Section 2 we provide relevant background. Section 3 presents the extension of best-first search to the $m$-best task. In particular, we define m-A*, the extension of A* algorithm to finding the $m$ best solutions (3.1), and prove its main properties (3.2). Section 4 describes algorithm m-BB, an extension of depth-first branch and bound algorithm to solving the $m$-best task. In Section 5 we discuss the adaptation of the two newly proposed $m$-best search algorithms for AND/OR search spaces over graphical models, including a hybrid method BE+m-BF that incorporates both variable elimination and heuristic search. Section 6 elaborates on the related work and contrasts it with our methods. Section 7 presents the empirical evaluation of our $m$-best schemes and Section 8 concludes.

## 2. Background

We begin by formally defining the graphical models framework and providing background on heuristic search.

### 2.1 Graphical Models

We denote variables by upper-case letters (e.g., $X, Y, Z$) and their values of variables by lower-case letters (e.g., $x, y, z$). Sets of variables are denoted by upper-case letters in bold (e.g. $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$). The assignment $(X_1 = x_1, \ldots, X_n = x_n)$ can be abbreviated as $\mathbf{x} = (x_1, \ldots, x_n)$.

We denote functions by letters $f, g, h$ etc., and a set of functions by $\mathbf{F}$. A function $f$ over a scope $\mathbf{S}_1 = \{X_1, \ldots, X_r\}$ is denoted by $f_{\mathbf{S}_1}$. The summation operator $\sum_{\mathbf{x} \in \mathbf{X}}$ defines a sum over all possible values of variables in $\mathbf{X}$, namely $\sum_{x_1 \in X_1}, \ldots, \sum_{x_n \in X_n}$. Minimization $\min_{\mathbf{x} \in \mathbf{X}}$ and maximization $\max_{\mathbf{x} \in \mathbf{X}}$ operators are defined in a similar manner. Note that we use terms elimination and marginalization interchangeably. For convenience we sometimes use $\min_{\mathbf{x}}$ ($\max_{\mathbf{x}}$, $\sum_{\mathbf{x}}$) to denote $\min_{\mathbf{x} \in \mathbf{X}}$ ($\max_{\mathbf{x} \in \mathbf{X}}$, $\sum_{\mathbf{x} \in \mathbf{X}}$).

A graphical model is a collection of local functions over subsets of variables that conveys probabilistic, deterministic, or preferential information, and whose structure is described by a graph. The graph captures independencies or irrelevance information inherent in the model, that can be useful for interpreting the modeled data and, most significantly, can be exploited by reasoning algorithms. The set of local functions can be combined in a variety of ways to generate a global function, whose scope is the set of all variables.

**DEFINITION 1 (Graphical model).** *A* graphical model $\mathcal{M}$ *is a 4-tuple* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes \rangle$*:*

1. $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a finite 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 non-negative real-valued discrete functions, defined over scopes of variables* $\mathbf{S}_i \subseteq \boldsymbol{X}$. *They are called* local *functions.*

4. $\bigotimes$ *is a combination operator, e.g.,* $\bigotimes \in \{\prod, \sum\}$ *(product, sum)*

*The graphical model represents a* global function*, whose scope is* $\boldsymbol{X}$ *and which is the combination of all the local functions:* $\bigotimes_{j=1}^{r} f_j$.

When $\bigotimes = \sum$ and $f_i : \mathbf{D}_{S_i} \to \mathbb{N}$ we have Weighted Constraint Satisfaction Problems (WCSPs). When $\bigotimes = \prod$ and $f_i = P_i(X_i \mid \mathbf{pa}_i)$ we have a Bayesian network. The probabilities $\mathbf{P}$ are defined relative to a directed acyclic graph $G$ over $\mathbf{X}$, where the set $X_{i_1}, \ldots, X_{i_k}$ are the parents $\mathbf{pa}_i$ of $X_i$, i.e. for each $X_{i_j}$ there is an edge pointing from $X_{i_j}$ to $X_i$. For illustration, consider the Bayesian network with 5 variables whose directed acyclic graph (DAG) is given in Figure 1(a).

The **most common optimization task** for Bayesian network is the *most probable explanation* (MPE) also known as *maximum a posteriori hypothesis* (MAP),[1] where the goal is to compute the optimal value

$$C^* = \max_{\mathbf{x}} \prod_{j=1}^{r} f_j(\mathbf{x}_{\mathbf{S}_j})$$

and its optimizing configuration

$$\mathbf{x}^* = \operatorname*{argmax}_{\mathbf{x}} \prod_{j=1}^{r} f_j(\mathbf{x}_{\mathbf{S}_j})$$

The related task, typical for WCSP, is the min-sum, namely computing a minimal cost assignment (min-sum): $C^* = \min_{\mathbf{x}} \sum_j f_j(\mathbf{x})$ and the optimizing configuration $\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x}} \sum_j f_j(\mathbf{x})$. Historically this task is also sometimes referred to as *energy minimization*. It is equivalent to an MPE/MAP task in the following sense: if $C^*_{max} = \max_{\mathbf{x}} \prod_j f_j(\mathbf{x})$ is a solution to an MPE problem, then $C^*_{max} = \exp(-C^*_{min})$, where $C^*_{min}$ is a solution to a min-sum problem $C^*_{min} = \min_{\mathbf{x}} \sum_j g_j(\mathbf{x})$ and $\forall j, g_j(\mathbf{x}) = -\log(f_j(\mathbf{x}))$.

A graphical model defines the **primal graph** that captures dependencies between the problem's variables. It has variables as its vertices. An edge connects any two vertices whose variables appear in the scope of the same function. An important property of a graphical model, characterizing the complexity of its reasoning tasks is the *induced width*. An *ordered graph* is a pair $(G, o)$ where $G$ is an undirected graph, and $o = (X_1, \ldots, X_n)$ is an ordering of nodes. The *width of a node* is the number of the node's neighbors that precede it in the ordering. The *width of a graph along an ordering* $o$ is the maximum width over all nodes. An *induced ordered graph* is obtained from an ordered graph as follows: nodes are processed from last to first based on $o$; when node $X_j$ is processed, all its preceding neighbors are connected. The width of an ordered induced graph along the ordering $o$ is called *induced width along* $o$ and is denoted by $w^*(o)$. The *induced width* of a graph, denoted by $w^*$, is the minimal induced width over all its orderings. Abusing notation we sometimes use $w^*$ to denote the induced width along a particular ordering, when the meaning is clear from the context.

Figure 1(b) depicts the primal graph of the Bayesian network from Figure 1(a). Figures 1(c) and 1(d) show the induced graphs of the primal graph from Figure 1(a) respectively along the orderings

1. In some communities MAP also refers to the task of optimizing a partial assignment to the variables. However, in this paper we use MAP and MPE as interchangeable, both referring to an optimal full variable assignment.

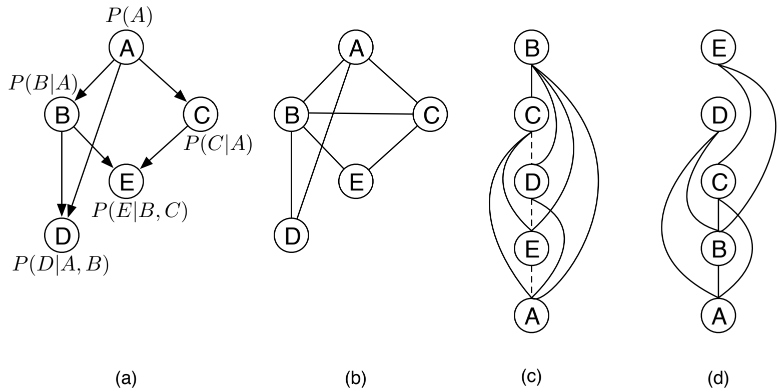



Figure 1: (a) A DAG of a Bayesian network, (b) its primal graph (also called moral graph), (c) its induced graph along $o = (A, E, D, C, B)$, and (d) its induced graph along $o = (A, B, C, D, E)$, example by Gogate (2009).

$o = (A, E, D, C, B)$ and $o' = (A, B, C, D, E)$. The dashed lines in Figure 1(c) represent the induced edges, namely edges that are absent from the moral graph, but were introduced in the induced graph. We can see that the induced width along ordering $o$ is $w^*(o) = 4$ and the induced width along ordering $o'$ is $w^*(o') = 2$, respectively.

## 2.2 Heuristic Search

Our analysis focuses on *best-first search* (BFS), whose behaviour for the task of finding a single optimal solution is well understood. Assuming a minimization task, best-first search always expands the node with the *best* (i.e., smallest) value of the heuristic evaluation function. It maintains a graph of explored paths, a list CLOSED of expanded nodes and a frontier of OPEN nodes. BFS chooses from OPEN a node $n$ with the smallest value of a heuristic evaluation function $f(n)$, expands it by generating its successors $succ(n)$, places it on CLOSED, and places $succ(n)$ in OPEN. The most popular variant of best-first search, A*, uses the heuristic evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the root to $n$, and $h(n)$ is a *heuristic function* that estimates the optimal cost to go $h^*(n)$ from $n$ to a goal node. A heuristic function is called *admissible* if it never overestimates (for minimization) the true minimal cost to reach the goal $h^*(n)$. Namely, $\forall n\ h(n) \leq h^*(n)$. A heuristic is called *consistent* or *monotonic*, if for every node $n$ and for every successor $n'$ of $n$ the following inequality holds: $h(n) \leq c(n, n') + h(n')$. If $h(n)$ is consistent, then the values of evaluation function $f(n)$ along any path are non-decreasing. It is known that regardless of the tie-breaking rule A* expands any node $n$ reachable by a strictly $C^*$-bounded path from the root, and such a node is referred to as *surely expanded by A** (Dechter & Pearl, 1985). A path $\pi$ is $C^*$*-bounded* relative to $f$, if $\forall n \in \pi : f(n) < C^*$, where $C^*$ is the cost of optimal solution.

A* search has a number of attractive properties (Nillson, 1980; Pearl, 1984; Dechter & Pearl, 1985):

- **Soundness and completeness**: A* terminates with the optimal solution.
- When $h$ is consistent, A* explores only the set of nodes $S = \{n|f(n) \leq C^*\}$ and it surely expands all the nodes having $S = \{n|f(n) < C^*\}$.
- **Optimal efficiency under consistent heuristic**: When $h$ is consistent, any node surely expanded by A* must be expanded by any other sound and complete search algorithm using the same heuristic information.
- **Optimal efficiency for node expansions**: When the heuristic function is consistent, A*, when searching a graph, expands each node at most once, and at the time of node's expansion A* has found the shortest path to it.
- **Dominance**: Given two heuristic functions $h_1$ and $h_2$, s.t. $\forall n\, h_1(n) < h_2(n)$, $A_1^*$ will expand every node surely expanded by $A_2^*$, where $A_i^*$ uses heuristic $h_i$.

Although best-first search is known to be the best algorithm in terms of number of nodes expanded (Dechter & Pearl, 1985), it requires exponential memory in the worst-case.

A popular alternative is the *depth-first branch and bound* (DFBnB), whose most attractive feature, compared to best-first search, is that it can be executed with linear memory. Yet, when the search space is a graph, it can exploit memory to improve its performance by flexibly trading space and time. Depth-first branch and bound expands nodes in a depth-first manner, maintaining the cost of the best solution found so far which is an upper bound $UB$ on the cost of the optimal solution. If the heuristic evaluation function of the current node $n$ is greater or equal to the upper bound, the node is *pruned* and the subtree below it is never explored. In the worst case depth-first branch and bound explores the entire search space. In the best case the first solution found is optimal, in which case its performance can be as good as BFS. However, if the solution depth is unbound, depth-first search might follow an infinite branch and never terminate. Also, if the search space is a graph, DFBnB may expand nodes numerous time, unless it uses caching and checks for duplicates.

### 2.3 Search in Graphical Models

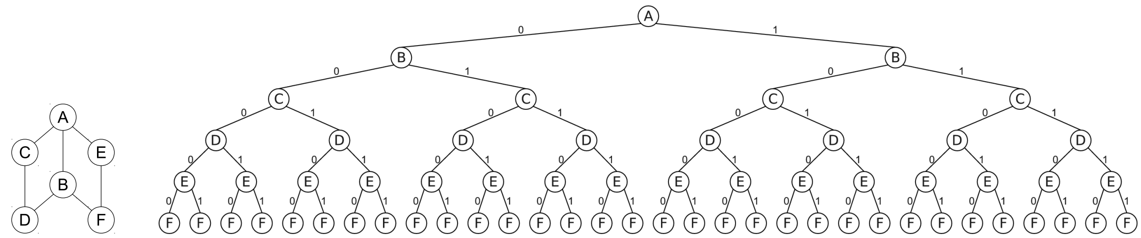
Search algorithms provide a way to systematically enumerate all possible assignments of a given graphical model. Optimization problems over graphical models can be naturally presented as the task of finding an optimal cost path in an appropriate search space.

The simplest variant of a search space is the so-called *OR search tree*. Each level corresponds to a variable from the original problem. The nodes correspond to partial variable assignments and the arc weights are derived from problems' input functions. The size of such a search tree is bounded by $O(k^n)$, where $n$ is the number of variables and $k$ is the maximum domain size.

Throughout this section we are going to illustrate the concepts using an example problem with six variables $\{A, B, C, D, E, F\}$ and six pairwise functions. Its primal graph is shown in Figure 2(a). Figure 2(b) displays the OR search tree corresponding to the lexicographical ordering.

#### 2.3.1 AND/OR SEARCH SPACES

OR search trees are blind to the problem decomposition encoded in graphical models and can therefore be inefficient. They do not exploit the independencies in the model. AND/OR search spaces



(a) Primal graph

(b) OR search tree along ordering A, B, C, D, E, F

Figure 2: An example problem with 6 variables $\{A, B, C, D, E, F\}$ and 5 pairwise functions.

for graphical models have been introduced to better capture the problem structure (Dechter & Mateescu, 2007). The AND/OR search space is defined relative to a *pseudo tree* of the primal graph that captures problem decomposition. Figure 3(a) shows the pseudo tree of the example problem.

**DEFINITION 2.** *A* pseudo tree *of an undirected graph $G = (V, E)$ is a directed rooted tree $\mathcal{T} = (V, E')$, such that every arc of G not included in $E'$ is a back-arc in $\mathcal{T}$, namely it connects a node in $\mathcal{T}$ to an ancestor in $\mathcal{T}$. The arcs in $E'$ may not all be included in $E$.*

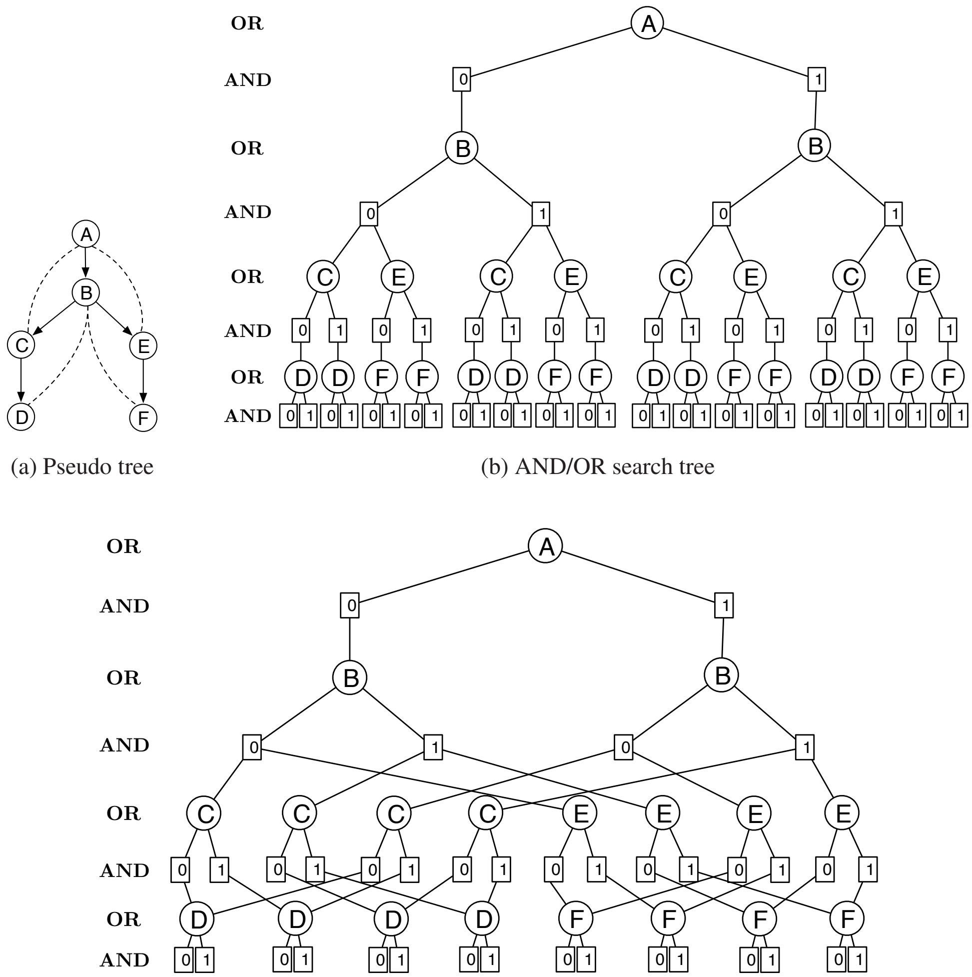
Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ with primal graph $G$ and a pseudo tree $\mathcal{T}$ of $G$, the *AND/OR search tree* $S_{\mathcal{T}}$ contains alternating levels of OR and AND nodes. Its structure is based on the underlying pseudo tree $\mathcal{T}$. The root node of $S_{\mathcal{T}}$ is an OR node labelled by the variable at the root of $\mathcal{T}$. The children of an OR node $X_i$ are AND nodes labelled with value assignments $\langle X_i, x_i \rangle$ (or simply $\langle x_i \rangle$). The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labelled with the children of $X_i$ in $\mathcal{T}$, representing conditionally independent subproblems. An AND/OR tree corresponding to the pseudo tree in Figure 3(a) is shown in Figure 3(b). The arcs from nodes $X_i$ to $\langle X_i, x_i \rangle$ in an AND/OR search tree are annotated by the weights derived from the cost functions in $\mathbf{F}$:

**DEFINITION 3** (arc weight)**.** *The weight $w(X_i, x_i)$ of the arc $(X_i, \langle X_i, x_i \rangle)$ is the combination (i.e. sum for WCSP and product for MPE) of all the functions, whose scope includes $X_i$ and is fully assigned along the path from root to the node corresponding to $\langle X_i, x_i \rangle$, evaluated at all values along the path.*

Some identical subproblems can be identified by their *context* (namely, a partial instantiation of their ancestors that separates the subproblem from the rest of the problem graph), can be merged, yielding an *AND/OR search graph* (Dechter & Mateescu, 2007). Merging all context-mergeable nodes yields the *context-minimal AND/OR search graph*, denoted by $C_{\mathcal{T}}$. An example can be seen in Figure 3(c). The size of the context-minimal AND/OR search graph can be shown to be exponential in the induced width of $G$ along the pseudo tree $\mathcal{T}$ (Dechter & Mateescu, 2007).

A *solution tree* $T$ of $C_{\mathcal{T}}$ is a subtree such that: (1) it contains the root node of $C_{\mathcal{T}}$; (2) if an internal AND node $n$ is in $T$, then all its children are in $T$; (3) if an internal OR node $n$ is in $T$, then exactly one of its children is in $T$; (4) every tip node in $T$ (i.e., nodes with no children) is a terminal node. The cost of a solution tree is the product, for MPE or sum for WCSP, of the weights associated with its arcs.

Each node $n$ in $C_{\mathcal{T}}$ is associated with a *value* $v(n)$ capturing the optimal solution cost of the conditioned subproblem rooted at $n$. Assuming an MPE/MAP problem, it was shown that $v(n)$ can




(a) Pseudo tree

(b) AND/OR search tree

(c) Context-minimal AND/OR search graph

Figure 3: AND/OR search spaces for graphical models.

be computed recursively based on the values of $n$'s successors: OR nodes by maximization, AND nodes by multiplication. For WCSPs, $v(n)$ for OR and AND nodes is updated by minimization and by summation, respectively (Dechter & Mateescu, 2007).

We next provide an overview of a depth-first branch and bound and best-first search algorithms, that explore AND/OR search spaces (Marinescu & Dechter, 2009b, 2009a; Otten & Dechter, 2011). These schemes use heuristics generated either by the mini-bucket elimination scheme (2.3.4) or through soft arc-consistency schemes (Marinescu & Dechter, 2009a, 2009b; Schiex, 2000; Darwiche, Dechter, Choi, Gogate, & Otten, 2008) or their composite (Ihler, Flerova, Dechter, & Otten, 2012). As it is customary in the heuristic search literature, when defining search algorithms we assume without loss of generality a minimization task (i.e., min-sum optimization problem).

**Algorithm 1:** AOBF exploring the AND/OR search tree (Marinescu & Dechter, 2009b)

**Input**: A graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, pseudo tree $\mathcal{T}$ rooted at $X_1$, heuristic function $h(\cdot)$<br>
**Output**: Optimal solution to $\mathcal{M}$<br>
1 Create root OR node $s$ labelled by $X_1$ and let $\mathcal{G}$ (explored search space) $= \{s\}$;<br>
2 Initialize $v(s) = h(s)$ and best partial solution tree $T^*$ to $\mathcal{G}$;<br>
3 **while** *$s$ is not SOLVED* **do**<br>
4     Select non-terminal tip node $n$ in $T^*$. If there is no such node then **break**;<br>
5     **if** *$n$ is an OR node labeled $X_i$* **then**<br>
6         **forall the** $x_i \in D(X_i)$ **do**<br>
7             Create AND child $n' = \langle X_i, x_i \rangle$;<br>
8             **if** *$n'$ is TERMINAL* **then**<br>
9                 Mark $n'$ SOLVED;<br>
10             $succ(n) \leftarrow succ(n) \cup n'$;<br>
11     **else if** *$n$ is an AND node labeled $\langle X_i, x_i \rangle$* **then**<br>
12         **forall the** *successor $X_j$ of $X_i$ in $\mathcal{T}$* **do**<br>
13             Create OR child $n' = X_j$;<br>
14             $succ(n) \leftarrow succ(n) \cup n'$;<br>
15     Initialize $v(n') = h(n')$ for all new nodes;<br>
16     Add new nodes to the explores search space $\mathcal{G} \leftarrow \mathcal{G} \cup \{succ(n)\}$;<br>
17     Let $S \leftarrow \{n\}$;<br>
18     **while** $S \neq \emptyset$ **do**<br>
19         Let $p$ be a node in $S$ that has no descendants in $\mathcal{G}$ still in $S$;<br>
20         $S \leftarrow S - \{p\}$;<br>
21         **if** *$p$ is OR node* **then**<br>
22             $v(p) = \min_{k \in succ(p)}(w(p,k) + v(k))$;<br>
23             Mark best successor $k$ of OR ancestors $p$, such that $k = \arg\min_{k \in succ(p)}(w(p,k) + v(k))$ (maintaining previously marked successor if still best);<br>
24             Mark $p$ as SOLVED if its best marked successor is solved;<br>
25         **else if** *$p$ is AND node* **then**<br>
26             $v(p) = \sum_{k \in succ(p)} v(k)$;<br>
27             Mark all arcs to the successors;<br>
28             Mark $p$ as SOLVED if all its children are SOLVED;<br>
29         **if** *$p$ changes its value or $p$ is marked SOLVED* **then**<br>
30             Add to $S$ all those parents of $p$ such that $p$ is one of their successors through a marked arc;<br>
31     Recompute $T^*$ by following marked arcs from the root $s$;<br>
32 **return** $\langle v(s), T^* \rangle$;

### 2.3.2 BEST-FIRST AND/OR SEARCH

The state-of-the-art version of best-first search for AND/OR search spaces for graphical models is the Best-First AND/OR search algorithm (AOBF) (Marinescu & Dechter, 2009b). AOBF is a variant of AO* (Nillson, 1980) that explores the context-minimal AND/OR search graph.

AOBF is described by Algorithm 1. For simplicity, we present the algorithm for traversing an AND/OR search tree. AOBF maintains the explicated part of the search space $\mathcal{G}$ and also keeps track of the current best partial solution tree $T^*$. It interleaves iteratively a top-down node expansion step (lines 4-16), which selects a non-terminal tip node of $T^*$ and generates its children in $\mathcal{G}$, with a bottom-up cost revision step (lines 17-30), which updates the values of the internal nodes based on their children's values. If a newly generated child node is terminal it is marked solved (line 9).

During the bottom-up phase, OR nodes that have at least one solved child and AND nodes who have all children solved are also marked as solved. The algorithm also marks the arc to the best AND child of an OR node through which the minimum is achieved (line 23). Following the backward step, a new best partial solution tree $T^*$ is recomputed (line 31). AOBF terminates when the root node is marked solved. If the heuristic used is admissible, at the point of termination $T^*$ is the optimal solution with cost $v(s)$, where $s$ is the root node of the search space.

Extending the algorithm to explore the context-minimal AND/OR search graph is straightforward and can be done as follows. When expanding a non-terminal AND node in lines 11-14, AOBF does not generate the corresponding OR children that are already present in the explicated search space $\mathcal{G}$ but rather links to them. All these identical OR nodes in $\mathcal{G}$ are easily recognized based on their contexts (Marinescu & Dechter, 2009b).

**THEOREM 1** (complexity, Marinescu & Dechter, 2009b)**.** *Algorithm AOBF traversing the context-minimal AND/OR graph has time and space complexity of $O(n \cdot k^{w^*})$, where $n$ is the number of variable in the problem, $w^*$ is the induced width of the pseudo tree and $k$ bounds the domain size.*

### 2.3.3 DEPTH-FIRST AND/OR BRANCH AND BOUND

The depth-first AND/OR Branch and Bound (AOBB) (Marinescu & Dechter, 2009a) algorithm traverses the AND/OR search space in a *depth-first* rather than a best-first manner, while keeping track of the current upper bound on the minimal solution cost.

As before and for simplicity, we present the variant of the algorithm that explores an AND/OR search tree. AOBB described by Algorithm 2 interleaves forward node expansion (lines 4-17) with a backward cost revision (or propagation) step (lines 19-29) that updates node values (capturing the current best solution to the subproblem rooted at each node), until search terminates and the optimal solution has been found. A node $n$ will be pruned (lines 12-13) if the current upper bound is higher than the node's heuristic lower bound, computed recursively using the procedure described in Algorithm 3.

In the worst case, AOBB explores the entire search space, namely $O(n \cdot k^{w^*})$ nodes (assuming a context-minimal AND/OR search graph). In practice, however, AOBB is likely to expand more nodes than AOBF using the same heuristic, but the empirical performance of AOBB depends heavily on the order in which the solutions are encountered, namely on how quickly the algorithm finds a close to optimal solution that it will use as an upper bound for pruning.

### 2.3.4 MINI-BUCKET HEURISTICS

The AND/OR search algorithms presented (AOBF and AOBB) most often use the *mini-bucket* (also known as MBE) heuristic $h(n)$. Mini-Bucket Elimination or MBE (Dechter & Rish, 2003) is an approximate version of an exact variable elimination algorithm called bucket elimination (BE) (Dechter, 1999). MBE (Algorithm 4) bounds the space and time complexity of full bucket elimination (which is exponential in the induced width $w^*$). Given a variable ordering, the algorithm associates each variable $X_i$ with a bucket which contains all functions defined on this variable, but not on higher index variables. Large buckets are partitioned into smaller subsets, called *mini-buckets*, each containing at most $i$ distinct variables. The parameter $i$ is called the i-bound. The algorithm processes buckets them from last to first (lines 2-10 in Algorithm 4). The mini-buckets of the same variable are processed separately. Assuming a min-sum problem, MBE calculates the sum

**Algorithm 2:** AOBB exploring the AND/OR search tree (Marinescu & Dechter, 2009b)

**Input**: A graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, pseudo tree $\mathcal{T}$ rooted at $X_1$, heuristic function $h(\cdot)$;
**Output**: Optimal solution to $\mathcal{M}$

Create root OR node $s$ labelled by $X_1$ and let the stack of created but not expanded nodes $OPEN \leftarrow \{s\}$;
Initialize $v(s) \leftarrow \infty$ and best partial solution tree rooted in $s$ $T^*(s) \leftarrow \emptyset$; $UB \leftarrow \infty$;
**while** $OPEN \neq \emptyset$ **do**
    Select top node $n$ in OPEN;
    **if** *$n$ is OR node labeled $X_i$* **then**
        **foreach** $x_i \in D(X_i)$ **do**
            Add AND child $n'$ labeled $\langle X_i, x_i \rangle$ to list of successors of $n$;
            Initialize $v(n') = 0$, best partial solution tree rooted in $n$ $T^*(n') = \emptyset$;
    **if** *$n$ is AND node labelled $\langle X_i, x_i \rangle$* **then**
        **foreach** *OR ancestor $k$ of $n$* **do**
            Recursively evaluate the cost of the partial solution tree rooted in $k$, based on heuristic function $h(\cdot)$, assign its cost to $f(k)$; // see Algorithm 3
            **if** *evaluated partial solution is not better than current upper bound at k (e.g. $f(k) \geq v(k)$* **then**
                Prune the subtree below the current tip node $n$;
            **else**
                **foreach** *successor $X_j$ of $X_i \in \mathcal{T}$* **do**
                    Add OR child $n'$ labeled $X_j$ to list of successors of $n$;
                    Initialize $v(n') \leftarrow \infty$, best partial solution tree rooted in $n$, $T^*(n') \leftarrow \emptyset$;
    Add successors of $n$ on top of OPEN;
    **while** *list of successors of node $n$ is empty* **do**
        **if** *node $n$ is the root node* **then**
            **return** *solution: $v(n), T^*(n)$ ;*
        **else**
            **if** *$p$ is AND node* **then**
                $v(p) \leftarrow v(p) + v(n)$, $T^*(p) \leftarrow T^*(p) \cup T^*(n)$;
            **else if** *$p$ is OR node* **then**
                **if** *the new value of better than the old one, e.g. $v(p) > (c(p, n) + v(n))$ for minimization* **then**
                    $v(p) \leftarrow w(p, n) + v(n)$, $T^*(p) \leftarrow T^*(p) \cup \langle x_i, X_i \rangle$;
        Remove $n$ from the list of successors of $p$;
        Move one level up: $n \leftarrow p$;

of the functions in each mini-bucket and eliminates its variable using the min operator (line 9). The new function is placed in the appropriate lower bucket (line 10). MBE generates a bound (lower for minimization and upper for maximization) on the optimal value. Higher $i$ values take more computational resources, but yield more accurate bounds. When $i$ is large enough (i.e., $i \geq w^*$), MBE coincides with full Bucket Elimination.

**THEOREM 2** (complexity, Dechter & Rish, 2003)**.** *Given a graphical model with variable ordering $o$ having induced width $w^*(o)$ and an i-bound parameter $i$, the time of the mini-bucket algorithm MBE(i) is $O(n \cdot k^{min(i,w^*(o))+1})$ and space complexity is $O(n \cdot k^{min(i,w^*(o))})$, where $n$ is the number of problem variables and $k$ is the maximum domains size.*

Mini-bucket elimination can be viewed as message passing from leaves to root along a *mini-bucket tree*. A mini-bucket tree of a graphical model $\mathcal{M}$ has the mini-buckets as its nodes. $Bucket_X$

**Algorithm 3:** Recursive computation of the heuristic evaluation function

**function** `evalPartialSolutionTree`$(T(n), h(n))$
**Input**: Partial solution subtree $T(n)$ rooted at node $n$, heuristic function $h(n)$;
**Output**: Heuristic evaluation function $f(T(n))$;
**if** $succ(n) == \emptyset$ **then**
    **if** *$n$ is an AND node* **then**
        **return** $0$;
    **else**
        **return** $h(n)$;
**else**
    **if** *$n$ is an AND node* **then**
        let $k_1, \ldots, k_l$ be the OR children of $n$;
        **return** $\sum_{i=1}^{l}$ `evalPartialSolutionTree`$(T(k_i), h(k_i))$;
    **else if** *$n$ is an OR node* **then**
        let $k$ be the AND child of $n$;
        return $w(n, k)$ + `evalPartialSolutionTree`$(T(k), h(k))$;

**Algorithm 4:** Mini-Bucket Elimination

**Input**: A model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, ordering $o$, parameter $i$
**Output**: Approximate solution to $\mathcal{M}$, and the ordered augmented buckets
**Initialize:** Partition the functions in **F** into $Bucket_1, \ldots, Bucket_n$, where $Bucket_i$ contains all functions whose highest variable is $X_i$.
`// Backward pass`
**for** $p \leftarrow n$ *downto 1* **do**
    Let $h_1, \ldots, h_j$ be the functions (original and intermediate) in $Bucket_p$; let $S_1, \ldots, S_j$ be their scopes;
    **if** *$X_p$ is instantiated ($X_p = x_p$)* **then**
        Assign $X_p = x_p$ to each $h_i$ and put each resulting function into its appropriate bucket;
    **else**
        Generate an $i$-partitioning;
        **foreach** $Q_k \in Q'$ **do**
            Generate the message function $h_{k \to b}$: $h_{k \to b} = \min_{x_p \in X_p} \sum_{i=1}^{j} h_i$;
            Add $h_{k \to b}$ to the bucket of $X_b$, the largest-index variable in $scope(h_{k \to b})$;
`// Forward pass`
Assign a value to each variable in the ordering $o$ so that the combination of the functions in each bucket is minimal;
**return** *The function computed in the bucket of the first variable and the corresponding assignment;*

is a child of $Bucket_Y$ is the function $h_{X \to Y}$ generated in $Bucket_X$ when variable $X$ is eliminated, is placed in $Bucket_Y$. Therefore, every vertex other than the root has one parent and possibly several child vertices. Note that a mini-bucket tree corresponds to a pseudo tree, where the mini-buckets of the same variables are combined to form what we call augmented buckets, corresponding to variable nodes (Dechter & Mateescu, 2007).

Mini-bucket elimination is often used to generate heuristics for search algorithms over graphical models, formulated for OR search spaces by Kask and Dechter (1999a, 1999b) and extended to AND/OR search by Marinescu and Dechter (2005).

**DEFINITION 4** (MBE heuristic for the AND/OR search space, Marinescu & Dechter, 2005)**.** *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:*
*1. If $n$ is an AND node, labeled by $\langle X_p, x_p \rangle$, then:*

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

*Namely, it is the sum of the intermediate functions $h_{k\to j}$ 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 bucket $B(X_p^1 \ldots X_p^q) = \{B(X_p^1), \ldots, B(X_p^q)\}$ 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:*

$$h(n) = \min_{m \in succ(p)} (w(n, m) + h(m))$$

*where $m$ are the AND children of $n$ labeled with values $x_p$ of $X_p$.*

Having established the necessary background, we will now turn to the main part of the paper, presenting our contributions, beginning with the extension of best-first search to the $m$-best task. As it is customary in the heuristic search literature and without loss of generality, we will assume in the remaining of the paper a min-sum optimization problem.

## 3. Best-First Search for Finding the *M* Best Solutions

Extending best-first search (Section 2.2) and in particular its most popular version, A*, to the $m$-best task is fairly straightforward and was suggested, for example, by Charniak and Shimony (1994). Instead of stopping after finding the optimal solution, the algorithm continues exploring the search space, reporting the next discovered solutions up until $m$ of them are obtained. We will show that these solutions are indeed the $m$ best and that they are found in a decreasing order of their optimality. In particular, the second solution reported is the second best solution and, in general, the $i^{th}$ solution discovered is the $i^{th}$ best.

### 3.1 m-A*: Definition

The $m$-best tree-search variant of A* denoted m-A* (Algorithm 5, assumes a consistent heuristic) solves an $m$-best optimization problem over any general search graph. We will show later how it can be extended to general admissible heuristics.

The scheme expands the nodes in the order of increasing value of $f$ in the usual A* manner. It keeps the lists of created nodes OPEN and expanded nodes CLOSED, as usual, maintaining a search tree, denoted by $Tr$. Beginning with the start node $s$, m-A* picks the node with the smallest evaluation function $f(n)$ in OPEN and puts it in CLOSED (line 7). If the node is a goal, a new solution is reported (lines 8-13). Otherwise, the node is expanded and its children are created (lines 15-23). The algorithm may encounter each node multiple times and will maintain up to $m$ of its

**Algorithm 5:** m-A* exploring a graph, assuming consistent heuristic

**Input**: An implicit directed search graph $G = (N, E)$, with a start node $s$ and a set of goal nodes $Goals$, a consistent heuristic evaluation function $h(n)$, parameter $m$

**Output**: the $m$ best solutions

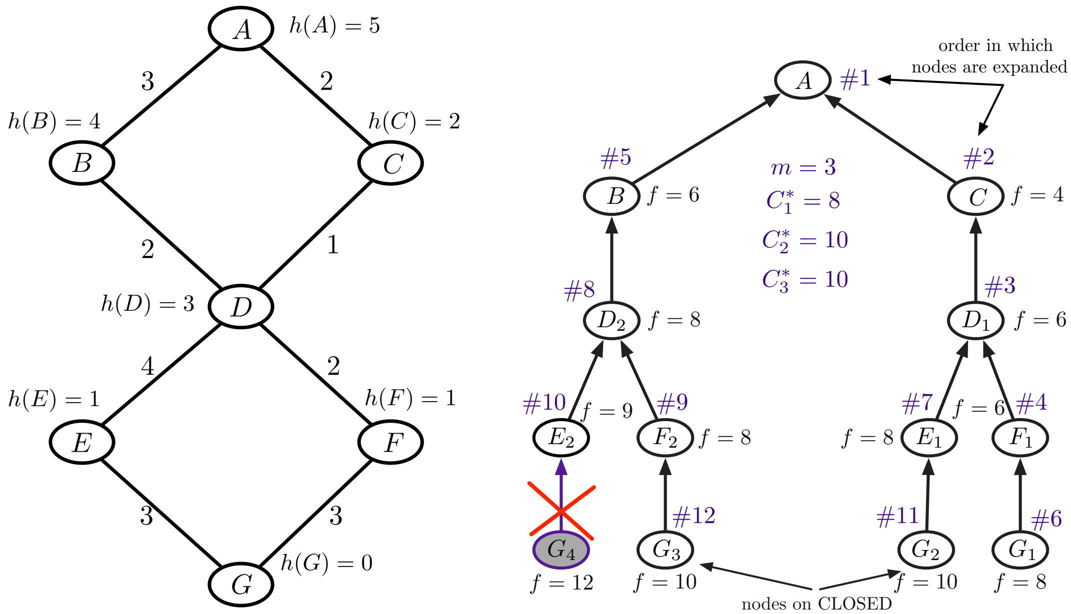
1 Initialize: OPEN=$\emptyset$, CLOSED=$\emptyset$, a tree $Tr = \emptyset$, $i = 1$ ($i$ counts the current solution being searched for)
2 OPEN $\leftarrow \{s\}$; $f(s) = h(s)$;
3 Make $s$ the root of $Tr$;
4 **while** $i \leq m$ **do**
5     **if** *OPEN is empty* **then**
6         **return** the solutions found so far;
7     Remove a node, denoted $n$, in OPEN having a minimum $f$ (break ties arbitrarily, but in favour of goal nodes and deeper nodes) and put it in CLOSED;
8     **if** *$n$ is a goal node* **then**
9         Output the current solution obtained by tracing back pointers from $n$ to $s$ (pointers are assigned in step 22); denote this solution as $Sol_i$;
10         **if** $i = m$ **then**
11             **return**;
12         **else**
13             $i \leftarrow i + 1$;
14     **else**
15         Expand node $n$, generating all its children $Ch$ ;
16         **foreach** $n' \in Ch$ **do**
17             **if** *$n'$ already appears in OPEN or CLOSED $m$ times* **then**
18                 Discard node $n'$;
19             **else**
20                 Compute current path cost $g(n') = g(n) + c(n, n')$;
21                 Compute evaluation function $f(n') = g(n') + h(n')$ ;
22                 Attach a pointer from $n'$ back to $n$ in $Tr$;
23                 Insert $n'$ into the right place in OPEN based on $f(n')$;
24 **return** *The set of the $m$ best solutions found*

copies in the OPEN and CLOSED lists combined (line 17), with separate paths to each copy in the explored search tree (lines 22-23). Nodes encountered beyond $m$ times are discarded (line 18). We denote by $C_i^*$ the $i^{th}$ best solution cost, by $f_i^*(n)$ the cost of the $i^{th}$ best solution going through node $n$, by $f_i(n)$ the heuristic evaluation function estimating $f_i^*(n)$ and by $g_i(n)$ and $h_i(n)$ the estimates of the $i^{th}$ best costs from $s$ to $n$ and from $n$ to a goal, respectively.

If the heuristic is not consistent, whenever the algorithm reaches a node it has seen before (if the search space is a graph and not a tree), there exists a possibility of the new path improving on the previously discovered ones. Therefore, lines 17-18 should be revised in the following way to account for the possibility that a better path to $n'$ is discovered:

**17** **If** *$n'$ appears already more than $m$ times in the union of OPEN or CLOSED* **then**
**18**     **If** *$g(n')$ is strictly smaller than $g_m(n')$, the current $m$-best path to $n'$* **then**
**19**         Keep $n'$ with a pointer to $n$ and put $n$ back in OPEN
**20**         Discard the earlier subtree rooted at $n$

Figure 4 shows an example of m-A* for finding the $m = 3$ shortest paths on a toy problem. The left hand side of Figure 4 shows the problem graph with 7 variables and 8 edges, together with the




(a) Problem graph (b) Trace of m-A*

Figure 4: Example problem. On the left: problem graph and heuristic values $h(n)$ for each node. On the right: trace of m-A* for finding $m = 3$ best solutions and evaluation function $f(n)$ for each node. White nodes are in CLOSED, the grey one was created, but discarded.

admissible heuristic functions for each node. Note that the heuristic is not consistent. For example, $h(A) > h(C) + c(A, C)$. $A$ is the start node, $G$ is the goal node. On the right side of the Figure we present the trace of m-A*, with the evaluation function for each copy of the nodes created by the time the $3^{rd}$ solution is found. The white nodes are in CLOSED, the grey one (node $G_4$) was created, but never put in OPEN. The algorithm expands the nodes in OPEN in increasing order of the evaluation functions. We assume that the ties are broken in favour of deeper nodes. First, m-A* discovers the solution $A - C - D - F - G$ with cost $C_1^* = 8$, next the solution $A - C - D - E - G$ with cost $C_1^* = 10$ is found. The third solutions is $A - B - D - F - G$ with cost $C_1^* = 10$. Note that two copies of each node $D$, $E$ and $F$ and four copies of $G$ were created. The goal node $G_4$ was discarded, because we bound the total number of copies of a particular node by $m = 3$.

**THEOREM 3.** *Given a graphical model $\mathcal{M} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{F}, \bigotimes \rangle$ with $n$ variables whose domain size is bounded by $k$, the worst case time and space complexity of m-A* exploring an OR search tree of $\mathcal{M}$ is $O(k^n)$.*

*Proof.* In worst case m-A* would explore the entire OR search tree, whose size is $O(k^n)$ (Section 2.3). Since the underlying search space is a tree, the algorithm will never encounter any of the nodes more than once, thus no nodes will be duplicated. $\square$

### 3.2 Properties of m-A*

In this section we extend the desirable properties of A*, listed in Section 2.2, to the $m$-best case. For simplicity and without loss of generality, we assume throughout that the search graph accommodates at least $m$ distinct solutions.

**THEOREM 4.** *Given an optimization task, its implicit directed search graph $G$ and some integer parameter $m \geq 1$, m-A* guided by an admissible heuristic has the following properties:*

1. *Soundness and completeness:* m-A* *terminates with the $m$ best solutions generated in order of their costs.*
2. *Optimal efficiency under consistent heuristic: Any node that is surely expanded*[2] *by* m-A* *must be expanded by any other search algorithm traversing $G$ that is guaranteed to find the $m$ best solutions having the same heuristic information.*
3. *Optimal efficiency for node expansions:* m-A* *expands each node at most $m$ times when the heuristic is consistent. The $i^{th}$ path found to a node is the $i^{th}$ best path.*
4. *Dominance: Given two heuristic functions $h_1$ and $h_2$, such that for every $n$ $h_1(n) < h_2(n)$,* m-A*$_1$ *will expand every node surely expanded by* m-A*$_2$*, when* m-A*$_i$ *is using heuristic $h_i$.*

We prove the properties of m-A* in Sections 3.2.1-3.2.2.

#### 3.2.1 SOUNDNESS AND COMPLETENESS

Algorithm m-A* maintains up to $m$ copies of each node and discards the rest. We will next show that this restriction does not compromise completeness.

**PROPOSITION 1.** *Any node discarded by m-A* does not lead to any of the $m$-best solutions.*

*Proof.* Consider a consistent heuristic first (as described in Algorithm 5). At the moment when m-A* discovered a node $n$ for the $(m+1)^{th}$ time, $m$ copies of $n$ reside on OPEN or CLOSED and the algorithm maintains $m$ distinct paths to each. Let $\pi_m$ be the $(m+1)^{th}$ path. As we will prove in Theorem 10, when node $n$ is discovered for the $(m+1)^{th}$ time, the cost $C_{new}$ of the newly discovered path $\pi_{new}$ is the $(m+1)^{th}$ best, namely it is no better than the costs already discovered: $C_{new} \geq C_{\pi_m}$. Therefore, the eliminated $(m+1)^{th}$ path to node $n$ is guaranteed to be worse than the remaining $m$ ones and thus can not be a part of any of the potential $m$-best optimal solutions that might be passing through node $n$.

If the heuristic is not consistent, m-A* can be modified to replace the worst of the previously discovered paths $\pi_m$ with the newly found $\pi_{new}$, if the cost of the latter is better and place the new copy in OPEN. Thus, again, it is safe to bound the number of copies by $m$. □

It is clear that along any particular solution path $\pi$ the evaluation function over all the nodes on $\pi$ is bounded by the path's cost $C(\pi)$, when the heuristic is admissible.

**PROPOSITION 2.** *The following is true regarding m-A*:*

1. *For any solution path $\pi$, forall $n \in \pi$, $f(n) \leq C(\pi)$.*

2. To be precisely defined in Section 3.2.3

2. *Unless $\pi$ was already discovered by m-A*, there is always a node $n$ on $\pi$ which resides in OPEN.*

3. *Therefore, as long as m-A* did not discover $\pi$ there must be a node in OPEN having $f(n) \leq C(\pi)$.*

*Proof.* 1. $f_\pi(n) = g_\pi(n) + h(n)$ and since $h(n) \leq c_\pi(n,t)$ due to admissibility, where $c_\pi(n,t)$ is the actual cost from $n$ to the goal node $t$ along $\pi$, we conclude that $f(n) \leq g_\pi(n) + h(n) = C(\pi)$.
2. Any reachable path from the root always has a leaf in OPEN unless all the nodes along the path are expanded and are in CLOSED.
3. Follows easily from 1 and 2. □

It follows immediately from Proposition 2 (stated similarly by Nilsson, 1982) that:

**PROPOSITION 3.** *[Necessary condition for node expansion] Any node $n$ expanded during m-A* when searching for the $i^{th}$ best solution ($1 \leq i \leq m$) satisfies $f(n) \leq C_i^*$.*

and it is also clear that

**PROPOSITION 4.** *[Sufficient condition for node expansion] Every node $n$ in OPEN, such that $f(n) < C_i^*$, must be expanded by m-A* before the $i^{th}$ best solution is found.*

Soundness and completeness of m-A* follows quite immediately.

**THEOREM 5** (soundness and completeness)**.** *Algorithm m-A* generates the $m$-best solutions in order, namely, the $i^{th}$ solution generated is the $i^{th}$ best solution.*

*Proof.* Let us assume by contradiction that this is not the case. Let the $i^{th}$ generated solution path $\pi_i$ be the first one that is not generated according to the best-first order. Namely the $i^{th}$ solution generated has a cost $C$ such that $C > C_i^*$. However, when the algorithm selected the goal $t_i$ along $\pi_i$, its evaluation function was $f(t_i) = g_{\pi_i}(t_i) = C$, while, based on Proposition 2, there was a node $n'$ in OPEN whose evaluation function was at most $C_i^*$. Thus $n'$ should have been selected for expansion instead of $t_i$. We have a contradiction and therefore the result follows. □

#### 3.2.2 THE IMPACT OF THE HEURISTIC STRENGTH

Like for A*, the performance of m-A* improves with more accurate heuristic.

**PROPOSITION 5.** *Consider two heuristic functions $h_1$ and $h_2$. Let us denote by m-A*$_1$ the algorithm that uses heuristic $h_1$ and by m-A*$_2$ the one using heuristic $h_2$. If the heuristic $h_1$ is more informed than $h_2$, namely for every node $n$, $h_2(n) < h_1(n)$, algorithm m-A*$_2$ will expand every node that will be expanded by the algorithm m-A*$_1$ before finding the $j^{th}$ solution for any $j \in \{1, 2, \ldots, m\}$, assuming the same tie-breaking rule.*

*Proof.* Since $h_1$ is more informed than $h_2$, $h_1(n) > h_2(n)$ for every non-goal node $n$. Let us assume that m-A*$_1$ expands some non-terminal node $n$ before finding the $j^{th}$ best solution with cost $C_j^*$. If node $n$ is expanded, it means that (a) at some point it is on OPEN and (b) its evaluation function satisfies $f_1(n) = g(n) + h_1(n) \leq C_j^*$ (Proposition 3). Consider the current path $\pi$ from start node to $n$. Each node $n' \in \pi$ on the path was selected at some point for expansion and thus

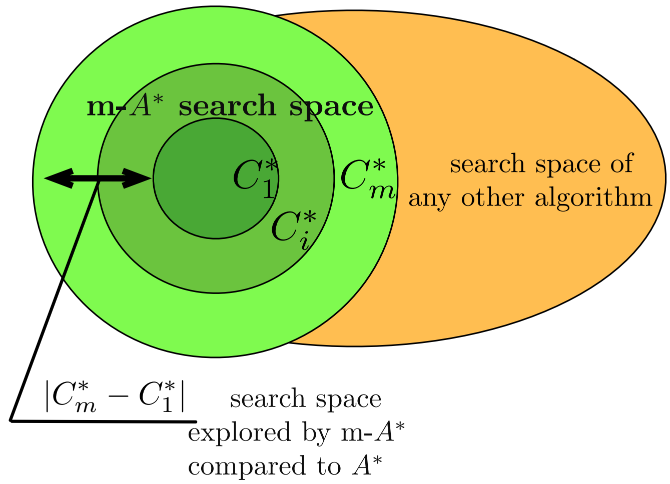



Figure 5: The schematic representation of the search spaces explored by the m-A* algorithm, depending on $m$ and cost $C_m^*$

the evaluation functions of all these nodes are also bounded by the cost of the $j^{th}$ best solution: $f_1(n') \leq C_j^*$. Since $h_1(n') > h_2(n')$ for every node $n'$ along the path $\pi$, their evaluation functions according to heuristic $h_2(n)$ obey:

$$f_2(n') = g(n') + h_2(n') < g(n') + h_1(n') < C_j^* \tag{1}$$

and thus each node $n'$ must also be expanded by m-A*$_2$. □

Consider the case of the exact heuristic. It is easy to show that:

**THEOREM 6.** *If $h = h^*$ is the exact heuristic, then m-A* generates solutions only on $j$-optimal paths $1 \leq j \leq m$.*

*Proof.* Since $h$ is exact, the $f$ values on OPEN are expanded in sequence of values $C_1^* \leq C_2^* \leq \ldots \leq C_i^* \ldots \leq C_m^*$. All the generated nodes having evaluation function $f = C_1^*$ are by definition on optimal paths (since $h = h^*$), all those who have $f = C_2^*$ must be on paths that can be second best and so on. Notice that some solutions can have the same costs. □

When $h = h^*$, m-A*'s complexity is clearly linear in the number of nodes having evaluation function $f^* \leq C_m^*$. However, when the cost function has only a small range of values, there may be an exponential number of solution paths having the cost $C_m^*$. To avoid this exponential frontier we chose the tie-breaking rule of expanding deeper nodes first, yielding a number of node expansions bounded by $m \cdot n$, when $n$ bounds the solution length. Clearly:

**THEOREM 7.** *When m-A* has access to $h = h^*$, then, using a tie-breaking rule in favour of deeper nodes, it expands at most $\#N = \sum_i \#N_i$ nodes, where $\#N_i$ is the length of the $i^{th}$ optimal solution path. Clearly, $\#N \leq m \cdot n$.*

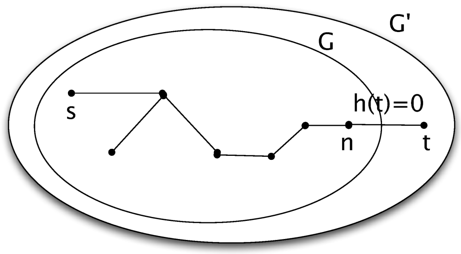



Figure 6: The graph $G'$ represents a new problem instance constructed by appending a branch leading to a new goal node $t$ to node $n$.

#### 3.2.3 M-A* WITH CONSISTENT HEURISTIC

When m-A* uses a consistent heuristic, it has several useful properties.

**Optimal efficiency under consistent heuristic.** Algorithm A* is known to be optimally efficient for consistent heuristic (Dechter & Pearl, 1985). Namely, any other algorithm that extends search paths from the root and uses the same heuristic information as A* will expand every node that is *surely expanded by A**, i.e., it will expand every $n$, such that $f(n) < C^*$. We extend the notion of nodes surely expanded by A* to the $m$-best case:

**PROPOSITION 6.** *Algorithm m-A* will expand any node $n$ reachable by a strictly $C^*_m$-bounded path from the root, regardless of the tie-breaking rule. The set of such nodes is referred to as surely expanded by m-A*.*

*Proof.* Let us consider the strictly $C^*_m$-bounded path $\pi = \{s, n_1, n_2, \ldots n\}$. The start node $s$ is clearly expanded at the beginning of the search and its children, including node $n_1$, are placed on OPEN. Since $f(n_1) < C^*_m$, node $n_1$ must be expanded by m-A* before finding the $m^{th}$ best solution (Proposition 4), its children, including $n_2$, in turn are placed in OPEN. The same is true for all nodes of $\pi$, including $n$. □

**THEOREM 8** (m-optimal efficiency)**.** *Any search algorithm, that is guaranteed to find the $m$-best solutions and that explores the same search space as m-A* and has the same consistent heuristic, will have to expand any node that is surely expanded by m-A*. Namely it will expand every node that lies on any path $\pi$ that is dominated by $C^*_m$, i.e. $f(n') < C^*_m, \forall n' \in \pi$.*

The proof idea is similar to the work by Dechter and Pearl (1985). Namely we can show that any algorithm that does not expand a node $n$, surely expanded by m-A*, can miss one of the $m$-best solutions, when applied to a slightly modified problem:

*Proof.* Let us consider a problem having the search graph $G$ and a consistent heuristic $h$. Assume that node $n$ is surely expanded by m-A* before finding the $j^{th}$ best solution. Let $B$ be an algorithm that uses the same heuristic $h$ and is guaranteed to find the $m$ best solutions. Let also assume that node $n$ is not expanded by $B$. A consistency of the heuristic also allows us to better characterize the nodes expanded by m-A*.

We can create a new problem graph $G'$ (see Figure 6) by adding a new goal node $t$ with $h(t) = 0$, connecting it to $n$ by an edge having cost $c = h(n) + \delta$, where $\delta = 0.5(C^*_j - D)$

and $D = \max_{n' \in S_j} f(n')$, where $S_j$ is the set of nodes surely expanded by *m-A** before finding the $j^{th}$ solution. It is possible to show that the heuristic $h$ is admissible for the graph $G'$ (Dechter & Pearl, 1985). Since $\delta = 0.5(C_j^* - D)$, $C^* = D - 2\delta$. By construction, the evaluation function of the new goal node is:

$$f(t) = g(t) + h(t) = g(n) + c = g(n) + h(n) + \delta = f(n) + \delta \leq D + \delta = C_j^* - \delta < C_j^* \quad (2)$$

which means that $t$ is reachable from $s$ by a path whose cost is strictly bounded by $C_j^*$. That guarantees that m-A* will expand $t$ (Proposition 6), discovering a solution with cost $C_j^* - \delta$. On the other hand, algorithm $B$, that does not expand node $n$ in the original problem, will still not expand it and thus will not reach node $t$ and will only discover the solution with cost $C_j^*$, not returning the true set of $m$ best solutions to the modified problem. From the contradiction the theorem follows. $\square$

**PROPOSITION 7.** *If the heuristic function employed by m-A* is consistent, the values of the evaluation function $f$ of the sequence of expanded nodes are non-decreasing.*

The proof is a straightforward extension of a result by Nilsson (1980).

*Proof.* Let node $n_2$ be expanded immediately after $n_1$. If $n_2$ was already in OPEN at the time when $n_1$ was expanded, then from the node selection rule it follows that $f(n_1) \leq f(n_2)$. If $n_2$ was not in OPEN, then it must have been added to it as a result of expansion of $n_1$, i.e., be a child of $n_1$. In this case the cost of getting to $n_2$ from the start node is $g(n_2) = g(n_1) + c(n_1, n_2)$ and the evaluation function of node $n_2$ is $f(n_2) = g(n_2) + h(n_2) = g(n_1) + c(n_1, n_2) + h(n_2)$. Since $h(n)$ is consistent, $h(n_1) \leq c(n_1, n_2) + h(n_2)$ and $f(n_2) \geq g(n_1) + h(n_1)$. Namely, $f(n_2) \geq f(n_1)$. $\square$

If the heuristic function is consistent, we have a stronger condition of Proposition 4:

**THEOREM 9.** *Algorithm m-A* using a consistent heuristic function:*

1. *expands all nodes $n$ such that $f(n) < C_m^*$;*
2. *never expands any nodes with evaluation function $f(n) > C_m^*$;*
3. *expands some nodes such that $f(n) = C_m^*$, subject to a tie-breaking rule.*

*Proof.* 1. Assume that there exists a node $n$ such that $f(n) < C_m^*$ and node $n$ is never expanded by m-A*. Such a situation can only arise if node $n$ has never been in the OPEN list, otherwise it would have been expanded, according to Proposition 4. That implies that the parent of node $n$ in the search space (let us denote it by node $p$) has never been expanded. However, similarly how it is done in the proof of Proposition 7, it is easy to show that $f(p) \leq f(n)$ and, consequently $f(p) < C_m^*$. Thus node $p$ must also have never been in OPEN, otherwise it would be expanded. Clearly, this is true for all the ancestors of $n$, up to the start node $s$. Since node $s$ is clearly in OPEN at the beginning of the search, the initial assumption is incorrect and the property follows.

2. and 3. Follow directly from Proposition 3. $\square$

Figure 7 provides a schematic summary of the search space explored by m-A* having a consistent heuristic.

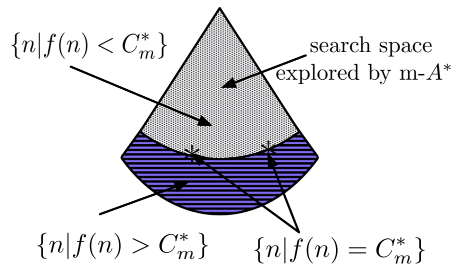



Figure 7: The nodes explored by the m-A* algorithm with a consistent heuristic.

**Optimal efficiency for node expansions.** Whenever a node $n$ is selected for expansion for the first time by m-A*, the algorithm has already found the shortest path to that node. We can extend this property as follows:

**THEOREM 10.** *Given a consistent heuristic $h$, when* m-A* *selects a node $n$ for expansion for the $i^{th}$ time, then $g(n) = g^*_i(n)$, namely it has found the $i^{th}$ best path from start node $s$ to $n$.*

*Proof.* By induction. For $i = 1$ (basic step) the theorem holds (Nillson, 1980). Assume that it also holds for $(i-1)^{th}$ expansion of node $n$. Let us consider the $i^{th}$ case, $i > 1$ (inductive step). We have already expanded the node $n$ $(i-1)$ times and due to the inductive hypothesis we have already found the $(i-1)$ distinct best paths to the node $n$. Let us assume that the cost of the newly found solution path is greater than the $i^{th}$ optimal one, i.e. $g_i(n) > g^*_i(n)$. Then, there exists a different, undiscovered path $\pi$ from $s$ to $n$ with cost $g_\pi(n) = g^*_i(n) < g_i(n)$. From Proposition 2 there exists in OPEN a node $n_0 \in \pi$. Obviously, node $n_0$ must be located between the start node $s$ and node $n$. Denoting by $C_\pi(n_0, n) = c(n_0, n_1) + \cdots + c(n_k, n)$, from the heuristic consistency it easily follows that $h(n_0) < C_\pi(n_0, n) + h(n)$ and that the evaluation function of node $n_0$ along path $\pi$ is $f_\pi(n_0) = g_\pi(n_0) + h(n_0) < g_\pi(n_0) + C_\pi(n_0, n) + h(n)$. Seeing that the cost of path $\pi$ from $s$ to $n$ is $g_\pi(n) = g_\pi(n_0) + C_\pi$, we conclude that $f_\pi(n_0) < f_\pi(n)$. However, that contradicts our assumption that node $n$ was expanded for the $i^{th}$ time before node $n_0$. The theorem follows. ☐

### 3.2.4 THE IMPACT OF THE REQUIRED BEST SOLUTIONS $m$

The sequence of the sizes of search spaces explored by m-A* as a function of $m$ is obviously monotonically increasing with $m$. Denoting by $j$-A* and $i$-A* the versions of the m-A* algorithm that search respectively for $j$ and $i$ best solutions, we can make the following straightforward characterization:

**PROPOSITION 8.** *Given a search graph and consistent heuristic,*

1. *Any node expanded by i-A* is expanded by j-A* if $i < j$ and if both use the same tie-breaking rule.*

2. *The set $S(i,j)$ of nodes defined by $S(i,j) = \{n | C_i^* < f(n) < C_j^*\}$ will surely be expanded by $j$-A* and surely not expanded by $i$-A*.*

3. *If $C_j^* = C_i^*$, the difference in the number of nodes expanded by $i$-A* and $j$-A* is determined by the tie-breaking rule.*

The proof follows trivially from Theorem 9. As a result, larger discrepancy between the respective costs $C_j^* - C_i^*$ yields larger difference in the search spaces explored by $j$-A* and $i$-A*. This difference, however, also depends on the granularity with which the values of a sequence of observed evaluation functions increase, which is related to the arc costs (or weights) of the search graph. If $C_i^* = C_j^* = C$, then the search space explored by $i$-A* and $j$-A* will differ only in the frontier of nodes satisfying $f(n) = C$. Figure 5 represents schematically the search spaces explored by the $i$-A* algorithm.

## 4. Depth-First Branch and Bound for Finding the *M* Best Solutions

Along with its valuable properties, m-A* inherits also the disadvantages of A*: its exponential space complexity, which makes the algorithm infeasible for many applications. An alternative approach is searching using depth-first branch and bound (DFBnB), which can be implemented in linear space if necessary and is therefore often more practical. DFBnB finds the optimal solution by exploring the search space in a depth-first manner. The algorithm maintains a cost $U$ of the best solution encountered so far and prunes search nodes whose lower-bounding evaluation function $f(n) = g(n) + h(n)$ is larger than $U$. Extending DFBnB to the $m$-best task is straightforward, as we describe next.

### 4.1 The m-BB Algorithm

Algorithm m-BB, the depth-first branch and bound extension to the $m$-best task, that explores a search tree is presented in Algorithm 6. As usual, the algorithm maintains lists of OPEN and CLOSED nodes. It also maintains a sorted list of CANDIDATE nodes that contains the best $m$ solutions found so far. Nodes on OPEN are organized in a "last in - first out" manner in order to facilitate a depth-first exploration of the search space (i.e., OPEN is a stack). At each step, m-BB expands the next node $n$ in OPEN (line 5). If it is a goal node, a new complete solution is found (line 6) and it is stored in the CANDIDATE list (line 7-9), which is then re-sorted (line 10). Only up to $m$ best solutions are maintained (lines 11-13).

The main modification to the depth-first branch and bound, when extended to the $m$-best task, is in its pruning condition. Let $U_1 \leq U_2 \leq \ldots \leq U_m$ denote the costs of the $m$ best solutions encountered thus far. Then $U_m$ is the upper bound used for pruning. Before $m$ solutions are discovered, no pruning takes place. Algorithm m-BB expands the current node $n$, generates its children (lines 15-17) and computes their evaluation function (line 18-19). It prunes a subproblem below $n$ iff $f(n) \geq U_m$ (lines 20-23). It is easy to see that when the algorithm terminates, it outputs the $m$-best solutions to the problem.

**THEOREM 11.** *Algorithm m-BB is sound and complete for the $m$-best solutions task.*

*Proof.* Algorithm m-BB explores the search space systematically. The only solutions that are skipped are the ones satisfying $f(n) \geq U_m$ (see step 22). Since $U_m \geq C_m^*$, where $C_m^*$ is the $m$

**Algorithm 6:** m-BB exploring a graph, assuming a consistent heuristic

**Input**: An implicit directed search graph $G = (N, E)$ with a start node $n_0$ and a set of goal nodes $Goals$. A heuristic function $h(n)$. Parameter $m$ (the number of desired solutions).

**Output**: the $m$ best solutions

1 Initialize: OPEN=$\emptyset$, CLOSED=$\emptyset$, a tree $Tr = \emptyset$, sorted list CANDIDATE = $\emptyset$, UpperBound = $-\infty$, $i = 1$ ($i$ counts the current solution being searched for);
2 Put the start node $n_0$ in OPEN, $g(n_0) = 0$, $f(n_0) = h(n_0)$;
3 Assign $n_0$ to be the root of $Tr$;
4 **while** *OPEN is not empty* **do**
5     Remove the top node from OPEN, denoted $n$, and put it on CLOSED;
6     **if** *$n$ is a goal node* **then**
7         $sol_i \leftarrow$ solution obtained by tracing back pointers from $n$ to $n_0$ (pointers assigned at step 17);
8         $C_i \leftarrow$ cost of $sol_i$;
9         Place solution $sol_i$ on CANDIDATE;
10         Sort CANDIDATE in increasing order of solution costs;
11         **if** *size of CANDIDATE list $\geq m$* **then**
12             $U_m \leftarrow$ cost of the $m^{th}$ element in CANDIDATE;
13             Keep first $m$ elements of CANDIDATE, discard the rest;
14     **else**
15         Expand node $n$, generating its children $succ(n)$;
16         **forall the** $n' \in succ(n)$ **do**
17             Attach a pointer from $n'$ back to $n$ in $Tr$;
18             $g(n') = g(n) + c(n, n')$;
19             $f(n') = g(n') + h(n')$;
20             **if** $f(n') < U_m$ **then**
21                 Place $n'$ in OPEN;
22             **else**
23                 Discard $n'$;
24 **return** *the solutions on CANDIDATE list*

best solution cost, it implies $f(n) \geq C^*_m$ and therefore that path cannot lead to a newly discovered $m$-best cost. □

**THEOREM 12.** *Given a graphical model $\mathcal{M} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{F}, \bigotimes \rangle$, the worst case time complexity of m-BB that explores an OR search tree of $\mathcal{M}$ is $O(k^n + \log m)$, where $n$ is the number of variables, $k$ is the domain size and $m$ is the number of required solutions. Space complexity is $O(n)$.*

*Proof.* In worst case m-BB would explore the entire OR search tree of size $O(k^n)$. The maintaining of CANDIDATE list introduces additional time overhead of $O(\log m)$. Since the OR search tree yields no caching, m-BB uses space linear in the number of variables. □

## 4.2 Characterization of the Search Space Explored by m-BB

We have already shown that m-A* is superior to any exact search algorithm for finding the $m$-best solutions when the heuristic is consistent (Theorem 8). In particular, m-BB must expand all the nodes that are surely expanded by m-A*, namely the set of nodes $\{n|f(n) < C^*_m\}$. From Theorem 8 and the pruning condition it is clear that:

**PROPOSITION 9.** *Given a consistent heuristic m-BB must expand any node in the set* $\{n|f(n) < C^*_m\}$*. Also, there are instances for which m-BB will expands nodes satisfying* $f(n) > C^*_m$.

Several sources of overhead of m-BB are discussed next.

#### 4.2.1 m-BB vs. BB

Pruning in m-BB does not occur until the upper bound on the current $m^{th}$ best solution is assigned a valid value, i.e., until $m$ solutions are found. In the absence of determinism, when all solutions are consistent, the time it takes to find $m$ arbitrary solutions in depth-first manner is $O(m \cdot n)$, where $n$ is the length of solution (for graphical models $n$ coincides with the number of variables). If the problem contains determinism it may be difficult to find even a single solution. This means that for m-BB the search may be exhaustive for quite some time.

#### 4.2.2 The Impact of Solution Order

The difference in the number of nodes expanded by BB and m-BB depends greatly on the variance between the solution costs. If all the solutions have the same cost, then $U_1 = U_m$. However, such a situation is unlikely and therefore the conditions for m-BB's node expansions are impacted by the order in which solutions are discovered. Let $\{U^1_m, \ldots, U^j_m\}$ be the non-increasing sequence of the upper bounds on the $m^{th}$ best solution, up to a point when m-BB uncovered the $j^{th}$ solution. Initially $U^j_m = \infty$, for $j \in \{1, \ldots, m-1\}$.

**PROPOSITION 10.** *Between the discovery of the* $(j-1)^{th}$ *and the* $j^{th}$ *solutions the set of nodes expanded by m-BB are included in* $S_j = \{n \mid f(n) \leq U^{j-1}_m\}$*, where* $C^*_m \leq U^j_m \leq U^{j-1}_m \leq \infty$.

*Proof.* Between discovering the $(j-1)^{th}$ and $j^{th}$ solutions m-BB expands only nodes satisfying $\{n \mid f(n) \leq U^{j-1}_m\}$, hence $\forall j : C_j \leq U^{j-1}_m$. Once the $j^{th}$ solution is found, it either replaces the previous bound on $m^{th}$ solution $U^j_m = C_j$ or some $k^{th}$ upper bound, $k \in \{1, \ldots, m-1\}$, yielding $U^j_m = U^{j-1}_{m-1}$. Either way, $C^*_m \leq U^j_m \leq U^{j-1}_m$. $\square$

#### 4.2.3 Ordering Overhead

The need to keep a list of $m$ sorted solutions (the CANDIDATE list) implies $O(\log m)$ overhead for each new solution discovered. The total number of solutions encountered before termination is hard to characterize.

#### 4.2.4 Caching Overhead

The overhead related to caching arises only when m-BB explores a search graph and uses caching. This version of the algorithm (not explicitly presented) stores the $m$ best partial solutions to any fully explored subproblems (and a subset of $m$ when only a partial set is discovered) and re-uses these results whenever the subproblem is encountered again. In order to implement caching, m-BB requires to store a list of length $m$ for each node that is cached. Moreover, the cached partial solutions need to be sorted, which yields an $O(m \log m)$ time overhead *per cached node*.

## 5. Adapting m-A* and m-BB for Graphical Models

Our main task is to find the $m$ best solutions to optimization tasks over graphical models. Therefore, we adapt the $m$-best search algorithms m-A* and m-BB to explore the AND/OR search space over graphical models, yielding algorithms m-AOBF and m-AOBB, respectively. We will also describe a hybrid algorithm *BE+m-BF*, combining Bucket Elimination and m-A*.

### 5.1 m-AOBF: Best-First AND/OR Search for *M* Best Solutions in Graphical Models

The extension of algorithm AOBF (Section 2.3.2) to the $m$-best task seems fairly straightforward, in principle. m-AOBF is AOBF that continues searching after discovering the first solution, until the required number of $m$ best solutions is obtained. The actual implementation requires several modifications as we discuss next.

It is not easy to extend AOBF's bottom-up node values updates and corresponding arc marking mechanism to the $m$-best task. Therefore, in order to keep track of the current best partial solution tree while searching for the $i^{th}$ best solution we adopt a naive approach that maintains explicitly a list OPEN containing entire partial solution trees (not just nodes), sorted in ascending order of their heuristic evaluation costs. Algorithm 7 presents the pseudo-code of our simple scheme that explores the AND/OR search tree and generates solutions one by one in order of their costs. At each step, the algorithm removes the next partial solution tree $T'$ from OPEN (line 4). If $T'$ is a complete solution, it is added to the list of solutions along with its cost (lines 5-8), otherwise the algorithm expands a tip node $n$ of $T'$, generating its successors (line 10-17). Each such newly generated node $n'$ is added to $T'$ separately, yielding a new partial solution tree $T''$ (lines 19-23), whose cost is recursively evaluated using Algorithm 3, as in AOBB (line 28). These new partial trees are then placed in OPEN (line 29). Search stops when all $m$ solutions have been found.

We note that the maintenance of the OPEN list containing explicit partial solution subtrees is a source of significant additional overhead which will become apparent in the empirical evaluation in Section 7. Thus, the question whether the performance of m-AOBF can be improved further is open and is therefore a rich topic of future work.

All m-A* properties (Section 3.2) can be extended to m-AOBF. In particular, algorithm m-AOBF with an admissible heuristic is sound and complete, terminating with the $m$ best solutions generated in order of their costs. m-AOBF is also optimal in terms of the number of nodes expanded compared with any other algorithm that explores the same AND/OR search space with the same consistent heuristic function.

**THEOREM 13** (m-AOBF complexity). *The complexity of algorithm m-AOBF traversing either the* AND/OR search tree *or the* context minimal AND/OR search graph *is time and space $O(k^{deg^{h-1}})$, where $h$ is the depth of the underlying pseudo tree, $k$ is the maximum domain size, and $deg$ bounds the degree of the nodes in the pseudo tree. If the pseudo tree is balanced (each internal node has exactly $deg$ child nodes), then the time and space complexity is $O(k^n)$, where $n$ is the number of variables.*

The real complexity bound of m-AOBF comes from the cost function. It appears however that maintaining an OPEN list in a brute force manner does not lend itself easily to an effective way of enumerating all partial solution subtrees and therefore the search space of all partial solution subtrees is actually exponential in $n$. The detailed proof of Theorem 13 is given in the Appendix.

**Algorithm 7:** m-AOBF exploring an AND/OR search tree

**Input**: A graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, pseudo tree $\mathcal{T}$ rooted at $X_1$, heuristic function $h(\cdot)$, parameter $m$;
**Output**: The $m$ best solutions to $\mathcal{M}$

Create root OR node $s$ labelled by $X_1$, let $\mathcal{G} = \{s\}$ (explored search space) and $T = \{s\}$ (partial solution tree);
Initialize $\mathcal{S} \leftarrow \emptyset$; $OPEN \leftarrow \{T\}$; $i = 1$; ($i$ counts the current solution being searched for);
**while** $i \leq m$ *and* $OPEN \neq \emptyset$ **do**
    Select the top partial solution tree $T'$ and remove it from OPEN;
    **if** $T'$ *is a complete solution* **then**
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle f(T'), T' \rangle\}$;
        $i \leftarrow i + 1$;
        **continue**;
    Select a non-terminal tip node $n$ in $T'$;
    `// Expand node n`
    **if** $n$ *is OR node labeled* $X_i$ **then**
        **forall the** $x_i \in D(X_i)$ **do**
            Create AND child $n'$ labeled $\langle X_i, x_i \rangle$;
            $succ(n) \leftarrow succ(n) \cup \{n'\}$;
    **else if** $n$ *is AND node labeled* $\langle X_i, x_i \rangle$ **then**
        **forall the** *successor* $X_j$ *of* $X_i$ *in* $\mathcal{T}$ **do**
            Create an OR child $n'$ labeled $X_j$;
            $succ(n) \leftarrow succ(n) \cup \{n'\}$;
    $\mathcal{G} \leftarrow \mathcal{G} \cup \{succ(n)\}$;
    `// Generate new partial solution trees`
    $\mathcal{L} \leftarrow \emptyset$;
    **forall the** $n' \in succ(n)$ **do** Initialize $v(n') = h(n')$;
    **if** $n$ *is OR node* **then**
        **forall the** $n' \in succ(n)$ **do**
            Create a new partial solution tree $T'' \leftarrow T' \cup \{n'\}$;
            $\mathcal{L} \leftarrow \mathcal{L} \cup \{T''\}$;
    **else if** $n$ *is AND node* **then**
        Create a new partial solution tree $T'' \leftarrow T' \cup \{succ(n)\}$;
    **forall the** $T'' \in \mathcal{L}$ **do**
        Recursively evaluate and assign to $f(T'')$ the cost of the partial solution tree $T''$, based on heuristic function $h(\cdot)$; // see Algorithm 3
        Place $T''$ in OPEN, keeping it sorted in the ascending order of costs $f(T'')$;
**return** *The* $m$ *best solutions found* $\mathcal{S}$;

### 5.2 m-AOBB: AND/OR Branch and Bound for *M* Best Solutions in Graphical Models

Algorithm m-AOBB extends the AND/OR Branch and Bound search (AOBB, Section 2.3.3) to the $m$-best task. The main difference between AOBB and m-AOBB is in the value function computed for each node. m-AOBB tracks the costs of the $m$ best partial solutions of each solved subproblem. Thus it extends the node value $v(n)$ and solution tree $T^*(n)$ rooted by $n$ in AOBB to ordered sets of length $m$, denoted by $\overline{v}(n)$ and $\overline{T^*}(n)$, respectively, where $\overline{v}(n) = \{v_1(n), \ldots, v_m(n)\}$ is an ordered set of the costs of the $m$ best solutions to the subproblem rooted by $n$, and $\overline{T^*}(n) = \{T_1^*(n), \ldots, T_m^*(n)\}$ is a set of corresponding solution trees. This extension arises due to the depth-first manner of search space exploration of m-AOBB in conjunction with the AND/OR decomposition. Therefore, due to the AND/OR decomposition m-AOBB needs to completely solve

**Algorithm 8:** m-AOBB exploring an AND/OR search tree

**Input**: A graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, pseudo tree $\mathcal{T}$ rooted at $X_1$, heuristic function $h(\cdot)$, parameter $m$;
**Output**: The $m$ best solutions to $\mathcal{M}$

// INITIALIZE
Create root OR node $s$ labeled by $X_1$ and let the stack of created but not expanded nodes $OPEN = \{s\}$;
Initialize $\overline{v}(s) = \overline{\infty}$ (a set of bounds on $m$ best solutions under $s$) and a set of best partial solution trees rooted in $s$ $\overline{T^*}(s) = \emptyset$; $UB = \infty$, sorted list $CANDIDATE = \emptyset$;
**while** $OPEN \neq \emptyset$ **do**
    Select top node $n$ in OPEN;
    // EXPAND
    **if** *$n$ is OR node labeled $X_i$* **then**
        **foreach** $x_i \in D(X_i)$ **do**
            Add AND child $n'$ labeled $\langle X_i, x_i \rangle$ to list $succ(n)$ containing the successors of $n$;
            Initialize $\overline{v}(n') = \overline{0}$, a set of best partial solution trees rooted in $n$ $\overline{T^*}(n') = \overline{\emptyset}$;
    **if** *$n$ is AND node labeled $\langle X_i, x_i \rangle$* **then**
        Let $p$ be ancestor of $n$;
        Recursively evaluate and assign to $f(p)$ the cost of the partial solution tree rooted in $p$, based on the heuristic $h(\cdot)$; // see Algorithm 3
        **if** $v_m(p) < \infty$ *and* $f(p) \geq v_m(p)$ **then**
            Prune the subtree below the current tip node $n$;
        **else**
            **foreach** *successor $X_j$ of $X_i \in \mathcal{T}$* **do**
                Add OR child $n'$ labeled $X_j$ to list $succ(n)$ containing the successors of $n$;
                Initialize $\overline{v}(n') = \overline{\infty}$, a set of best partial solution trees rooted in $n$ $\overline{T^*}(n') = \overline{\emptyset}$;
    Remove $n$ from OPEN and add $succ(n)$ on top of OPEN;
    // PROPAGATE
    **while** *list of successors of node n is empty* **do**
        **if** *$n$ is the root node* **then**
            **return** *a set of solutions rooted at $n$ and their costs:* $\overline{T^*}(n), \overline{v}(n)$ ;
        **else**
            Update ancestors of $n$, AND and OR nodes $p$, bottom up:
            **if** *$p$ is AND node* **then**
                Combine the set of the partial solution trees to the subproblem rooted in $p$ $\overline{T^*}(p)$ and the set of partial solution trees rooted in $n$ $\overline{T^*}(n)$ and their costs $\overline{v}(p)$ and $\overline{v}(n)$; // see Algorithm 9
                Assign the resulting set of the costs and the set of the best partial solution trees respectively to $\overline{v}(p)$ and $\overline{T^*}(p)$;
            **else if** *$p$ is OR node* **then**
                **foreach** *solution cost $v_i(n)$ in the set $\overline{v}(n)$* **do**
                    Update the cost with the weight of the arc, creating a new set of costs $\overline{v'}(n)$: $v'_i(n) = c(p,n) + v_i(n)$;
                Merge the sets of partial solutions $\overline{v}(n)$ and $\overline{v}(p)$ and the sets of partial solution trees rooted in $p$ and $n$: $\overline{T^*}(p)$ and $\overline{T^*}(n)$, keeping $m$ best elements; //Algorithm 10
                Assign results of merging respectively to $\overline{v}(p)$ and $\overline{T^*}(p)$;
        Remove $n$ from the list of successors of $p$;
        Move one level up: $n \leftarrow p$;
**return** $\overline{v}(s)$ *and* $\overline{T^*}(s)$

the subproblems rooted in all the children $n'$ of an AND node $n$, before even a single solution to a subproblem above $n$ is acquired (unlike the m-BB case). Consequently, during the bottom-up phase sets of $m$ costs have to be propagated and updated. m-AOBF on the other hand, only maintains a set of partial solution trees.

**Algorithm 9:** Combining the sets of costs and partial solution trees

**function** Combine($\overline{v}(n)$, $\overline{v}(p)$, $\overline{T^*}(n)$,$\overline{T^*}(n)$)

**Input**: Input sorted sets of costs $\overline{v}(n)$, $\overline{v}(p)$, corresponding partial solution trees $\overline{T^*}(n)$, $\overline{T^*}(p)$, number of required solutions $m$

**Output**: A set of costs $m$ best combined solutions $\overline{v'}(p)$, corresponding partial solution trees $\overline{T^{*\prime}}(p)$

// INITIALIZE

Sorted list OPEN, initially empty; //contains potential cost combinations

$\overline{v'}(p) \leftarrow \emptyset$; $\overline{T^{*\prime}}(p) \leftarrow \emptyset$;

$k = 1$; //number of partial solutions already assembled, up to $m$ in total

// Search over possible combinations

OPEN$\leftarrow v_1(n) + v_1(p)$;

**while** $k < m$ *and* $OPEN$ *is not empty* **do**

    Remove the top node $V$ on OPEN, where $V = Sv_i(n) + v_j(p)$;

    $v'_k(p) \leftarrow V$;

    ${T^*}'(p) \leftarrow T^*_i(n) \cup T^*_j(p)$;

    **if** $v_{i+1}(n) + v_j(p)$ *not in* $OPEN$ **then**

        Put $v_{i+1}(n) + v_j(p)$ in $OPEN$;

    **if** $v_i(n) + v_{j+1}(p)$ *not on* $OPEN$ **then**

        Put $v_i(n) + v_{j+1}(p)$ in $OPEN$;

    $k \leftarrow k + 1$;

**return** $\overline{v'}(p)$, $\overline{T^*(p)}$;

Unlike m-AOBF that discovers solutions one by one in order of their costs, m-AOBB (pseudo-code in Algorithm 8) reports the entire set of $m$ solutions at once, at termination. m-AOBB interleaves forward node expansion (lines 5-18) with a backward propagation (or cost revision) step (lines 19-33) that updates node values until search terminates. A node $n$ will be pruned (lines 12-13) if the current upper bound on the $m^{th}$ solution under $n$, $v_m(n)$, is lower than the node's evaluation functions $f(n)$, which is computed recursively as in AOBB (Algorithm 3). During the bottom-up propagation phase at each AND node the partial solutions to the subproblems rooted in the node's children are combined (line 24-26, Algorithm 9). At each parent OR node $p$ $\overline{v}(p)$ and $\overline{T^*}(p)$ are updated to incorporate the new and possibly better partial solutions rooted in a child node $n$ (lines 27-31, Algorithm 10).

#### 5.2.1 CHARACTERIZING NODE PROCESSING OVERHEAD

In addition to the increase in the explored search space that m-BB experiences compared with BB due to the reduced pruning (Section 4.2), AND/OR search introduces additional overhead for m-AOBB. The propagation of a set of $m$ costs and of $m$ partial solution trees leads to an increase in memory by a factor of $m$ per node. Processing the partial solutions at both OR and AND nodes introduces an additional overhead.

**THEOREM 14.** *Algorithm m-AOBB exploring the* AND/OR search tree *has a time overhead of* $O(m \cdot deg \cdot \log m)$ *per AND node and* $O(m \cdot k)$ *per OR node, where* $deg$ *bounds the degree of the pseudo tree and* $k$ *is the largest domain size. Assuming* $k < deg \cdot \log(m)$*, the total worst case time complexity is* $O(n \cdot k^h deg \cdot m \log(m))$ *and the space complexity is* $O(mn)$*. The time complexity of m-AOBB exploring the* AND/OR search graph *is* $O(n \cdot k^{w^*} deg \cdot m \log(m))$*, space complexity* $O(mn \cdot k^{w^*})$.

**Algorithm 10:** Merging the sets of costs and partial solution trees

**function** Merge($\overline{v}(n)$,$\overline{v}(p)$,$\overline{T^*}(n)$,$\overline{T^*}(p)$)

**Input**: Input sorted cost sets $\overline{v}(n)$ and $\overline{v}(p)$, sets of corresponding partial solution trees $\overline{T^*}(n)$ and $\overline{T^*}(p)$, number of required solutions $m$

**Output**: $\overline{v'}(p)$, a merged set of $m$ best solution costs, $\overline{T^{*'}}(p)$ a set of corresponding partial solution trees

// INITIALIZE

$\overline{v'}(p) \leftarrow \emptyset$;

$\overline{T^{*'}}(p) \leftarrow \emptyset$;

$i, j \leftarrow 1$; //indices in the cost sets

$k \leftarrow 1$; //index in the resulting array

// Merge two sorted sets

**while** $k \leq m$ **do**

    **if** $v_i(p) \leq v_j(n)$ **then**

        $v'_k(p) \leftarrow v_i(p)$;

        $T^{*'}_k(p) \leftarrow T^*_i(p)$;

        $i \leftarrow i + 1$;

        $k \leftarrow k + 1$;

    **else**

        $v'_k(p) \leftarrow v_j(n)$;

        $T^{*'}_k(p) \leftarrow T^*_j(n)$;

        $j \leftarrow j + 1$;

        $k \leftarrow k + 1$;

**return** $\overline{v'}(p)$ and $\overline{T^{*'}}(p)$;

*Proof.* Combining the sets of current $m$-best partial solutions (Algorithm 9) introduces an overheard of $O(m \log(m))$. The resulting time overhead per AND node is $O(deg \cdot m \log(m))$. Merging two sorted sets of costs (Algorithm 10) can be done in $O(m)$ steps. If an OR node has $O(k)$ children, the resulting overhead is $O(m \cdot k)$. Assuming $k < deg \cdot \log(m)$, the complexity is dominated by processing of the AND nodes. In the worst case, the tree version of m-AOBB, called m-AOBB-tree, would explore the complete search space of size $O(n \cdot k^h)$, where $h$ bounds the depth of the pseudo tree, while the graph version, called m-AOBB-graph, would visit a space of of size $O(n \cdot k^{w^*})$, where $w^*$ is the induced width of the pseudo tree. The space complexity of m-AOBB-tree follows from the need to propagate the sets of $O(m)$ partial solutions of length $O(n)$. The time overhead for m-AOBB is the same for AND/OR trees and AND/OR graphs. The space complexity of m-AOBB-graph is explained by the need to store $m$ partial solutions for each cached node. □

### 5.3 Algorithm BE+m-BF

It is known that exact heuristics for graphical models can be generated by the Bucket Elimination (BE) algorithm described in Section 2.3.4. We can therefore first compile the exact heuristics along an ordering using BE and then apply m-A* (or m-AOBF, both will work the same at this point), using these exact heuristics. The resulting algorithm is called BE+m-BF. Worst-case analysis of this algorithm will show that it yields the best worst-case complexity compared with any known $m$-best algorithm for graphical models.

**THEOREM 15.** *The time complexity of BE+m-BF is $O(nk^{w*+1} + nm)$ when $n$ is the number of variables, $k$ is the largest domain size, $w^*$ is the induced width of the problem and $m$ is the desired number of solutions. The space complexity is $O(nk^{w*} + nm)$.*

*Proof.* BE's time complexity is $O(nk^{w*+1})$ and space complexity of $O(nk^{w*})$ (Dechter, 1999). Since BE compiles an exact heuristic function, m-A* with this exact heuristic expands nodes for which $f(n) = C^*_j$ only while searching for $i^{th}$ solution. If the algorithm breaks ties in favour of deeper nodes, it will only expand nodes on solution paths. Each path has length $n$, yielding the total time and space complexity of this step of the algorithm equal to $O(n \cdot m)$. □

## 6. Related Work

We can distinguish several primary approaches employed by earlier $m$-best exact algorithms, some mentioned already in the Introduction. Note that some of the original works do not include space complexity analysis and the bounds provided are often our own.

The first and most influential approach was introduced by Lawler (1972). It aimed to use of-the-shelf optimization schemes for best solutions. Lawler showed how to extend any given optimization algorithm to the $m$-best task. At each step, the algorithm seeks the best solution to a re-formulation of the original problem that excludes the solutions already discovered. The scheme has been improved over the years and is still one of the primary strategies for finding the $m$-best solutions. The time and space complexity of Lawler's scheme are $O(n \cdot m \cdot T(n))$ and $O(S(n))$ respectively, where $T(n)$ and $S(n)$ are the time and space complexity of finding a single best solution. For example, if we use AOBF as the underlying optimization algorithm, the use of Lawler's method yields time complexity of $O(n^2 mk^{w^* \log n})$ and space complexity of $O(nk^{w^* \log n})$.

Hamacher and Queyranne (1985) built upon Lawler's work but used as building blocks algorithms that find both the first and second best solutions. Once two best solutions are generated, a new problem is formulated so that the second best solution is the best solution to the new problem. Then, the second best solution for the new problem becomes the overall third best solution and the procedure is repeated. The algorithm has time complexity of $O(m \cdot T_2(n))$ and space complexity of $O(S_2(n))$, where $T_2(n)$ and $S_2(n)$ are respectively the time and space for finding the second best solution. The complexity of this method is always bounded from above by that of Lawler, seeing that Lawler's scheme can be used as an algorithm for finding the second best task. Using m-AOBF to find the two best solutions, we obtain time complexity of $O(2mnk^{w^* \log n})$ and space complexity $O(2nk^{w^* \log n})$.

Nilsson (1998) applied Lawler's method using a join-tree algorithm. On top of that his algorithm reuses computations from previous iterations. His scheme, called max-flow algorithm, uses message-passing on a junction tree to calculate the initial max-marginal functions for each cluster (e.g. probability values of the most probable assignments task) yielding the best solution. Note that this step is equivalent to running the bucket-elimination algorithm. Subsequent solutions are recovered by conditioning search which consult the generated function. The time complexity analysis by Nilsson (1998) is $O(2p|C| + 2mp|R| + pm \log (pm))$, where $p$ is the number of cliques in the joint tree, $|C|$ is the size of the largest clique and $|R|$ is the size of the largest residual (i.e. the number of variables in a cluster but not in neighbouring clusters). The space complexity can be bounded by $O(p|C| + p(|S|))$, where $|S|$ is the size of a separator between the clusters. If applied to a bucket-tree, Nilsson's scheme has time and space complexity of $O(2nk^{w*+1} + mn(2k + \log(mn))$ and $O(nk^{w^*+1} + nk^{w^*})$ respectively, since the the bucket tree has $p = n$ cliques, whose size is bounded by $|C| = k^{w^*+1}$ and the residual in each cluster is $|R| = k$ (the domain of a single variable). Thus the algorithm has better time complexity than all other schemes mentioned so far, except for BE+m-BF.

Two works by de Campos et al. build upon Nilsson's approach, extending it to solving the m-best MAP task using genetic algorithms (de Campos, Gámez, & Moral, 1999) and probability trees (de Campos, Gámez, & Moral, 2004), respectively. These schemes are approximate and the authors provide no theoretical analysis of their complexity.

Fairly recently, Yanover and Weiss (2004) developed an iterative scheme based on belief propagation, called BMMF. At each iteration BMMF uses Loopy Belief Propagation to solve two new problems obtained by restricting the values of certain variables. When applied to a junction tree having induced width $w^*$ (whose largest cluster size is bounded by $k^{w^*+1}$) it is an exact algorithm having time complexity $O(2mnk^{w*+1})$ and space complexity $O(nk^{w*} + mnk)$. When applied on a loopy graph, BMMF is not guaranteed to find exact solutions.

Another approach based on Lawler's idea uses optimization via the LP-relaxation (Wainwright & Jordan, 2003), formulated by Fromer and Globerson (2009). Their method, called Spanning TRee Inequalities and Partitioning for Enumerating Solutions, or STRIPES, also partitions the search space, while systematically excluding all previously determined assignments. At each step new constraints are added to an LP optimization problem, which is solved via an off-the-shelf LP-solver. In general, the algorithm is approximate. However, on trees or junction-trees it is exact if the underlying LP solver reports solutions within the time limit. PESTEELARS is an extension of the above scheme by Batra (2012) that solves the LP relaxation using message-passing approach that, unlike conventional LP solvers, exploits the structure of the problem's graph. The complexity of these LP-based algorithm is hard to characterize using the usual graph parameters.

Another approach extends variable elimination (or dynamic programming) schemes to directly obtain the $m$ best solutions. In our recent paper (Flerova et al., 2011) we extended bucket elimination and mini-bucket elimination to the $m$-best solutions task, yielding an exact scheme called *elim-m-opt* and its approximate version called *mbe-m-opt*, respectively. This work also embeds the $m$-best optimization task within the semi-ring framework. The time and space complexities of algorithm elim-m-opt are bounded by $O(m \log mnk^{w^*+1})$ and $O(mnk^{w^*})$, respectively.

Two related dynamic programming based ideas are by Seroussi and Golmard (1994) and Elliot (2007). Seroussi and Golmard extract the $m$ solutions directly, by propagating the $m$ best partial solutions along a junction tree. Given a junction tree with $p$ cliques, largest cluster size $|C|$, separator size bounded by $|S|$ and branching degree $deg$, the time complexity of the algorithm is $O(m^2 \cdot p \cdot |C| \cdot deg)$ and the space complexity is $O(m \cdot p \cdot |S|)$. Adapted to a bucket tree, this algorithm has time complexity $O(m^2 nk^{w^*+1} deg)$ and space complexity of $O(mnk^{w^*})$. Elliot propagates the $m$ best partial solutions along a representation called Valued And-Or Acyclic Graph, also known as a smooth deterministic decomposable negation normal form (sd-DNNF) (Darwiche, 2001). The time complexity of Elliot's algorithm is $O(nk^{w^*+1} m \log(m \cdot deg))$ and the space complexity is $O(mnk^{w^*+1})$.

Several methods focus on search schemes obtaining multiple optimal solution for the $k$ shortest paths task (KSP). For a survey see the paper by Eppstein (1994). The majority of these algorithms assume that the entire search graph is available in memory and thus are not directly applicable. A recent exception is by Aljazzar and Leue (2011), whose $K^*$ algorithm finds the $k$ shortest paths during search "on-the-fly" and thus can be potentially useful for graphical models. The algorithm interleaves A* search on the problem's implicit graph $G$ and Dijkstra's algorithm (1959) on a spe-

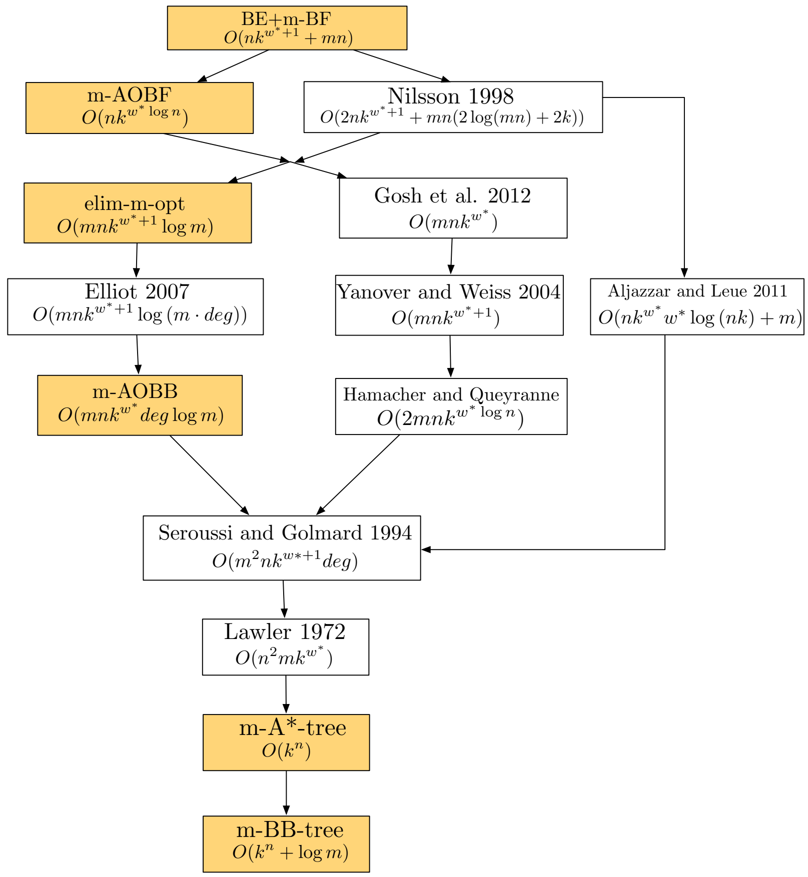



Figure 8: Time complexity comparison of the exact $m$-best algorithms specified for a bucket tree. A parent node in the graph has a better complexity than its children. Problem parameters: $n$ - number of variables, $k$ - largest domain size, $w^*$ - induced width, $deg$ - the degree of the join (bucket) tree. Our algorithms are highlighted.

cific path graph structure denoted $P(G)$. $P(G)$ is a directed graph, the vertices of which correspond to edges in the problem graph $G$. Given a consistent heuristic, $K^*$, when applied to an AND/OR search graph is time and space $O(nk^{w^*} w^* \log(n \cdot k) + m)$.

More recently, Gosh, et al., (2012) introduced a best-first search algorithm for generating ordered solutions for *explicit* AND/OR trees or graphs. The time complexity of their algorithm can be bounded by $O(mnk^{w*})$, when applied to a context-minimal AND/OR search graph. The space complexity is bounded by $O(s \cdot nk^{w*+1})$, where $s$ is the number of candidate solutions generated and stored by the algorithm, hard to quantify using usual graph parameters. However, this approach, which explores the space of complete solutions, does not seem to be practical for graphical models because it requires the entire AND/OR search space to be fully explicated in memory before attempting to generate even the second best solution. In contrast, our algorithms generate the $m$ best solutions while traversing the space of partial solutions.

Figure 8 provides a visual comparison between the worst-case time complexity bounds of the discussed schemes in a form of a directed graph, where each node corresponds to an algorithm and a parent in the graph has a better complexity than its child. We assume in our analysis that $n >> m > k > 2$.

We see that the best emerging scheme, as far as worst-case performance goes is BE+m-BF. However, since it requires compiling of the exact heuristics, it is often infeasible. We see also that algorithm elim-m-opt appears to have relatively good time complexity superior, for example, to m-AOBB search. However, as we showed in our previous work (Flerova et al., 2011), it is quite limited empirically. Note that the worst-case analysis often fails to capture the practical performance, either because the algorithms that have good worst-case performance require too much memory, or because it ignores the power of the cost function in bounding the performance.

## 7. Experimental Evaluation

Our experiments consist of two parts: evaluation of the $m$-best search algorithms on the benchmarks from recent UAI and Pascal2 competitions and comparison of our schemes with some of the previously developed algorithms on randomly generated networks, whose parameters and structure had to be restricted due to the limitations of the available implementations of the competing schemes. We defer the discussion of the second part of experiments until Section 7.5, concentrating now on the evaluation our $m$-best search schemes only.

### 7.1 Overview and Methodology

We used 6 benchmarks, all, except for binary grids, came from real-world domains:

1. Pedigrees
2. Binary grids
3. WCSP
4. Promedas
5. Proteins
6. Segmentation

The **pedigrees benchmark (“pedigree*”)** was used in the UAI 2008 competition.[3] They arise from the domain of genetic linkage analysis and are associated with the task of haplotyping. A haplotype is a sequence of alleles at different loci inherited by an individual from one parent, and the two haplotypes (maternal and paternal) of an individual constitute this individual’s genotype. When genotypes are measured by standard procedures, the result is a list of unordered pairs of alleles, one pair for each locus. The maximum likelihood haplotype problem consists of finding a joint haplotype configuration for all members of the pedigree which maximizes the probability of data. 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 (see the paper by Fishelson and Geiger (2002) for more details).

3. http://graphmod.ics.uci.edu/group/Repository

| Benchmark | # inst | $n$ | $k$ | $w^*$ | $h_T$ |
|---|---|---|---|---|---|
| Pedigrees | 13 | 581-1006 | 3-7 | 16-39 | 52-104 |
| Grids | 32 | 144-2500 | 2 | 15-90 | 48-283 |
| WCSP | 61 | 25-1057 | 2-100 | 5-287 | 11-337 |
| Promedas | 86 | 197-2113 | 2 | 5-120 | 34-187 |
| Proteins | 72 | 15-242 | 18-81 | 5-16 | 7-44 |
| Segmentation | 47 | 222-234 | 2-21 | 15-18 | 47-67 |

Table 1: Benchmark parameters: # inst - number of instances, $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h_T$ - pseudo tree height.

In each of the **binary grid networks (“50-*”**, **“75-*”** and **“90-*”)**[4] the nodes corresponding to binary variables are arranged in an $N$ by $N$ square and the functions are defined over pairs of variables and are generated uniformly randomly.

The **WCSP (“*.wcsp”)** benchmark includes random binary WCSPs, scheduling problems from the SPOT5 benchmark, and radio link frequency assignment problems, providing a large variety of problem parameters.

**Protein side-chain prediction (“pdb*”)** networks correspond to side-chain conformation prediction tasks in the protein folding problem (Yanover, Schueler-Furman, & Weiss, 2008). The resulting instances have relatively few nodes, but very large variable domains, generally rendering most instances very complex.

**Promedas (“or_chain_*”)** and **segmentation (“*_s.binary”)** are probabilistic networks that come from the set of problems used in the 2011 Probabilistic Inference Challenge.[5] **Promedas** instances are based on a Bayesian network model developed for expert systems for medical diagnosis (Wemmenhove, Mooij, Wiegerinck, Leisink, Kappen, & Neijt, 2007). **Segmentation** is a common benchmark used in computer vision, modeling the task of image segmentation as an MPE problem, namely assigning a label to every pixel in an image, such that pixels with the same label share certain characteristics.

Table 1 describes the benchmark parameters: # inst - number of instances, $n$ - number of variables, $k$ - maximum domain size, $w^*$ - induced width of the ordering used, $h_T$ - pseudo-tree height. The induced width is not only one of the crucial parameters indicating the difficulty of the problem, but the difference between the induced width and the mini-bucket i-bound signifies the strength of the heuristic. When the i-bound is considerably smaller than the induced width, the heuristic is weak, while the i-bound equal or greater than the induced width yields an exact heuristic, which in turn yields much faster search. Clearly, a large number of variables, a high domain size or a large pseudo tree height suggest harder problems.

#### 7.1.1 ALGORITHMS

We can distinguish 6 algorithms: **BE+m-BF**, **m-A*-tree** and **m-BB-tree** exploring a regular OR search tree and their modifications that explore an AND/OR search tree, denoted **m-AOBF-tree** and **m-AOBB-tree**. We also consider a variant of m-AOBF that explores the AND/OR search graph **m-AOBF-graph**. We did not implement the m-AOBB over AND/OR search graph, because the

4. http://graphmod.ics.uci.edu/repos/mpe/grids/
5. http://www.cs.huji.ac.il/project/PASCAL/archive/mpe.tgz

overhead due to book-keeping looked prohibitive. We used m-AOBF as representative for AND/OR graph search, and as we will see it proved indeed to be not cost-effective. All algorithms were guided by pre-compiled mini-bucket heuristics, described in Section 2.3.4. We used 10 i-bounds, ranging from 2 to 22. However, for some hard problems computing the mini-bucket heuristic with the larger i-bounds proved infeasible, so the actual range of i-bounds varies among the benchmarks and among instances within a benchmark. All algorithms were restricted to a static variable ordering computed using a min-fill heuristic (Kjærulff, 1990). Both AND/OR schemes used the same pseudo tree. In our implementation algorithms m-BB, m-BF and m-AOBF break ties lexicographically, while algorithm m-AOBB solves the independent subproblems rooted at an AND node in increasing order of their lower bound heuristic estimates.

The algorithms were implemented in C++ (32-bit) and the experiments were run on a 2.6GHz quad-core processor. The memory limit was set for 4 GB per problem, the time limit to 3 hours. We report the CPU time (in seconds) and the number of nodes expanded during search. For uniformity we consider the task throughout to be the maximization-product problem, also known as Most Probable Explanation task (MPE or MAP). We focus on complete and exact solutions only and thus do not report the results if the algorithm found less than $m$ solutions (for best-first schemes) or if the optimality of the solutions was not proved (for branch and bound schemes).

#### 7.1.2 GOALS OF THE EMPIRICAL EVALUATION

We will address the following aspects:
1. Comparing best-first and depth-first branch and bound approaches
2. The impact of AND/OR decomposition on the search performance
3. Scalability of the algorithms with the number of required solutions $m$
4. Comparison with earlier proposed algorithms

### 7.2 The Main Trends in the Behavior of the Algorithms

Tables 2, 4, 6, 8, 10, and 12 present for each of our algorithms the raw results in the form of runtime in seconds and number of expanded nodes for select instances from each benchmark, selected to best illustrate the prevailing trends. For each benchmark we show the results for two values of the i-bound, corresponding, in most cases, to relatively weak and strong heuristics. Note that the i-bound has no impact on the BE+m-BF, since it always calculates the exact heuristic. We show three values of number of solutions $m$, equal to 1 (ordinary optimization problem), 10 and 100.

In order to see the bigger picture, in Figures 9-14 we show bar charts representing for each benchmark a median runtime and a number of instances solved by each algorithm for a particular strength of the heuristic (i-bound) for $m \in \{1, 2, 5, 10, 100\}$. The y-axis is on logarithmic scale. The numbers above the bars indicate the actual values of median time in seconds and number of solved instances, respectively. It is important to note that in these figures we only account for harder instances, for which the i-bound did not yield exact heuristic. We acknowledge that the median times are not strictly comparable since they are calculated over a varied number of instances solved by each algorithm. However, this metric is robust to outliers and gives us an intuition about the algorithms' relative success. In addition, Tables 3, 5, 7, 9, 11 and 13 show for each benchmark the number of instances, for which a given algorithm is the best in terms of runtime and in terms of number of expanded nodes. If several algorithms show the same best result, it counts towards the score of all of them.

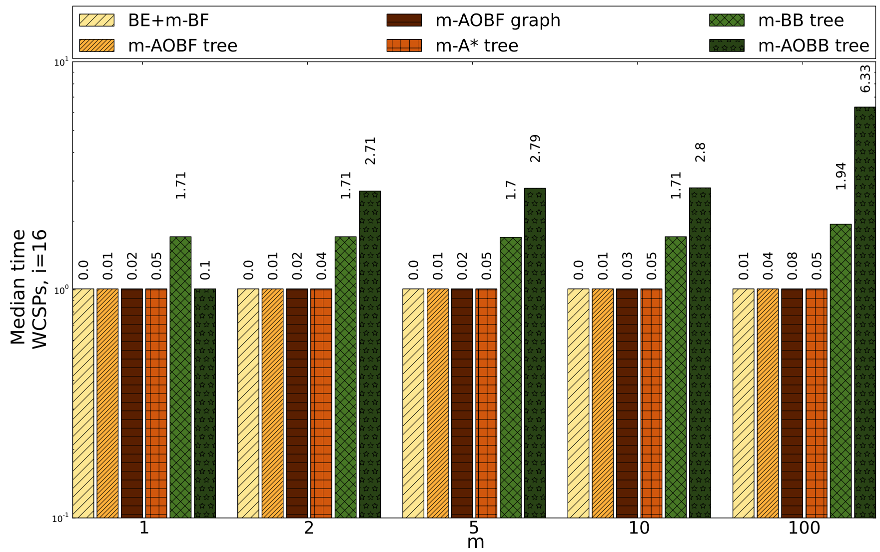
| instance $(n,k,w^*,h)$ | i-bound | algorithm | m=1 time | m=1 nodes | m=10 time | m=10 nodes | m=100 time | m=100 nodes |
|---|---|---|---|---|---|---|---|---|
| 503.wcsp | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | Timeout | | Timeout | | Timeout | |
| | | BE+m-BF | **0.05** | **6647** | **0.06** | **6671** | **0.06** | **6984** |
| (144, 4, 9, 44) | 8 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | 1269.0 | 348648825 | 1275.65 | 348648869 | 1255.46 | 348651775 |
| | | m-AOBB tree | 22.99 | 2320223 | 164.72 | 12110559 | 8010.42 | 568148386 |
| | | BE+m-BF | **0.05** | **6647** | **0.06** | **6671** | **0.06** | **6984** |
| myciel5g_3.wcsp | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | OOT | | OOT | | OOT | |
| | | m-AOBB tree | **1461.76** | **46419482** | **2389.32** | **74629839** | **3321.47** | **83802828** |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (47,2, 19, 46) | 8 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | 151.49 | 33563300 | **152.27** | 33609110 | **148.08** | 36255491 |
| | | m-AOBB tree | **107.03** | **4274313** | 185.66 | **7245553** | 251.98 | **8319419** |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| satellite01ac.wcsp | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | **5760.25** | **14260410** | OOT | | OOT | |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (79, 8, 19, 56) | 8 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 793.56 | 2579416 | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | **484.69** | **1530768** | **551.67** | **1995114** | **858.29** | **3507104** |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| 29.wcsp | 4 | m-AOBF tree | 10.23 | 464134 | 10.22 | 464182 | 10.29 | 464698 |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 11.59 | 938812 | 11.58 | 938869 | 11.57 | 939508 |
| | | m-BB tree | 12.96 | 2243619 | 12.89 | 2245137 | 12.77 | 2279587 |
| | | m-AOBB tree | 1.81 | 87717 | 2.63 | 147851 | 115.3 | 9189667 |
| | | BE+m-BF | **0.0** | **111** | **0.0** | **168** | **0.01** | **739** |
| (83, 4, 18, 58) | 8 | m-AOBF tree | 0.05 | 2347 | 0.05 | 2395 | 0.08 | 2899 |
| | | m-AOBF graph | 0.09 | 1401 | 0.09 | 1447 | 0.13 | 1482 |
| | | m-A* tree | 0.02 | 2098 | 0.02 | 2155 | 0.02 | 2724 |
| | | m-BB tree | 0.17 | 37629 | 0.17 | 38463 | 0.25 | 55125 |
| | | m-AOBB tree | 0.02 | 1577 | 0.33 | 24239 | 79.38 | 6731546 |
| | | BE+m-BF | **0.0** | **111** | **0.0** | **168** | **0.01** | **739** |

Table 2: WCSP: CPU time (in seconds) and number of nodes expanded. A 'Timeout' stands for exceeding the time limit of 3 hours. 'OOM' indicates out of 4GB memory. In bold we highlight the best time and number of nodes for each $m$. Parameters: $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h$ - pseudo tree height.

We next provide some elaboration and interpretation of the results.

### 7.2.1 WCSP

Table 2 shows the results for two values of the i-bound for select instances chosen to best illustrate the common trends seen across the WCSP benchmark. Figure 9 presents the median time and number of solved instances for each algorithm for i=16. Table 3 shows for the same i-bound (i=16) the number of instances for which each of the schemes had the best runtime and best number of expanded nodes. For many problem instances of this benchmark the mini-bucket elimination with the large i-bounds is infeasible, thus we present the results for a small and a medium i-bounds.



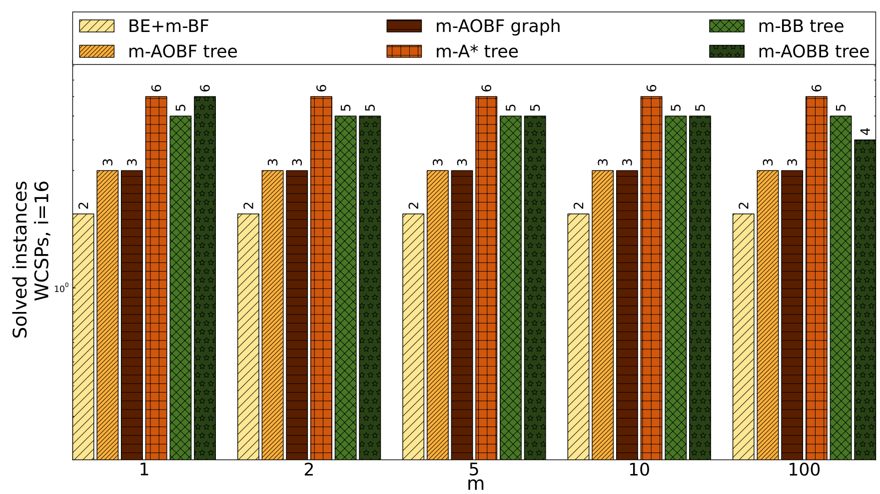




Figure 9: Median time and number of solved instances (out of 49) for select values of $m$ for WCSPs, i-bound=16. Numbers above bars - actual values of time (sec) and # instances. Total instances in benchmark: 61, discarded instances due to exact heuristic: 12.

**BE+m-BF.** As suggested by theory, whenever BE+m-BF does not run out of memory, it is the most efficient scheme. See for example Table 2, 503.wcsp and 29.wcsp. However, calculation of the exact heuristic is only feasible for easier instances and, as Figure 9 shows, it can only solve

| algorithm | WCSPs: # inst=61, $n$=14-1058<br>$k$=2-100, $w^*$=6-287, $h_T$=8-585, i-bound=16 | | | | |
|---|---|---|---|---|---|
| | m=1 | m=2 | m=5 | m=10 | m=100 |
| | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN |
| Not solved | 43 | 43 | 43 | 43 | 43 |
| m-AOBF tree | 1 / 2 | 1 / 1 | 0 / 1 | 1 / 1 | 0 / 0 |
| m-AOBF graph | 1 / 2 | 0 / 2 | 0 / 2 | 0 / 2 | 0 / 3 |
| m-A* tree | 5 / 1 | 4 / 3 | 5 / 3 | 4 / 3 | 5 / 2 |
| m-BB tree | 1 / 0 | 2 / 0 | 1 / 0 | 1 / 0 | 0 / 0 |
| m-AOBB tree | 1 / 2 | 2 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| BE+m-BF | 2 / 1 | 2 / 1 | 2 / 1 | 2 / 1 | 2 / 1 |

Table 3: Number of instances, for which each algorithm has the best runtime (#BT) and best number of expanded nodes (#BN), WCSPs. Out of 61 instances 12 have exact heuristics. The table accounts for remaining 49, i-bound= 16 .

2 WCSP instances. As seen in Table 3, on these two instances BE+m-BF demonstrated the best runtime among all the schemes.

**m-AOBB-tree.** For a number of problems for small values of $m$, m-AOBB-tree is superior to m-BB-tree both in terms of the runtime and in number of expanded nodes. For example, for 29.wcsp, i=4, m=10 m-AOBB-tree requires 2.63 seconds to solve the problem and expands 147851 nodes while the runtime of m-BB-tree is 12.89 seconds and it expands 2245137 nodes. However, on the majority of instances m-AOBB-tree is slower than all other schemes, as seen in Figure 9. Moreover, m-AOBB-tree scales poorly with the number of solutions. For $m = 100$ it very often has both the worst runtime and the largest explored search space among all the schemes, e.g. i=8, 503.wcsp. Such striking decrease in performance as $m$ grows is consistent across various benchmarks and can be explained by the need to combine sets of partial solutions at AND nodes as we described earlier. The overhead connected to AND/OR decomposition also accounts for the larger time per node ratio of m-AOBB-tree, compared to other schemes. For example, in Table 2 for instance myciel5g_3.wcsp, i=8, for m=10 and m=100 m-AOBB-tree expands less nodes than m-BB-tree, but its runtime is larger. Nevertheless, m-AOBB-tree has its benefits. Since it is more space efficient than the other algorithms, it is often the only scheme able for find solutions for the harder instances, especially when the heuristic is weak, as we see for myciel5g_3.wcsp for i=4 and satellite01ac.wcsp for both i=4 and i=8, respectively.

**m-BB-tree.** In Figure 9 we see that m-BB-tree solves almost the same number of problems as m-AOBB-tree while having considerably better median time.

**m-AOBF-tree and m-AOBF-graph.** Unsurprisingly, best-first search algorithms often run out of space on problems feasible for branch and bound, such as 503.wcsp and myciel5g_3.wcsp for i=8. m-AOBF-based schemes are overall inferior to other algorithms, solving, as Figure 9 shows, the least number of problems. Both schemes run out of memory much more often than m-A*-tree. We believe this is due to overhead of maintaining an OPEN list of partial solution trees, as opposed of an OPEN list of individual nodes as m-A*-tree does. Whenever the m-AOBF schemes do manage to find solutions, as for example for instance 29.wcsp, i=8, m-AOBF-graph explores the smallest search space among the schemes, except for BE+m-BF. At the same time m-AOBF-tree sometimes expands more nodes compared not only to m-AOBF-graph, but also to m-A*-tree, which is not what we would normally expect, since m-A*-tree traverses an OR search space, which is inherently larger than the AND/OR one. However, it is important to remember that though better space efficiency of AND/OR schemes is often observed, it is not guaranteed. Many factors, such as tie breaking between the nodes with the same value of evaluation function, can impact the performance of m-AOBF-tree. m-AOBF-tree and m-AOBF-graph have almost the same median time and number of solved problems, as seen in Figure 9.

**m-A*-tree.** Out of the three best first algorithms m-A*-tree is overall the best. In Figure 9 we see that it solves more instances than all other schemes for all values of $m$ and its median runtime is close to that of BE+m-BF. Table 3 proves that for i-bound=16 this scheme is the fastest among all the schemes on largest number of instances, showing best runtime on 4-5 instances, depending on $m$. This is explained in part by the relatively reduced overhead for maintaining the search space and OPEN list in memory, compared for example with the m-AOBF schemes.

### 7.2.2 PEDIGREES

Table 4 displays the results for select instances from the Pedigree benchmark for two i-bounds each. Overall, the difference between the results for the algorithms greatly diminishes as the heuristic strength increases. Figure 10 shows the median time and number of solved instances for select values of $m$ for i=16. The number of instances for which each of the schemes had the best runtime and best number of expanded nodes for the same i-bound is presented in Table 5.

**BE+m-BF.** Here too BE+m-BF is often superior to other algorithms, especially when the other schemes use lower values of the i-bound, e.g. pedigree23, i=12, all $m$s. For large i-bounds and thus more accurate heuristics the difference is much smaller. Moreover, sometimes BE+m-BF can be slower than other schemes, due to the time required to calculate the exact heuristic, e.g. pedigree23, i=16. Table 5 shows that BE+m-BF is overall the fastest. We see that on the Pedigree benchmark this algorithm is quite successful, as is evident from the many instances it solved (see Figure 10).

**m-AOBB-tree.** For low values of $m$ m-AOBB-tree is slightly superior to all other algorithms, solving the most number of instances, (see Figure 10). On the other hand, its median time is the largest. It fails to solve any instances for m=100. From Table 4 we see that m-AOBB-tree is the slowest, (e.g., pedigree23, i=16, all $m$s). Yet, for instance pedigree33, i=12, m=1, this scheme is the only one to find any solution.

**m-BB-tree.** As expected, m-BB-tree is inferior to the best-first search schemes unless the latter run out of memory. As was the case for WCSP, this scheme is often faster than m-AOBB-tree, for example, on pedigree30, i=16, all values of $m$. The bar charts show that m-BB-tree has the second worst median time for all values of $m$, but solves the most number of problems for m=100.

**m-AOBF schemes.** Both m-AOBF algorithms are unsuccessful on the Pedigree benchmark. They often run out of memory even for $m = 1$ (e.g. pedigree33, i=22). For most instances where they do report solution m-AOBF-tree is faster than m-AOBF-graph, though the difference is usually not very large.

**m-A*-tree.** As we saw for WCSPs, on some pedigree instances m-A*-tree is faster than the two m-AOBF schemes, as seen in Figure 10, all values of $m$. Moreover, it is superior on harder instances

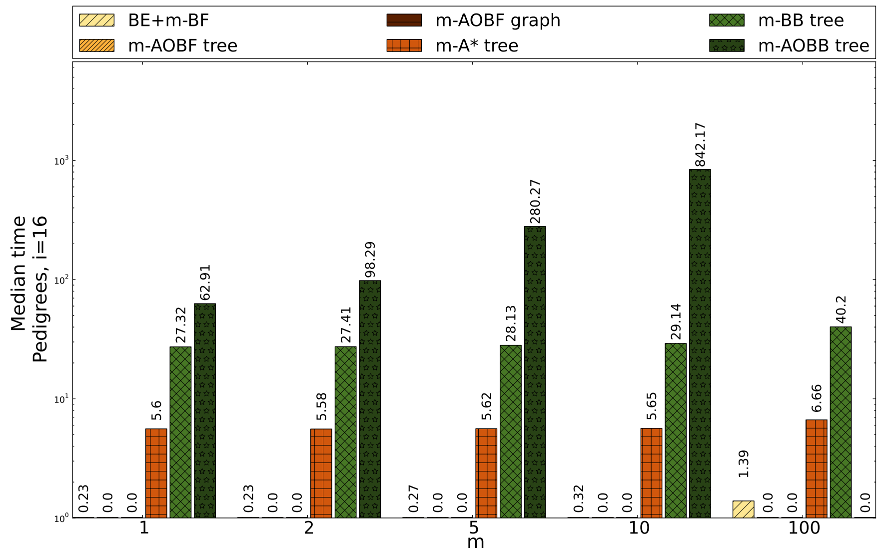
| instance ($n$,$k$,$w^*$,$h$) | i-bound | algorithm | number of solutions m=1 time | m=1 nodes | m=10 time | m=10 nodes | m=100 time | m=100 nodes |
|---|---|---|---|---|---|---|---|---|
| pedigree33 | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | **7814.77** | **145203641** | Timeout | | Timeout | |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (798, 4, 24, 132) | 22 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **1.32** | 73625 | **1.55** | **77138** | **3.76** | **112422** |
| | | m-BB tree | 2.98 | 145717 | 4.15 | 177397 | 21.48 | 655141 |
| | | m-AOBB tree | 2.88 | **70644** | Timeout | | Timeout | |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| pedigree30 | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | 2510.59 | 33453995 | OOT | | OOT | |
| | | BE+m-BF | **7.90** | **6423** | **7.99** | **7611** | **9.22** | **23028** |
| (1290, 5, 20, 105) | 16 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 65.43 | 4866388 | 65.86 | 4867551 | 67.01 | 4882985 |
| | | m-BB tree | 84.28 | 12243789 | 85.72 | 12298570 | 127.25 | 13027245 |
| | | m-AOBB tree | 594.36 | 6907399 | Timeout | | Timeout | |
| | | BE+m-BF | **7.90** | **6423** | **7.99** | **7611** | **9.22** | **23028** |
| pedigree23 | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 8.44 | 713664 | 8.46 | 715729 | 11.15 | 904802 |
| | | m-BB tree | 23.0 | 4446224 | 24.56 | 4676953 | 35.46 | 6179124 |
| | | m-AOBB tree | 32.11 | 831096 | 1077.9 | 75355901 | Timeout | |
| | | BE+m-BF | **7.11** | **630** | **7.24** | **2482** | **7.68** | **19297** |
| (403, 5, 21, 64) | 16 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **0.52** | 53862 | **0.58** | 55927 | **1.17** | 85300 |
| | | m-BB tree | 4.5 | 837288 | 5.61 | 959931 | 14.14 | 1641751 |
| | | m-AOBB tree | 13.39 | 346145 | 713.73 | 50252107 | OOM | |
| | | BE+m-BF | 7.11 | **630** | 7.24 | **2482** | 7.68 | **19297** |
| pedigree20 | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 27.0 | 2321986 | 26.66 | 2324701 | 27.47 | 2353927 |
| | | m-BB tree | 34.23 | 7239379 | 37.63 | 7434961 | 66.81 | 9155747 |
| | | m-AOBB tree | 88.85 | 4365855 | 1019.76 | 63940515 | Timeout | |
| | | BE+m-BF | **24.95** | **491** | **24.99** | **3482** | **26.16** | **32643** |
| (438, 5, 20, 65) | 16 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **24.94** | 2279192 | **24.93** | 2281907 | **25.97** | 2311133 |
| | | m-BB tree | 27.32 | 5857301 | 29.14 | 5946423 | 40.2 | 6617200 |
| | | m-AOBB tree | 54.46 | 2531322 | 970.61 | 59333828 | Timeout | |
| | | BE+m-BF | 24.95 | **491** | 24.99 | **3482** | 26.16 | **32643** |

Table 4: Pedigrees: CPU time (in seconds) and number of nodes expanded. A 'Timeout' stands for exceeding the time limit of 3 hours. 'OOM' indicates out of 4GB memory. In bold we highlight the best time and number of nodes for each $m$.Parameters: $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h$ - pseudo tree height.

infeasible for both m-AOBF schemes and BE+m-BF, e.g. pedigree23, i=16. As shown in Figure 10, it solves 5 instances for i=16, for all $m$s, which is the best or second best results, depending on the number of solutions. However, the median time of m-A*-tree is considerably larger than that of BE+m-BF, while for this i-bound the latter solves only a single instance less.

### 7.2.3 BINARY GRIDS

Table 6 shows the results for select instances from the grid networks domain. Figure 11 shows the median runtime and number of solved instances for i=18, while Table 7 presents the number of instances, for which an algorithm is the best, for the same i-bound. Most trends in the algorithms'



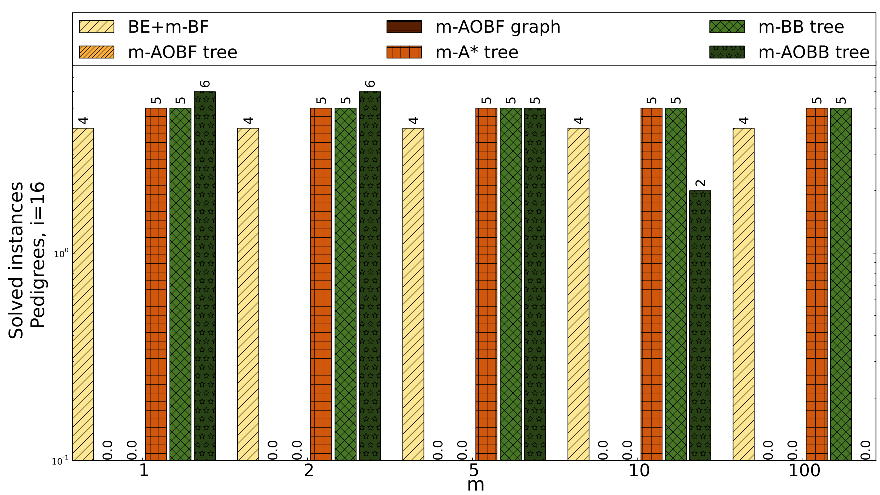




Figure 10: Median time and number of solved instances (out of 12) for select values of $m$ for Pedigrees, i-bound=16. Numbers above bars - actual values of time (sec) and # instances. Total instances in benchmark: 13 , discarded instances due to exact heuristic: 1.

behavior observed on WCSP and Pedigree benchmarks can be also noticed on the Grid benchmark as well. In particular, m-AOBB-tree is very successful when $m$ is small, even solving the most instances, as seen in Figure 11. But it shows worse results for m=100 and for any number of solutions has the largest median time. m-BB-tree has smaller median time for all $m$s, but is still

| algorithm | Pedigrees: # inst=13, $n$=335-1290<br>$k$=3-7, $w^*$=15-47, $h_T$=52-204, i-bound=16 | | | | |
|---|---|---|---|---|---|
| | m=1 | m=2 | m=5 | m=10 | m=100 |
| | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN |
| Not solved | 6 | 6 | 6 | 6 | 6 |
| m-AOBF tree | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| m-AOBF graph | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| m-A* tree | 1 / 1 | 1 / 1 | 1 / 1 | 1 / 1 | 1 / 1 |
| m-BB tree | 0 / 0 | 0 / 0 | 1 / 1 | 1 / 1 | 1 / 1 |
| m-AOBB tree | 1 / 1 | 1 / 1 | 0 / 0 | 0 / 0 | 0 / 0 |
| BE+m-BF | 4 / 4 | 4 / 4 | 4 / 4 | 4 / 4 | 4 / 4 |

Table 5: Number of instances, for which each algorithm has the best runtime (#BT) and best number of expanded nodes (#BN), Pedigrees. Out of 13 instances 1 have exact heuristics. The table accounts for remaining 12, i-bound= 16 .

considerably slower than any of the best-first schemes. m-A*-tree presents the best compromise between a small medium running time and a relatively large number of solved instances. Table 7 shows that for majority of grid instances it is the fastest algorithm. The two m-AOBF schemes have results quite similar to each other, solving almost the same number of instances for all $m$s with little difference in median runtimes, as is shown in Figure 11. They both are consistently inferior to all other schemes except for BE+m-BF, which often runs out of memory. The main difference of the Grid benchmark compared with the previously discussed domains lies in the behaviour of BE+m-BF when the i-bound is high. Even though it expands less nodes, for many problems BE+m-BF is slower than the other schemes due to the large time required to compute the exact heuristic. For example, on grid 75-19-5, i=18, for m=10 the runtime of BE+m-BF is 143.11 seconds, while even m-AOBB-tree, known to be slow, terminates in just 94.0 seconds. At the same time, for this instance BE+m-BF explores the smallest search space for all values of $m$.

#### 7.2.4 PROMEDAS

Table 8 shows the results for the Promedas benchmark. Figure 12 presents the median time and number of solved instances for the benchmark for i=16. Table 9 shows for the same i-bound the number of instances for which each of the schemes had the best runtime and best number of expanded nodes. A significant fraction of the instances is not solved by any of the algorithms, especially for low and medium i-bounds. Unlike the other benchmarks, m-AOBB-tree not only solves the most instances for small $m$s, but also is quite successful for m=100, solving only one instance less than the best scheme for this value of $m$, m-BB-tree. Moreover, sometimes m-AOBB-tree is the only scheme to report any solutions, especially for weak heuristic, e.g. or_chain_50.fg and or_chain_212.fg, i=12. BE+m-BF runs out of memory on most instances, as seen in Table 8. Overall, the variance of the algorithms' performance is more significant for Promedas than for the previously discussed benchmarks. For example, as we see in Figure 12, for i=16 m-A*-tree, m-BB-tree and m-AOBB-tree solve between 25 and 33 instances for $m \in [1, 10]$, while BE+m-BF and both m-AOBF-based schemes solve only between 4 and 8 instances. Table 9 demonstrates that m-A*-tree most often is the fastest of the algorithms.

| instance $(n,k,w^*,h)$ | i-bound | algorithm | number of solutions m=1 time | m=1 nodes | m=10 time | m=10 nodes | m=100 time | m=100 nodes |
|---|---|---|---|---|---|---|---|---|
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| 50-15-5 | 10 | m-A* tree | 8.83 | 866865 | 11.97 | 1177549 | 20.22 | 1931039 |
| | | m-BB tree | 8.75 | 1967152 | 11.91 | 2647393 | 22.06 | 4708311 |
| | | m-AOBB tree | 3.29 | 251502 | 34.28 | 2485393 | Timeout | |
| | | BE+m-BF | OOM | OOM | OOM | OOM | OOM | OOM |
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| (400, 2, 27, 99) | | m-AOBF graph | OOM | | OOM | | OOM | |
| | 18 | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | **347.24** | **17332742** | **692.59** | **28676212** | **2277.92** | **75442102** |
| | | BE+m-BF | OOM | OOM | OOM | OOM | OOM | OOM |
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| 50-17-5 | | m-AOBF graph | OOM | | OOM | | OOM | |
| | 10 | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | 82.95 | 3290939 | 368.18 | 16431707 | Timeout | |
| | | BE+m-BF | **18.45** | **289** | **18.47** | **1220** | **18.67** | **9534** |
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| (289, 2, 22, 84) | | m-AOBF graph | OOM | | OOM | | OOM | |
| | 18 | m-A* tree | **0.35** | 51205 | **0.39** | 55783 | **0.89** | 104621 |
| | | m-BB tree | 1.39 | 355700 | 1.83 | 421798 | 4.92 | 892065 |
| | | m-AOBB tree | 1.79 | 85289 | 116.77 | 7505310 | Timeout | |
| | | BE+m-BF | 18.45 | **289** | 18.47 | **1220** | 18.67 | **9534** |
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| 90-20-5 | 10 | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | **347.24** | **17332742** | **692.59** | **28676212** | **2277.92** | **75442102** |
| | | BE+m-BF | OOM | OOM | OOM | OOM | OOM | OOM |
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| (400, 2, 27, 99) | 18 | m-A* tree | **1.3** | **153399** | **1.88** | **211547** | **3.53** | **362344** |
| | | m-BB tree | 74.83 | 14968683 | 76.93 | 15403354 | 85.42 | 16631321 |
| | | m-AOBB tree | 46.53 | 2450725 | 118.26 | 4940247 | 563.22 | 18306275 |
| | | BE+m-BF | OOM | OOM | OOM | OOM | OOM | OOM |
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| 75-19-5 | 10 | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | 3591.1 | **1**19431966 | Timeout | | Timeout | |
| | | BE+m-BF | **143.11** | **361** | **143.11** | **2330** | **144.11** | **16897** |
| | | m-AOBF tree | OOM | | OOM | | OOM | |
| (361, 2, 25, 89) | | m-AOBF graph | OOM | | OOM | | OOM | |
| | 18 | m-A* tree | **14.3** | 1609506 | **18.76** | 2029844 | **28.04** | 2995437 |
| | | m-BB tree | 16.27 | 4005082 | 22.28 | 5320573 | 37.26 | 8191215 |
| | | m-AOBB tree | 39.66 | 1367955 | 94.0 | 3480629 | Timeout | |
| | | BE+m-BF | 143.11 | **361** | 143.11 | **2330** | 144.11 | **16897** |

Table 6: Grids: CPU time (in seconds) and number of nodes expanded. A 'Timeout' stands for exceeding the time limit of 3 hours. 'OOM' indicates out of 4GB memory. In bold we highlight the best time and number of nodes for each $m$. Parameters: $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h$ - pseudo tree height.

### 7.2.5 PROTEIN

Table 10 shows select Protein instances for i=4 and i=8, respectively. Figure 13 and Table 11 show the summary of the results for i=4. This benchmark is fairly difficult due to very large domain size (up to 81). The heuristic calculation is not feasible for higher i-bounds. In particular, BE+m-BF has considerable problems in calculating the exact heuristic. Even for low i-bounds only relatively easy instances are solved. Note that for instances pdb1ctk and pdb1dlw i-bound=8 yields exact heuristic. Both m-AOBF-tree and m-AOBF-graph fail to find any solutions within the memory limit on the majority of instances, e.g., pdb1b2v and pdb1cxy, i=4. There is not much difference between the runtimes of all algorithms, with an exception of m-AOBB-tree. For example, for pdb1b2v, i=8,

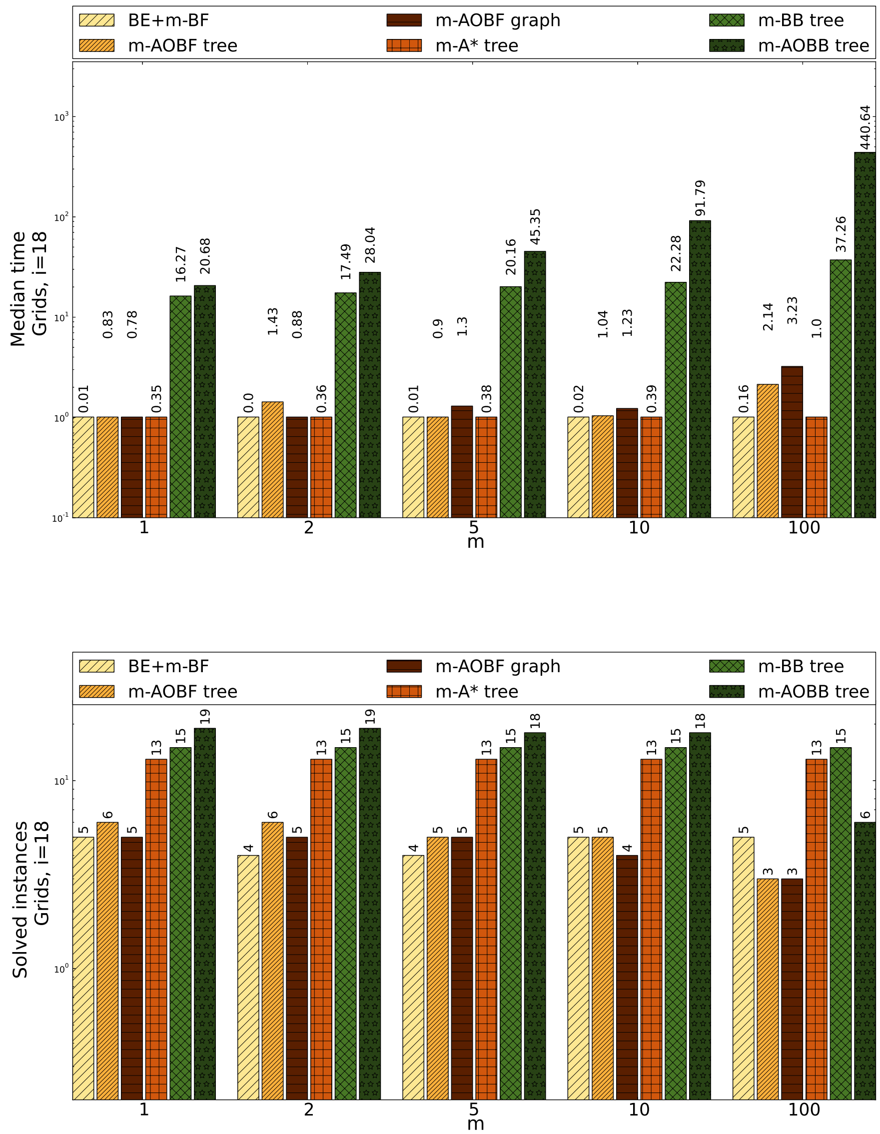



Figure 11: Median time and number of solved instances (out of 31) for select values of $m$ for Grids, i-bound=18. Numbers above bars - actual values of time (sec) and # instances. Total instances in benchmark: 32, discarded instances due to exact heuristic: 1.

m-AOBB-tree requires 6.46 seconds to find m=10 solutions, while the runtimes of other algorithms range from 0.03 to 0.09 seconds (except for BE+m-BF which runs our of memory). However, the slow performance of m-AOBB-tree on easier problems that are feasible for all algorithms is compensated by the fact that for many instances it is the only scheme to report any solution, solving

| algorithm | Grids: # inst=32, $n$=144-2500 $k$=2-2, $w^*$=15-74, $h_T$=48-312, i-bound=18 | | | | |
|---|---|---|---|---|---|
| | m=1 | m=2 | m=5 | m=10 | m=100 |
| | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN |
| Not solved | 12 | 12 | 13 | 13 | 14 |
| m-AOBF tree | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| m-AOBF graph | 0 / 0 | 0 / 1 | 0 / 2 | 0 / 1 | 0 / 3 |
| m-A* tree | 8 / 2 | 8 / 2 | 8 / 6 | 9 / 6 | 7 / 6 |
| m-BB tree | 1 / 0 | 1 / 0 | 0 / 0 | 1 / 0 | 2 / 2 |
| m-AOBB tree | 7 / 10 | 7 / 10 | 5 / 5 | 5 / 5 | 2 / 2 |
| BE+m-BF | 5 / 7 | 5 / 6 | 6 / 5 | 6 / 6 | 6 / 4 |

Table 7: Number of instances, for which each algorithm has the best runtime (#BT) and best number of expanded nodes (#BN), Grids. Out of 32 instances 1 have exact heuristics. The table accounts for remaining 31, i-bound= 18 .

most instances by considerable amount for $m \in [1, 10]$ (Figure 13). Table 11 shows that m-AOBB-tree is the best both in terms of time and space for the overwhelming majority of problems for all values of $m$ except for $m = 100$.

#### 7.2.6 SEGMENTATION

Table 12 shows the results for select instances from the Segmentation benchmark for two i-bounds, namely i=4 and i=12, while Figure 14 and Table 13 present the summary of the results for i=12. Unlike WCSP, for this benchmark we chose to display relatively low i-bounds not because calculating heuristic with larger i's is infeasible, but because the problems have low induced width and we wished to avoid displaying results obtained with exact heuristics. The main peculiarity of this benchmark is the striking success of BE+m-BF. Overall it solves as many instances as the usually superior m-A*-tree and m-BB-tree, as is seen in Figure 14. Moreover, its runtime is superior to the other schemes, as is true for all instances in Table 12 and is also illustrated by the results in the Table 13. When the heuristic is very weak, m-AOBB-tree is fairly successful, for example, finding solutions for all values of $m$ for 12_4_s.binary, i=4, which is infeasible for any other scheme except for BE+m-BF. However, as usual, m-AOBB-tree is the overall slowest of the schemes.

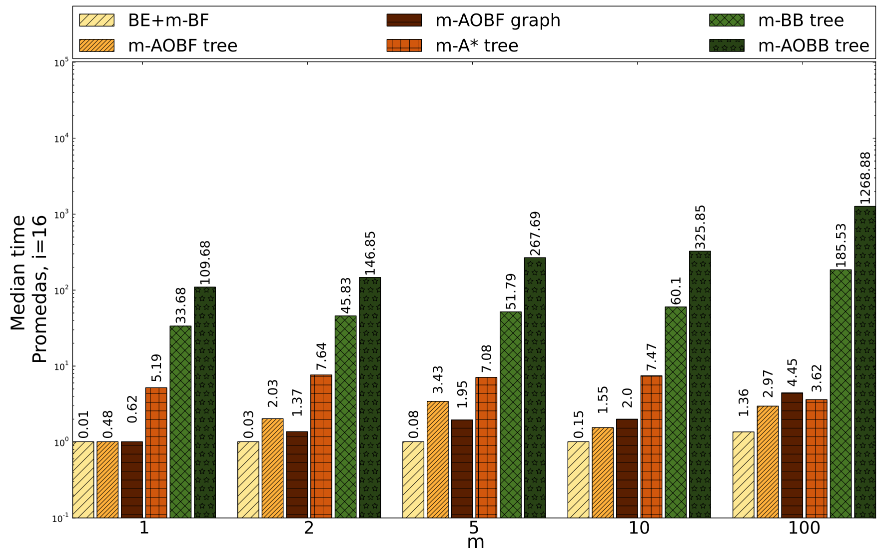
### 7.3 Best-first vs Depth-First Branch and Bound for the *M* Best Solutions

Let us again consider the data presented in Tables 2-13 and Figures 9-14 in order to summarize our observations and contrast the performance of best-first and depth-first branch and bound schemes. Among the best-first search schemes m-A*-tree is the most successful. It is often very effective, when armed with a good heuristic, and requires less space than the other best-first schemes. As we already noted, BE+m-BF shows good results on the Segmentation benchmark, where it is the best algorithm in terms of the median runtime, while solving at least the same number of problems as the other schemes. However, on the other benchmarks the calculation of the exact heuristic is often infeasible.

| instance $(n,k,w^*,h)$ | i-bound | algorithm | number of solutions | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | m=1 | | m=10 | | m=100 | |
| | | | time | nodes | time | nodes | time | nodes |
| or_chain_107.fg | 16 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **7.89** | **919865** | **18.89** | **2108122** | **44.61** | **4641627** |
| | | m-BB tree | 14.58 | 3139711 | 35.15 | 7051974 | 102.6 | 18494630 |
| | | m-AOBB tree | 67.95 | 1398364 | 229.49 | 5134280 | 627.94 | 13594667 |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (620, 2, 30, 64) | 22 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **9.2** | **1093564** | **20.83** | **2465364** | **56.59** | **6356871** |
| | | m-BB tree | 17.0 | 3861414 | 42.36 | 9205755 | 100.45 | 19217427 |
| | | m-AOBB tree | 122.01 | 3214924 | 418.46 | 11123810 | 855.84 | 21388619 |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| or_chain_141.fg | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | 9553.91 | 1276222668 | OOT | | OOT | |
| | | m-AOBB tree | **272.0** | **9878480** | **721.67** | **25481595** | **2091.26** | **64400241** |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (676, 2, 30, 70) | 16 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **14.16** | **1261489** | **38.48** | **3379926** | OOM | |
| | | m-BB tree | 279.61 | 56821714 | 460.15 | 87947802 | **885.72** | 160581726 |
| | | m-AOBB tree | 140.9 | 6490042 | 315.2 | 14103095 | 909.48 | **33842266** |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| or_chain_212.fg | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | **1772.8** | **49808550** | **4206.07** | **111853485** | Timeout | |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (773, 2, 33, 79) | 22 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **9.91** | **1118792** | **33.87** | **3669711** | **78.37** | **8186757** |
| | | m-BB tree | 78.08 | 15922806 | 141.66 | 27615033 | 342.51 | 58246101 |
| | | m-AOBB tree | 584.83 | 11336657 | 1239.88 | 24717964 | 5032.11 | 86444575 |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| or_chain_50.fg | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | **1404.27** | **33495406** | **3748.85** | **93992107** | **10070.0** | **245628104** |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| 661, 2, 36, 76) | 22 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **53.87** | **5673948** | OOM | | OOM | |
| | | m-BB tree | 91.15 | 18515503 | **176.14** | **34915510** | **447.46** | **85945673** |
| | | m-AOBB tree | Timeout | | Timeout | | Timeout | |
| | | BE+m-BF | OOM | | OOM | | OOM | |

Table 8: Promedas: CPU time (in seconds) and number of nodes expanded. An 'Timeout' stands for exceeding the time limit of 3 hours. 'OOM' indicates out of 4GB memory. In bold we highlight the best time and number of nodes for each $m$. Parameters: $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h$ - pseudo tree height.

The two m-AOBF-based schemes are overall inferior due to prohibitively large memory, solving fewer instances than the other algorithms. We believe that a non-trivial extension of AOBF from a single solution to the m-best task is not straightforward, because it is hard to represent multiple partial solution trees in an efficient manner. In order to have an efficient m-AOBF implementation, one needs to quickly identify which partial solution subtree to select and extend next, when searching for the $(k+1)^{th}$ solution after finding the $k^{th}$ best solution. While AOBF (for 1 solution) uses an arc-marking mechanism to efficiently represent the current best partial solution subtree during search, this is not easy to extend for the case when searching for the $m$ best solutions. Therefore, as was shown in Section 5.1, our m-AOBF implements a naive mechanism where each of the par-



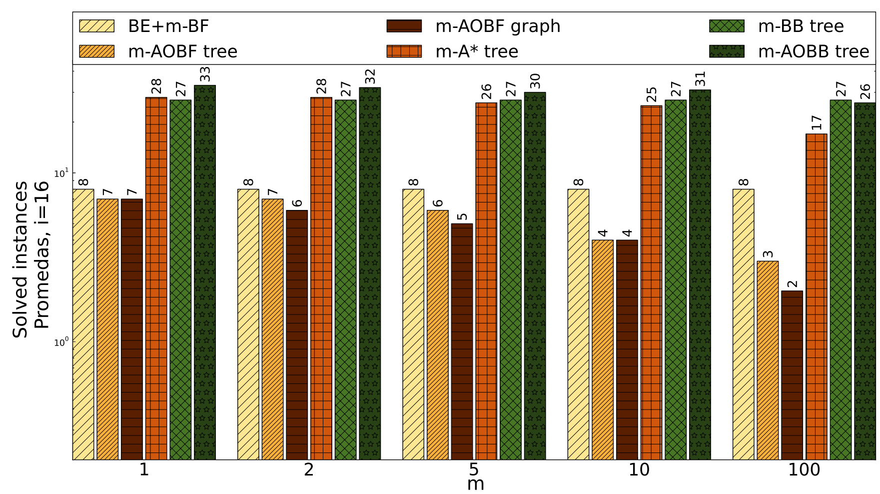




Figure 12: Median time and number of solved instances (out of 86) summary for select values of $m$ for Promedas, i-bound=16. Numbers above bars - actual values of time (sec) and # instances. Total instances in benchmark: 75, discarded instances due to exact heuristic: 11.

tial solution trees is represented explicitly in memory. This simple representation, however, incurs a considerable computational overhead when searching for the $m$ best solutions, which is indeed revealed by our experiments. A more efficient implementation of m-AOBF is left for future work.

| algorithm | Promedas: # inst=86, $n$=197-2113<br>$k$=2-2, $w^*$=5-120, $h_T$=34-187, i-bound=16 | | | | |
|---|---|---|---|---|---|
| | m=1 | m=2 | m=5 | m=10 | m=100 |
| | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN |
| Not solved | 42 | 42 | 45 | 44 | 46 |
| m-AOBF tree | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| m-AOBF graph | 0 / 1 | 0 / 2 | 0 / 2 | 0 / 3 | 0 / 2 |
| m-A* tree | 22 / 17 | 21 / 17 | 18 / 17 | 18 / 15 | 9 / 9 |
| m-BB tree | 1 / 0 | 1 / 0 | 1 / 0 | 2 / 0 | 10 / 5 |
| m-AOBB tree | 4 / 8 | 4 / 8 | 3 / 5 | 4 / 8 | 2 / 7 |
| BE+m-BF | 8 / 7 | 8 / 6 | 8 / 6 | 8 / 5 | 8 / 6 |

Table 9: Number of instances, for which each algorithm has the best runtime (#BT) and best number of expanded nodes (#BN), Promedas. Out of 86 instances 11 have exact heuristics. The table accounts for remaining 75, i-bound= 16 .

Unsurprisingly, the branch and bound algorithms are more robust in terms of memory and also dominate m-A*-tree and other best-first schemes on many benchmarks in terms of the number of instances solved. However, they tend to have considerably larger median time and expand more nodes. In particular, m-AOBB-tree does not scale well with the number of solutions and for large values of $m$ the runtime increases drastically. Unlike m-AOBF, whose inferior performance can be attributed to specifics of implementation, the depth-first m-AOBB suffers from issues inherent to solving the m-best problem in a depth-first manner. As Algorithm 10 describes, m-AOBB needs to merge the $m$ best partial solution at each internal node, which hurts the performance significantly, but cannot be avoided, unless the algorithmic approach itself is fundamentally changed. We did not see a way to overcome this limitation.

Overall, whenever the calculation of the exact heuristic is feasible, BE+m-BF should be the algorithm of choice. Otherwise, m-A*-tree is superior for the relatively easy problems, while m-AOBB-tree is the best scheme for hard memory intensive instances. This superiority of a best-first approach, whenever memory is available, is expected, based, on the one hand, on intuition derived from our knowledge of the task of finding a single solution, and on the other hand, on the theoretical results in Section 3.2.

### 7.4 Scalability of the Algorithms with the Number of Required Solutions

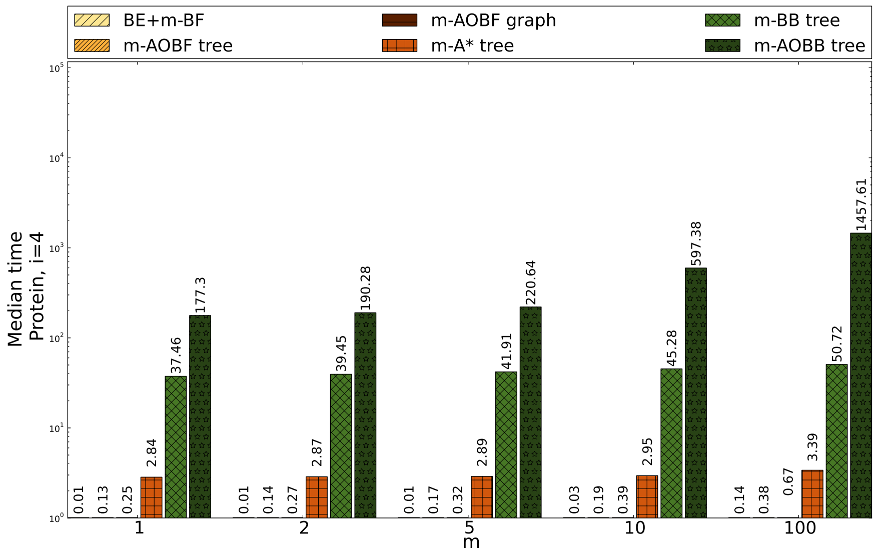
Figures 15-17 present the plots showing the runtime in seconds and the number of expanded nodes as a function of number of solutions $m$ (on a log scale) for two instances from each benchmark. Figure 15 displays results for WCSP and Pedigree benchmarks, Figure 16 - for Grids and Promedas, Figure 17 - for Proteins and Segmentation. Lower values (on the y-axis) are preferable. Each row contains two instances from each benchmarks for a specific value of the i-bound, the runtime plots being shown above the ones containing the expanded nodes. The examples are chosen to best illustrate the prevailing tendencies.

Note that the theoretical analysis suggests that the runtime of BE+m-BF, the best among the algorithms, should scale with $m$ since its worst case complexity is $O(nk^{w^*} + mn)$. The theoretical complexity of the best-first search schemes m-AOBF-tree and m-A*-tree is linear in the number of

| instance ($n$,$k$,$w^*$,$h$) | i-bound | algorithm | number of solutions m=1 time | m=1 nodes | m=10 time | m=10 nodes | m=100 time | m=100 nodes |
|---|---|---|---|---|---|---|---|---|
| pdb1b2v | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **0.12** | **2508** | **0.17** | **3186** | **0.34** | **6249** |
| | | m-BB tree | 10.23 | 948337 | 11.09 | 1034645 | 14.4 | 1404370 |
| | | m-AOBB tree | 6.51 | 100584 | 35.14 | 827365 | 4462.0 | 230849005 |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (133, 36, 13, 33) | 8 | m-AOBF tree | 0.02 | **95** | 0.06 | 294 | 0.42 | 1956 |
| | | m-AOBF graph | 0.03 | **95** | 0.09 | **108** | 0.64 | **135** |
| | | m-A* tree | **0.0** | 139 | **0.03** | 597 | **0.15** | 3051 |
| | | m-BB tree | 0.01 | 2401 | 0.07 | 8861 | 0.5 | 67330 |
| | | m-AOBB tree | 0.29 | 6563 | 6.46 | 256588 | Timeout | |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| pdb1cxy | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **0.38** | **3708** | **0.48** | **4434** | **0.94** | **10854** |
| | | m-BB tree | 0.4 | 51020 | 0.6 | 73849 | 1.45 | 191203 |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| (70, 81, 9, 19) | 8 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | **0.01** | **121** | **0.04** | **480** | **0.1** | **2870** |
| | | m-BB tree | 0.03 | 5791 | 0.07 | 11429 | 0.27 | 53702 |
| | | m-AOBB tree | 0.66 | 7029 | 2.04 | 34567 | 44.28 | 1335157 |
| | | BE+m-BF | OOM | | OOM | | OOM | |
| pdb1ctj | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 10.43 | 35400 | 13.23 | 43538 | 19.74 | 65340 |
| | | m-BB tree | 844.08 | 76609260 | 1039.96 | 94422614 | 1325.84 | 120786833 |
| | | m-AOBB tree | **5.64** | 74833 | 18.29 | 306054 | 157.43 | 5307198 |
| | | BE+m-BF | **0.01** | **62** | **0.02** | **265** | **0.07** | **1050** |
| (62, 81, 8, 21) | 8 | m-AOBF tree | **0.0** | 49 | 0.07 | 302 | 0.42 | 1825 |
| | | m-AOBF graph | 0.02 | **45** | 0.11 | **74** | 0.75 | **95** |
| | | m-A* tree | 0.01 | 62 | 0.03 | 265 | 0.08 | 1057 |
| | | m-BB tree | 0.01 | 1118 | 0.03 | 5385 | 0.15 | 31066 |
| | | m-AOBB tree | 0.22 | 3098 | 2.32 | 54324 | 71.86 | 3273324 |
| | | BE+m-BF | 0.01 | 62 | **0.02** | 265 | **0.07** | 1050 |
| pdb1dlw | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 46.17 | 579108 | 46.26 | 579405 | 46.49 | 582375 |
| | | m-BB tree | 47.27 | 6380302 | 47.33 | 6391107 | 50.72 | 6762911 |
| | | m-AOBB tree | 187.38 | 1451906 | 544.55 | 12759004 | OOT | |
| | | BE+m-BF | **0.01** | **294** | **0.05** | **635** | **0.39** | **4265** |
| (84, 81, 8, 29) | 8 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 0.14 | 6900 | 0.18 | 7240 | 0.52 | 10855 |
| | | m-BB tree | 1.06 | 162913 | 1.09 | 167037 | 1.86 | 280189 |
| | | m-AOBB tree | 18.53 | 154850 | 157.01 | 8632114 | OOT | |
| | | BE+m-BF | **0.01** | **294** | **0.05** | **635** | **0.39** | **4265** |

Table 10: Protein: CPU time (in seconds) and number of nodes expanded. An 'Timeout' stands for exceeding the time limit of 3 hours. 'OOM' indicates out of 4GB memory. In bold we highlight the best time and number of nodes for each $m$. Parameters: $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h$ - pseudo tree height.

solutions, while for m-BB-tree the overhead due to the $m$-best task is a factor of $(m \cdot \log m)$ and for m-AOBB-tree it is $(m \log m \cdot deg)$, where $deg$ is the degree of the pseudo tree. We observed that compared to other schemes the runtime of BE+m-BF indeed rises quite slowly as the number of solutions increases, even as $m$ reaches 100. The runtime m-A*-tree also scales well with $m$. The behaviour of m-BB-tree depends a lot on the benchmarks. On Pedigrees and Protein its runtime changes very little on most instances as the number of solutions grows, but on the other benchmarks, the runtime for m=100 tends to be significantly larger then for m=1. m-AOBF-tree and m-AOBF-graph often do not provide any solutions even for m=1 or, alternatively, run out of memory as $m$ slightly increases ($m \in [2, 10]$). These algorithms are clearly not successful in practice. As



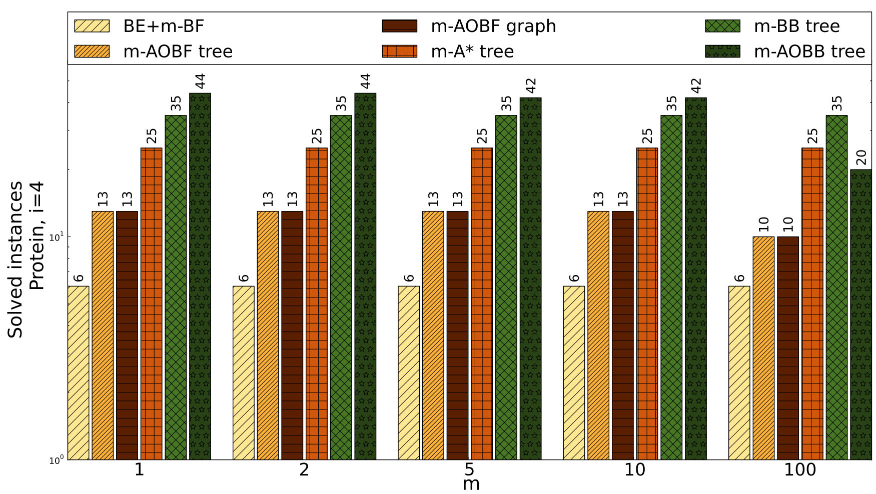




Figure 13: Median time and number of solved instances (out of 72) for select values of $m$ for Protein, i-bound=4. Numbers above bars - actual values of time (sec) and # instances. Total instances in benchmark: 72, discarded instances due to exact heuristic: 0.

we discussed before, both the runtime and number of expanded nodes of m-AOBB-tree increase drastically as $m$ gets larger.

| algorithm | Protein: # inst=72, $n$=15-242<br>$k$=18-81, $w^*$=5-16, $h_T$=7-44, i-bound=4 | | | | |
|---|---|---|---|---|---|
| | m=1 | m=2 | m=5 | m=10 | m=100 |
| | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN |
| Not solved | 26 | 26 | 27 | 27 | 34 |
| m-AOBF tree | 8 / 3 | 7 / 1 | 7 / 1 | 7 / 0 | 5 / 0 |
| m-AOBF graph | 1 / 10 | 2 / 12 | 2 / 12 | 1 / 13 | 1 / 10 |
| m-A* tree | 11 / 9 | 9 / 9 | 12 / 9 | 14 / 10 | 17 / 13 |
| m-BB tree | 8 / 1 | 9 / 2 | 8 / 3 | 6 / 3 | 11 / 10 |
| m-AOBB tree | 21 / 22 | 21 / 21 | 18 / 19 | 17 / 17 | 3 / 3 |
| BE+m-BF | 6 / 4 | 6 / 2 | 5 / 2 | 5 / 2 | 6 / 2 |

Table 11: Number of instances, for which each algorithm has the best runtime (#BT) and best number of expanded nodes (#BN), Protein. Out of 72 instances 0 have exact heuristics. The table accounts for remaining 72, i-bound= 4 .

### 7.5 Comparison with Competing Algorithms

We compare our methods with a number of previously developed schemes described in more details in Section 6: STRIPES, PESTEELARS and Nilsson's algorithm. The implementations of these schemes were provided by Dhruv Batra. The first two approaches are based on ideas of LP relaxations and are approximate, but are known to often find exact solutions, though they provide no guarantees of optimality. Nilsson's algorithm is an exact message-passing scheme operating on a junction tree. For the first set of experiments (on a tree benchmark) we also show results for STILARS algorithm, an older version of the PESTEELARS algorithm. However, this scheme is consistently inferior to the other two LP-based schemes and is not considered for the other two benchmarks. In the following, we collectively refer to these 4 algorithms as *"competing schemes"*.

#### 7.5.1 RANDOMLY GENERATED BENCHMARKS

The available to us code of the LP-based and Nilsson's approaches was developed to run on restricted inputs only, and so it could not be applied to the benchmarks used in the bulk of our evaluation described above. We concluded that re-implementing the competing codes to work on general input would be too time consuming and would not provide any additional insights. Thus we chose to compare our algorithms with the competitors using benchmarks that can be acceptable to the competing schemes.

Specifically, the comparison was performed on the following three benchmarks: random trees, random binary grids and random graphs with submodular potentials, that we call "submodular graphs" in the remainder of the section. Table 14 shows the parameters of the benchmarks. The instances were generated in the following manner. First, a vector of 12 logarithmically spaced integers between 10 and $10^{3.5}$ was generated, serving as the number of variables for the instances. For binary grids benchmarks each value was used to generate two problems with the same number of variables. The edges between the variables were generated uniformly randomly, while making sure that the end graph is a tree, grid or a loopy graph, depending on the benchmark. For each edge we define a binary potential and for each vertex a unary potential in an exponential form: $f = e^{\theta}$,

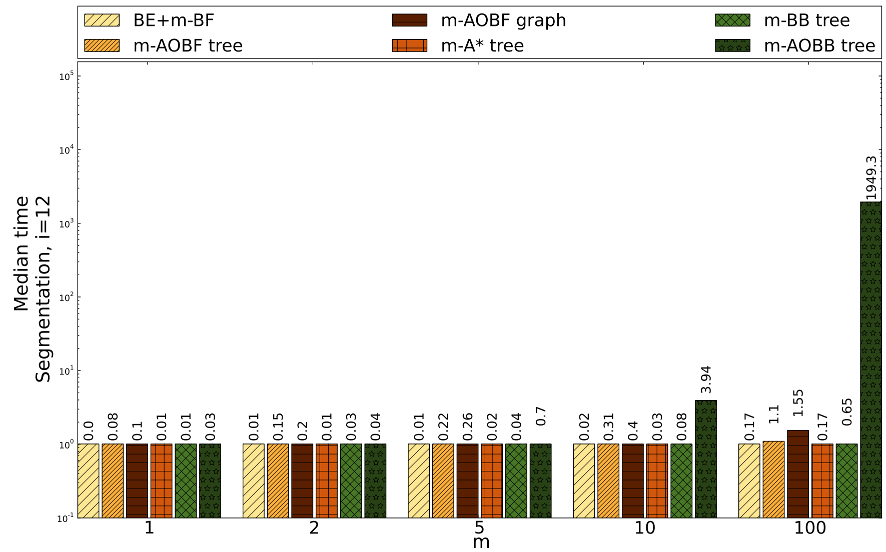
| instance $(n,k,w^*,h)$ | i-bound | algorithm | number of solutions m=1 time | m=1 nodes | m=10 time | m=10 nodes | m=100 time | m=100 nodes |
|---|---|---|---|---|---|---|---|---|
| 12_4_s.binary | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | 164.91 | 5653312 | 505.82 | 18888321 | 4371.05 | 189179726 |
| | | BE+m-BF | **0.0** | **225** | **0.02** | **1619** | **0.21** | **11194** |
| (225, 2, 16, 48) | 12 | m-AOBF tree | 7.31 | 103327 | 10.36 | 143333 | OOM | |
| | | m-AOBF graph | 10.47 | **1843** | OOM | | OOM | |
| | | m-A* tree | 0.03 | 3754 | 0.06 | 5692 | 0.3 | 18616 |
| | | m-BB tree | 0.04 | 8251 | 0.21 | 24349 | 1.32 | 131571 |
| | | m-AOBB tree | 0.08 | 4158 | 1.62 | 118074 | 489.57 | 40961080 |
| | | BE+m-BF | **0.0** | **225** | **0.02** | **1619** | **0.21** | **11194** |
| 16_16_s.binary | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | 71.71 | 2733703 | 360.14 | 14906212 | OOT | |
| | | BE+m-BF | **0.01** | **227** | **0.02** | **1365** | **0.19** | **11157** |
| (227, 2, 16, 57) | 12 | m-AOBF tree | 0.23 | 3338 | 3.75 | 46121 | OOM | |
| | | m-AOBF graph | 0.33 | 799 | 5.72 | 1827 | OOM | |
| | | m-A* tree | **0.01** | 585 | 0.09 | 9103 | 0.38 | 30542 |
| | | m-BB tree | 0.05 | 10687 | 0.19 | 30119 | 1.2 | 141591 |
| | | m-AOBB tree | 0.21 | 11076 | 14.28 | 1054628 | OOT | |
| | | BE+m-BF | **0.01** | **227** | **0.02** | **1365** | **0.19** | **11157** |
| 7_9_s.binary | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | 127.17 | 3337949 | 505.08 | 17976200 | OOT | |
| | | BE+m-BF | **0.01** | **234** | **0.03** | **1337** | **0.21** | **10212** |
| (234, 2, 16, 53) | 12 | m-AOBF tree | 8.85 | 122663 | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 0.02 | 1978 | 0.06 | 4170 | 0.28 | 15807 |
| | | m-BB tree | 0.03 | 4415 | 0.11 | 13357 | 0.95 | 89675 |
| | | m-AOBB tree | 0.05 | 2750 | 10.54 | 806490 | OOT | |
| | | BE+m-BF | **0.01** | **234** | **0.03** | **1337** | **0.21** | **10212** |
| 11_4_s.binary | 4 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | OOM | | OOM | | OOM | |
| | | m-BB tree | Timeout | | Timeout | | Timeout | |
| | | m-AOBB tree | 110.19 | 4227437 | 555.6 | 23302165 | OOT | |
| | | BE+m-BF | **0.01** | **231** | **0.03** | **1615** | **0.28** | **14241** |
| (231, 2, 16, 57) | 12 | m-AOBF tree | OOM | | OOM | | OOM | |
| | | m-AOBF graph | OOM | | OOM | | OOM | |
| | | m-A* tree | 1.02 | 102671 | 1.17 | 115407 | 1.86 | 167983 |
| | | m-BB tree | 2.07 | 428791 | 2.9 | 527967 | 7.1 | 1010155 |
| | | m-AOBB tree | 0.75 | 39170 | 11.99 | 809403 | 8497.93 | 617227854 |
| | | BE+m-BF | **0.01** | **231** | **0.03** | **1615** | **0.28** | **14241** |

Table 12: Segmentation: CPU time (in seconds) and number of nodes expanded. An 'Timeout' stands for exceeding the time limit of 3 hours. 'OOM' indicates out of 4GB memory. In bold we highlight the best time and number of nodes for each $m$. Parameters: $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h$-pseudo tree height.

where $\theta$ is a real number sampled from a uniform distribution. For the third benchmark the potentials are further modified to be submodular. On the random trees the $m$-best optimization LP problem is guaranteed to be tight, on the graphs with submodular potentials the LP optimization problem is tight, but its $m$-best extension is not, and on the arbitrary loopy graphs, including grids, the algorithms provide no guarantees.

#### 7.5.2 COMPETING ALGORITHMS' PERFORMANCE

Table 15 shows the runtimes for select instances from the random tree benchmark for our 5 $m$-best search schemes and the competing LP schemes STILARS, PESTEELARS and STRIPES. We



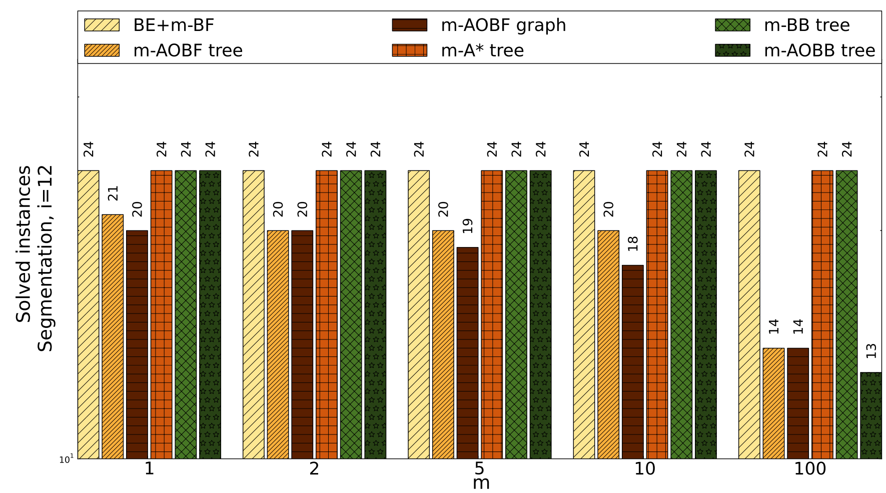




Figure 14: Median time and number of solved instances (out of 47) for select values of $m$ for Segmentation, i-bound=12. Numbers above bars - actual values of time (sec) and # instances. Total instances in benchmark: 47, discarded due to exact heuristic: 0.

observed on this benchmarks that STILARS was always inferior to the other two schemes and therefore it was excluded from the remainder of evaluation. Instead, in Tables 16 and 17, where we show results for the random binary grids and submodular graphs benchmarks, we added for comparison Nilsson's max-flow algorithm. In the Table 15-17 the time limit was set to 1 hour, memory limit to 3 GB. The schemes' behavior is quite consistent across the instances.

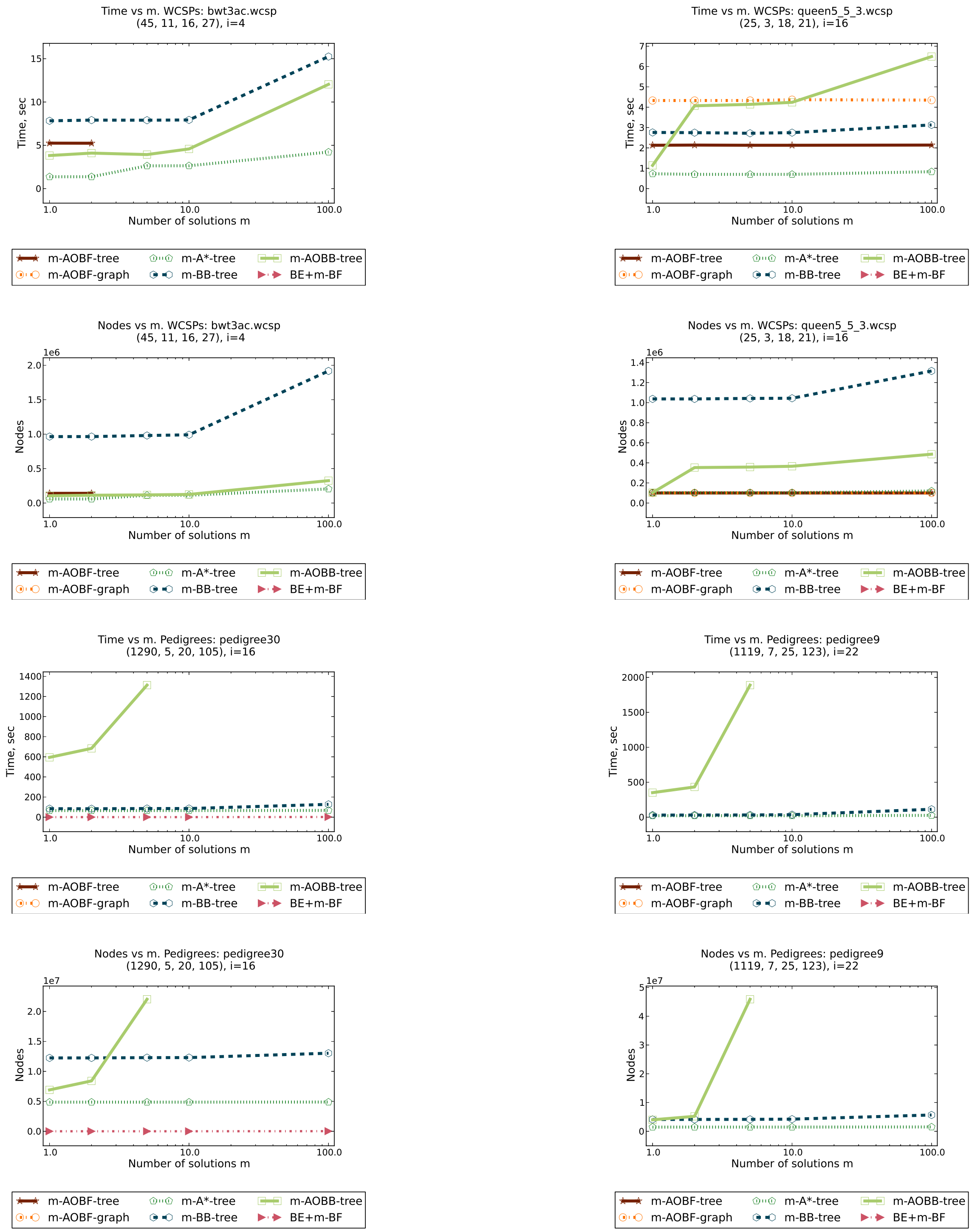



Figure 15: CPU time in seconds and number of expanded nodes as a function of number of solutions. WCSP and Pedigrees, 4 GB, 3 hours.

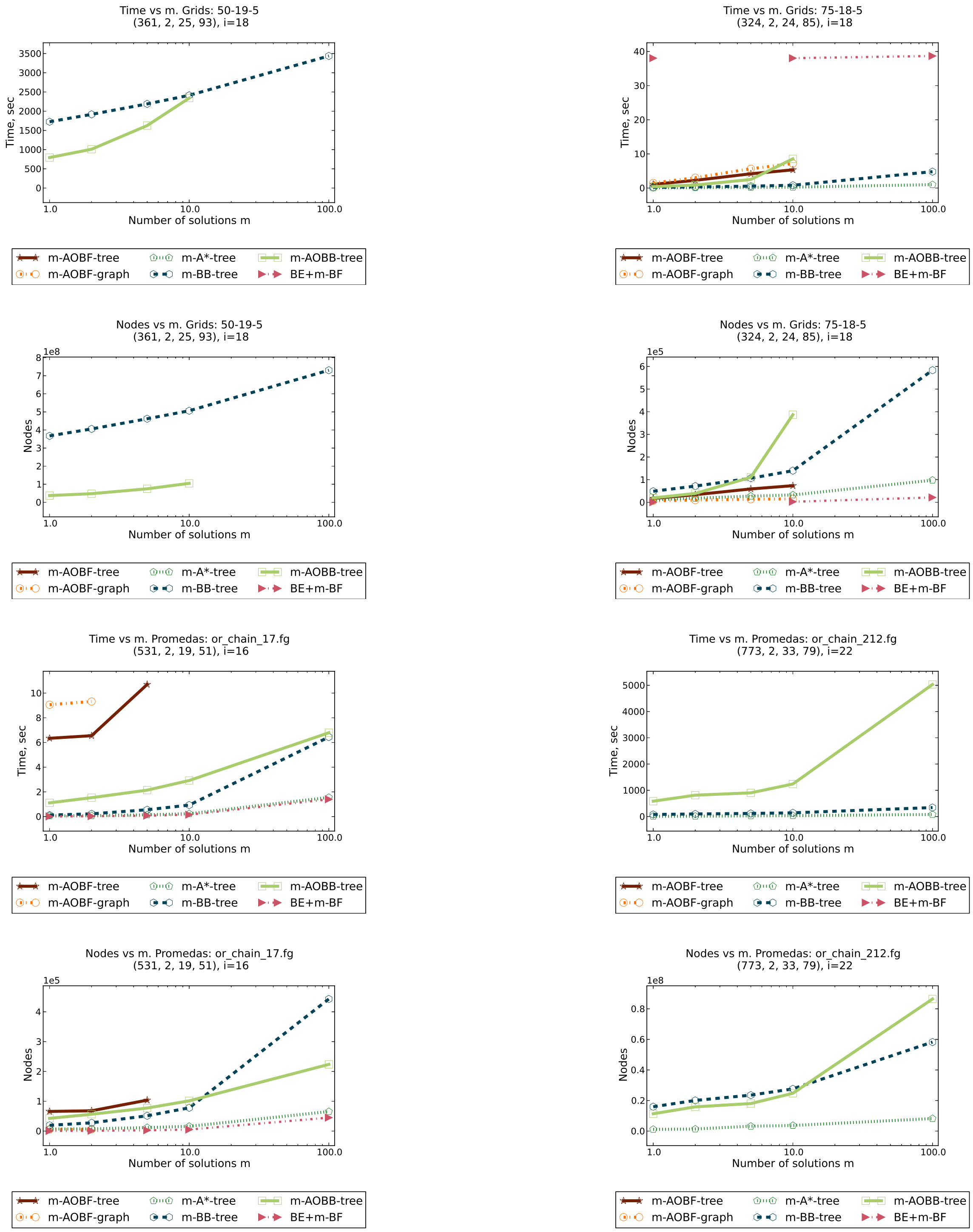



Figure 16: CPU time in seconds and a number of expanded nodes as a function of number of solutions. Grids and Promedas, 4 GB, 3 hours.

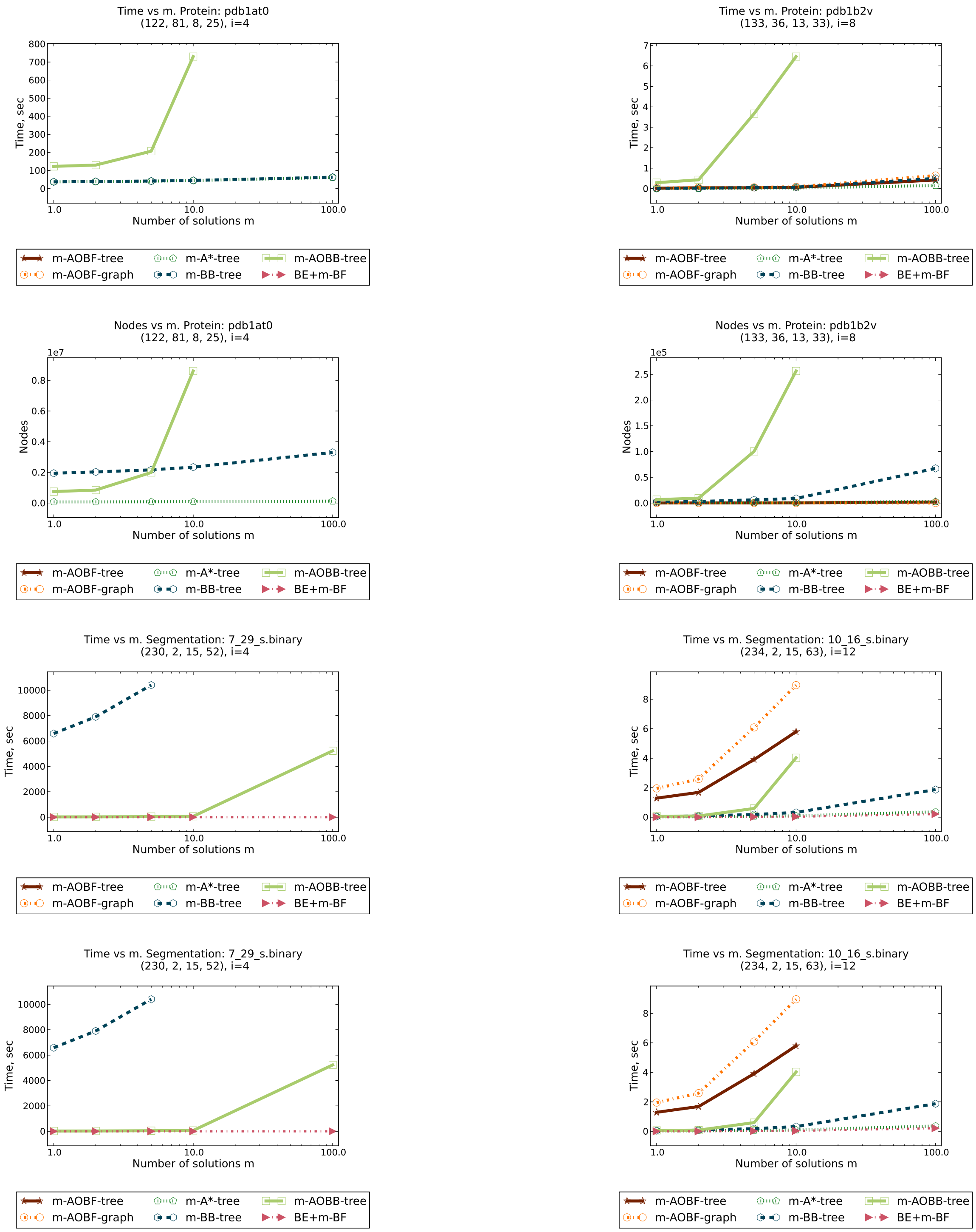



Figure 17: CPU time in seconds and number of expanded nodes as a function of number of solutions. Protein and Segmentation, 4 GB, 3 hours.

| algorithm | Segmentation: # inst=47, $n$=222-234<br>$k$=2-21, $w^*$=15-18, $h_T$=47-67, i-bound=12 | | | | |
|---|---|---|---|---|---|
| | m=1 | m=2 | m=5 | m=10 | m=100 |
| | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN | #BT / #BN |
| Not solved | 23 | 23 | 23 | 23 | 23 |
| m-AOBF tree | 0 / 5 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| m-AOBF graph | 0 / 5 | 0 / 7 | 0 / 11 | 0 / 15 | 0 / 14 |
| m-A* tree | 15 / 0 | 11 / 0 | 12 / 0 | 11 / 0 | 3 / 0 |
| m-BB tree | 7 / 0 | 5 / 0 | 3 / 0 | 2 / 0 | 0 / 0 |
| m-AOBB tree | 0 / 0 | 3 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| BE+m-BF | 21 / 19 | 20 / 17 | 22 / 13 | 24 / 9 | 24 / 10 |

Table 13: Number of instances, for which each algorithm has the best runtime (#BT) and best number of expanded nodes (#BN), Segmentation. Out of 47 instances 0 have exact heuristics. The table accounts for remaining 47, i-bound= 12 .

| Benchmark | # inst | $n$ | $k$ | $w^*$ | $h_T$ |
|---|---|---|---|---|---|
| Random trees | 12 | 10-5994 | 2-4 | 1 | 5-132 |
| Random Binary Grids | 24 | 16-3192 | 2 | 6-79 | 9-221 |
| Random submodular graphs | 12 | 16-3192 | 2 | 4-74 | 9-208 |

Table 14: Benchmark parameters: # inst - number of instances, $n$ - number of variables, $k$ - domain size, $w^*$ - induced width, $h_T$ - pseudo tree height.

STILARS and Nilsson's schemes are always dominated by the other two competing schemes in terms of runtime. STRIPES and PESTEELARS are sometimes faster than all our schemes for m=1, e.g. tree_nnodes880_ps1_k4, however, on all three benchmark they scale rather poorly with $m$. For $m \geq 5$ they are almost always inferior to our algorithms, provided that the latter report any results, with occasional exception of m-AOBB-tree, which also tends to be slow for large $m$. The only problems on which PESTEELARS and STRIPES are superior to our search schemes are the largest networks having over a 1000 variables, such as grid_nnodes3192_ps2_k2, which are infeasible for our algorithms. Overall, our five $m$-best algorithms proved superiority over the considered competing schemes on the majority of instances, often having better runtime, especially when $m > 2$, while guaranteeing solution optimality.

## 8. Conclusion

Most of the work on finding $m$ best solutions over graphical models was focused on either iterative schemes based on Lawler's idea or on dynamic programming (e.g., variable-elimination or tree-clustering). We showed for the first time that for combinatorial optimization defined over graphical models the traditional heuristic search paradigms are not only directly applicable, but often superior.

Specifically, we extended best-first and depth-first branch and bound search algorithms to solve the $m$-best optimization task, presenting m-A* and m-BB, respectively. We showed that the properties of A* extend to the m-A* algorithm and, in particular, proved that m-A* is superior to any

| instance | algorithm | i-bound=4. k=2 | | | | |
|---|---|---|---|---|---|---|
| | | m=1 | m=2 | m=5 | m=10 | m=100 |
| | | time | time | time | time | time |
| tree_nnodes245_ps1_k2 | m-AOBF tree | 0.02 | 0.02 | 0.05 | 0.09 | 0.61 |
| | m-AOBF graph | 0.02 | 0.03 | 0.08 | 0.13 | 0.95 |
| | m-A* tree | 0.0 | 0.0 | 0.01 | 0.02 | 0.12 |
| | m-BB tree | 0.0 | 0.0 | 0.02 | 0.03 | 0.37 |
| (245, 2, 2, 32) | m-AOBB tree | 0.02 | 0.02 | 0.06 | 14.14 | 3045.25 |
| | STILARS | 0.0 | 0.04 | 7.93 | 33.3 | 1757.41 |
| | STRIPES | 0.09 | 0.17 | 0.4 | 0.88 | 13.88 |
| | PESTEELARS | 0.0 | 0.13 | 0.51 | 1.32 | 47.32 |
| tree_nnodes880_ps1_k4 | m-AOBF tree | 0.3 | 0.46 | 1.06 | 1.9 | OOM |
| | m-AOBF graph | 0.48 | 0.76 | 1.8 | 3.28 | OOM |
| | m-A* tree | 0.08 | 0.17 | 0.24 | 0.48 | 3.67 |
| | m-BB tree | 0.1 | 0.3 | 0.54 | 1.23 | 14.38 |
| (880, 4, 2, 52) | m-AOBB tree | 1.17 | 1.37 | 52.36 | 927.12 | Timeout |
| | STILARS | 0.0 | 0.11 | 28.19 | 81.21 | 2440.22 |
| | STRIPES | 5.67 | 11.26 | 28.09 | 56.41 | 607.01 |
| | PESTEELARS | 0.0 | 0.87 | 6.13 | 9.26 | 79.0 |
| tree_nnodes5994_ps1_k4 | m-AOBF tree | OOM | OOM | OOM | OOM | OOM |
| | m-AOBF graph | OOM | OOM | OOM | OOM | OOM |
| | m-A* tree | 5.44 | 10.68 | 18.31 | 37.21 | 206.26 |
| | m-BB tree | 5.77 | 29.04 | 36.49 | 97.73 | 1112.2 |
| (5994, 4, 2, 189) | m-AOBB tree | 851.48 | 922.19 | Timeout | Timeout | Timeout |
| | STILARS | 0.05 | 2.72 | 64.48 | 250.36 | 7325.4 |
| | STRIPES | 248.53 | 506.25 | 1279.87 | 2576.87 | Timeout |
| | PESTEELARS | 0.05 | 18.28 | 91.17 | 169.39 | Timeout |

Table 15: Random trees, i-bound=4. Timeout - out of time, OOM - out of memory. 3 GB, 1 hour.

other search scheme for the $m$-best task. We also analyzed the overhead of both algorithms caused by the need to find multiple solutions. We introduced BE+m-BF, a hybrid of variable elimination and best-first search scheme and showed that it has the best worst-case time complexity among all $m$-best algorithms over graphical models known to us.

We evaluated our schemes empirically. We observed that the AND/OR decomposition of the search space, which significantly boosts the performance of traditional heuristic search schemes, was not cost-effective for $m$-best search algorithms, at least with our current implementation. As expected, the best-first schemes dominate the branch and bound algorithms whenever sufficient space is available, but fail on the memory-intensive problems. We compared our schemes with 4 previously developed algorithms: three approximate schemes based on an LP-relaxation of the problem and an algorithm performing message passing on a junction tree. We showed that our schemes often dominate the competing schemes, known to be efficient, in terms of runtime, especially when the required number of solutions is large. Moreover, our scheme guarantee solution optimality.

## Acknowledgement

This work was sponsored in part by NSF grants IIS-1065618 and IIS-1254071, and by the United States Air Force under Contract No. FA8750-14-C-0011 under the DARPA PPAML program.

| instance | algorithm | i-bound=20 | | | |
|---|---|---|---|---|---|
| | | m=2 | m=5 | m=10 | m=25 |
| | | time | time | time | time |
| Random binary grid | | | | | |
| grid_nnodes380_ps1_k2 (380, 2, 25, 379) | m-AOBF tree | OOM | OOM | OOM | OOM |
| | m-AOBF graph | OOM | OOM | OOM | OOM |
| | m-A* tree | 0.49 | 0.53 | 0.58 | 0.67 |
| | m-BB tree | 0.56 | 0.64 | 0.71 | 0.91 |
| | m-AOBB tree | 55.93 | 106.34 | 202.65 | 2027.66 |
| | Nilsson | 112.5 | 772.49 | 1860.46 | 5026.68 |
| | STRIPES | 5.06 | 46.57 | 172.95 | 361.04 |
| | PESTEELARS | 4.63 | 13.4 | 28.95 | 75.28 |
| grid_nnodes380_ps2_k2 (380, 2, 25, 61) | m-AOBF tree | OOM | OOM | OOM | OOM |
| | m-AOBF graph | OOM | OOM | OOM | OOM |
| | m-A* tree | 0.2 | 0.23 | 0.26 | 0.36 |
| | m-BB tree | 0.32 | 0.36 | 0.58 | 0.95 |
| | m-AOBB tree | 7.62 | 12.82 | 67.59 | 1964.18 |
| | Nilsson | 110.4 | 757.14 | 1820.45 | 4985.0 |
| | STRIPES | 2.23 | 19.41 | 38.54 | Timeout |
| | PESTEELARS | 3.98 | 11.5 | 24.34 | Timeout |
| grid_nnodes3192_ps2_k2 (3192, 2, 75, 217) | m-AOBF tree | OOM | OOM | OOM | OOM |
| | m-AOBF graph | OOM | OOM | OOM | OOM |
| | m-A* tree | OOM | OOM | OOM | OOM |
| | m-BB tree | Timeout | Timeout | Timeout | Timeout |
| | m-AOBB tree | Timeout | Timeout | Timeout | Timeout |
| | Nilsson | OOM | OOM | OOM | OOM |
| | STRIPES | 123.45 | 658.05 | 3035.29 | Timeout |
| | PESTEELARS | 26.86 | 81.27 | 172.35 | Timeout |

Table 16: Random binary grids, i-bound=20. Timeout - out of time, OOM - out of memory. 3 GB, 1 hour.

| instance | algorithm | i-bound=20 | | | |
|---|---|---|---|---|---|
| | | m=2 | m=5 | m=10 | m=25 |
| | | time | time | time | time |
| gen_nnodes132_ps1_k2 (132, 2, 13, 34) | m-AOBF tree | 0.01 | 0.02 | 0.03 | 0.05 |
| | m-AOBF graph | 0.01 | 0.03 | 0.06 | 0.09 |
| | m-A* tree | 0.0 | 0.01 | 0.02 | 0.02 |
| | m-BB tree | 0.0 | 0.0 | 0.03 | 0.05 |
| | m-AOBB tree | 0.03 | 0.09 | 5.44 | 120.67 |
| | Nilsson | 9.34 | 60.81 | 144.93 | 394.26 |
| | STRIPES | 0.5 | 1.32 | 3.13 | 13.24 |
| | PESTEELARS | 2.9 | 8.52 | 18.56 | 48.76 |
| gen_nnodes380_ps1_k2 (380, 2, 25, 61) | m-AOBF tree | OOM | OOM | OOM | OOM |
| | m-AOBF graph | OOM | OOM | OOM | OOM |
| | m-A* tree | 0.47 | 0.51 | 0.57 | 0.72 |
| | m-BB tree | 0.54 | 0.61 | 0.73 | 1.03 |
| | m-AOBB tree | 51.77 | 110.96 | 141.68 | 2027.05 |
| | Nilsson | 105.58 | 728.0 | 1753.98 | 4817.09 |
| | STRIPES | 2.07 | 6.2 | 13.21 | 76.0 |
| | PESTEELARS | 4.38 | 14.09 | 29.96 | 75.04 |
| gen_nnodes1122_ps1_k2 (1122, 2, 43, 112) | m-AOBF tree | OOM | OOM | OOM | OOM |
| | m-AOBF graph | OOM | OOM | OOM | OOM |
| | m-A* tree | OOM | OOM | OOM | OOM |
| | m-BB tree | Timeout | Timeout | Timeout | Timeout |
| | m-AOBB tree | Timeout | Timeout | Timeout | Timeout |
| | Nilsson | OOM | OOM | OOM | OOM |
| | STRIPES | 16.46 | 57.96 | 107.73 | 282.4 |
| | PESTEELARS | 9.69 | 28.84 | 61.04 | 158.7 |

Table 17: Random loopy graphs with submodular potentials, i-bound=20. Timeout - out of time, OOM - out of memory. 3 GB, 1 hour.

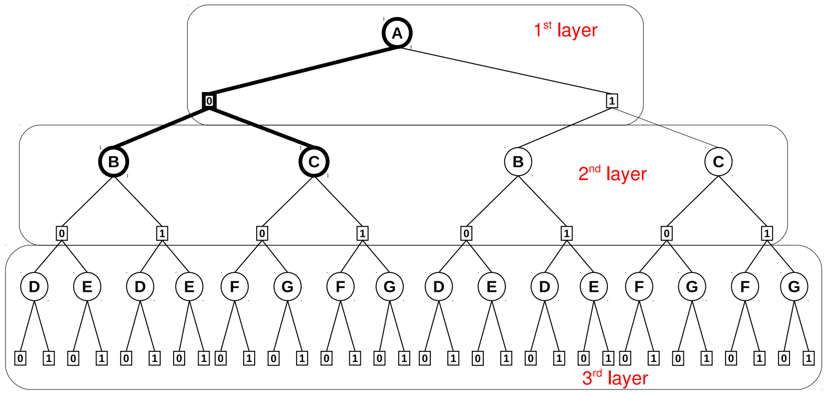



Figure 18: Example AND/OR search tree with 3 layers of OR and AND nodes.

## Appendix A. Proof of Theorem 13

Let $S_{\mathcal{T}}$ be the AND/OR search tree relative to pseudo tree $\mathcal{T}$ with depth $h$, $n$ be the number of variables, $k$ be the maximum domain size, and $deg$ be the maximum degree of the nodes in $\mathcal{T}$.

Define a *partial solution subtree* $T'$ to be a subtree of $S_{\mathcal{T}}$ such that: (1) $T'$ contains the root $s$ of $S_{\mathcal{T}}$; (2) if a non-terminal OR node $n$ is in $T'$, then $T'$ contains exactly one AND child node $n'$ of $n$; (3) if a non-terminal AND node $n$ is in $T'$ then $T'$ contains all OR child nodes $n'_1, \ldots, n'_j$ of $n$; (4) a leaf or tip node of $T'$ doesn't have any successors in $T'$.

The nodes in $S_{\mathcal{T}}$ are grouped into *layers*. There are $h$ layers such that the $i^{th}$ layer, denoted by $\mathcal{L}_i$, where $1 \leq i \leq h$, contains all the OR nodes whose variables have depth $i$ in $\mathcal{T}$, together with their AND children. We assume that the root of $\mathcal{T}$ has depth 1. For illustration, Figure 18 depicts an AND/OR search tree with 3 layers, where for example $\mathcal{L}_1 = \{A, \langle A, 0\rangle, \langle A, 1\rangle\}$.

We denote by $T_i^{OR}$ the set of partial solution subtrees whose leaf nodes are OR nodes in $\mathcal{L}_i$. Similarly, $T_i^{AND}$ is the set of partial solution subtrees whose leaf nodes are AND nodes in $\mathcal{L}_i$. A partial solution subtree $T' \in T_2^{OR}$ whose leaf nodes are OR nodes belonging to the 2nd layer is highlighted in Figure 18, namely $T' = \{A, \langle A, 0\rangle, B, C\}$.

**LEMMA 1.** *Given $T' \in T_i^{OR}$ and $T'' \in T_i^{AND}$ such that $T''$ is an extension of $T'$, then $T'$ and $T''$ have the same number of leaf nodes.*

*Proof.* Let $m$ be the number of OR leaf nodes in $T'$. By definition, each of the $m$ nodes can be extended by exactly one AND child node in $T''$. It follows that $T''$ has also $m$ AND leaf nodes. □

**LEMMA 2.** *Given $T' \in T_i^{OR}$, the number of leaf nodes $T'$, denoted by $m_i$, is at most $deg^{i-1}$.*

*Proof.* We show by induction that $m_i = deg^{i-1}$. If $i = 1$ then $m_1 = 1$. Assume that for $i = p-1$, $m_{p-1} = deg^{p-2}$, and let $T' \in T_{p-1}^{OR}$. We first extend $T'$ to $T'' \in T_{p-1}^{AND}$. By Lemma 1, $T''$ and $T'$ have the same number of leaf nodes, namely $m_{p-1}$. Next, we extend $T''$ to $T''' \in T_p^{OR}$. Since each of the $m_{p-1}$ AND leaf nodes in $T''$ can have at most $deg$ OR child nodes in $T'''$, it follows that $m_p$, the number of leaf nodes in $T'''$ is $m_p = m_{p-1} \cdot deg = deg^{p-2} \cdot deg = deg^{p-1}$. □

*Proof of Theorem 13* Consider the number of partial solution subtrees $N$ that are contained by $S_{\mathcal{T}}$:

$$N = \sum_{i=1}^{h} (N_i^{OR} + N_i^{AND}) \tag{3}$$

where $N_i^{OR} = |T_i^{OR}|$ and $N_i^{AND} = |T_i^{AND}|$, respectively.

Given $T' \in T_{i-1}^{AND}$, it is easy to see that $T'$ can be extended to a single partial solution subtree $T'' \in T_i^{OR}$ such that each of the leaf nodes in $T'$ has at most $deg$ OR child nodes in $T''$. Therefore:

$$N_i^{OR} = N_{i-1}^{AND} \tag{4}$$

Given $T' \in T_i^{OR}$, $T'$ can be extended to at most $k^m$ partial solution subtrees $T'' \in T_i^{AND}$ because each of the $m$ OR leaf nodes in $T'$ can have exactly one AND child node in $T''$ and $k$ bounds the domain size. By Lemmas 1 and 2, we then have that:

$$N_i^{AND} = N_i^{OR} \cdot k^{deg^{i-1}} \tag{5}$$

Using Equations 4 and 5, as well as $N_1^{OR} = 1$, we rewrite Equation 3 as follows:

$$\begin{aligned} N = & (1 + k) \\ & + (k + k^{deg+1}) \\ & + (k^{deg+1} + k^{deg^2+deg+1}) \\ & + ... \\ & + (k^{deg^{h-2}+deg^{h-3}+...+1} + k^{deg^{h-1}+deg^{h-2}+...+1}) \\ & \approx O(k^{\frac{deg^h-1}{deg-1}}) \end{aligned} \tag{6}$$

Thus, the worst-case number of partial solution subtrees that need to be stored in OPEN is $N \approx O(k^{deg^{h-1}})$. Therefore, the time and space complexity of m-AOBF follows as $O(k^{deg^{h-1}})$. When the pseudo tree $\mathcal{T}$ is balanced, namely each internal node has exactly $deg$ child nodes, the time and space complexity bound is to $O(k^n)$, since $n \approx O(deg^{h-1})$.

## References


Aljazzar, H., & Leue, S. (2011). K : A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, *175*(18), 2129–2154.

Batra, D. (2012). An efficient message-passing algorithm for the M-best MAP problem. *Uncertainty in Artificial Intelligence*.

Charniak, E., & Shimony, S. (1994). Cost-based abduction and MAP explanation. *Artificial Intelligence*, *66*(2), 345–374.

Darwiche, A. (2001). Decomposable negation normal form. *Journal of the ACM (JACM)*, *48*(4), 608–647.

Darwiche, A., Dechter, R., Choi, A., Gogate, V., & Otten, L. (2008). Results from the probablistic inference evaluation of UAI08, a web-report in http://graphmod.ics.uci.edu/uai08/Evaluation/Report. *In: Uncertainty in Artificial Intelligence applications workshop*.

de Campos, L. M., Gámez, J. A., & Moral, S. (1999). Partial abductive inference in bayesian belief networks using a genetic algorithm. *Pattern Recognition Letters*, *20*(11), 1211–1217.

de Campos, L. M., Gámez, J. A., & Moral, S. (2004). Partial abductive inference in bayesian networks by using probability trees. In *Enterprise Information Systems V*, pp. 146–154. Springer.

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

Dechter, R., & Mateescu, R. (2007). AND/OR search spaces for graphical models. *Artificial Intelligence*, *171*(2-3), 73–106.

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

Dechter, R. (2013). Reasoning with probabilistic and deterministic graphical models: Exact algorithms. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *7*(3), 1–191.

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

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269–271.

Elliott, P. (2007). Extracting the K Best Solutions from a Valued And-Or Acyclic Graph. Master's thesis, Massachusetts Institute of Technology.

Eppstein, D. (1994). Finding the k shortest paths. In *Proceedings 35th Symposium on the Foundations of Computer Science*, pp. 154–165. IEEE Comput. Soc. Press.

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

Fishelson, M. a., Dovgolevsky, N., & Geiger, D. (2005). Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, *59*(1), 41–60.

Flerova, N., Dechter, R., & Rollon, E. (2011). Bucket and mini-bucket schemes for m best solutions over graphical models. In *Graph structures for knowledge representation and reasoning workshop*.

Fromer, M., & Globerson, A. (2009). An lp view of the m-best map problem. *Advances in Neural Information Processing Systems*, *22*, 567–575.

Ghosh, P., Sharma, A., Chakrabarti, P., & Dasgupta, P. (2012). Algorithms for generating ordered solutions for explicit AND/OR structures. *Journal of Artificial Intelligence (JAIR)*, *44*(1), 275–333.

Gogate, V. G. (2009). *Sampling Algorithms for Probabilistic Graphical Models with Determinism DISSERTATION*. Ph.D. thesis, University of California, Irvine.

Hamacher, H., & Queyranne, M. (1985). K best solutions to combinatorial optimization problems. *Annals of Operations Research*, *4*(1), 123–143.

Ihler, A. T., Flerova, N., Dechter, R., & Otten, L. (2012). Join-graph based cost-shifting schemes. *arXiv preprint arXiv:1210.4878*.

Kask, K., & Dechter, R. (1999a). Branch and bound with mini-bucket heuristics. In *IJCAI*, Vol. 99, pp. 426–433.

Kask, K., & Dechter, R. (1999b). Mini-bucket heuristics for improved search. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 314–323. Morgan Kaufmann Publishers Inc.

Kjærulff, U. (1990). Triangulation of graphs–algorithms giving small total state space. *Tech. Report R-90-09*.

Lawler, E. (1972). A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, *18*(7), 401–405.

Marinescu, R., & Dechter, R. (2009a). AND/OR Branch-and-Bound search for combinatorial optimization in graphical models. *Artificial Intelligence*, *173*(16-17), 1457–1491.

Marinescu, R., & Dechter, R. (2009b). Memory intensive AND/OR search for combinatorial optimization in graphical models. *Artificial Intelligence*, *173*(16-17), 1492–1524.

Marinescu, R., & Dechter, R. (2005). AND/OR branch-and-bound for graphical models. In *International Joint Conference on Artificial Intelligence*, Vol. 19, p. 224. Lawrence Erlbaum Associates Ltd.

Nillson, N. J. (1980). *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA.

Nilsson, D. (1998). An efficient algorithm for finding the M most probable configurations in probabilistic expert systems. *Statistics and Computing*, *8*(2), 159–173.

Nilsson, N. (1982). *Principles of artificial intelligence*. Springer Verlag.

Otten, L., & Dechter, R. (2011). Anytime AND/OR depth first search for combinatorial optimization. In *SOCS*.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies*. Addison-Wesley.

Schiex, T. (2000). Arc consistency for soft constraints. *International Conference on Principles and Practice of Constraint Programming (CP)*, 411–424.

Seroussi, B., & Golmard, J. (1994). An algorithm directly finding the K most probable configurations in Bayesian networks. *International Journal of Approximate Reasoning*, *11*(3), 205–233.

Wainwright, M. J., & Jordan, M. I. (2003). Variational inference in graphical models: The view from the marginal polytope. In *Proceedings of the Annual Allerton congerence on communication control and computing*, Vol. 41, pp. 961–971. Citeseer.

Wemmenhove, B., Mooij, J. M., Wiegerinck, W., Leisink, M., Kappen, H. J., & Neijt, J. P. (2007). Inference in the promedas medical expert system. In *Artificial intelligence in medicine*, pp. 456–460. Springer.

Yanover, C., & Weiss, Y. (2004). Finding the M Most Probable Configurations Using Loopy Belief Propagation. In *Advances in Neural Information Processing Systems 16*. The MIT Press.

Yanover, C., Schueler-Furman, O., & Weiss, Y. (2008). Minimizing and learning energy functions for side-chain prediction. *Journal of Computational Biology*, *15*(7), 899–911.