

# Subproblem Ordering Heuristics for AND/OR Best-First Search

William Lam<sup>a,1,\*</sup>, Kalev Kask<sup>a</sup>, Javier Larrosa<sup>b</sup>, Rina Dechter<sup>a</sup>

<sup>a</sup>*Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA 92697, USA*

<sup>b</sup>*Computer Science Department, UPC Barcelona Tech, Campus Nord, Calle Jordi Girona, 1-3, 08034 Barcelona, Spain*

---

## Abstract

Best-first search can be regarded as anytime scheme for producing lower bounds on the optimal solution, a characteristic that is mostly overlooked. We explore this topic in the context of AND/OR best-first search, guided by the MBE heuristic, when solving graphical models. In that context, the impact of the secondary heuristic for subproblem ordering may be significant, especially in the anytime context. Indeed, our paper illustrates this, showing that a new concept of bucket errors can advise in providing effective subproblem orderings in AND/OR search for both exact and anytime solutions.

*Keywords:* graphical models; heuristic search; combinatorial optimization; artificial intelligence

---

## 1. Introduction

An important problem in the field of artificial intelligence is the *min-sum* problem over *graphical models*, which includes the *most probable explanation*, or *maximum a posterior* query over probabilistic graphical models [1]. This problem has numerous applications including *scheduling*, *genetic linkage analysis*, and *protein side chain prediction* [2, 3, 4]. It is often solved by search schemes such as depth-first branch-and-bound or best-first search such as A\*. However, since it is too complex to be solved exactly, research often settles on anytime schemes. Indeed, AND/OR Branch-and-Bound (AOBB) is one such anytime algorithm that generates upper bounds through a sequence of improved sub-optimal solutions over time [5, 6]. However, there has been little work on the symmetrical problem of **generating lower bounds by search in an anytime manner**, which is the focus of this paper.

---

\*Corresponding author

*Email addresses:* [wllmlam@uci.edu](mailto:wllmlam@uci.edu) (William Lam), [kkask@ics.uci.edu](mailto:kkask@ics.uci.edu) (Kalev Kask), [larrosa@cs.upc.edu](mailto:larrosa@cs.upc.edu) (Javier Larrosa), [dechter@ics.uci.edu](mailto:dechter@ics.uci.edu) (Rina Dechter)

<sup>1</sup>Currently at Google Inc.

We turn to best-first search for this task, which explores the search space in  
15 frontiers of non-decreasing lower bounds and is thus inherently anytime for pro-  
ducing lower bounds. Specifically, we will explore *AND/OR* Best-First (AOBF)  
search, guided by the mini-bucket heuristic, which is known to be a state-of-  
the-art algorithm for the min-sum task, but which has been evaluated mostly  
20 on its performance for finding an optimal solution [7]. The scheme can be eas-  
ily adapted to yield a sequence of lower bounds by simply reporting the best  
heuristic evaluation it has seen so far. Indeed, this idea was used in recent work  
to produce algorithms that give both upper and lower bounds on the optimal  
solution [8, 9].

The focus of our paper is on the specific aspect of the impact of AND child  
25 node ordering on AOBF’s ability to generate lower bounds in an anytime man-  
ner. AOBF is guided by two heuristic evaluation functions. In the AND/OR  
search space, the “best” is a *partial solution graph* with the best potential solu-  
tion according to a heuristic evaluation function  $f_1$  amongst all partial solution  
graphs of the currently explored search space. A second heuristic  $f_2$  prioritizes  
30 which leaf (known as *tip nodes*) of the current best partial solution graph should  
be expanded next. We call this the AND subproblem ordering. Quoting Pearl  
(page 50) [10],

“These two functions, serving in two different roles, provide two dif-  
ferent types of estimates:  $f_1$  estimates some properties of the set  
35 of solution graphs that may emanate from a given candidate base,  
whereas  $f_2$  estimates the amount of information that a given node  
expansion may provide regarding the alleged superiority of its host-  
ing graph. Most works in search theory focus on the computation of  
 $f_1$ , whereas  $f_2$  is usually chosen in an ad-hoc manner.”

40 Indeed, in most current implementations of AOBF,  $f_2$  is simply chosen to be  
equal to  $f_1$ . We show in this paper that the choice of  $f_2$  has a significant impact  
on the anytime performance of AOBF for finding lower bounds. In our analysis,  
we show that a concept known as the *residual*, which captures a local accuracy  
of the heuristic evaluation function, is a natural choice for  $f_2$ . In [11], it was  
45 shown that the residual of the mini-bucket heuristic can be approximated by its  
*local errors*. We illustrate empirically that the local bucket errors can provide  
relevant information on the increase of the lower bound due to a given node  
expansion. While we work in the AND/OR search framework in this paper, the  
connection to similar search frameworks such as recursive conditioning [12] and  
50 backtracking with tree decomposition (BTD) [13, 14] is clear. Furthermore,  
[15] investigates subproblem ordering in the context of BTD, but for depth-  
first search only. To our knowledge, this is a first investigation of subproblem  
ordering for best-first search in the AND/OR search space and among the first  
investigations of anytime best-first search for generating lower bounds.

55 The rest of this paper is organized as follows: Section 2 presents the back-  
ground on graphical models, the AOBF algorithm, and mini-bucket heuristics.  
Section 3 analyzes the impact of subproblem ordering and illustrates it with an

example. Section 4 introduces the subproblem ordering heuristic based on residuals and local bucket errors and suggests a way to approximate them. Section 5 presents the experiments and section 6 concludes.

## 2. Background

### 2.1. Graphical Models

$X_k, x_k$	variable, assigned variable
$f_j, S_{f_j}$	function, scope
$\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$	graphical model
$G = (V, E)$	primal graph
$G^*(d)$	induced graph relative to order $d$
$w^*(d)$	induced width relative to order $d$

Table 1: Notation on graphical models.

We will use  $X_k$  to denote a variable and  $D_k$  its domain. A generic domain value will be noted  $x_k$ . A variable assignment will be denoted  $(X_k, x_k)$  or, when the context is clear, just  $x_k$ . We will use  $f_j$  to denote a function and  $S_{f_j}$  its scope, which is the set of variables for which the function is defined (i.e.,  $f_j : \prod_{X_k \in S_{f_j}} D_k \rightarrow \mathbb{R}$ ). Table 1 provides a summary of the notation. A graphical model is a collection of functions over subsets of variables, which we define next.

**Definition 1 (graphical model  $\mathcal{M}$ ).** A graphical model  $\mathcal{M}$  is a tuple  $(\mathbf{X}, \mathbf{D}, \mathbf{F})$ , where

1.  $\mathbf{X} = \{X_1, \dots, X_n\}$  is a finite set of variables
2.  $\mathbf{D} = \{D_1, \dots, D_n\}$  is a set of finite domains associated with each variable.
3.  $\mathbf{F} = \{f_1, \dots, f_m\}$  is a set of local functions with scope  $S_{f_j} \subseteq \mathbf{X}$  for all  $f_j$ .

A graphical model represents a *global* function which is the sum of all the local functions, denoted  $\sum_{f_j \in \mathbf{F}} f_j(\cdot)$ . Graphical models are used to model complex systems and their main virtue is allowing compact representation and their structure can often allow efficient query processing. Our focus is on the min-sum problem, defined as follows:

**Definition 2 (min-sum problem).** Given a graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ , the min-sum problem seeks an optimal assignment of its variables that minimizes the global function. Namely, finding  $x^*$ , satisfying,

$$x^* = \operatorname{argmin}_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot)$$

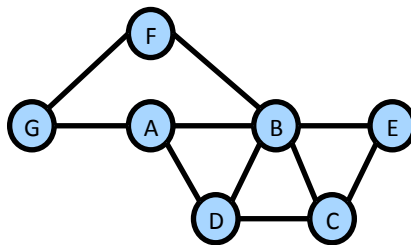


Figure 1: A primal graph of a graphical model with 7 variables.

80 **Definition 3 (primal graph  $G$ ).** The primal graph  $G = (V, E)$  of a graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$  has one node associated with each variable (i.e.,  $V = \mathbf{X}$ ) and edges  $(X_k, X_{k'}) \in E$  for each pair of variables that appear in the same scope  $S_{f_j}$  of some local function  $f_j \in \mathbf{F}$ .

Consider a graphical model with 7 variables indexed from  $A$  to  $G$  with  
 85 binary functions  $\mathbf{F} = \{f(A), f(A, B), f(A, D), f(A, G), f(B, C), f(B, D), f(B, E), f(B, F), f(C, D), f(C, E), f(F, G)\}$ . The primal graph appears in **Figure 1** which has a node for each variable and edges connecting variables which appear in the same function scope.

**Definition 4 (induced width [1]).** Given a primal graph  $G = (V, E)$ , an  
 90 ordered graph is a pair  $(G, d)$ , where  $d = (X_1, \dots, X_n)$  is an ordering of the nodes. The nodes adjacent to  $X_k$  that precede it in the ordering are called its parents. The width of a node in an ordered graph is its number of parents. The width of an ordered graph  $(G, d)$ , denoted  $w(d)$ , is the maximum width over all nodes. The width of a graph is the minimum width over all orderings of the graph. The induced graph of an ordered graph  $(G, d)$  is an ordered graph  
 95  $(G^*, d)$ , where  $G^*$  is obtained from  $G$  as follows: the nodes of  $G$  are processed from last to first along  $d$ . When a node  $X_k$  is processed, all of its parents are connected. The induced width of an ordered graph  $(G, d)$ , denoted  $w^*(d)$ , is the maximum number of parents a node has in the induced ordered graph  $(G^*, d)$ .  
 100 The induced width of a graph  $w^*$ , is the minimum induced width over all its orderings.

The complexity of the *min-sum* problem for a given graphical model can be bounded by the *induced width*  $w^*$  of its associated primal graph [5, 1].

## 2.2. AND/OR Search for Graphical Models

105 In the context of graphical models, the conditional independencies in the model can be exploited via the AND structures in AND/OR search spaces. AND/OR search spaces for graphical models are defined relative to a *pseudo tree* of the primal graph [16].

**Definition 5 (pseudo tree [16, 17, 18]).** Given an undirected graph  $G = (V, E)$ , a directed rooted tree  $\mathcal{T} = (V, E')$  defined on all its nodes is a pseudo tree if  
 110

$\mathcal{T}$	pseudo-tree (nodes correspond to variables)
$\bar{X}_k$	pseudo-tree path from root to $X_k$
$\mathcal{T}_k$	sub-tree rooted by $X_k$
$\mathcal{T}_{k,d}$	sub-tree rooted by $X_k$ with depth $d$
$c(X_k, x_k)$	cost of arc from OR node $X_k$ to AND node $x_k$
$\bar{x}_k$	path from root to AND node $x_k$

Table 2: Notation on AND/OR search for graphical models.

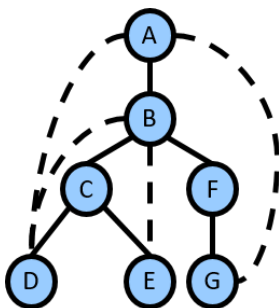


Figure 2: A pseudo tree for the running example. Solid arcs form the main tree structure and dotted arcs the back-arcs.

any arc of  $G$  which is 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  $E$ .

**Figure 2** shows a pseudo tree for our running example.

In general, an AND/OR search tree [19, 10] has two types of nodes. OR nodes representing branching points where a decision has to be made (i.e. choosing a value for a variable), and AND nodes representing sets of conditionally independent subproblems that need to be solved. In mapping graphical models to AND/OR search, the children of OR nodes are AND nodes, and the children of AND nodes are OR nodes. There is a cost associated with each edge between an OR node and its child AND node extracted from the functions of the model.

**Definition 6 (AND/OR search tree [16]).** Given a graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$  and a pseudo tree  $\mathcal{T}$ , its AND/OR search tree consists of alternating levels of OR and AND nodes. OR nodes are labeled with a variable  $X_k \in \mathbf{X}$ . Its children are AND nodes, each labeled with an instantiation  $x_k$  of  $X_k$ . Children of AND nodes are OR nodes, labeled with the children of  $X_k$  in  $\mathcal{T}$ . Each child represents a conditionally independent subproblem given assignments to their ancestors. The root of the AND/OR search tree is an OR node labeled by the variable at the root of  $\mathcal{T}$ .

The path from the root to an AND node  $x_k$  represents a unique assignment to the variables in  $\bar{X}_k$ , that will be denoted  $\bar{x}_k$  (see Table 2). In the AND/OR search tree, the costs of the OR-to-AND arcs denoted  $c(X_k, x_k)$  (abusing notation, since they are dependent on the path to  $x_k$ ) are defined as follows:

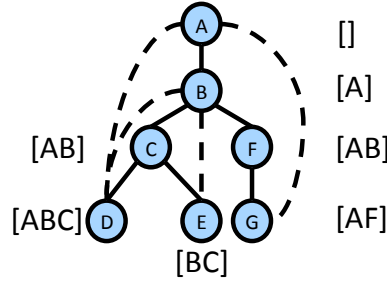


Figure 3: The same pseudo-tree as Figure 2, annotated with contexts.

**Definition 7 (arc cost  $c(X_k, x_k)$ ).** The cost  $c(X_k, x_k)$  of the arc  $(X_k, x_k)$  along a path  $\bar{x}_k$  is the sum of all the functions whose scope includes  $X_k$  that are fully assigned by the values specified along the path  $\bar{x}_k$  from the root to node  $x_k$ .

For completeness, we define the costs of edges from AND nodes to OR nodes to be 0.

A more compact search space can be obtained if identical subproblems in the AND/OR tree are merged, producing an AND/OR graph [16]. A class of identical subproblems can be identified in terms of their OR context,

**Definition 8 (OR context).** The OR context of a variable  $X_k$  in a pseudo tree  $\mathcal{T} = (V, E')$  is the set of ancestor variables connected to  $X_k$  or its descendants by arcs in  $E'$ .

Figure 3 shows the same pseudo-tree in the running example with OR contexts.

**Definition 9 (context-minimal AND/OR search graph [16]).** Identical subproblems can be merged based on the OR context. We can merge two nodes if they have the same assignment to the context variables. This yields a context-minimal AND/OR search graph  $C_{\mathcal{T}}$  whose size can be shown to be bounded exponentially in the induced width of  $G$  along the pseudo-tree  $\mathcal{T}$ .

A solution tree  $T$  of the AND/OR search tree or graph corresponds to complete assignments of the variables in the graphical model, defined next.

**Definition 10 (solution tree).** A solution tree  $T$  is a subtree of the AND/OR search graph  $C_{\mathcal{T}}$  such that:

1. It contains the root node of  $C_{\mathcal{T}}$ ;
2. If it contains an internal AND node  $n$ , then all children of  $n$  are also in  $T$ ;
3. If it contains an internal OR node  $n$ , then exactly one AND node child is in  $C_{\mathcal{T}}$ ;
4. Every tip node in  $T$  (nodes with no children) is a terminal node of  $C_{\mathcal{T}}$ .

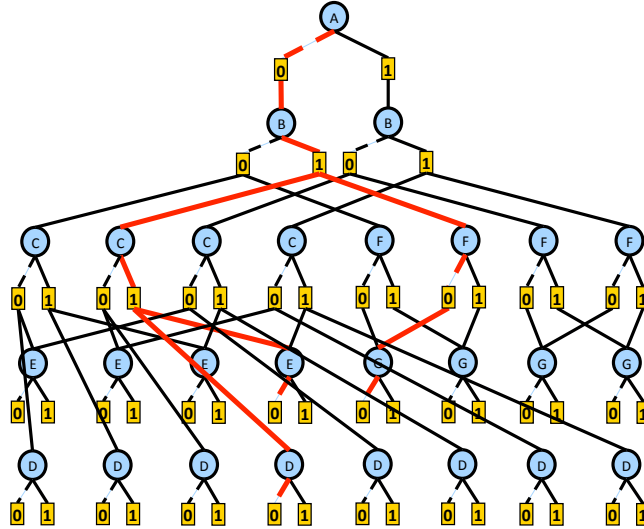


Figure 4: Example AND/OR search graph. A solution tree is highlighted in red.

The cost of a solution tree is the sum of its arc weights associated with the arcs of  $C_{\mathcal{T}}$ .

The cost of a solution tree corresponds to the cost of the assignment it represents as given by the global function of the model. Thus, for the min-sum  
 165 problem, the optimal solution tree corresponds to the optimal solution.

**Figure 4** shows the corresponding context-minimal AND/OR search graph guided by the pseudo tree in **Figure 3**. Since the context of variable  $E$  is only over  $B, C$ , the corresponding OR nodes have been merged with respect to  $A$ , which is an ancestor not in the context. Similarly, the OR nodes of  $G$  are  
 170 merged with respect to  $B$ . The *solution tree* corresponding to the assignment  $(A = 0, B = 1, C = 1, D = 0, E = 0, F = 0, G = 0)$  is highlighted in red.

### 2.2.1. AND/OR Best-First Search

$\mathcal{G}$	currently explicated search graph
$T$	current best solution tree to $\mathcal{G}$
$r$	root of the search graph (initial node in $\mathcal{G}$ )
$n$	node in $\mathcal{G}$
$l(n)$	current best lower bound on below $n$
$succ(n)$	set of successors (children) of $n$

Table 3: Notation on AND/OR Best-First search.

The AO\* algorithm is a best-first search algorithm for AND/OR search spaces [19]. Our AOBF (AND/OR Best First) algorithm is a variant of AO\*  
 175 specialized for graphical models [7]. In the following, we use the notation in

---

**Algorithm 1: AND/OR Best-First (AOBF) [7]**

---

**Input:** A graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ , pseudo-tree  $\mathcal{T}$ , heuristic function  $h$ , ordering function  $f_2$ . ( $f_1$  is defined by  $l(n)$ , which depends on  $h$ )

**Output:** Lower-bound to solution of  $\mathcal{M}$

- 1 Create the root OR node  $r$  labeled by  $X_1$  and let  $\mathcal{G} = \{r\}$
- 2 Initialize value  $l(r) = h(r)$  and best partial solution tree  $T$  to  $\mathcal{G}$
- 3  $n \rightarrow r$  // Initial node  $n$  to expand
- 4 **while**  $r$  is not marked SOLVED and memory is available **do**
  - // Expand
  - 5 **if**  $n$  is an OR node (labeled  $X_k$ ) **then**
    - 6 Create AND node  $n'$  for each  $x_k \in D_k$
    - 7 **if**  $n'$  is TERMINAL **then** mark  $n'$  as SOLVED;
  - 8 **else if**  $n$  is an AND node (labeled  $x_k$ ) **then**
    - 9 Create OR node  $n'$  for each child  $X_j$  of  $X_k \in \mathcal{T}$  if  $n' \notin \mathcal{G}$
  - 10 **foreach** generated  $n'$  **do**
    - 11  $\text{succ}(n) \leftarrow \text{succ}(n) \cup n'$
    - 12  $l(n') \leftarrow h(n')$  // Initial lower bound is the heuristic
  - 13  $\mathcal{G} \leftarrow \mathcal{G} \cup \{\text{succ}(n)\}$
  - // Revise
  - 14  $S \leftarrow \{n\}$  // Set to keep track of nodes needing revision
  - 15 **while**  $S \neq \emptyset$  **do**
    - 16 Select  $p$  from  $S$  s.t.  $p$  has no descendants in  $\mathcal{G} \cap S$
    - 17  $S \leftarrow S - \{p\}$
    - 18 **if**  $p$  is an OR node **then**
      - 19  $l(p) \leftarrow \min_{m \in \text{succ}(p)} (c(p, m) + l(m))$
      - 20 Mark  $k = \text{argmin}_m$  as the best successor of  $p$
      - 21 **if**  $k$  is SOLVED **then** mark  $p$  as SOLVED;
      - 22 **if**  $l(p)$  changed and  $p = r$  **then**
        - 23  $\text{Output } l(p)$  // Report new lower bound
    - 24 **else if**  $p$  is an AND node **then**
      - 25  $l(p) \leftarrow \sum_{m \in \text{succ}(p)} l(m)$
      - 26 **if**  $\forall m \in \text{succ}(p)$  are SOLVED **then** mark  $p$  as SOLVED;
    - 27 **if**  $l(p)$  changed or  $p$  is SOLVED **then**
      - 28  $S \leftarrow S \cup \{\text{parent}(p)\}$
  - 29 Update  $T$  to new best partial solution tree w.r.t.  $f_1$  by including all nodes visited by DFS on  $\mathcal{G}$  that only includes the best marked AND nodes.
  - 30 **if**  $r$  is not solved **then**
    - // Choose best tip based on  $f_2$  for next expansion
    - 31  $n \leftarrow \text{argmax}_{n \in \text{tips}(T)} f_2(n)$
- 32 **return**  $\langle l(r), T \rangle$

---



Table 3. The algorithm works by gradually expanding a portion (denoted  $\mathcal{G}$ ) of the context-minimal AND/OR search graph  $C_{\mathcal{T}}$ , always identifying the best *partial solution tree*  $T$  in  $\mathcal{G}$ . After node expansion, every node  $n \in \mathcal{G}$  needs to be updated with its best lower bound based on the current state of  $\mathcal{G}$ , denoted  $l(n)$ , which corresponds to the  $f_1$  evaluation function. As usual in AO\* search, AOBF [7], which explores the context-minimal AND/OR search graph, presented in Algorithm 1, interleaves a top-down expansion step and a bottom-up revision step. The top-down expansion step generates the children of the next node to expand  $n$  (initially the root  $r$ ), which are appended to  $\mathcal{G}$ . (lines 4-13). Note that if an OR child of an AND node already exists, we do not generate it again to ensure we are searching the search graph  $C_{\mathcal{T}}$  (line 9) A bottom-up revision step then updates the internal nodes values that represent the current best lower bounds below them (lines 15-26). In this step, the values of newly expanded children are propagated to their parents, and recursively up to the root (stored in  $l(n)$ ). During these value updates, the algorithm marks the best child of each OR node, which serves as a bookkeeping step to keep track of the assignment corresponding to propagated values. In addition, OR nodes are marked *solved* if their current best child is marked *solved* and AND nodes are marked *solved* if all of their children are marked *solved*. Subsequently, a new best partial solution tree  $T$  w.r.t.  $f_1$  is identified by following the marked best children from the root. If the root  $r$  is not solved, then the next node  $n$  to expand from  $T$  is determined by  $f_2$ . Otherwise,  $T$  is necessarily a full solution tree that corresponds to the optimal solution, with a cost equal to value of the root node  $l(r)$ . and AOBF terminates.

In earlier literature the algorithm is provided as a purely exact algorithm [19, 7]. However, since lower bound values are constantly updated as the algorithm searches,  $l(r)$  provides a *anytime lower bound* on the optimal solution (lines 22-23).

On the  $f_2$  function for selecting the next node to expand, it is common to choose an  $f_2$  which orders the non-terminal tips using the  $f_1$  heuristic in ascending order. The focus of our work is in proposing a more informed  $f_2$ .

### 2.3. Mini-Bucket Elimination Heuristics

$B_k, S_{B_k}$	bucket associated to pseudo-tree node $X_k$ , scope
$B_k^r$	mini-bucket associated to pseudo-tree node $X_k$
$\lambda_{k \rightarrow p}$	message computed at $B_k$ and sent to $B_p$
$h(\bar{x}_p)$	heuristic value of node $\bar{x}_p$

Table 4: Notation on bucket elimination for graphical models.

A commonly used heuristic guiding AND/OR search is based on the MBE (mini-bucket elimination) algorithm [20], which generates a lower bound on the optimal cost. It has a parameter known as the  $i$ -bound which allows trading off pre-processing time and space for heuristic accuracy with actual search. MBE works relative to the same pseudo-tree  $\mathcal{T}$  which defines the AND/OR search

graph. See Table 4 for a summary of the notation used in the rest of this section.

---

**Algorithm 2:** Mini-Bucket Elimination [20]

---

**Input:** Graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ , pseudo tree  $\mathcal{T}$ , bounding parameter  $i$ -bound

**Output:** Lower bound to min-sum on  $\mathcal{M}$  and messages  $\lambda_{q \rightarrow k}^s$

```

1 foreach  $X_k \in \mathbf{X}$  in bottom up order according to  $\mathcal{T}$  do
2    $B_k \leftarrow \{f_j \in \mathbf{F} \mid X_k \in S_j\}$ 
3    $\mathbf{F} \leftarrow \mathbf{F} - \mathbf{F}_k$ 
4   Put all generated messages  $\lambda_{q \rightarrow k}^s$  in  $B_k$ 
5   Partition the  $B_k$  into mini-buckets  $B_k^1, \dots, B_k^r$  with scope bounded by
   the  $i$ -bound
6   foreach  $B_k^s \in B_k^1, \dots, B_k^r$  do
7     Let  $X_a$  be closest ancestor variable of  $X_k$  in the mini-bucket
8     Generate message:  $\lambda_{k \rightarrow a}^s \leftarrow \min_{X_k} \sum_{f_j \in B_k^s} f_j$ 
9 return All  $\lambda$ -messages generated (root message is the min-sum lower
   bound)

```

---

215 Each variable  $X_k \in \mathcal{T}$  is associated with a bucket  $B_k$  that includes a subset of functions from  $\mathbf{F}$ . A function  $f_j$  is placed into a bucket  $B_k$  if  $X_k$  is the deepest variable in  $\mathcal{T}$  such that  $X_k \in S_{f_j}$ . The scope of a bucket  $B_k$  (denoted  $S_{B_k}$ ) is the union of the scopes of its functions. The overall idea is that each bucket  $B_k$  generates a *message*  $\lambda_{k \rightarrow p} = \min_{X_k} B_k$ , where  $p$  is the parent of  $k$  in the pseudo tree  $\mathcal{T}$ . Processing a bucket may require partitioning the bucket's  
220 functions into *mini-buckets*  $B_k = \cup_r B_k^r$ , where each  $B_k^r$  includes no more than  $i$  variables. In this case, each mini-bucket is processed separately as if they were associated with different variables. If no partitioning is necessary, then the message computed by the root variable is the optimal cost. (The algorithm  
225 reduces to the exact bucket elimination algorithm [1] in this case.) Otherwise, it is a lower bound.

We provide details in Algorithm 2. The main loop (lines 1-9) partitions and generates the  $\lambda$  messages which are propagated upward. The time and space complexity is  $O(nrk^i)$ , where  $r$  is the maximum number of mini-buckets for any  
230 variable [1].

We provide an example in **Figure 5** for our example problem. Here, we use an  $i$ -bound of 3. In this case, starting with variable  $D$ , we have its bucket which contains the functions  $f(A, D), f(B, D), f(C, D)$  that all contain variable  $D$ . However, the total scope size here is 4, which exceeds the  $i$ -bound of 3.  
235 Therefore, we partition it into two mini-buckets and each generates a separate  $\lambda$  message, as if they were separate variables. For the rest of the variables, the  $i$ -bound is satisfied, so there is no need to partition them.

MBE messages can be used to construct a heuristic for search [21]. These heuristics are referred to as *static* heuristics since they are pre-compiled before

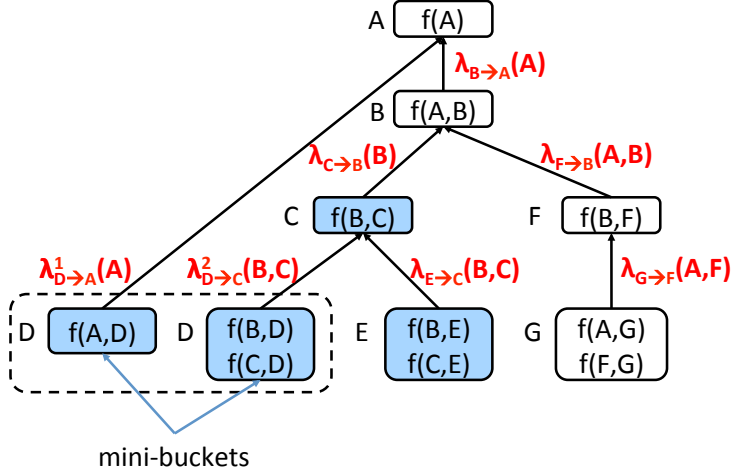


Figure 5: Example of mini-bucket elimination on the running example using an  $i$ -bound of 3.

240 search starts. During search, they can be extracted efficiently by table lookups.

**Definition 11 (MBE heuristic).** Let  $\bar{x}_p$  be a partial assignment and  $\bar{X}_p$  be the set of corresponding instantiated variables.  $\lambda_{k \rightarrow q}^s(\bar{x}_p)$  denotes the message sent from  $B_k$  to  $B_q$ . (We abuse notation here by stating its argument as  $\bar{x}_p$ , though it may only contain a subset of  $\bar{x}_p$ .) The heuristic value for  $\bar{x}_p$  is given by:

$$h(\bar{x}_p) = \sum_{X_k \in \mathcal{T}_p} \sum_{s \in \{1, \dots, r_k \mid X_q \in \bar{X}_p\}} \lambda_{k \rightarrow q}^s(\bar{x}_p) \quad (1)$$

where  $X_k \in \mathcal{T}_p$  denotes the set of variables in the pseudo subtree rooted by  $X_p$ , excluding  $X_p$  and  $r_k$  denotes the number of mini-buckets for variable  $X_k$ .

245 *Example.* In the example, (see **Figure 5**), the heuristic function of the partial assignment  $(A = 0, B = 1)$  is  $h(A = 0, B = 1) = \lambda_{D \rightarrow A}(A = 0) + \lambda_{C \rightarrow B}(B = 1) + \lambda_{F \rightarrow B}(A = 0, B = 1)$ .

### 2.3.1. Residuals and Local Bucket Error

250 Look-ahead is a method of improving accuracy of the heuristic value for a given node  $n$  by expanding the search tree below a node to a certain depth  $d$  and backing up the heuristic values of descendant nodes in the search space. This improved heuristic is denoted  $h^d(n)$ . The residual is the gain produced by look-ahead:

**Definition 12 (residual).** The depth  $d$  residual of a node  $n$  is

$$res^d(n) = h^d(n) - h(n) \quad (2)$$

The notion of local bucket error was introduced as a function that captures the local error induced by mini-bucket elimination [11]. It was introduced to capture the heuristic error and was shown to be identical to the depth 1 look-ahead residual. The bucket error is the difference between the message that would have been computed in a bucket without partitioning (called an *exact bucket message*  $\mu_k^*$ ) and the message computed by the mini-buckets (called a *combined mini-bucket message*  $\mu_k$ ).

We define these notions below.

**Definition 13 (combined bucket and mini-bucket messages).** *Given a mini-bucket partition  $B_k = \cup_r B_k^r$ , we define the combined mini-bucket message at  $B_k$ ,*

$$\mu_k(\cdot) = \sum_r \left( \min_{x_k} \sum_{f \in B_k^r} f(\cdot) \right) \quad (3)$$

*In contrast, the exact bucket message without partitioning at  $B_k$  is*

$$\mu_k^*(\cdot) = \min_{x_k} \sum_{f \in B_k} f(\cdot) \quad (4)$$

Note that although we say that  $\mu_k^*$  is exact, it is exact only *locally* to  $B_k$  since it may contain partitioning errors introduced by messages computed in earlier processed buckets. We now define the local error for MBE,

**Definition 14 (local bucket error of MBE [11]).** *Given a completed run of MBE, the local bucket error function at  $B_k$  denoted  $E_k$  is*

$$E_k(\cdot) = \mu_k^*(\cdot) - \mu_k(\cdot) \quad (5)$$

*The scope of  $E_k$  is the set of variables in  $B_k$  excluding  $X_k$ .*

An alternative measure is the *relative local bucket error*, computed by dividing each  $E_k(\cdot)$  term by the exact bucket message  $\mu_k^*(\cdot)$ . This serves as a way to normalize the error with respect to the function values, which can vary in scale amongst the bucket errors in practice.

---

**Algorithm 3: Local Bucket Error Evaluation (LBEE)**

---

**Input:** A Graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ , a pseudo tree  $\mathcal{T}$ ,  $i$ -bound

**Output:** Error function  $E_k$  for each bucket  $B_k$

**Initialization:** Run  $MBE(i)$  w.r.t.  $\mathcal{T}$ .

**foreach**  $B_k, X_k \in \mathbf{X}$  **do**

Let  $B_k = \cup_r B_k^r$  be the partition used by  $MBE(i)$

$\mu_k = \sum_r (\min_{x_k} \sum_{f \in B_k^r} f)$

$\mu_k^* = \min_{x_k} \sum_{f \in B_k} f$

$E_k = \mu_k^* - \mu_k$

**return**  $E_k$  functions

---

**Algorithm Local Bucket Error Evaluation (LBEE) 3** (LBEE) computes the local bucket error for each bucket. Following the execution of  $MBE(i)$ ,  
 270 a second pass is performed from leaves to root along the pseudo tree. When  
 processing a bucket  $B_k$ , LBEE computes the combined mini-bucket message  $\mu_k$ ,  
 the exact bucket message  $\mu_k^*$ , and the error function  $E_k$ . The complexity of processing  
 each bucket is exponential in the scope of the bucket after the execution  
 of  $MBE(i)$ . The total complexity is therefore dominated by the largest scope  
 275 of the output buckets. This number is called the *pseudo-width*, defined next.

**Definition 15 (pseudo-width(i)).** *Given a run of  $MBE(i)$  along pseudo tree  $\mathcal{T}$ , the pseudo-width of  $B_k$ ,  $psw_k^{(i)}$  is the number of variables in  $B_k$  after all messages have been received. The pseudo-width of  $\mathcal{T}$  relative to  $MBE(i)$  is  $psw(i) = \max_{X_k} \{psw_k^{(i)}\}$*

**Theorem 1 (complexity of LBEE).** *The time and space complexity of LBEE is  $O(nk^{psw(i)})$ , where  $n$  is the number of variables,  $k$  bounds the domain size, and  $psw(i)$  is the pseudo-width along  $\mathcal{T}$  relative to  $MBE(i)$ .*

### 3. Illustrating the Impact of Subproblem Ordering

We will now show that the  $f_2$  heuristic can have a potentially large impact  
 285 on AOB performance, starting with an example. In order to reason about  
 the sequence of expansions that occur during AOB, we define the notion of a  
 profile of  $f_2$  relative to  $f_1$ . In the following,  $T_i$  refers to the best partial solution  
 tree after the  $i$ -th node expansion of AOB.

**Definition 16 (profile).** *Given a primary and secondary heuristic function  $f_1$  and  $f_2$  respectively, the sequence  $p_{f_2} = \{f_1(T_i) | i = 1 \dots j\}$  produced with a particular  $f_2$  when  $f_1$  is kept fixed, is the profile of  $f_2$ , under  $f_1$ .*

The sequence of  $f_1$  values seen at each step yields lower bounds on  $C^*$ ,  
 the optimal cost, whose quality increases with steps (for a monotone heuristic  
 function). At termination, we get the cost  $C^*$ . Yet, one sequence may be  
 295 superior to another.

**PROPOSITION 1.** *Given AOB using an evaluation function  $f_1$ , there exist two tip node evaluation function  $f_2$  and  $f'_2$  such that the profiles  $p_{f_2}$  and  $p_{f'_2}$  under  $f_1$  differ.*

*Proof of Proposition 1.* We prove this by construction. **Figure 6** depicts a  
 300 graphical model defined over variables  $A, B, C$  having two functions  $f(A, B)$  and  
 $f(A, C)$ . The full AND/OR search space is shown explicitly and the optimal  
 solution ( $A = 1, B = 1, C = 0$ ) is marked in red. Assume that our heuristic  
 evaluation function is a constant 0. Assume two  $f_2$  evaluation functions:  $f_2$   
 which orders the subproblems by from left to right ( $B \prec C$ ), while  $f'_2$  reverses  
 305 the orderings from right to left. The profile of  $f_2$  under  $f_1$  is: (0,1,5,11), while  
 the profile of  $f'_2$  is: (0,6,11). In particular, with  $f_2$  the algorithm explores all

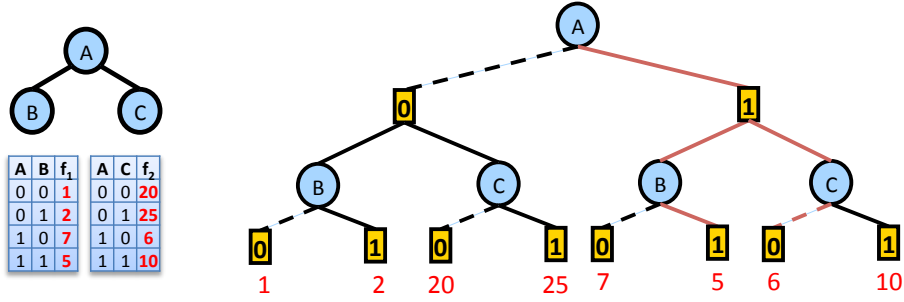


Figure 6: A simple graphical model over 3 variables with two functions. Shown above are the primal graph (also a valid pseudo-tree in this case), the function tables, and the associated AND/OR search space with weights labeled. The optimal solution tree is highlighted and has a cost of 11.

solution subtrees while with  $f'_2$ , it will never expand node  $B$  under  $A = 0$ , since expanding  $C$  proves that the  $A = 0$  branch has a cost of at least 20. We would never return to the  $A = 0$  branch since the  $A = 1$  branch never exceeds a cost of 20 at any point. In fact, it yields the optimal solution. Clearly, profile  $p_{f'_2}$  dominates that of  $p_{f_2}$  in this case.  $\square$

In the following we show further that the choice of  $f_2$  can, in the worst case, make an *exponential* impact on the number of expansions needed in order to cross a given lower bound threshold  $L$ .

**Theorem 2.** *Given a weighted AND/OR search graph, a heuristic evaluation function  $f_1$ , and two secondary  $f_2$  and  $f'_2$  functions, there exists an AND/OR search graph and a threshold  $L$  where the profile until reaching  $L$  for  $p_{f_2}$  is exponentially longer than for  $p_{f'_2}$ .*

*Proof.* Let  $T$  be a partial solution tree of an AND/OR search tree that is currently selected for extension. Let  $C = f_1(T)$  where  $C < C^*$ . Assume that  $T$  cannot be extended to an optimal solution, namely  $f_1^*(T) > C^*$ . Let  $A$  and  $B$  be two variables labeling OR tip nodes of  $T$  which are direct child nodes of an AND parent  $X = 0$ .

Let a subtree below a variable  $X$  be denoted as  $t_X$  and the  $d$ -depth truncated subtree be denoted as  $t_X^d$ . Assume that the best extension of  $T$  into  $t_B$  has  $f_1$  smaller than  $C^*$ . Furthermore, we want to force all of the nodes in  $t_B$  to be explored by AOBF in order to establish the optimal cost in  $t_B$ . This can be accomplished if the arc costs in  $t_B$  are monotonically increasing along a breadth-first ordering of the arcs in  $t_B$ . In contrast, assume that  $t_A^1$  ( $t_A$  truncated to depth 1), provides an extension to  $T$  having  $f_1 > C^*$ , namely  $f(T \cup t_A^1) \geq C^*$ .

Under these assumptions, an  $f'_2$  that prefers expanding all of  $t_B$  before any of  $t_A$  (and such exists) will yield a profile with more nodes than that of an  $f_2$  that expands  $t_A$  first. Since  $f(T \cup t_A^1) \geq C^*$ , it will never expand any of  $t_B$ . Therefore,  $p_{f'_2}$  would be exponentially longer than  $p_{f_2}$  for the threshold of  $C^*$ .  $\square$

#### 4. Local Bucket Errors for AOBF

The overall target for an effective subproblem ordering heuristic should be to select the subproblem which increases the lower bound the most. Assume a greedy scheme where given a frontier of tip nodes, we choose the node which would lead to the largest increase in  $f_1$  in a *single* expansion. We will show that this largest increase in  $f_1$  is the *residual*, and therefore we can use *look-ahead* as a guide to decide which subproblem to expand next. Therefore, the pre-computed (Algorithm 3) local bucket error functions which are identical to the residual can be used to order tip node expansion.

Clearly, this greedy scheme may be too greedy. For example, consider a  $t_X^d$  having  $d > 1$  and  $f_1(T \cup t_X^d) > C^*$ . To illustrate where the 1-level greedy scheme may fail, we consider the situation where  $f_1(T \cup t_X^1) = f_1(T)$ . Using a greedy  $f_2$ ,  $X$  would be ordered last because its depth-1 residual is zero, yet a greater increase in  $f_1$  may occur with a deeper look into the subproblem rooted by  $X$ .

The actual quantity of interest is the exact residual.

**Definition 17 (exact residual).** *The exact residual  $res^*(n)$  is defined as  $h^*(n) - h(n)$ .*

The exact residual is equal to a  $d$ -level residual when  $d$  is the depth of the search space below node  $n$ . Computing the exact residual  $res^*(n)$  is equivalent to full look-ahead. Since this is computationally too expensive, we propose to approximate this by a sum of depth-1 residuals over all the nodes in the look-ahead subtree. This can be accomplished by adding up all the local bucket errors of all variables in the subtree, a quantity we call *subtree error*. Two specific approximations for these quantities will be explored.

##### 4.1. A Constant Measure for Average Subtree Error

The first approach is to use the *average local bucket error* defined here:

**Definition 18 (average local error [11]).** *The average local error of  $B_k$  given a run of  $MBE(i)$  is*

$$\bar{E}_k = \frac{1}{|D_{S_{B_k}}|} \sum_{\bar{x}_k} E_k(\bar{x}_k) \quad (6)$$

It is the average of local bucket error values. Whenever the scope of the local bucket error function is too large, we can sample it, using the empirical average as an estimator. Computing this quantity is linear in the number of samples. It is easy to see that the empirical average is an unbiased estimate of the average local bucket error.

The average subtree errors, denoted  $\bar{E}_k^t$ , is the sum of the average local bucket errors over all the variables rooted by  $X_k$  in the pseudo-tree  $\mathcal{T}$ . In the following,  $desc(X_k)$  and  $ch(X_k)$  denote the descendants and children of  $X_k$  in  $\mathcal{T}$ , respectively. Namely,

**Definition 19 (average subtree error).** Let  $\bar{E}_k$  be the average local bucket error for  $X_k$ . The average subtree errors, denoted  $\bar{ET}_k$  is defined by:

$$\bar{ET}_k = \bar{E}_k + \sum_{j \in \text{desc}(X_k)} \bar{E}_j \quad (7)$$

and it can be expressed recursively:

$$\bar{ET}_k = \bar{E}_k + \sum_{c \in \text{ch}(X_k)} \bar{ET}_c \quad (8)$$

#### 4.2. A Functional Measure for Subtree Error

We next propose a more refined *function-based subtree error*. We will use the following elimination operator.

**Definition 20 (average elimination operator  $\Downarrow$ ).** Let  $f(\cdot)$  be a function with scope  $S_{f_j}$ , and  $S \subseteq S_{f_j}$ , and  $\mathbf{D}_S$  be the domains of the variables in  $S$ . The average elimination  $\Downarrow_S$  of  $f(\cdot)$  is defined by

$$\Downarrow_S f(\cdot) = \frac{1}{|\mathbf{D}_S|} \sum_S f(\cdot) \quad (9)$$

**Definition 21 (subtree error function).** Let  $E_k(\cdot)$  be the local bucket error function of  $X_k$ . The subtree error function relative to a pseudo-tree  $\mathcal{T}$  is

$$ET_k(\cdot) = E_k(\cdot) + \sum_{c \in \text{ch}(X_k)} \Downarrow_{(S_{E_c} - S_{E_k})} ET_c(\cdot) \quad (10)$$

375 A discounted version is defined, using discount factor  $\gamma_k \leq 1$ ,

$$ET_k(\cdot) = E_k(\cdot) + \gamma_k \cdot \sum_{c \in \text{ch}(X_k)} \Downarrow_{S_{E_c} - S_{E_k}} ET_c(\cdot) \quad (11)$$

We can view each element of summation over the children as a message computed by *averaging* out the variables which are not present in the parent's bucket. Each message computed at a node  $c$  can be interpreted as the expected error of the subproblem rooted by  $X_c$  as a function of its ancestor variables.  
 380 In the discounted version the contribution of errors from variables that are far away from the current variable are discounted.

**Algorithm 4**, which we call Bucket Error Propagation (BEP), generates the subtree error functions starting from the local bucket error functions as input. We denote by  $\lambda_{c \rightarrow k}^e$  as the message sent from child node  $c$  to  $k$ .

385 **PROPOSITION 2 (complexity of BEP).** The time and space complexity of algorithm BEP is  $O(nk^{psw(i)})$ .



---

**Algorithm 4: Bucket Error Propagation (BEP)**

---

**Input:** A Graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ , a pseudo-tree  $\mathcal{T}$ , local bucket error functions  $E_k(\cdot)$ , discount factors  $\gamma_k$

**Output:** Subtree error functions  $ET_k(\cdot)$

- 1 **Initialize**, for all leaves  $u$  of  $\mathcal{T}$ ,  $ET_u(\cdot) = E_u(\cdot)$
  - 2 *Compute bottom-up over  $\mathcal{T}$ , for each variable  $X_k$ :*
  - 3 
$$\left[ \begin{array}{l} ET_k(\cdot) = E_k(\cdot) + \gamma_k \cdot \sum_{c \in ch(X_k)} \lambda_{c \rightarrow k}^e(\cdot) \\ \lambda_{k \rightarrow pa(k)}^e(\cdot) = \Downarrow_{S_{E_k} - S_{E_{pa(k)}}} ET_k(\cdot) ; \end{array} \right. \quad // \text{ Compute message}$$
  - 4 **return**  $ET_k(\cdot)$  for each variable  $X_k \in \mathbf{X}$
- 

*Proof.* Computing the subtree error messages generated by a subtree error function  $ET_k(\cdot)$  requires enumeration over all the assignments of its scope, which is bounded by  $O(k^{psw(i)})$  time. Incorporating a message into a function  $ET_k(\cdot)$  from a child is also bounded by  $O(k^{psw(i)})$  time. Since we have  $n$  variables, the total time complexity is  $O(nk^{psw(i)})$ , The space complexity is also  $O(nk^{psw(i)})$ , since the scopes of  $ET_k(\cdot)$  are the same as the scopes of  $E_k(\cdot)$ .  $\square$

Thus, the complexity of BEP is dominated by the complexity of LBEE.

### 4.3. Approximating the Error Functions

The main advantage of using the average subtree error rather than the subtree error function explicitly is a matter of memory usage. Since the complexity of BEP is dominated by LBEE, BEP may not be feasible if the pseudo-width is large. Thus, we consider additional simplification to provide a finer level of control of this trade-off by modifying LBEE.

#### 4.3.1. Scope Bounding

We can bound further the complexity of the error measure by truncating the scopes of the local bucket error functions and aggregating over the eliminated variables.

**Definition 22 (scope-bounded bucket error function).** *Given the bucket error function  $E_k(\cdot)$ , the scope-bounded bucket error function, denoted  $EB_k(\cdot)$  relative to  $S \subseteq S_{E_k}$  is defined by*

$$EB_k(\cdot) = \Downarrow_{(S_{E_k} - S)} E_k(\cdot) \quad (12)$$

We can bound the scope size of  $S$  by an integer  $s$ , which we call the *s-bound*. Typically the  $s$ -bound is smaller than or equal to the  $i$ -bound of the MBE heuristic. We select the bounded scopes by eliminating variables from  $S_{E_k}$  that are closest to  $X_k$  in the pseudo-tree  $\mathcal{T}$ , until  $|S| \leq s$ .

Algorithm 5 presents the modified version of LBEE, called SB-LBEE, to account for the bounded bucket errors. It differs from LBEE (Algorithm 3) in having an  $s$ -bound parameter (line 4). If  $s = 0$ , the errors reduce to the *average local bucket error*.

---

**Algorithm 5: Scope-Bounded Local Bucket Error Evaluation (SB-LBEE)**


---

**Input:** A Graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ , a pseudo-tree  $\mathcal{T}$ ,  $i$ -bound,  $s$ -bound

**Output:** Scope-bounded error function  $EB_k(\cdot)$  for each variable  $X_k$

- 1 **Initialization:** Run  $MBE(i)$  for  $\mathcal{M}$  w.r.t.  $\mathcal{T}$
  - 2 **foreach**  $X_k \in \mathbf{X}$  **do**
  - 3     Let  $B_k = \cup_r B_k^r$  be the partition used by  $MBE(i)$  for  $X_k$
  - 4     Choose  $S$  s.t.  $S \subseteq S_{B_k}$ ,  $X_k \in S$ , and  $|S| \leq s + 1$
  - 5      $\mu_k^b(\cdot) = \Downarrow_{(S_{B_k}-S)} \sum_r (\min_{x_k} \sum_{f \in B_k^r} f(\cdot))$
  - 6      $\mu_k^{b^*}(\cdot) = \Downarrow_{(S_{B_k}-S)} \min_{x_k} \sum_{f \in B_k} f(\cdot)$
  - 7      $EB_k(\cdot) = \mu_k^{b^*}(\cdot) - \mu_k^b(\cdot)$
  - 8 **return**  $EB_k(\cdot)$  functions
- 

PROPOSITION 3 (**complexity of SB-LBEE**). *Given an  $s$ -bound, and an  $i$ -bound for  $MBE$ , the time complexity of SB-LBEE is  $O(nk^{psw(i)})$  and the space complexity is  $O(nk^s)$ , where  $n$  is the number of variables,  $k$  bounds the maximum domain size,  $psw(i)$  is the pseudo-width along  $\mathcal{T}$  relative to  $MBE(i)$ .*

*Proof.* The time complexity for each variable is dominated by computing the exact ( $s$ -bounded) bucket message  $\mu_k^{b^*}(\cdot)$ . Although the resulting scope is bounded by  $s$  using the  $\Downarrow$  operator, the min-sum computation over the bucket variables is still bounded by the scope size of  $B_k$  at the time of processing, which is the pseudo-width. Therefore, processing each bucket is bounded by  $O(k^{psw(i)})$  time, yielding a total time complexity of  $O(nk^{psw(i)})$ . For space complexity, the messages and error functions are bounded by  $s$  by design, thus for each variable, the algorithm needs  $O(k^s)$  space, yielding a total space complexity of  $O(nk^s)$ .  $\square$

Since the space complexity of BEP (Algorithm 4) depends on the maximum scope size of the error functions, which is the *pseudo-width*, we can replace the full local bucket error functions  $E_k(\cdot)$  with the scope-bounded versions computed by SB-LBEE to generate *s-bounded subtree error functions*.

#### 4.3.2. Sampling

Since the time complexity of SB-LBEE is bounded by  $O(nk^{psw(i)})$  primarily due to the computation of the exact bucket message  $\mu_k^{b^*}(\cdot)$  (Proposition 3), we estimate it to reduce the complexity. Given  $S \subseteq S_{B_k}$ , we can write the argument explicitly, yielding

$$\begin{aligned} \mu_k^{b^*}(\bar{x}_S) &= \Downarrow_{(S_{B_k}-S)} \min_{x_k} \sum_{f \in B_k} f(\bar{x}_S, \bar{x}_{S_{B_k}-S}) \\ &= \frac{1}{|\mathbf{D}_{S_{B_k}-S}|} \sum_{S_{B_k}-S} \left( \min_{x_k} \sum_{f \in B_k} f(\bar{x}_S, \bar{x}_{S_{B_k}-S}) \right) \end{aligned} \quad (13)$$

We now define the following estimator for  $\mu_k^b(\bar{x}_S)$  by sampling over  $\mathbf{D}_{S_{B_k}-S}$ .

$$\hat{\mu}_k^{b*}(\bar{x}_S) = \frac{1}{m} \sum_{i=1}^m \left( \min_{x_k} \sum_{f \in B_k} f(\bar{x}_S, \bar{x}_i) \right) \quad \bar{x}_i \sim U(\mathbf{D}_{S_{B_k}-S}) \quad (14)$$

where  $m$  is the number of samples and  $U(\mathbf{D}_{S_{B_k}-S})$  is a uniform distribution over its argument  $S_{B_k} - S$ , the set of variables we need to eliminate by averaging. It is easy to see that  $\hat{\mu}_k^{b*}(\bar{x}_S)$  is an unbiased estimator of  $\mu_k^{b*}(\bar{x}_S)$ . Since we sample  $m$  times for each of the  $k^s$  instantiations over  $n$  variables, the time complexity of this sampling procedure is  $O(nmk^s)$ , which carries over to SB-LBEE.

#### 4.4. Algorithm: Subtree Error Compilation

---

##### Algorithm 6: Subtree Error Compilation (SEC)

---

- Input:** A Graphical model  $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ , a pseudo-tree  $\mathcal{T}$ ,  $i$ -bound,  $s$ -bound, discount factors  $\gamma_k$
- Output:**  $s$ -bounded Subtree error functions  $EBT_k(\cdot)$
- 1 **Initialize:** Run MBE( $i$ ) for  $\mathcal{M}$  w.r.t. pseudo-tree  $\mathcal{T}$
  - 2 Generate  $s$ -bounded bucket error functions  $EB_k(\cdot)$  with SB-LBEE (possibly with sampling)
  - 3 Generate  $s$ -bounded subtree error functions  $EBT_k(\cdot)$  for each variable  $X_k \in \mathbf{X}$  with BEP using scope-bounded  $EB_k(\cdot)$  and discount factors  $\gamma_k$
  - 4 **return**  $EBT_k(\cdot)$
- 

We summarize the approach in Algorithm 6. Before search begins, we compile the subtree errors with SEC (Algorithm 6), which includes the standard procedure of compiling the MBE heuristic, then we subsequently use SB-LBEE to generate the scope-bounded bucket error functions. The algorithm then uses BEP (Algorithm 4) to generate the subtree error functions. Note that SB-LBEE reduces to computing average local bucket errors when  $s = 0$ . The subtree error functions are exactly the average subtree errors described in section 4.1, but with the discount factors applied.

During search, for any current assignment of the partial solution tree  $T$ , AOBF selects a tip node of  $T$  having the largest value according to its corresponding subtree error function. Namely it selects  $X_k$  by  $\operatorname{argmax}_{k \in \operatorname{tips}(T)} EBT_k(\bar{x}_T)$ , which defines the  $f_2$  ordering function. Since the error functions are compiled before search, evaluating  $ET_k$  is a lookup with constant complexity.

## 5. Experiments

We experimented with 4 variants of our error functions, formed by using either *constant* or *scope-bounded* errors (denoted **Const** and **ScpBnd**)

respectively), and *absolute* or *relative* errors (denoted **Abs** and **Rel** respectively). Thus, the 4 variants are named **Const-Abs (C-A)**, **Const-Rel (C-R)**, **ScpBnd-Abs (SB-A)**, and **ScpBnd-Rel (SB-R)**.

455 Using Algorithm 6, the **Const** variants set  $s = 0$ , while otherwise we use a default  $s = 10$  for the **ScpBnd** variants. (In practice, we found  $s = 10$  to give a reasonable balance between efficiency and overhead.) The absolute error **Abs** computes each error function value exactly as described in the previous section. We also consider the relative error **Rel** (see Definition ), which normalizes the  
 460 error values to express them as a percentage instead. In all variants, for each variable  $X_k$ ,  $\gamma_k = \frac{1}{|D_k|}$ . Intuitively, this is the branching factor  $X_k$  and thus we proportionally penalize the error for each assignment of  $X_k$ .

We compare against the baseline subproblem ordering that uses the heuristic evaluation function,  $f_1$  (denoted **Heur (H)** in the experiments). We will generally refer to the 4 variants as *error-guided orderings* and we break ties using  
 465 the baseline ordering. For all experiments, we use the mini-bucket elimination with moment-matching (MBE-MM) heuristic [22]. We used the MinFill variant of the iterative greedy variable ordering algorithm [23] to pre-compute elimination orderings used for pseudo-tree construction. In practice, strong heuristics  
 470 are preferred for search (i.e. using the highest possible  $i$ -bound given a memory bound). This is an easy decision for depth-first search algorithms that use linear memory. However, since best-first search uses exponential memory, the optimal trade-off is problem dependent and thus non-trivial. In our setup, we varied the  $i$ -bound in a way to focus on how the results depend on the levels  
 475 of heuristic error. For  $i$ -bounds less than 10, we set the  $s$ -bound to be equal to the  $i$ -bound. Whenever necessary, we used a maximum sample size of  $10^5$  shared across the entire scope-bounded error function. In preliminary experiments with computing bucket errors in our previous work [11], we found that  $10^5$  was a reasonable bound to keep the pre-processing time low relative to the  
 480 time spent with constructing MBE heuristics. In some cases, the number of possible assignments to a particular scope-bounded error function exceeds  $10^5$ . Since the sampling procedure requires at least one sample per assignment, we reduce  $s$  for these cases until this condition is satisfied. Given these settings, the memory used by the pre-compiled error functions were no greater than that  
 485 of the MBE heuristics (i.e. MBE dominates the space complexity for anything computed during pre-processing).

We experimented with benchmarks from genetic linkage analysis [24] (*pedigree*, *type4*), and medical diagnosis networks [25] (*promedas*). We included only instances that had a significant amount of search. In particular, having more  
 490 than  $10^5$  nodes expanded when using the baseline ordering with the lowest  $i$ -bound. Overall we report results on 13 pedigrees, 34 promedas networks, and 65 type4 instances, yielding a total of 112 problem instances. Based on the fixed variable orderings computed for each instance and their resulting induced width, the benchmarks represent a variety of problem difficulties ranging from  
 495 easy to hard and are presented in that order. Table 5 provides ranges of the various benchmark parameters.

Benchmark	# inst	$n$	$k$	$w^*$	$h$	$ F $	$a$
Pedigree	13	387	4	16	58	438	4
		1006	7	39	143	1273	5
Promedas	34	661	2	31	66	669	3
		1911	2	94	165	1928	3
Type4	65	3907	5	21	300	5749	4
		9838	5	68	1535	14765	4

Table 5: Benchmark statistics for. # inst - number of instances,  $n$  - number of variables,  $k$  - maximum domain size,  $w^*$  - induced width,  $h$  - pseudotree height,  $|F|$  - number of functions,  $a$  - maximum arity. The top value is the minimum and the bottom value is the maximum for that statistic.

The implementation is in C++ (64-bit) and was executed on a 2.66GHz processor with 24GB of RAM, which was shared between AOBF and MBE-MM.

### 5.1. Evaluating for Exact Solutions

Tables 6, 8, and 10 report the pre-processing time for compiling the MBE heuristic and subtree error functions (in seconds), total CPU time including pre-processing (in seconds), and number of OR nodes (in thousands of nodes) expanded for a subset of the problems which could be solved exactly by AOBF over the benchmarks. For each instance we also mention the problem’s parameters such as the number of variables ( $n$ ), maximum domain size ( $k$ ), induced width ( $w^*$ ), and pseudo-tree height ( $h$ ). Each column is indexed by the  $i$ -bound of the MBE-MM. Each row for each instance shows the various subproblem ordering schemes we experimented with. Additionally, we summarize over all of the problems that could be solved, by  $i$ -bound, for each benchmark in Tables 7, 9, and 11.

We aim to assess whether an ordering strategy had a positive impact in terms of the number of nodes expanded and whether the impact is cost-effective time-wise.

**Pedigree.** Table 6 shows the results for four different  $i$ -bounds for selected instances from the **pedigree** benchmark. At an  $i$ -bound of 6, we see that for 2 of the instances (*pedigree9* and *pedigree33*) AOBF runs out of memory when using the baseline ordering. For *pedigree20*, we see when using the **Const-Rel** heuristic, the number of nodes expanded is about half of that of the baseline. Still, the baseline can be better (e.g. see *pedigree39*). Moving to higher  $i$ -bounds, we see that for instances other than *pedigree39*, the error-guided orderings usually result in fewer nodes expanded at termination which usually translates to improved runtime, except at the highest  $i$ -bounds. This is due to the small difference in the number of nodes expanded relative to the extra pre-processing time. For example, on *pedigree9* with  $i=18$ , although **Const-Abs** expands 60K fewer nodes, pre-processing took 2 seconds longer compared with the baseline, resulting in a total time that is 1 second worse.

instance ( $n, k, w^*, h$ )	heur	$i = 6$		$i = 10$		$i = 14$		$i = 18$	
		pre	time / nodes	pre	time / nodes	pre	time / nodes	pre	time / nodes
<b>pedigree9</b> (935,7,25,137)	H	0	oom / -	0	154 / 4137	1	14 / 512	6	<u>10</u> / 193
	C-A	0	602 / 6368	1	73 / <b>1741</b>	3	12 / <b>350</b>	8	11 / <b>133</b>
	C-R	0	673 / 6361	1	<b>72</b> / 1751	3	11 / 350	8	11 / 134
	SB-A	0	oom / -	1	448 / 9241	3	18 / 617	8	13 / 216
	SB-R	0	<b>588</b> / <b>6355</b>	1	119 / 3034	2	<b>10</b> / 350	8	11 / <b>133</b>
<b>pedigree20</b> (387,5,21,58)	H	0	85 / 2678	0	28 / 1019	0	16 / 595	6	<u>6</u> / 5
	C-A	0	78 / 1933	0	<b>24</b> / <b>812</b>	2	16 / <b>503</b>	7	7 / 5
	C-R	0	58 / <b>1523</b>	1	28 / 833	1	15 / 505	7	7 / 5
	SB-A	0	232 / 5521	-	- / err	1	22 / 831	7	7 / 4
	SB-R	0	<b>57</b> / 1579	0	24 / 818	1	<b>15</b> / 512	7	7 / <b>4</b>
<b>pedigree33</b> (581,4,24,116)	H	0	oom / -	0	11 / 448	1	<u>4</u> / 163	6	<u>6</u> / 28
	C-A	0	576 / 10568	1	13 / 392	4	8 / 160	8	9 / 26
	C-R	0	503 / <b>10498</b>	1	13 / 434	4	7 / 156	8	9 / 26
	SB-A	0	<b>452</b> / 10527	1	<b>10</b> / <b>380</b>	3	6 / <b>145</b>	8	8 / 26
	SB-R	0	514 / 10513	1	13 / 415	3	7 / 159	8	8 / <b>26</b>
<b>pedigree39</b> (953,5,20,82)	H	0	<b>146</b> / <b>3869</b>	0	<b>11</b> / <b>490</b>	0	<u>0</u> / 3	6	<u>6</u> / 1
	C-A	0	230 / 4964	1	17 / 604	1	1 / 2	6	7 / 1
	C-R	0	223 / 4921	0	15 / 604	1	1 / 2	6	7 / 1
	SB-A	0	217 / 4974	0	15 / 605	1	1 / 2	6	6 / 1
	SB-R	0	220 / 4920	0	15 / 593	1	1 / 1	6	6 / <b>1</b>
<b>pedigree41</b> (885,5,33,100)	H	0	oom / -	0	oom / -	2	oom / -	34	180 / 4995
	C-A	0	oom / -	3	oom / -	6	<b>487</b> / <b>13188</b>	39	<b>166</b> / <b>3942</b>
	C-R	0	oom / -	3	oom / -	6	oom / -	39	168 / 3942
	SB-A	0	oom / -	2	oom / -	5	oom / -	38	oom / -
	SB-R	0	oom / -	2	oom / -	5	oom / -	38	167 / 3967
<b>pedigree42</b> (390,5,21,67)	H	0	23 / 935	0	28 / 1113	12	<b>15</b> / 130	170	<b>170</b> / 20
	C-A	0	20 / <b>671</b>	2	<b>22</b> / 780	14	16 / 83	171	171 / 14
	C-R	0	<b>20</b> / 679	2	22 / <b>780</b>	14	16 / 123	173	174 / 14
	SB-A	0	40 / 1109	2	30 / 986	13	15 / <b>83</b>	171	171 / <b>12</b>
	SB-R	0	20 / 675	2	29 / 1053	13	16 / 102	171	171 / 15
<b>pedigree44</b> (644,4,24,79)	H	0	oom / -	0	664 / 12209	1	36 / 1254	4	<u>9</u> / 223
	C-A	0	oom / -	1	388 / 9683	3	36 / 1230	6	11 / 221
	C-R	0	oom / -	1	412 / 9783	3	<b>35</b> / <b>1138</b>	6	11 / 221
	SB-A	0	oom / -	1	<b>336</b> / <b>9117</b>	3	<b>35</b> / 1154	6	10 / <b>213</b>
	SB-R	0	oom / -	1	497 / 11041	2	36 / 1175	6	11 / 213

Table 6: Exact evaluation for **pedigree** instances. The best times and node counts are **bolded** per  $i$ -bound and the best times and node count overall for a given instance are also **underlined**. If the optimal solution was not found, then we report ‘oom’ to denote that the experiment ran out of memory. We also report the pre-processing time to the left of each total time.  $n$  - number of variables,  $k$  - maximum domain size,  $w^*$  - induced width,  $h$  - pseudo-tree height

Win counts for <b>pedigree</b> instances				
heuristic	$i = 6$ (solved=6)	$i = 10$ (solved=7)	$i = 14$ (solved=8)	$i = 18$ (solved=11)
	wins by time(%) / nodes(%)	wins by time(%) / nodes(%)	wins by time(%) / nodes(%)	wins by time(%) / nodes(%)
C-A	4 (66.7) / 4 (66.7)	4 (57.1) / 5 (71.4)	4 (50.0) / 8 (100.0)	2 (18.2) / 7 (63.6)
C-R	4 (66.7) / 4 (66.7)	4 (57.1) / 5 (71.4)	3 (37.5) / 7 (87.5)	3 (27.3) / 7 (63.6)
SB-A	1 (16.7) / 1 (16.7)	2 (28.6) / 3 (42.9)	1 (12.5) / 4 (50.0)	0 (0.0) / 4 (36.4)
SB-R	4 (66.7) / 5 (83.3)	3 (42.9) / 5 (71.4)	3 (37.5) / 7 (87.5)	2 (18.2) / 7 (63.6)

Table 7: Summary for exact solutions on **pedigree** instances: win counts of instances that the ordering heuristic had a lower time or lower number of nodes than the baseline. The number of instances solved by any heuristic (including the baseline) is shown under each  $i$ -bound label. Each number in parentheses is the percentage of solved instances that performed better for that particular  $i$ -bound.

Table 7 shows the win counts for time and nodes expanded for each of the error-guided orderings relative to the baseline. Examining the number of wins based on nodes expanded, more than half of the instances that could be solved benefited from error-guided ordering (excluding **ScpBnd-Abs**). For example, at an  $i$ -bound of 10, every method except ScpBnd-Abs has fewer nodes expanded than the baseline on 71.4% of the solved instances. At the lower  $i$ -bounds of 6 and 10, we see that the savings in the number of nodes translates to savings in time as well in most cases. However, this decreases as the  $i$ -bound increases, due a combination of the pre-processing overhead of the computing the error-based heuristics and the relative ease of the benchmark problems using stronger heuristics. Overall, the error-guided orderings demonstrate their moderate impact on a variety of  $i$ -bounds here, but their overhead prevents them from being useful in situations where problems are already easy to solve with the correct  $f_1$  heuristic and the baseline orderings.

**Promedas.** Table 8 reports on a selection of **promedas** instances. We used higher  $i$ -bounds for this benchmark because the instances are harder compared to the **pedigrees**. As in the previous benchmark, we observe that the number of nodes expanded by an error-guided ordering is better than the baseline ordering in many cases. Notably, we see here for the high  $i$ -bound of 18, where the savings in nodes on a few harder instances translated well to savings in the runtime. For example, on *or\_chain\_25.fg*, both the runtime and number of nodes expanded using any of the error-guided orderings were about half of those of the baseline ordering. Still, the error-guided orderings may still be worse than the baseline (e.g. *or\_chain\_140.fg*, but this is usually the exception).

Table 9 provides the win counts for each error-guided ordering heuristic. The **ScpBnd-Abs** heuristic is the worst performer, as we saw before. Both relative error base orderings (**Const-Rel** and **ScpBnd-Rel**) have similar positive performance, demonstrating positive impact of the orderings on at least 50% of the instances across all  $i$ -bounds. Overall, **Const-Abs** was best, yielding improvements on at least 70% of the instances. The time savings in nodes carries over to most cases using lower  $i$ -bounds and slightly fewer at higher  $i$ -bounds. For example, at an  $i$ -bound of 12 using **Const-Abs**, 9 out of the 10 instances (90%) had a positive impact of ordering also had better runtime than the baseline. In contrast, at an  $i$ -bound of 18, only 12 out of the 16 instances (75%) had improved times.

**Type4.** Table 10 shows all the instances of the **type4** benchmark that could be solved by AOBF with the given  $i$ -bounds. This benchmark is the hardest of the 3 benchmarks and thus we used even higher  $i$ -bounds compared to the **promedas** benchmark. Even then, only with an  $i$ -bound of 20 are we able to solve all 6 of the instances. Like with the previous two benchmarks, we see that the error-guided orderings yield better performance on many instances. However, the variation is smaller. For example, on *type4-haplo\_100\_19* using an  $i$ -bound of 20, the number of nodes expanded using the **Const-Abs** ordering is still about 87% of what the baseline ordering yields, compared with the lower 50% rates seen on some instances in the other two benchmarks. On two of the instances here

instance ( $n, k, w^*, h$ )	heur	$i = 12$		$i = 14$		$i = 16$		$i = 18$	
		pre	time / nodes	pre	time / nodes	pre	time / nodes	pre	time / nodes
or_chain_25.fg (1075,2,43,80)	H	0	oom / -	0	239 / 10020	0	164 / 7009	1	319 / 13152
	C-A	2	oom / -	3	172 / 6686	4	<b>115</b> / <b>4629</b>	4	161 / 6225
	C-R	2	oom / -	3	175 / <b>6668</b>	4	117 / 4671	4	<b>159</b> / 6226
	SB-A	2	oom / -	2	oom / -	3	117 / 4678	4	163 / 6225
	SB-R	2	oom / -	2	<b>170</b> / 6670	3	115 / 4657	4	161 / <b>6224</b>
or_chain_40.fg (988,2,43,87)	H	0	oom / -	0	oom / -	1	195 / 8642	2	109 / 5084
	C-A	2	oom / -	3	oom / -	5	168 / 7039	6	60 / <b>2550</b>
	C-R	2	oom / -	3	oom / -	5	<b>144</b> / <b>5721</b>	7	<b>60</b> / 2550
	SB-A	2	oom / -	3	oom / -	4	302 / 12748	5	231 / 9579
	SB-R	2	oom / -	3	oom / -	4	152 / 6151	6	61 / 2550
or_chain_63.fg (731,2,38,81)	H	0	42 / 2036	0	11 / 563	0	<b>7</b> / 349	1	9 / 411
	C-A	1	<b>31</b> / <b>1391</b>	2	10 / 350	3	8 / <b>226</b>	4	<b>8</b> / 251
	C-R	1	32 / 1391	2	10 / <b>350</b>	3	8 / 244	4	9 / <b>251</b>
	SB-A	1	52 / 2266	2	15 / 608	3	10 / 382	4	12 / 467
	SB-R	1	44 / 2038	2	<b>10</b> / 361	3	7 / 230	4	9 / 273
or_chain_80.fg (840,2,50,108)	H	0	128 / 5374	0	80 / 3398	1	47 / 1977	2	46 / 1904
	C-A	2	112 / 4550	4	<b>75</b> / 3019	5	47 / 1724	7	42 / <b>1443</b>
	C-R	2	<b>99</b> / <b>4550</b>	4	76 / 3029	5	47 / 1738	7	41 / 1443
	SB-A	2	248 / 9001	3	174 / 6222	5	83 / 3108	6	84 / 3127
	SB-R	3	114 / 4551	3	76 / <b>3018</b>	4	<b>46</b> / <b>1724</b>	6	<b>40</b> / 1444
or_chain_94.fg (762,2,32,97)	H	0	74 / 3659	0	58 / 2945	1	45 / 2498	1	28 / 1582
	C-A	1	39 / 1690	3	32 / 1416	4	28 / 1318	4	21 / 937
	C-R	1	<b>35</b> / 1690	3	32 / 1416	4	28 / <b>1298</b>	4	21 / 937
	SB-A	1	370 / 12931	2	206 / 8068	3	114 / 5048	4	59 / 2787
	SB-R	1	37 / <b>1624</b>	2	<b>30</b> / <b>1357</b>	3	<b>27</b> / 1307	4	<b>21</b> / <b>931</b>
or_chain_140.fg (1260,2,32,79)	H	0	<b>108</b> / 5103	0	79 / 3946	1	<b>41</b> / 2224	1	<b>32</b> / <b>1714</b>
	C-A	2	121 / <b>4943</b>	3	66 / <b>2913</b>	4	51 / 2290	4	55 / 2448
	C-R	2	133 / 5633	3	66 / 2934	4	49 / 2288	4	69 / 3328
	SB-A	2	132 / 4979	2	70 / 2962	3	43 / <b>2008</b>	4	51 / 2240
	SB-R	2	132 / 5442	2	<b>65</b> / 2944	3	47 / 2119	4	67 / 3220
or_chain_178.fg (1012,2,35,97)	H	0	oom / -	0	284 / 13893	1	253 / 14811	1	231 / 11752
	C-A	2	265 / 11758	3	134 / <b>6389</b>	5	<b>145</b> / <b>6958</b>	5	114 / 5339
	C-R	2	<b>264</b> / <b>11724</b>	3	<b>133</b> / 6390	5	254 / 12591	6	<b>111</b> / <b>5339</b>
	SB-A	2	oom / -	3	oom / -	4	oom / -	5	oom / -
	SB-R	2	300 / 12810	3	157 / 7303	4	278 / 13411	5	113 / 5461
or_chain_199.fg (917,2,33,79)	H	0	33 / 1677	0	<b>42</b> / 2259	0	33 / 1850	1	<b>20</b> / 1098
	C-A	1	<b>29</b> / <b>1332</b>	2	43 / <b>2094</b>	3	<b>30</b> / <b>1496</b>	4	20 / 971
	C-R	1	39 / 1708	2	71 / 2921	3	30 / 1498	4	20 / <b>971</b>
	SB-A	1	93 / 3330	2	79 / 3048	2	70 / 3050	3	34 / 1527
	SB-R	1	39 / 1693	2	58 / 2596	2	38 / 1789	3	22 / 1065
or_chain_212.fg (773,2,33,79)	H	0	103 / 5031	0	60 / 3110	0	43 / 2382	1	17 / 911
	C-A	1	55 / <b>2476</b>	2	<b>33</b> / <b>1569</b>	3	<b>26</b> / 1165	4	14 / <b>549</b>
	C-R	1	<b>55</b> / 2476	2	33 / 1570	3	26 / <b>1164</b>	4	14 / 549
	SB-A	1	oom / -	2	282 / 12138	3	215 / 9986	3	86 / 4305
	SB-R	1	66 / 2830	2	35 / 1670	3	31 / 1399	3	<b>14</b> / 566
or_chain_226.fg (735,2,42,87)	H	0	197 / 8434	0	212 / 9580	1	75 / 3443	2	34 / 1528
	C-A	2	156 / 5322	4	<b>90</b> / <b>3330</b>	5	<b>45</b> / 1611	6	28 / <b>921</b>
	C-R	2	<b>154</b> / <b>5320</b>	4	104 / 3905	5	45 / <b>1611</b>	5	26 / 976
	SB-A	2	291 / 11595	3	346 / 15606	4	117 / 4854	5	62 / 2525
	SB-R	2	159 / 5482	3	98 / 3663	4	46 / 1697	4	<b>25</b> / 976

Table 8: Exact evaluation for **promedas** instances. The best times and node counts are **bolded** per  $i$ -bound and the best times and node count overall for a given instance are also **underlined**. If the optimal solution was not found, then we report ‘oom’ to denote that the experiment ran out of memory. We also report the pre-processing time to the left of each total time.  $n$  - number of variables,  $k$  - maximum domain size,  $w^*$  - induced width,  $h$  - pseudo-tree height

Win counts for <b>promedas</b> instances								
heuristic	$i = 12$		$i = 14$		$i = 16$		$i = 18$	
	wins by (solved=14)		wins by (solved=17)		wins by (solved=19)		wins by (solved=20)	
	time(%)	nodes(%)	time(%)	nodes(%)	time(%)	nodes(%)	time(%)	nodes(%)
C-A	10 (71.4)	10 (71.4)	11 (64.7)	12 (70.6)	11 (57.9)	14 (73.7)	12 (60.0)	16 (80.0)
C-R	8 (57.1)	8 (57.1)	10 (58.8)	12 (70.6)	10 (52.6)	13 (68.4)	10 (50.0)	14 (70.0)
SB-A	2 (14.3)	4 (28.6)	2 (11.8)	3 (17.6)	3 (15.8)	4 (21.1)	2 (10.0)	2 (10.0)
SB-R	7 (50.0)	7 (50.0)	10 (58.8)	11 (64.7)	9 (47.4)	15 (78.9)	10 (50.0)	14 (70.0)

Table 9: Summary for exact solutions on **promedas** instances: win counts of instances that the ordering heuristic had a lower time or lower number of nodes than the baseline. The number of instances solved by any heuristic (including the baseline) is shown under each  $i$ -bound label. Each number in parentheses is the percentage of solved instances that performed better for that particular  $i$ -bound.



instance ( $n, k, w^*, h$ )	heur	$i = 14$			$i = 16$			$i = 18$			$i = 20$		
		pre	time	nodes	pre	time	nodes	pre	time	nodes	pre	time	nodes
<b>type4-haplo_100_19</b> (3927,5,28,362)	H	4	oom	-	9	oom	-	23	2436	11898	61	208	1415
	C-A	16	oom	-	24	oom	-	36	<b>1965</b>	<b>10036</b>	70	176	<b>1231</b>
	C-R	16	oom	-	25	oom	-	35	2024	10036	70	177	1231
	SB-A	14	oom	-	21	oom	-	33	2953	13625	68	222	1654
	SB-R	14	oom	-	21	oom	-	33	2039	10343	68	<b>175</b>	1231
<b>type4-haplo_120_17</b> (4302,5,23,300)	H	3	oom	-	7	568	1846	17	32	162	32	34	10
	C-A	12	oom	-	16	325	<b>1823</b>	21	29	155	34	34	8
	C-R	12	oom	-	14	311	1825	21	29	<b>155</b>	34	34	8
	SB-A	10	oom	-	14	<b>211</b>	1880	20	<b>25</b>	155	33	<b>34</b>	<b>8</b>
	SB-R	10	oom	-	14	338	1828	20	28	155	33	34	8
<b>type4-haplo_170_23</b> (6933,5,21,396)	H	2	<b>1390</b>	<b>5299</b>	3	41	<b>300</b>	8	12	21	13	16	6
	C-A	9	3239	5820	11	31	359	11	12	16	14	14	6
	C-R	9	3323	5821	11	32	385	11	12	16	14	14	6
	SB-A	7	2046	6675	9	<b>22</b>	398	10	11	22	14	14	6
	SB-R	8	2067	6484	10	31	383	10	<b>11</b>	<b>16</b>	14	14	<b>6</b>
<b>type4b_100_19</b> (3938,5,29,354)	H	5	oom	-	11	oom	-	30	oom	-	101	880	7757
	C-A	18	oom	-	26	oom	-	43	oom	-	111	832	7664
	C-R	18	oom	-	26	oom	-	43	oom	-	111	678	<b>7588</b>
	SB-A	16	oom	-	22	oom	-	35	oom	-	109	973	7891
	SB-R	16	oom	-	22	oom	-	40	oom	-	109	<b>634</b>	7598
<b>type4b_120_17</b> (4072,5,24,319)	H	3	oom	-	8	508	1956	22	<b>24</b>	119	38	<b>39</b>	48
	C-A	11	oom	-	16	336	1922	27	31	104	41	42	35
	C-R	11	oom	-	14	<b>307</b>	1919	27	31	104	41	42	<b>35</b>
	SB-A	10	oom	-	14	364	1937	26	29	<b>103</b>	40	41	38
	SB-R	10	oom	-	14	329	<b>1919</b>	26	30	105	40	41	37
<b>type4b_170_23</b> (5590,5,21,427)	H	3	382	2968	4	<b>5</b>	37	7	<b>7</b>	20	9	<b>9</b>	5
	C-A	8	287	2879	9	10	36	8	9	6	9	10	5
	C-R	8	333	2809	9	10	35	8	8	6	9	9	5
	SB-A	7	<b>176</b>	<b>2630</b>	8	9	53	8	8	<b>6</b>	9	10	5
	SB-R	7	281	2736	8	9	<b>34</b>	8	8	6	9	10	<b>5</b>

Table 10: Exact evaluation for **type4** instances. The best times and node counts are also **bolded** per  $i$ -bound and the best times and node count overall for a given instance are also **underlined**. If the optimal solution was not found, then we report ‘oom’ to denote that the experiment ran out of memory. We also report the pre-processing time to the left of each total time.  $n$  - number of variables,  $k$  - maximum domain size,  $w^*$  - induced width,  $h$  - pseudo-tree height

Win counts for <b>type4</b> instances				
heuristic	$i = 14$ (solved=2)	$i = 16$ (solved=4)	$i = 18$ (solved=5)	$i = 20$ (solved=6)
	wins by time(%) / nodes(%)	wins by time(%) / nodes(%)	wins by time(%) / nodes(%)	wins by time(%) / nodes(%)
C-A	1 (50.0) / 1 (50.0)	3 (75.0) / 3 (75.0)	3 (60.0) / 5 (100.0)	3 (50.0) / 4 (66.7)
C-R	1 (50.0) / 1 (50.0)	3 (75.0) / 3 (75.0)	3 (60.0) / 5 (100.0)	3 (50.0) / 4 (66.7)
SB-A	1 (50.0) / 1 (50.0)	3 (75.0) / 1 (25.0)	2 (40.0) / 3 (60.0)	2 (33.3) / 2 (33.3)
SB-R	1 (50.0) / 1 (50.0)	3 (75.0) / 3 (75.0)	3 (60.0) / 5 (100.0)	3 (50.0) / 4 (66.7)

Table 11: Summary for exact solutions on **type4** instances: win counts of instances that the ordering heuristic had a lower time or lower number of nodes than the baseline. The number of instances solved by any heuristic (including the baseline) is shown under each  $i$ -bound label. Each number in parentheses is the percentage of solved instances that performed better for that particular  $i$ -bound.

(*type4-haplo\_170\_23* and *type4b\_170\_23*), there is no difference in the number of nodes expanded, indicating that most of the errors likely evaluated to zero, thus falling back on the baseline ordering. Still, we see improved runtime on the hardest of the instances here (*type4-haplo\_100\_19* and *type4b\_100\_19*), where any savings in the number of nodes expanded did carry over to overall savings in runtime.

Table 11 aggregates the results by win counts as before. Across the  $i$ -bounds, every error-guided ordering was better on at 50-100% of the instances except for **ScpBnd-Abs**. Notably, improvement was achieved on all instances by these 3 orderings at an  $i$ -bound of 18 and a majority at the highest  $i$ -bound of 20. In terms of overall runtime improvement, most instances also had better runtimes when their orderings were better in terms of node expansions.

### 5.1.1. Discussion on Finding Exact Solutions

Our experiments, show that the *error-guided orderings can benefit AOBF*. For most combinations of benchmark and  $i$ -bound, at least 60-70% of the solved instances had positive impact when using the error-guided orderings in terms of nodes expanded, despite the multiple levels of approximation performed (summing 1-level residuals to approximate the exact residual, bounding the scopes of the error functions, and sampling). Furthermore on the harder benchmarks (**promedas** and **type4**), the superior orderings node-wise usually also carried over to improved overall runtime. Specifically, when considering time on the same benchmarks, the percentage of instances that exhibited positive impact around 50-60%, though in some cases, the error-guided orderings were only a bit slower, since the difference in pre-processing time is usually in the order of a few seconds. For easy instances, the impact was negative in terms of the number of nodes (and obviously time-wise). This is partly because on easy instances, the MBE-MM heuristic is strong and thus there are no errors, thus making the error-guided orderings not cost-effective. Overall, all of the error-guided orderings (except **ScpBnd-Abs**) have similar performance to each other.

## 5.2. Anytime Lower Bounding

Next, we will evaluate AOBF for generating lower bounds in an anytime fashion and the impact of the  $f_2$  ordering heuristic on this anytime performance. Figures 7, 9, and 11 report the lower bound obtained as a function of time for each subproblem ordering  $f_2$ . A profile that is higher earlier in time is superior. The first point of each line is always the bound returned by the MBE-MM heuristic itself, recorded whenever search starts following all pre-processing. If known, the exact solution is also plotted as a dashed gray line. For each benchmark, we select 2 representative instances, one of which was exactly solved and another which was not. For each instance, we show 4 different  $i$ -bounds. For pedigrees both instances are exactly solved (Figure 7) Each ordering is labeled with abbreviated names for clarity.

We also aggregated the results per benchmark in Figures 8, 10, and 12. We normalized the time scale for each instance to that of the baseline, ranked the

bounds yielded by each variant across time, and aggregated across the instances by averaging. The number of instances varies with the different  $i$ -bounds since for some large instances, compiling the MBE heuristics at the highest  $i$ -bounds exceeds our memory limit of 24GB.

620 **Pedigrees.** Figure 7 shows the lower bounds as a function of time on 2 selected instances from the **pedigree** benchmark. First, on *pedigree9* (which is solved exactly). For the lower  $i$ -bounds of 6 and 10, all of the methods except **ScpBnd-Abs** perform better early on. Since the problem is easy at higher  $i$ -bounds, AOBF quickly finds the optimal solution after the initial bound generated by  
625 MBE-MM. Still, at an  $i$ -bound of 14, everything except **ScpBnd-Abs** improves over the baseline. At the highest  $i$ -bound of 18, the pre-processing overhead makes the error-guided orderings not cost-effective. In *pedigree51*, where AOBF ran out of memory before finding the exact solution, the lowest  $i$ -bounds of 6 and 10 also benefited from error-guided orderings. Increasing to the  $i$ -bound  
630 14 yields marginally better performance compared with the baseline (except for **ScpBnd-Abs**). Finally, at the highest  $i$ -bound the baseline performs best.

Figure 8 presents the average ranks for each ordering heuristic based on normalizing the time across the instances and averaging as explained earlier. As seen in the instance-by-instance results, all error-guided orderings except  
635 **ScpBnd-Abs** outperform the baseline. As the  $i$ -bound increases the average rank of the baseline improves. For an  $i$ -bound of 18, only **ScpBnd-Rel** ranks similarly to the baseline, but the baseline is better overall.

**Promedas.** Figure 9 shows results on 2 selected instances from the **promedas** benchmark. For the first instance (*or-chain-178.fg*), the error-guided orderings  
640 improve significantly over the baseline, except for **ScpBnd-Abs** across all  $i$ -bounds. Next, on *or-chain-108.fg*, most of the error-guided orderings improve over the baseline at an  $i$ -bound of 12. Once we increase the  $i$ -bound to 14, the baseline manages to get a profile that is more similar to the 3 dominating methods, but still falls short. At  $i$ -bounds of 16 and 18, all methods that were  
645 performing well before show a better profile, generating a higher lower bound early, as expected

Figure 10 presents the ranking summary over this benchmark. For all  $i$ -bounds the baseline seems superior early on due to the pre-processing overhead of the error-guided orderings. However, it is overtaken by the other methods  
650 eventually at different points on the normalized time scale. At an  $i$ -bound of 12, the **Const** methods outrank the baseline early on. Moving to  $i$ -bounds of 14 and 16, everything but **ScpBnd-Abs** approaches the baseline eventually and outrank it with time. Finally, at the highest  $i$ -bound of 18, the **ScpBnd-Rel** method performs better early around 0.1 on the time scale, with the **Const**  
655 methods overtaking the baseline at around 0.6 on the time scale.

**Type4.** Figure 11 shows lower bounds over time for 2 instances of our last benchmark. The first instance (*t4-haplo-100-19*) could not be solved at  $i$ -bounds of 14 and 16, but were solved with higher  $i$ -bounds. Once again, all of the error-guided heuristics except **ScpBnd-Abs** improve performance over all  $i$ -bounds

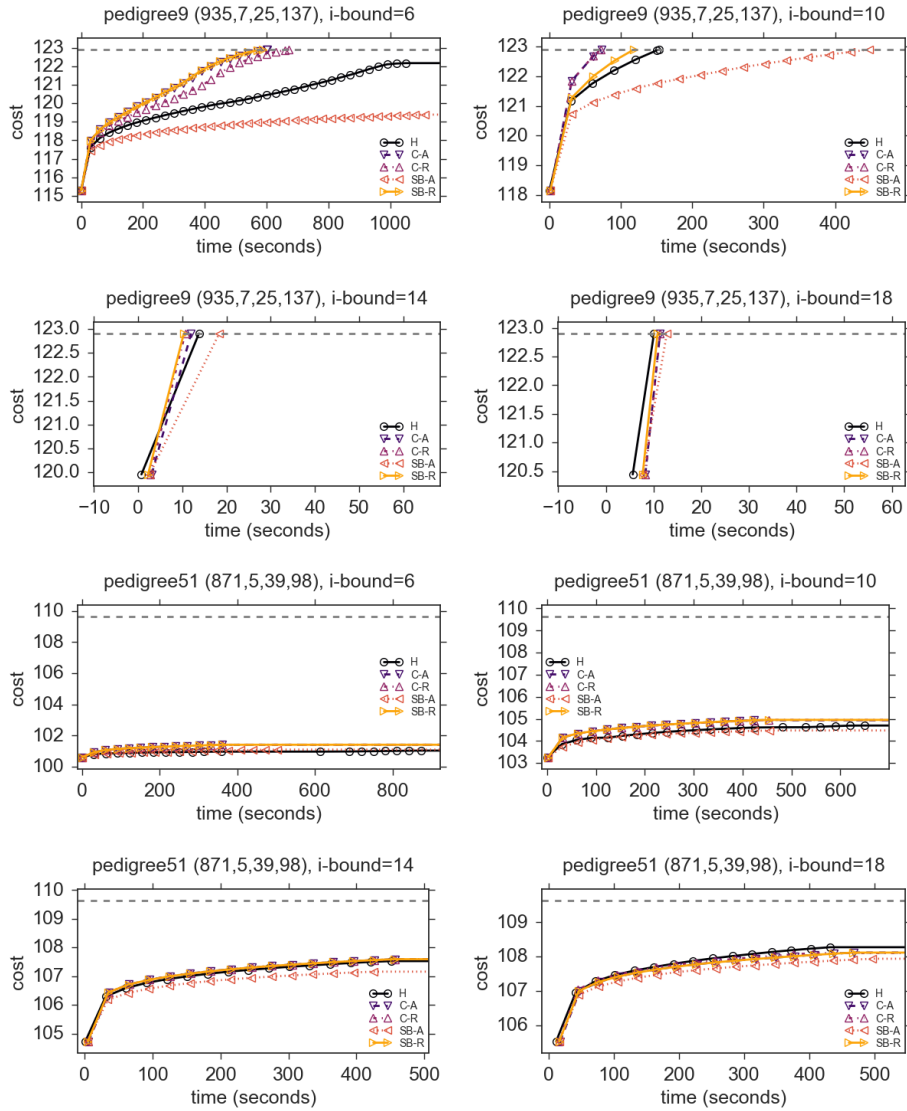


Figure 7: Lower bounds as a function of time for two instances from the **pedigree** benchmark. Higher is better.

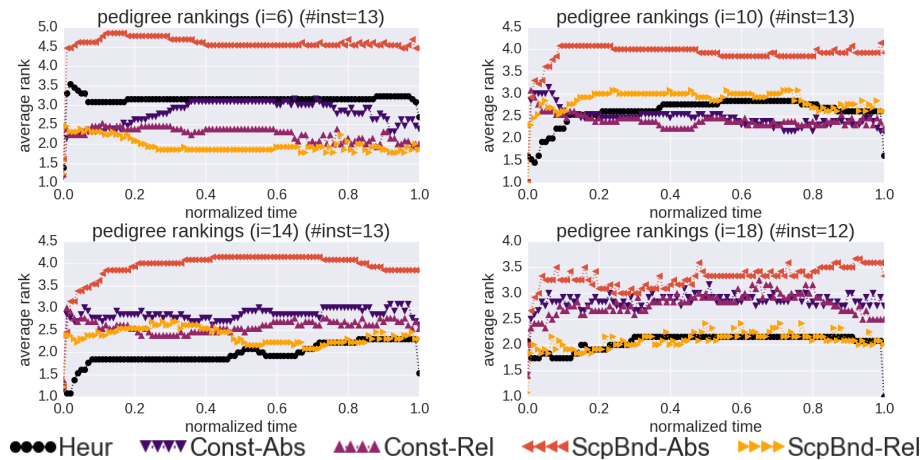


Figure 8: Average rank of each ordering as a function of normalized time across all of the instances in the **pedigree** benchmark. Lower is better.

660 of 16 and up. At the lowest  $i$ -bound of 14, the performance is close to the baseline. Moving to *t4-haplo-190-20* which was not solved, we have similar behavior. Specifically, **ScpBnd-Abs** is slightly better than the rest at  $i$ -bounds up to 18, but performs significantly worse than both **Rel** methods at the highest  $i$ -bound of 20.

665 Figure 12 provides summary rankings of the orderings over normalized time for all of the instances that did not run out of memory for each  $i$ -bound. Notably, the number of instances reported decreases significantly as we increase the  $i$ -bound because we were unable to compile the MBE heuristics given our memory bound for many instances. At an  $i$ -bound of 14, the baseline performs best.  
 670 Many of the instances had a large number of variables, which substantially increased the time and memory needed to compile the MBE heuristics. In turn, this extra cost carried over to increasing the pre-processing time for the error-guided ordering heuristics, leading to a negative impact. For higher  $i$ -  
 675 bounds, we observe the same behavior as in the previous benchmark, where the baseline ordering is best initially. However, the error-guided orderings other than **ScpBnd-Abs** dominate with more time. Notably, with the highest  $i$ -bound, the **ScpBnd-Rel** ordering outperforms the baseline early and maintains its continuously.

### 5.2.1. Discussion on Anytime Performance

680 In contrast with the performance for finding exact solutions, we see that **ScpBnd-Rel** was overall the winning ordering heuristic. This is illustrated by its ranking at the highest  $i$ -bound on the two *harder* benchmarks we evaluated (**promedas** and **type4**). However, it was still outperformed by the baseline on the **pedigree** benchmark, though not significantly. Overall, from the instance-  
 685 by-instance lower bound plots **ScpBnd-Rel** was most consistent in producing

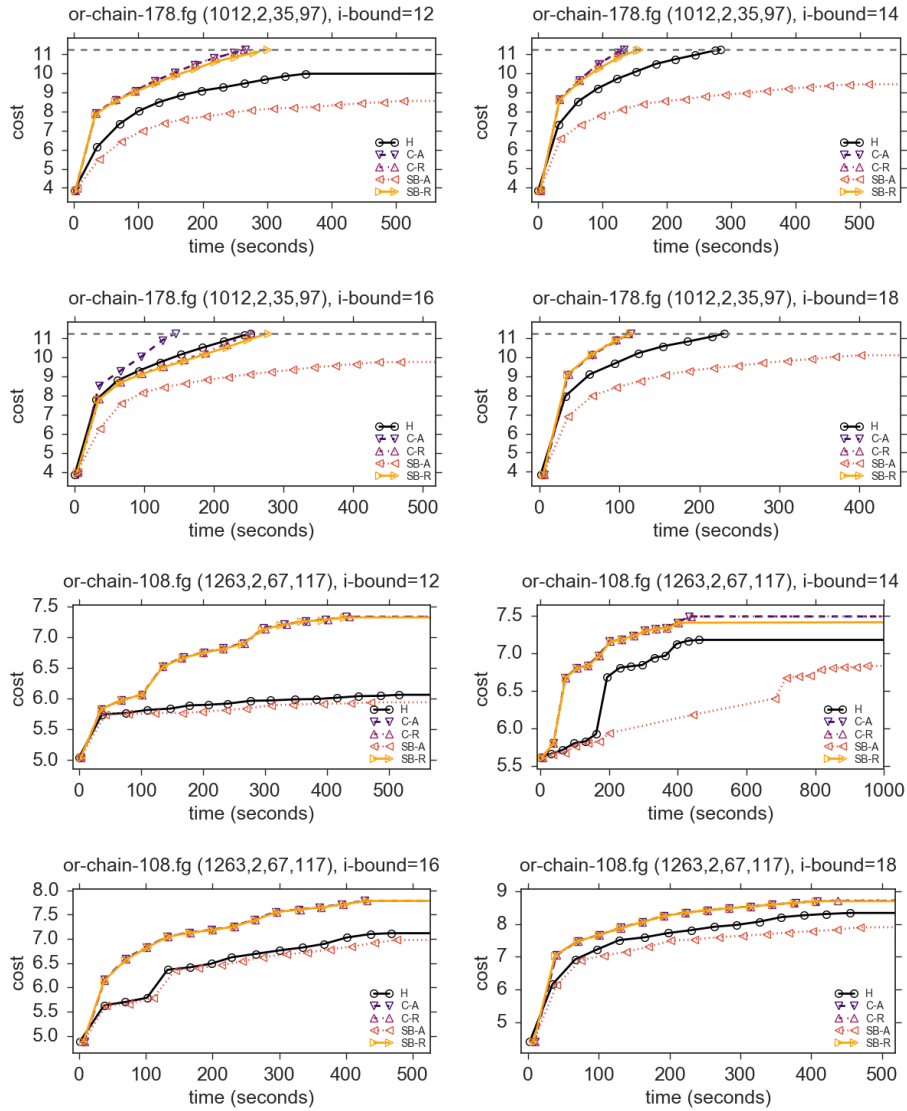


Figure 9: Lower bounds as a function of time for two instances from the **promedas** benchmark. Higher is better.

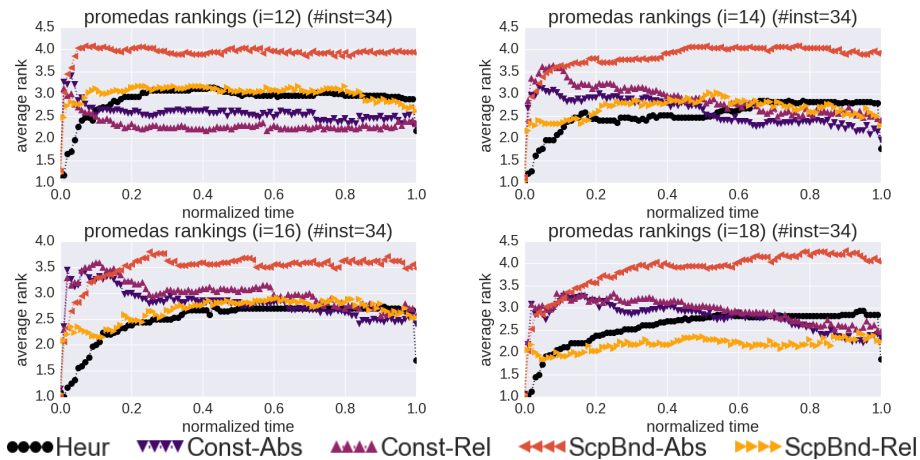


Figure 10: Average rank of each ordering as a function of normalized time across all of the instances in the **promedas** benchmark. Lower is better.

superior lower bounds.

## 6. Conclusion

This paper focuses on the potential of using AND/OR best-first search for generating lower bounds in an anytime fashion. Within this context, it explores the impact of subproblem ordering heuristics (the so-called secondary evaluation function [10]) for both exact and anytime performance. We present new heuristics for subproblem ordering which are based on pre-compiled information regarding the error associated with the primary heuristic, guided by a recently defined notion of bucket error. In an extensive empirical evaluation, we showed that in the context of evaluating the exact solution, most of our proposed error-based variants, were equally good, improving the runtime on 50-60% of the instances. This accounted for most of the hard instances which had enough error to impact subproblem ordering. In the anytime evaluation, we found that **ScpBnd-Rel** (the scope-bounded relative error functions) was the best scheme overall, illustrating the advantage of having more informed functional error information over the constant-based ones.

Various issues remain to be explored. First, all averages are taken by enumeration or sampling and using a simple average. However this assumes that each assignment has equal impact. Exploration into a weighted average estimate (i.e. *importance sampling*) could potentially improve the estimates. Also, the method of truncating the scopes for the **ScpBnd** methods is quite arbitrary. A more informed truncation procedure may reduce the loss of information. Lastly, we saw that **ScpBnd-Abs** tended to be much inferior to all ordering heuristics in nearly all cases, which suggests that the truncation and message passing pro-

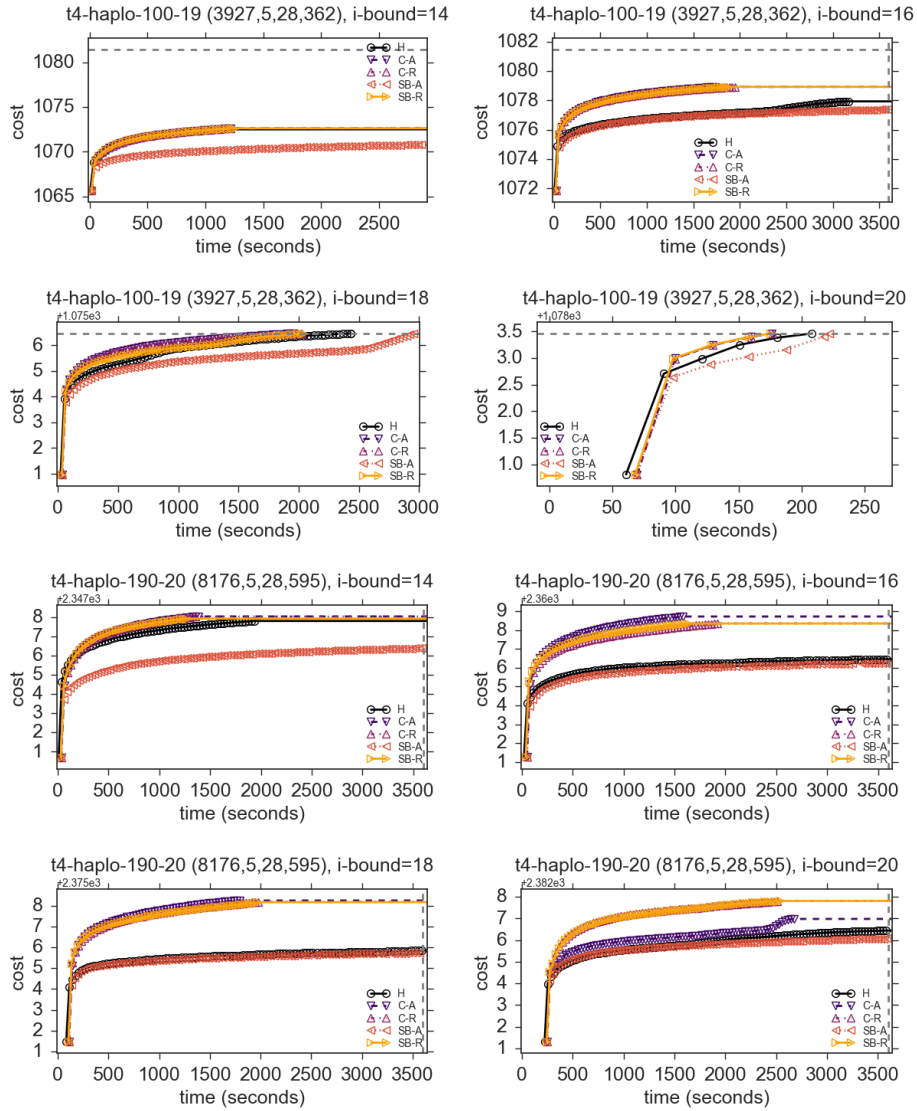


Figure 11: Lower bounds as a function of time for two instances from the **type4** benchmark. Higher is better.



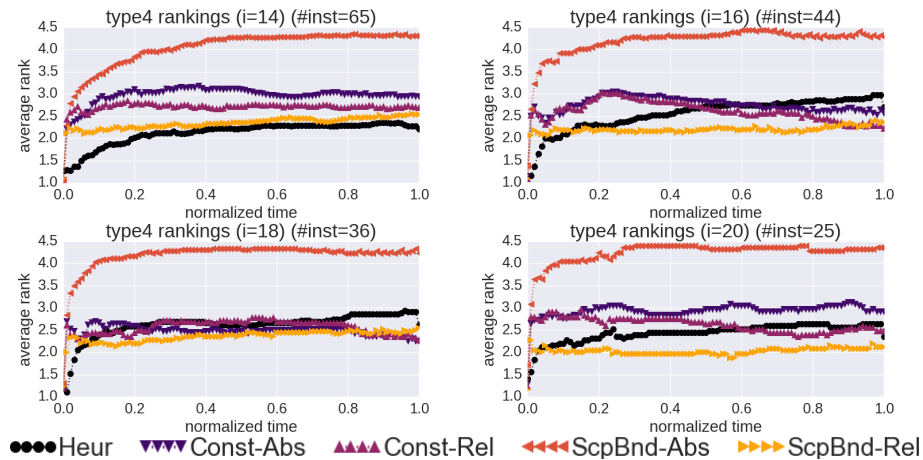


Figure 12: Average rank of each ordering as a function of normalized time across all of the instances in the `type4` benchmark. Lower is better.

710 cess is sensitive to the scale of values. The relative error variant which expresses errors in terms of percentages on the other hand, was informative.

As far as we know, there has been little focus on subproblem ordering in AND/OR Best-First search, our work illustrates that there is potential to be realized by ordering the subproblems in an informed manner. The ideas presented here generalize to any type of AND/OR search. In particular, various memory-efficient A\* variants (e.g. IDA\*, RBFS) [26, 27] use the idea of repeatedly *deleting and re-expanding* nodes, which would potentially also benefit from the savings yielded by better subproblem orderings.

715

## Acknowledgments

720 This work was supported in part by NSF grants IIS-1526842 and IIS-1254071, by the US Air Force under Contract No. FA8750-14-C-0011 under the DARPA PPAML program, and MINECO/FEDER under project TIN2015-69175-C4-3-R.

## References

- 725 [1] R. Dechter, Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2013. doi:10.2200/S00529ED1V01Y201308AIM023. URL <http://dx.doi.org/10.2200/S00529ED1V01Y201308AIM023>
- 730 [2] E. Bensana, M. Lemaitre, G. Verfaillie, Earth observation satellite management., Constraints 4 (3) (1999) 293–299.

- [3] M. Fishelson, D. Geiger, Optimizing exact genetic linkage computations, *Journal of Computational Biology* 11 (2-3) (2004) 263–275.
- 735 [4] C. Yanover, O. Schueler-Furman, Y. Weiss, Minimizing and learning energy functions for side-chain prediction, *Journal of Computational Biology* 15 (7) (2008) 899–911.
- [5] R. Marinescu, R. Dechter, And/or branch-and-bound search for combinatorial optimization in graphical models, *Artif. Intell.* 173 (16-17) (2009) 1457–1491.
- 740 [6] L. Otten, R. Dechter, Anytime and/or depth-first search for combinatorial optimization, *AI Communications* 25 (3) (2012) 211–227.
- [7] R. Marinescu, R. Dechter, Memory intensive and/or search for combinatorial optimization in graphical models, *Artif. Intell.* 173 (16-17) (2009) 1492–1524.
- 745 [8] D. Allouche, S. De Givry, G. Katsirelos, T. Schiex, M. Zytnicki, Anytime hybrid best-first search with tree decomposition for weighted csp, in: *International Conference on Principles and Practice of Constraint Programming*, Springer, 2015, pp. 12–29.
- [9] R. Marinescu, J. Lee, A. Ihler, R. Dechter, Anytime best+depth-first search for bounding marginal map, in: *AAAI*, 2017.
- 750 [10] J. Pearl, *Heuristics: Intelligent Search Strategies*, Addison-Wesley, 1984.
- [11] W. Lam, K. Kask, J. Larrosa, R. Dechter, Residual-guided look-ahead in and/or search for graphical models, *Journal of Artificial Intelligence Research* 60 (2017) 287–346.
- 755 [12] A. Darwiche, Recursive conditioning, *Artificial Intelligence* 126 (1-2) (2001) 5–41.
- [13] C. Terrioux, P. Jegou, Hybrid backtracking bounded by tree-decomposition of constraint networks, in: *Artificial Intelligence*, 2003.
- 760 [14] J. Philippe, T. Cyril, Decomposition and good recording for solving max-csps, in: *Proceedings of the 16th European Conference on Artificial Intelligence*, IOS Press, 2004, pp. 196–200.
- [15] P. Jégou, S. Ndiaye, C. Terrioux, Dynamic heuristics for backtrack search on tree-decomposition of csps., in: *International Joint Conference on Artificial Intelligence*, 2007, pp. 112–117.
- 765 [16] R. Dechter, R. Mateescu, And/or search spaces for graphical models, *Artif. Intell.* 171 (2-3) (2007) 73–106.

- [17] J. R. Bayardo, D. P. Miranker, On the space-time trade-off in solving constraint satisfaction problems, in: Fourteenth International Joint Conference on Artificial Intelligence(I95), 1995, pp. 558–562.
- 770 [18] E. C. Freuder, M. J. Quinn, Taking advantage of stable sets of variables in constraint satisfaction problems, in: Joint International Conference of Artificial Intelligence, 1985.
- [19] N. J. Nilsson, Principles of Artificial Intelligence, Tioga, Palo Alto, CA, 1980.
- 775 [20] R. Dechter, I. Rish, Mini-buckets: A general scheme for approximating inference, *Journal of the ACM* 50 (2) (2002) 107–153.
- [21] K. Kask, R. Dechter, Branch and bound with mini-bucket heuristics, in: Proc. IJCAI-99, 1999.
- [22] A. T. Ihler, N. Flerova, R. Dechter, L. Otten, Join-graph based cost-shifting schemes, in: Proceedings of the 28th Conference on Uncertainty of Artificial Intelligence (UAI 2012), 2012.
- 780 [23] K. Kask, A. Gelfand, L. Otten, R. Dechter, Pushing the power of stochastic greedy ordering schemes for inference in graphical models., in: AAAI, 2011.
- [24] M. Fishelson, D. Geiger, Exact genetic linkage computations for general pedigrees., *Bioinformatics* 18 (suppl.1) (2002) S189–S198.
- 785 [25] B. Wemmenhove, J. M. Mooij, W. Wiegerinck, M. Leisink, H. J. Kappen, J. P. Neijt, Inference in the promedas medical expert system, in: Artificial Intelligence in Medicine, Vol. 4594 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 456–460. doi:10.1007/978-3-540-73599-1\_61.  
790 URL [http://dx.doi.org/10.1007/978-3-540-73599-1\\_61](http://dx.doi.org/10.1007/978-3-540-73599-1_61)
- [26] R. E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, *Artificial intelligence* 27 (1) (1985) 97–109.
- [27] R. Korf, Linear-space best-first search, in: Artificial Intelligence, 1993, pp. 795 41–78.