On the Impact of Subproblem Orderings on Anytime AND/OR Best-First Search for Lower Bounds

William Lam and **Kalev Kask** and **Rina Dechter** [1] and **Javier Larrosa** [2]

Abstract. Best-first search can be regarded as an anytime scheme for producing lower bounds on the optimal solution, a characteristic that is mostly overlooked. In this paper we explore this topic in the context of AND/OR best-first search, guided by the mini-bucket heuristic, when solving graphical models. In that context, the impact of the secondary heuristic for subproblem ordering becomes apparent, especially when viewed in the anytime context. Specifically, we show how the concept of *bucket errors*, introduced recently, can yield effective subproblem orderings in AND/OR search and that it is often superior to the baseline approach which uses the same heuristic for node selection (OR nodes) and for subproblem orderings (AND nodes). Our experiments show an improvement in performance both for proving optimality and also in the anytime performance.

1 INTRODUCTION

AND/OR best-first search is guided by two heuristic evaluations functions. The first, f_1, evaluates the potential cost of the best solution extending a current partial solution graph. The second heuristic, f_2, prioritizes the tip nodes of a current partial solution which will be expanded next. Quoting Pearl (page 54): ”These two functions, serving in two different roles, provide two different types of estimates: f_1 estimates some properties of the set 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 priority of its hosting graph. Most work in search theory focus on the computation of f_1, whereas f_2 is usually chosen in an ad-hoc manner.” [10]

Our paper investigates the impact of secondary heuristics for ordering subproblems in AND/OR search [10]. This is done in the context of exploring the potential of AND/OR best-first search as an anytime scheme for generating lower bounds on the optimal solution. We consider the min-sum problem over a graphical model (which include, among others, the *most probable explanation* task of Markov networks [11] and weighted CSPs). This problem is usually solved by depth-first *branch-and-bound*, which generates successively better solutions and therefore tightens the *upper bound* on the optimal solution in an anytime fashion. AND/OR branch-and-bound (AOBB) improve its performance by exploiting decomposition in the graphical model. However, its anytime performance can be severely degraded due to the default practice to completely solve a subproblem before moving onto another. This was remedied, for example, by the breadth-rotating AOBB algorithm (BRAOBB) which rotates through subproblems rather than solving each one exactly first [9].

There has not been much work in the symmetrical problem of generating lower bounds by search in an anytime manner. Here, we explore the potential of best-first search as a potentially effective anytime scheme for lower bounds for AND/OR search spaces over graphical models. It is easy to see that best-first search schemes are ideal candidates since they explore the search space in frontiers of non-decreasing lower bounds and are thus inherently anytime for this task.

AND/OR best-first (AOBF), guided by the mini-bucket heuristic, is known to be a state-of-the art algorithm for combinatorial optimization (e.g., MPE and weighted CSP). Like any best-first scheme it is optimal under common conditions (admissibility and monotonicity of its heuristic function), yet it often cannot run to completion due to its memory issues. The algorithm is an extension of the A* algorithm to the AND/OR search space known as the AO* algorithm [8, 7]. Thus far, it has only been evaluated in its performance on finding an optimal solution. As an A*-like algorithm, it can be easily made to generate a sequence of lower bounds by simply reporting the best evaluation function, f_1, seen so far.

While exploring AOBF as an anytime scheme, our focus will be on investigating a class of secondary heuristics f_2.

Contributions. The paper investigates the potential of *bucket errors* introduced recently [2] in yielding secondary heuristic for subproblem ordering and its impact on anytime lower bounding. We analyze the problem and show empirically, on three benchmarks that the bucket errors provide relevant information on how much we can improve the lower bound when expanding a particular node. In particular, we show that it can improve the anytime lower bound often, when compared with a baseline ordering. To our knowledge, this is the first investigation of subproblem ordering in AOBF and among the first investigations of anytime best-first search.

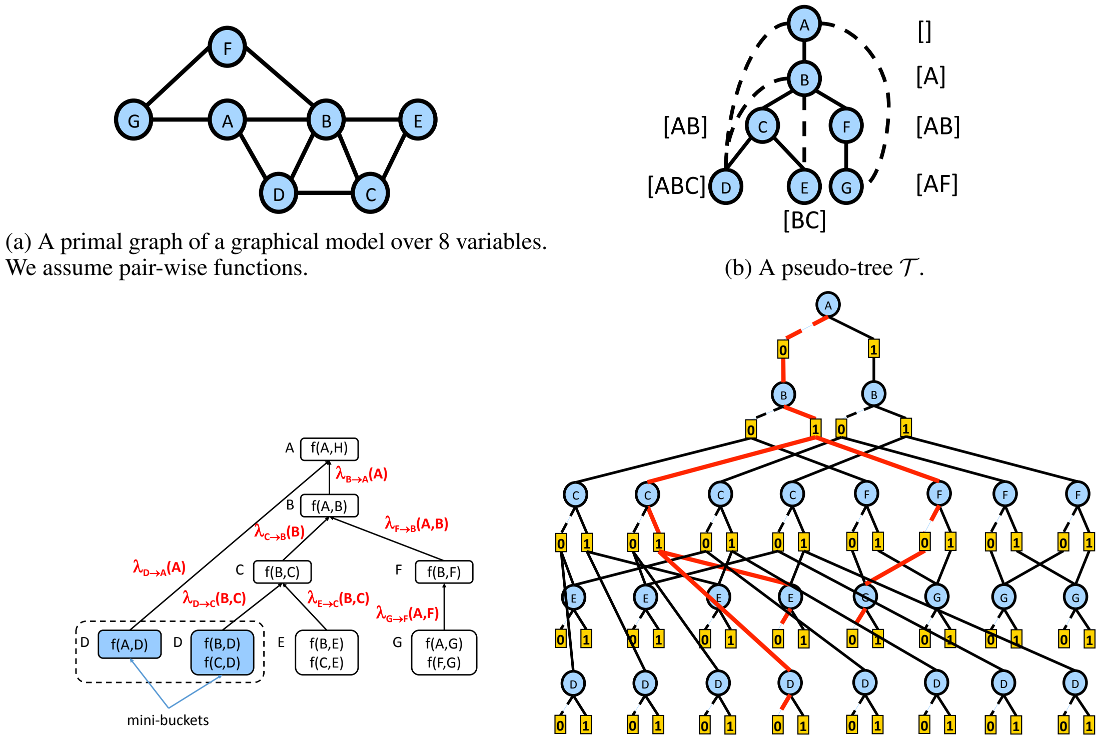
In section 2, we provide background. Section 3 discusses the impact of properly choosing a node for expansion. Section 4 relates the bucket error on the effect of a node expansion in AOBF and proposes several variants of a subproblem ordering heuristic based on bucket error. Section 5 presents the experiments and discussion. Section 6 concludes the paper.

2 BACKGROUND

Graphical Models A *graphical model* is a tuple $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, where $\mathbf{X} = \{X_i : i \in V\}$ is a set of variables indexed by a set V, $\mathbf{D} = \{D_i : i \in V\}$ is a set of finite domains of values for each X_i, and $\mathbf{F}$ is a set of discrete functions over subsets of $\mathbf{X}$. We focus on the *min-sum problem*, $C^* = \min_x \sum_{f \in F} f(x)$ which covers a wide

[1] University of California, Irvine, USA, {willmlam, kkask, dechter}@ics.uci.edu. Supported in part by NSF grants IIS-1526842, and IIS-1254071, the US Air Force under Contract No. FA8750-14-C-0011 under the DARPA PPAML program.

[2] UPC Barcelona Tech, Spain, larrosa@lsi.upc.edu. Supported in part by MINECO/FEDER under project TIN2015-69175-C4-3-R.



(a) A primal graph of a graphical model over 8 variables. We assume pair-wise functions.

(b) A pseudo-tree $\mathcal{T}$.

(c) Mini-bucket tree with i-bound of 3.

(d) The AND/OR context minimal search graph.

Figure 1

range of reasoning problem such as MAP in graphical models or cost minimization problem in weighted constraint satisfaction problems (WCSP). The *scope* of f (the subsets of X that $f \in \mathbf{F}$ are over) is denoted as *scope(f)*.

Primal graph, pseudo-tree. The *primal graph* G of a graphical model $\mathcal{M}$ is a graph where each variable X_i is represented by a node and edges connect variables which appear in the same scope of any $f \in \mathbf{F}$. A *pseudo-tree* $\mathcal{T} = (V, E')$ of the primal graph $G = (V, E)$ is a rooted tree over the same nodes V such that every edge in $E - E'$ connects a node to an ancestor in $\mathcal{T}$. We denote $\bar{X}_i$ as a set containing X_i and its *ancestors* in $\mathcal{T}$. $ch(X_i)$ denotes the children of X_i in $\mathcal{T}$.

Example. Figure 1a is a primal graph of a graphical model with 10 binary functions, where each edge represents a function. Figure 1b is a corresponding pseudo-tree.

AND/OR search. A state-of-the-art method to solve reasoning tasks on graphical models is to search their AND/OR graph [7]. The AND/OR search tree, which is guided by a *pseudo-tree* $\mathcal{T}$ of the primal graph G, consists of alternating levels of OR and AND nodes. Each OR node is labeled with a variable $X_i \in \mathbf{X}$. Its children are AND nodes, each labeled with an instantiation x_i of X_i. The weight of the edge, $w(X_i, x_i)$, is the sum of costs of all the functions $f \in F$ that are completely instantiated at x_i but are not at its parent X_i. Children of AND nodes are OR nodes, labeled with the children of X_i in $\mathcal{T}$. Each child represents a conditionally independent subproblem given assignments to their ancestors. Those edges are not weighted. The root of the AND/OR search *tree* is the root of $\mathcal{T}$. The path from an AND node labeled x_i to the root corresponds to a partial assignment to $\bar{X}_i$ denoted $\bar{x}_i$.

A *solution tree* in the AND/OR tree is a subtree that (1) contains its root, (2) if an OR node is in the solution subtree, then exactly one of its children is in the solution tree, (3) if an AND node is in the solution tree, then all its children are. A solution tree corresponds to a complete assignment and its cost is the sum-cost of its arc weights.

The AND/OR search tree can be converted into a *graph* by merging nodes that root identical subproblems. It was shown that identical subproblems can be identified by their *context* (a partial instantiation that decomposes the subproblem from the rest of the problem graph), yielding an *AND/OR search graph* [7].

Example. The pseudo-tree in Figure 1b labels each node with its context. Figure 1d displays the AND/OR search graph of the running example. A solution tree corresponding to the assignment $(A = 0, B = 1, C = 1, D = 0, E = 0, F = 0, G = 0)$, is highlighted.

AND/OR best-first (AOBF) is a state-of-the-art algorithm for solving combinatorial optimization problems over graphical models [7]. It is a variant of the AO* algorithm [8] specialized for graphical models. It works by maintaining the explicated part of the context-minimal AND/OR search graph $\mathcal{G}$ and keeps track of a best *partial solution tree* T^*. To achieve this, every node in $\mathcal{G}$ needs to be updated with its best lower bound based on the current state of $\mathcal{G}$. We present the AOBF in Algorithm 1. It interleaves a top-down expansion step and a bottom-up revision step. Once T^* is determined (using f_1), the top-down expansion step selects a non-terminal tip node of T^* and generates its children in $\mathcal{G}$ (lines 4-9) (this is f_2). A bottom-up revision step follows, which updates the internal nodes values that represent the best lower bound that we know for that node, based on its expanded descendants (lines 10-15). In this step, we just need to propagate the values of newly expanded children to its parents and repeat this process for every updated node (up to the root if necessary). This step also marks which children are the best of a particular node. In addition, for OR nodes, we mark them as solved if it's best child is solved and for AND nodes, we mark them as solved if all of its children are solved. Finally, we find a new best partial solution tree T^* by following the marked best children from the root of $\mathcal{G}$. When the root node of $\mathcal{G}$ is marked as solved, then the best solution tree T^* is the optimal solution whose cost is the revised value of the root node $v(r)$ and AOBF terminates. In previous literature the algorithm is provided as a purely exact algorithm. However, since values are constantly updated as the algorithm searches, $v(r)$ is actually a lower bound on the solution. In particular, when running out of memory, it can return $v(r)$ as a lower bound on the solution.

Algorithm 1: AOBF

```
Input: A graphical model $\mathcal{M} = (X, D, F)$, pseudo-tree $\mathcal{T}$,
       heuristic function $h$
Output: Lower-bound to solution of $\mathcal{M}$
Create the root OR node $r$ labeled by $X_1$ and let $\mathcal{G} = \{r\}$
Initialize value $v(r) = h(r)$ and best partial solution tree $T^*$ to $\mathcal{G}$
while $r$ is not solved and memory is available do
    // Expand
    $n$ := Select-Node($T^*$)
    if $n = X_i$ is an OR node then
        add AND child $n'$ for each $x_i \in D(X_i)$ to $\mathcal{G}$
    else if $n = \langle X_i, x_i \rangle$ is an AND node then
        add OR child $n'$ for each child of $X_i$ in $\mathcal{T}$ to $\mathcal{G}$
    Initialize $v(n') = h(n')$ for each generated $n'$
    // Revise
    Update $n$ and its ancestors $p$ in $\mathcal{G}$ bottom-up (repeat until
      the root):
    if $p$ is an OR node then
        $v(p) = \min_{m \in children(p)}(c(p,m) + v(m))$
        Mark arg $\min_m$ as the best child of $p$
    else if $p$ is an AND node then
        $v(p) = \sum_{m \in children(p)} v(m)$
    Update $T^*$ to new best partial solution tree by following
      marked best children from root $r$
return $\langle v(r), T^* \rangle$
```

Mini-Bucket Elimination Heuristic. Current AOBF [6, 7] algorithms are guided by the mini-bucket heuristic. This heuristic is based on *mini-bucket elimination*, MBE(i), where the *i-bound* allows trading off pre-processing time and space for heuristic accuracy with actual search. It is a *static heuristic*, meaning that it is pre-processed before search starts. MBE(i) works relative to the same pseudo-tree $\mathcal{T}$ which defines the AND/OR search graph. Each node X_k of $\mathcal{T}$ is associated with a bucket B_k that includes a set of functions. The *scope of a bucket* is the union of scopes of all its functions before it is processed as described next. First, each function f of the graphical model is placed in a bucket B_k if X_k is the deepest variable in $\mathcal{T}$ s.t. $X_k \in scope(f)$. Then MBE(i) processes the buckets of the pseudo-tree from leaves towards the root. Processing a bucket may require partitioning the bucket's functions into *mini-buckets* $B_k = \cup_r B_k^r$, where each B_k^r includes no more than i variables. Then, processing each mini-bucket separately, all its functions are combined (by sum in our case) and the bucket's variable (X_p) is eliminated (by min in our case). Each resulting new function, also called a *message*, is placed in the closest ancestor bucket whose variable is contained in its scope. By design, MBE(i) is time and space exponential in the i-bound.

Bucket Elimination (BE). When the i-bound is high enough so there is no partitioning the mini-bucket algorithm becomes the exact bucket-elimination (BE) scheme [1]. Its time and space complexity is exponential in the size of the largest bucket scope encountered which is called the *induced width* and is denoted w^*, for that ordering.

Notation. In the following, f_k denotes an original function placed in Bucket B_k (if there is more than one, they are all summed into a single function), and $\lambda_{j \to k}$ denotes a message created at bucket B_j and placed in bucket B_k. Processing bucket B_k produces messages $\lambda_{k \to i}$ for some ancestors X_i of X_k (i.e, $X_i \in \bar{X}_k - X_k$).

Example. Figure 1c illustrates the execution of MBE(3) in our running example by means of the so-called mini-bucket tree (nodes are buckets and tree-edges show message exchanges). In this example, bucket B_D is the only one that needs to be partitioned into mini-buckets. Each mini-bucket generates a message. In the figure, messages are displayed along an edge to emphasize the bucket where they are generated and the bucket where they are placed. For instance, $\lambda_{D \to C}$ is generated in bucket D (as $\lambda_{D \to C} = \min_D \{f(B,D) + f(B,D)\}$) and placed in bucket C.

Extracting the mini-bucket heuristic [5]. The (static) mini-bucket heuristic used by AOBF requires a MBE(i) execution prior to search keeping all the message functions. Let $\bar{x}_k$ be a partial assignment. Λ_j denotes the sum of the messages sent from bucket B_j to all of the instantiated ancestor variables,

$$\Lambda_j(\bar{x}_k) = \sum_{X_l \in \bar{X}_k} \lambda_{j \to l}(\bar{x}_k) \tag{1}$$

The heuristic value at node $\bar{x}_k$ is,

$$h(\bar{x}_k) = \sum_{X_j \in \mathcal{T}_k} \Lambda_j(\bar{x}_k) \tag{2}$$

Example. In our example, the heuristic function of partial assignment $(A = 0, B = 1)$ is $h(A = 0, B = 1) = \lambda_{D \to A}(A = 0) + \lambda_{C \to B}(B = 1) + \lambda_{F \to B}(A = 0, B = 1)$

Bucket Error [2]. The notion of bucket error was introduced recently as a function that captures the local error induced by mini-bucket elimination. It 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 and the message computed by the mini-buckets.

DEFINITION **1** (**bucket and mini-bucket messages at** B_k). *[2] Given a mini-bucket partition $B_k = \cup_k B_k^r$, the combined mini-bucket message B_k is,*

$$\mu_k = \sum_r \left(\min_{X_k} \sum_{f \in B_k^r} f \right) \tag{3}$$

while the exact message that would have been generated with no partitioning at B_k is,

$$\mu_k^* = \min_{X_k} \sum_{f \in B_k} f \tag{4}$$

It is important to note that μ_k^* is only exact for the current bucket B_k; it may contain errors introduced by partitioning from its descendants.

DEFINITION 2 (**local bucket error at B_k**). *Given a run of MBE, the local bucket error function Err_k of B_k is,*

$$Err_k = \mu_k^* - \mu_k \tag{5}$$

where the scope of Err_k is the same as the set variables in B_k.

Computing and storing the bucket functions takes time and space exponential in the *pseudo-width*, which is between the i-bound and induced width w^* of the problem. The pseudo-width is the maximum number of variables in a bucket once the mini-bucket execution finishes. Since the pseudo-width may be too high, often those functions can be computed but cannot be stored. We therefore define an aggregate notion of a bucket error which is the average value of the bucket error function.

DEFINITION 3 (**average bucket error at B_k**). *[2] The average bucket error of B_k given a run of MBE is,*

$$\tilde{E}_k = \frac{1}{\prod_{v \in \bar{X}_k} |D_v|} \sum_{\bar{x}_k} Err_k(\bar{x_k})$$

Alternatively, we can divide the $Err_k(\bar{x}_k)$ term by $\mu^*(\bar{x}_k)$ to yield *relative error*.

3 ILLUSTRATING THE IMPACT OF SUBPROBLEM ORDERING

We refer to Algorithm AOBF which is described in Algorithm 1 in the following. The *Select-Node* call in line 4 of the AOBF algorithm uses f_2, where the heuristic function h corresponds to f_1. In all implementations of AOBF so far, f_2 is based on the mini-bucket heuristic.

THEOREM **1.** *Given an AO* algorithm that uses a given f_1 evaluation function. Then, for different choices of f_2, when AO* terminates (with a full solution which is optimal) its explicated subgraph G' is not the same, even if we assume identical tie breaking rules. Namely, even though the final value of a solution tree seen at termination is the same, with two different secondary functions f_2 and f_2', the sequence of f_1 values seen for successive partial solution graphs: $t_1, t_2, \ldots, t_i, \ldots$ and $t_1', t_2', \ldots, t_i', \ldots$, produced when using f_2 and f_2' respectively, can differ significantly.*

Note that the sequence of f_1 values for each respective sequence, yields lower bounds whose quality increases (for a monotone heuristic function), yet one sequence may be superior to the other.

DEFINITION **4** (**profile**). *Let us call the sequence $p_{f_2} = \{f_1(t_i) | i = 1..j\}$ produced with a particular f_2 when f_1 is kept fixed, the profile of f_2, under f_1. Let L be a constant smaller than C*. We denote by $I_{f_2}(L)$ the first index of a profile where it crossed the lower bound L and call it the threshold index of f_2 at cost L. Namely when using f_2 as a secondary heuristic, $I_{f_2}(L) = k$ if $f_1(t_{k-1}) < L$ and $f_1(t_k) \geq L$.*

If t is a solution base that can be extended to an optimal solution tree, then $f^*(t) = C^*$. In the following we show that the choice of f_2 can, in the worst case, make an exponential difference in the number of expansions needed to cross a lower bound threshold L. This means that the impact of f_2 can be quite high for both anytime performance and when looking for an optimal solution.

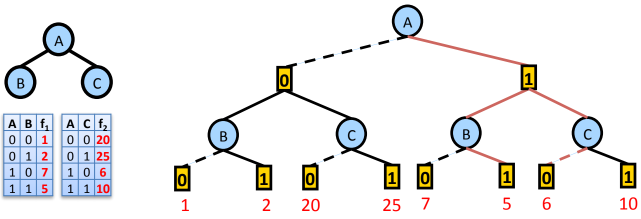
THEOREM **2.** *Assume the search space is a weighted AND/OR tree, having a heuristic evaluation function $f_1(t)$ for a partial solution tree t. Let f_2 and f_2' be two secondary heuristic functions deciding which tip node of t to expand next. Then, there exists an AND/OR search tree, a heuristic evaluation function f_1, and 2 secondary heuristic functions f_2 and f_2', and a lower bound threshold L, $L \leq C^*$, where C^* is the optimal cost for the AND/OR tree, such that there is an exponential gap between $I_{f_2}(L)$ and $I_{f_2'}(L)$.*

Proof. Let t be a partial solution base that is currently selected for extension by AOBF search a weighted AND/OR tree, T. Let $C = f_1(t)$ where $C < C^*$. Assume that t cannot be extended to an optimal solution, namely $f^*(t) > C^*$. Let A and B be two variables labeling OR tip nodes of t which are direct child nodes of an OR parent $X = 0$. We will not make assumptions regarding the subtrees below A and below B.

Let a subtree below by a variable X be denoted as T_X and the d-depth truncated subtree be denoted as T_X^d. Now, assume that T_B is an OR tree having depth n (i.e., there are n variables below it). Assume that the best extension of t into T_B has f_1 smaller than C^*. Furthermore, we want to make sure all of the nodes in T_B would be explored by AOBF in order to establish the optimal cost in T_B. This can be enforced if the weights on the arcs in T_B are monotonically increasing along a breadth-first ordering of the arcs in T_B. In contrast, assume that T_A^1 (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 about the AND/OR tree and f_1, a secondary heuristic function f_2' that prefers expanding all of T_B before any of T_A (and such exists) will expand many more nodes than an f_2 that expands T_A first, and which will never expand any of T_B since its $f(t \cup T_A^1) \geq C^*$. Therefore, with f_2', we generate exponentially more nodes before finding an optimal solution.

To generalize, let $f_1(t) = C_1$, $f_1(t \cup T_A^1) = C_2$, and $f_1(t \cup T_B^n) = C_3$, where $C_1 \leq C_3 < C_2 < C^*$. Assume that the profiles of f_2 and f_2' until subtree t is encountered are identical and appear at index j of their profiles. Clearly, if we apply f_2 at this point, we get that the index of lower bound C_2 is $j + 1$. On the other hand, if we apply f_2', its index for lower bound C_3 is $j + k^n$, where k is the maximum domain size of a variable in the problem. Since $C_3 < C_2$, its index to C_2 must be even larger. Therefore, $I_{f_2}(C_2) = j + 1$, while $I_{f_2'}(C_2) > j + k^n$, which proves our claim. □



A	B	f_1
0	0	1
0	1	2
1	0	7
1	1	5

A	C	f_2
0	0	20
0	1	25
1	0	6
1	1	10

Figure 2: 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.

Example. Figure 2 depicts a graphical model defined over 3 variables A, B, C having two functions $f(A, B)$ and $f(A, C)$. The full AND/OR search space is shown here and the optimal solution

$(A = 1, B = 1, C = 0)$ is marked in red. Let us assume that our heuristic evaluation function is 0, and that the general primary f_1 of a partial tree is obtained by backing up recursively the values of h up to the root of the tree, as usual. Let's also assume that we have two secondary functions for subproblem orderings: f_2 which orders the subproblems by choosing the left-most branch first ($B \prec C$), while f_2' reverses the orderings from right to left. It is easy to see that the sequence of f_1 values seen with f_2 is: (0,1,5,11) while the sequence seen with f_2' is shorter: (0,6,11). In particular, with f_2 we explore all solution subtrees while with f_2' we will never expand the B node 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 due to the $A = 1$ never exceeding a cost of 20 at any point and in fact yields the optimal solution. Clearly, the profile of f_2' dominates that of f_2 in this case.

To illustrate further the potential impact of subproblem ordering, consider a partial solution tree encountered t, that cannot be extended to an optimal solution. If $f_1(t) < C^*$, the value of the optimal solution, the secondary function f_2 can influence the size explored below it considerably. It can guide in selecting a subproblem that yields the largest increase in the heuristic evaluation function first, then avoiding expanding alternative branches. Overall, we wish to prune partial solution graphs t that lead to suboptimal solutions in as few node expansions as possible. The pruning condition is $f_1 > C^*$, since best-first search will never expand a partial solution graph that has been proven to be suboptimal. Therefore, the tip node selection given by f_2 should lead to the largest increase in f_1.

4 BUCKET ERRORS FOR AOBF

The overall target for an effective ordering heuristic should select the subproblem which it can increase the bound the most in fewer expansions. First, let us assume a simple, greedy scheme for subproblem ordering, that is, given a frontier of tip nodes to select from, we aim to select the one that will lead to the greatest increase in f_1 in a single expansion. Clearly, *lookahead* can be used to answer this question. As such the depth-1 lookahead residual of each tip node corresponds to the greatest increase in f_1, which, in turn, is equivalent to the bucket error [2].

However, this greedy scheme does not capture well the cases where there exists a T_X^d such that $d > 1$ and $f(t \cup T_X^d) > C^*$. In particular, if $f(t \cup T_X^1) = f(t)$, then the subproblem rooted by X would be ordered last because its depth-1 residual is zero.

Let us therefore consider the opposite end of the spectrum by considering the error of $f_1(t)$ itself.

DEFINITION 5 **(full residual).** *Let the full residual $res^*(n) = h^*(n) - h(n)$. This is equivalent to the d-level residual with d equal to the depth of the search space below node n.*

Computing the full residual $res^*(n)$ is equivalent to performing look-ahead in a way that would be equivalent to solving the problem, so we consider approximating it instead by summing over the depth-1 residuals (or the bucket errors). To distinguish this measure from bucket errors, we call them *subtree errors*. We will clarify what it means to sum over bucket errors in the following sections that detail two different approaches we propose. Overall, using bucket errors over look-ahead during search has the advantage that all computation of the secondary heuristic is done as pre-processing before search, much like with static MBE heuristics.

4.1 Constant Subtree Error

The first approach is to use the *average bucket errors* to represent the error at each bucket. This is the simplest and most efficient approach, as we only require a single constant value for each bucket variable. Summation over them is then trivial; we simply add all of the average bucket errors for variables within the subtree of interest.

We define the *constant subtree error* recursively as

$$\tilde{E}_k^{st} = \tilde{E}_k + \sum_{c \in ch(X_k)} \tilde{E}_c^{st} \tag{6}$$

where $\tilde{E}_k$ are the average bucket errors (see Definition 3) and $ch(X_k)$ denotes the children of variable X_k in the pseudo tree. For leaves in the pseudo tree, this expresses the base case that for those variables, the constant subtree error is identical to the average bucket error.

An immediate shortcoming is that this is *context-independent*, since the average is computed over all of the instantiations to variables within a particular bucket. This means that they contain information from all instantiations of the error functions, whereas we would ideally only want the error for the single instantiation we are interested in during search. The quality of this constant subtree error approach is therefore dependent on the variance of the bucket error functions. Still, due to its simplicity, it consumes a very minimal amount of memory, which can allow AOBF to expand mode nodes before running out of memory.

4.2 Subtree Error Functions

In order to address the context-independent nature of constant-value subtree errors, we can consider storing functions, much like with local bucket error functions. Ignoring the possibly intractable complexity of storing such functions for now, we first describe how to generate *subtree error functions* from the local bucket error functions. A subtree error function for a particular variable will have the same scope as its local bucket error function.

In extending Equation 6 to error *functions*, we take a natural bottom-up approach. We define a one pass message passing scheme similar in execution to bucket elimination, but with some differences.

DEFINITION 6 **(subtree error message).** *Let Err_k^{st} be a subtree error function with scope S_k, X_p be the parent of X_k in pseudo tree $\mathcal{T}$, and Err_p be the local bucket error function for X_p with scope S_p. Then the* subtree error message *is defined as*

$$\lambda_{k \to p}^e(S_p) = \frac{1}{|\mathbf{D}_{S_k - S_p}|} \sum_{S_k - S_p} Err_k^{st}(S_k) \tag{7}$$

The message is generated by *averaging* out the variables which are not present in the parent which the message will be sent to. Similar to constant subtree errors, the subtree error functions are identical to the local bucket error functions, and forms the base case for the recurrence. **Algorithm 2** present a procedure for computing all of the subtree error functions for a given problem with local bucket error functions already computed. We denote the parent of i in $\mathcal{T}$ as $pa(i)$.

4.3 Approximating Error Functions

We now return to deal with the issue of the error functions being possibly intractable to compute and store due to its complexity in the pseudowidth. We choose to approximate the error functions as a result.

```
Algorithm 2: Subtree Error Propagation
  Input: A Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, a pseudo-tree $\mathcal{T}$,
      $i$-bound, local bucket error functions $Err_k$
  Output: Subtree error functions $Err^{st}_k(S_k)$
1 Initialize, for all leaves $u$ of $\mathcal{T}$, $Err^{st}_u(S_u) = Err_u(S_u)$
2 Compute bottom-up over $\mathcal{T}$, for each variable $X_i$,
3     $Err^{st}_{X_i}(S_i) = Err_{X_i}(S_i) + \sum_{c \in ch(X_i)} \lambda^e_{c \to i}(S_i)$
4     $\lambda^e_{i \to pa(i)}(S_{pa(i)}) = \frac{1}{|\mathbf{D}_{S_i - S_{pa(i)}}|} \sum_{S_i - S_{pa(i))}} Err^{st}_i(S_i)$
5 return $Err^{st}_{X_i}$ for each variable $X_i in \mathbf{X}$
```

First, we address the space aspect first by bounding the scope of the error functions. Unlike the MBE scheme, the error functions are single functions rather than a collection of functions (buckets), so it is not possible to consider any sort of partitioning-based scheme. We take the approach of quantization by truncating the scopes of the local bucket error functions and aggregating over the eliminated variables by averaging.

DEFINITION **7 (scope-bounded local bucket error function).** *Let s^b_k be a scope such that $s^b_k \subseteq s_k$, where s_{X_i} is the scope of the local bucket error function. Then the* scope-bounded local bucket error function*for a variable X_k is*

$$Err^b_k(S^b_k) = \frac{1}{|\mathbf{D}_{S_k - S^b_k}|} \sum_{S_k - S^b_k} Err_k(S_k) \tag{8}$$

With this, we can select a scope $S^b_{X_k}$ for each variable such that $|S^b_{X_k}|$ is smaller then a specified integer (typically the i-bound). We currently select $S^b_{X_k}$ in an ad-hoc fashion by removing variables from S_{X_k} that are closest to X_k until the condition above is satisfied. Next, we can replace the full local bucket error functions used in **Algorithm 2** with scope-bounded versions to generate *scope-bounded subtree error functions* which are tractable to compute and store in memory. This is a generalization of everything presented thus far. If $S^b_{X_k} = \emptyset$ for all variables, then each local bucket error functions becomes the average bucket error. Running **Algorithm 2** on such functions reduces to constant subtree errors.

The other aspect of computation time is still present here, since we still have to enumerate over a possibly large function when averaging out a subset of variables (i.e. we may need to truncate a large number of variables) For this, we can replace the averaging steps of Equations 7 and 8 with a sampling procedure. In particular, given an expression of the form $\frac{1}{\mathbf{D}_s} \sum_s Err$, we replace this with $\frac{1}{\#samples} \sum_{\bar{x}} Err, \bar{x} \sim U(s)$, where s is a scope and $U(s)$ denotes a uniform distribution over the assignments to s.

4.4 Weighted Subtree Error

Although we now have methods of approximating of $res^*(n)$, they are actually invariant to the amount of AOBF expansions it takes to actually achieve that difference. Let us consider an example where there are two tip nodes n and m to choose from. Assume that the variable for node n has non-zero subtree error, but its descendants have zero subtree error, while the variable for node m has zero subtree error, but it has a single descendant variable for node m_d that is d nodes deeper with non-zero subtree error. In this case, the approximation to $res^*(n)$ is exact for both. Furthermore, now let us assume that $R = res^*(n) = res^*(m)$. Using this directly, the tip nodes are tied in priority. However, we know that expanding n is less costly since in order for AOBF to improve the lower bound by R via expanding m first, it must at expand every single node on the path from m to m_d, spending at least d more node expansions.

A natural approach is to incorporate weights for the messages sent to the parents in **Algorithm 2** as a way to penalize their contributions.

To apply this idea, we modify line 3 of **Algorithm 2** to

$$Err^{st}_{X_i} = Err_{X_i} + w_i \cdot \sum_{c \in ch(X_i)} \lambda^e_{c \to i} \tag{9}$$

where w_i is the weight for variable X_i.

To decide what the weights should be, we go back to the reasoning in Theorem 2. Let t be a current partial solution tree. Let A and B be two variables labeling OR tip nodes of t. Let $t_A = t \cup T^{d_1}_A$ and $t_B = t \cup T^{d_2}_B$ be the partial solution trees expanded below A to depth d_1 and below B to depth d_2 such that $f(t_A) = f(t_B)$ and $d_1 < d_2$. Then it follows that $I(t_A) < I(t_B)$, with a difference as large as $k^{d_2 - d_1}$. Therefore, we should choose to expand nodes in t_B over t_A if the error of t_B is at least $k^{d_2 - d_1}$ larger than that of t_A. Intuitively, this prioritizes errors which are closer and falls in line with our objective of increasing the bound the most with fewer expansions. Based on this analysis, we choose w_i as $\frac{1}{|D_i|}$, where D_i is the domain of variable X_i, so $|D_i|$ is a worst case estimate of the number of extra expansions needed from a variable X_i before moving on to expand its children (i.e. we expand each value of X_i before proceeding finding the best one). We take the inverse of this so it penalizes the children. Since *weighted* errors are now propagated, the penalizing weights along a path to the root serve to exponentially penalize errors based on their depth relative to any particular node on the path, which matches our estimate above.

Finally, our ordering heuristic f_2 for a particular variable X_i is given directly by the subtree error function $Err^{st}_{X_i}$, which varies depending on the choice of $S^b_{X_k}$ for each variable.

5 EXPERIMENTS

We evaluated our proposed subproblem ordering heuristics on the min-sum task in graphical models. Using the framework for scope-bounded subtree error functions described in the previous section, we experimented with 4 variants. In particular, we considered the case of absolute vs. relative errors and constant vs. scope-bounded subtree error functions. We compare these 4 also against the baseline subproblem ordering based on the heuristic evaluation function, f_1. In all of the results, we refer to the baseline as *Baseline*, and denote our 4 variants with hyphenated combinations of *Absolute/Relative* and *Constant/ScopeBounded* The version of MBE we used for all experiments is mini-bucket elimination with moment-matching (MBE-MM) [4]. We used a fixed pseudo-tree for all settings. We also varied the i-bound to show how the results depend on the levels of heuristic error. When necessary, in approximating each error function, we used a fixed scope-bound of 10 for the scope bounding and a maximum sample size of 10^5 values for sampling.

We used benchmarks from genetic linkage analysis [3] (`pedigree`, `type4`), and medical diagnosis networks [12] (`promedas`). We included only instances that had a fair amount of search (number of nodes expanded is at least on the order of 10^5 when using the baseline ordering). Overall we report results on 10 pedigrees, 34 promedas networks, and 44 type4 instances, for a total of 88 instances. The implementation was in C++ (64-bit) and was executed on a 2.66GHz processor with 24GB of RAM.

Exact Inference. In **Table 1**, we report the CPU time (seconds) and number of OR nodes expanded for a subset of the problems which

instance (n, k, w^*, h)	heur	$i = 10$ time	 nodes	$i = 12$ time	 nodes	$i = 14$ time	 nodes	$i = 16$ time	 nodes	$i = 18$ time	 nodes
				pedigree networks							
pedigree9 (935,4,27,100)	Baseline	792	10412048	226	4391784	237	5672090	**27**	950622	**12**	261861
	Absolute-Constant	840	10408138	**219**	4389160	103	3890902	31	947791	15	283633
	Relative-Constant	288	7642412	233	**4388686**	103	3889300	31	947712	15	**254102**
	Absolute-ScopeBounded	956	10829863	245	4456735	236	5669618	31	951332	14	283398
	Relative-ScopeBounded	**252**	**7011689**	233	4415602	**103**	**3888749**	32	**947379**	15	270359
pedigree19 (693,5,21,117)	Baseline		oom	1102	6908120	284	3695085	54	1028323	**98**	**192875**
	Absolute-Constant		oom	**515**	9292892	**212**	**3322455**	42	927227	101	220920
	Relative-Constant		oom	568	**4538644**	272	3679115	**42**	**922953**	101	220920
	Absolute-ScopeBounded		oom	598	9837857	274	7672713	51	1428580	102	234270
	Relative-ScopeBounded		oom	764	7669632	271	3663181	44	1025588	100	220920
pedigree44 (644,5,24,79)	Baseline	574	11673224	258	7806467	**31**	1166724	**13**	510651	**8**	206145
	Absolute-Constant	578	11466903	232	**7151730**	33	**1142691**	17	516318	11	**194343**
	Relative-Constant	398	9783861	228	7377908	34	1227131	16	517032	11	216053
	Absolute-ScopeBounded	**312**	**9065438**	294	8005335	32	1152958	15	**505324**	9	213716
	Relative-ScopeBounded	455	10794795	**227**	7331955	33	1169720	14	537312	9	213524
				promedas networks							
orchain80.fg (840,3,50,108)	Baseline		oom	128	5374693	80	3398901	47	1977044	46	1904356
	Absolute-Constant	**370**	**14358065**	112	4550622	**75**	3019573	47	1724537	42	**1443359**
	Relative-Constant		oom	**99**	**4550571**	76	3029016	47	1738411	41	1443415
	Absolute-ScopeBounded		oom	248	9001600	174	6222197	83	3108986	84	3127096
	Relative-ScopeBounded		oom	114	4551500	76	**3018916**	**46**	**1724171**	**40**	1444453
orchain208.fg (797,3,41,97)	Baseline		oom	336	13850972	79	3555258	39	1747772	41	1830746
	Absolute-Constant	**157**	**6592327**	**133**	5866565	**53**	**2138422**	25	**893042**	29	**1103890**
	Relative-Constant	157	6592583	136	**5865223**	54	2148716	25	902896	**26**	1111423
	Absolute-ScopeBounded		oom		oom	302	10415855	113	4199239	87	3399972
	Relative-ScopeBounded	184	7434017	161	6730016	54	2176748	**25**	912046	29	1125857
orchain226.fg (735,3,42,87)	Baseline		oom	197	8434233	212	9580054	75	3443357	34	1528349
	Absolute-Constant		oom	156	5322015	**90**	**3330653**	**45**	1611674	28	**921161**
	Relative-Constant	342	**11205154**	**154**	**5320502**	104	3905754	45	**1611614**	26	976171
	Absolute-ScopeBounded		oom	291	11595162	346	15606697	117	4854775	62	2525417
	Relative-ScopeBounded	**339**	11231185	159	5482198	98	3663610	46	1697173	**25**	976807

Table 1: Times and nodes expanded for finding the optimal solution.

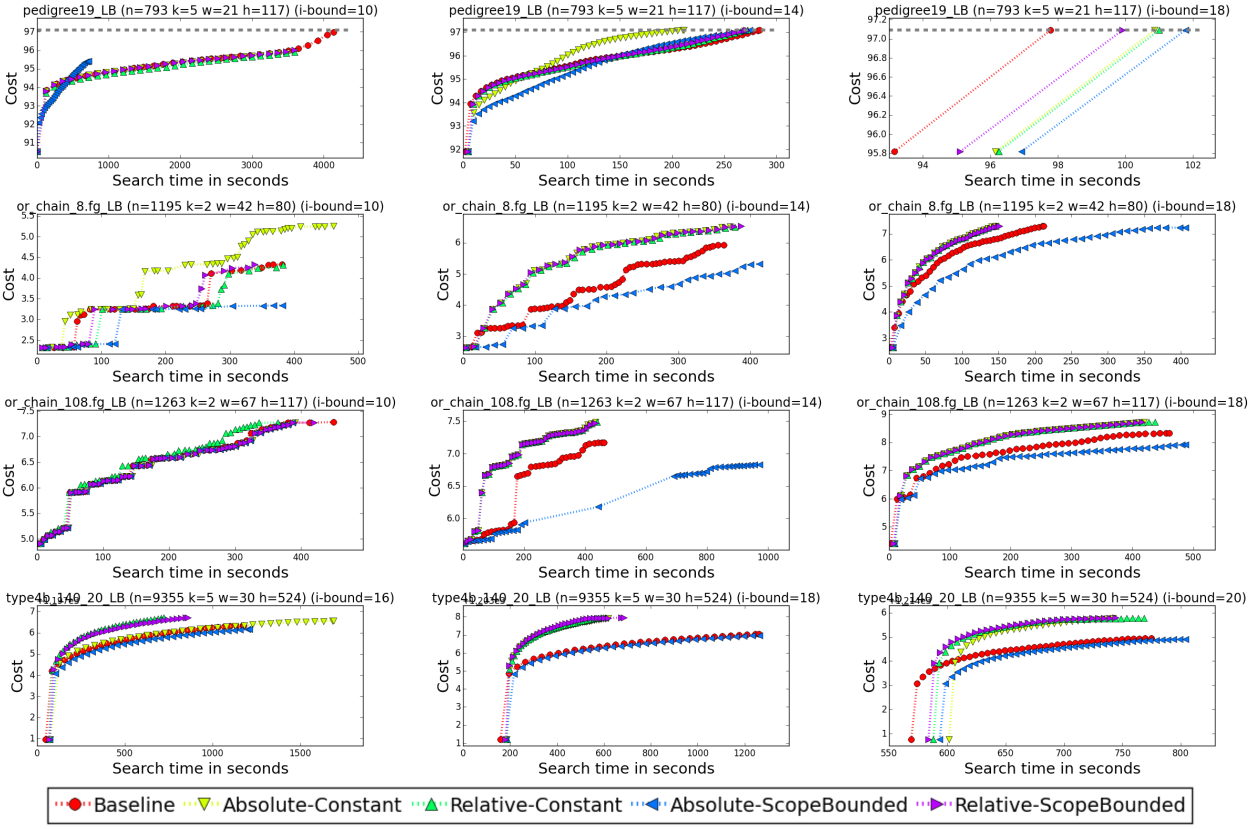


Figure 3: Plots of the lower bound over time. The exact solution is shown as the top dotted horizontal line if known. Higher is better.

could be solved exactly by AOBF. 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 ordering schemes we experimented with. The 24GB of memory was shared between the AOBF algorithm and MBE-MM. 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. The `type4` instances were generally too difficult to solve exactly (running out of memory), so they have been omitted from this

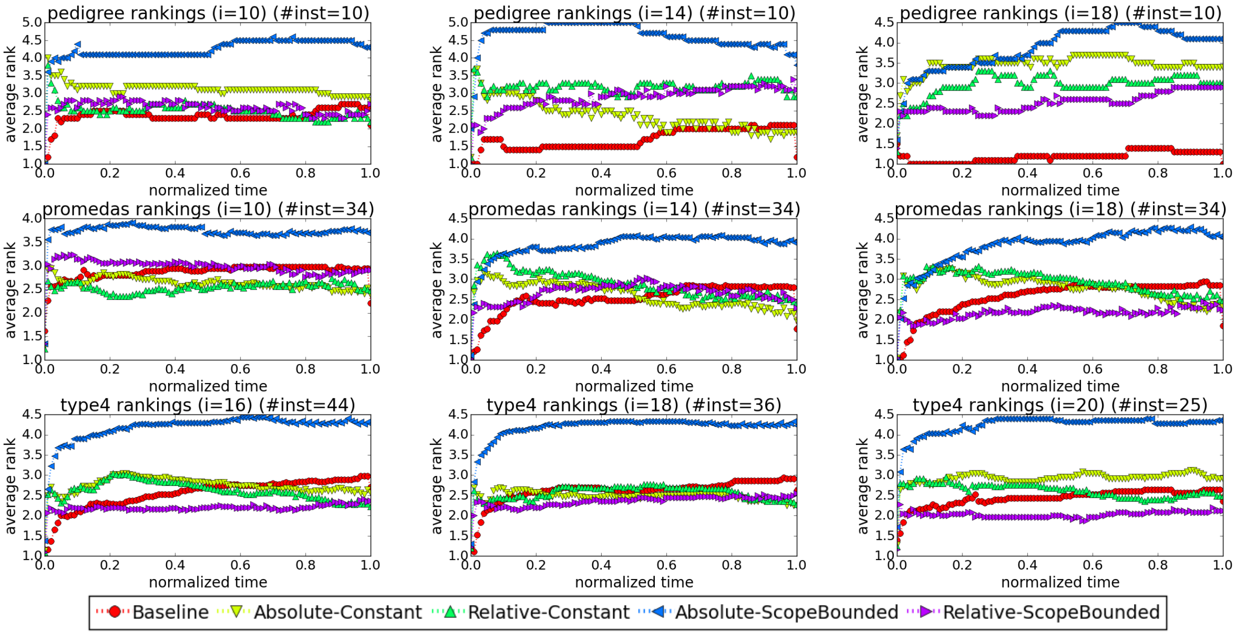


Figure 4: Average rankings over the normalized time relative to the baseline. Lower is better.

table.

We observe that the new ordering heuristic can improve over the baseline in many cases, especially when the i-bound is low. For example, on pedigree44 with and i-bound of 10, we reduce the time by a factor 1.8 when using *Absolute-ScopeBounded*. As the i-bound increases, the heuristic accuracy improves and the errors decrease, having less impact. Specifically, tip node ordering has a smaller effect since the heuristics are mostly accurate, and most expansions would not tighten the bound by much regardless of the choice of the node. However, for all of the promedas instances with an i-bound of 18, an error-based method is better than the baseline.

Anytime Lower Bounding. In **Figure 3**, we report the lower bound on the cost of the optimal solution as a function of time. A plot that is higher earlier on indicates superior performance. The first point of each line is always the bound returned by the MBE itself, recorded whenever search starts following all pre-processing. We show one instance from each of our benchmarks over 3 different i-bounds. For pedigree19, the exact solution is known and we show the evolution of the bound over time for the different secondary heuristic functions. For an i-bound of 14, we see that early on, the *Absolute-Constant* ordering heuristic results in AOBF tightening of the lower bound more quickly than for the others, beating out the rest of the orderings at around 75 seconds. For the other two instances, the optimal solution is not known, but we can compare the ordering heuristics based on their relative performance. In or_chain_8, for the i-bounds of 14, every method except the *Absolute-ScopeBounded* has superior performance overall. Lastly, type4b_140_20 at the highest i-bound (20) demonstrates that computing the ordering heuristic is cost-effective. Note that the baseline method starts search earlier at around 575 seconds, while all of the other methods start 10 to 20 seconds later due to the overhead in computing the error functions. However, within a few seconds, the bound increases much more compared to the baseline and ultimately leads to a better bound at termination when we run out of memory.

In **Figure 4**, we aggregated the results over each benchmark. 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 some instances run out of memory when computing the MBE heuristic with higher i-bounds.

Pedigree. The baseline performs the best overall, but falls behind towards the end, with the relative error based measures ranking slightly better. Inspecting the general distribution of bucket errors for pedigree instances, it is zero for many of the variables, making the extra initial preprocessing time not cost-effective.

Promedas. For $i = 10$, we see that the *Constant* methods perform similarly to the baseline at the beginning, but start to rank better with time. This is once again due to the extra preprocessing time at the start. The *ScopeBounded* methods tend to have a disadvantage due to additional overhead during search itself. Compared with the *Constant* methods, they require a table lookup as opposed to a simple retrieval of a constant. However, at $i = 18$, even though there is less error, the buckets with error have larger error function scopes and the *Relative-ScopeBounded* method is able to work well here.

Type4. For the type4 instances with $i = 16$, the *Relative-ScopeBounded* heuristic is best at the start, though its performance starts to approach that of the *Relative-Constant* method, once again due to computation overhead during search. With $i = 20$, nearly half of the instances could not run due to MBE running out of memory, but we also see that the *Relative-ScopeBounded* method maintains the best rank here for the same reasons as the promedas instances. Overall, we see that the baseline is outperformed by most of our schemes on this benchmark.

To summarize, we have the following takeaways:

1. When the solution evaluation heuristics are weak, the subproblem ordering has a larger effect.
2. Although, the scope-bounded method is more informative, it expands nodes during search at a slower rate. As a consequence, its relative performance degrades as the bound approaches the optimal solution.
3. We see that the scope-bounded method using absolute error per-

forms poorly in most cases.

4. The context-independent heuristic is easy to compute and offers the best overall trade-off.

6 CONCLUSION

The paper explores the potential of using AND/OR best-first search for generating lower bounds in an anytime fashion. And, in particular, highlights the significance of subproblem orderings heuristics (the so-called secondary evaluation function [10]) within the anytime context. The paper presents new heuristics for subproblem ordering which are based on the notion of pre-compiling information about the primary heuristic error using the notion of bucket-error and demonstrate significant impact on many instances, empirically. As far was we know there has been little focus on this aspect of AND/OR search in recent literature. This technique should be applied to any type AND/OR search, and can be extended to various memory-efficient A* variants such as IDA* which would also benefit from a better subproblem ordering evaluation function.

REFERENCES

[1] Rina Dechter, *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2013.

[2] Rina Dechter, Kalev Kask, William Lam, and Javier Larrosa, 'Lookahead with mini-bucket heuristics for mpe', in *AAAI*, (2016).

[3] M. Fishelson and D. Geiger, 'Exact genetic linkage computations for general pedigrees.', *Bioinformatics*, (2002).

[4] Alexander Ihler, Natalia Flerova, Rina Dechter, and Lars Otten, 'Join-graph based cost-shifting schemes', in *Uncertainty in Artificial Intelligence (UAI)*, 397–406, AUAI Press, Corvallis, Oregon, (August 2012).

[5] K. Kask and R. Dechter, 'Branch and bound with mini-bucket heuristics', *Proc. IJCAI-99*, (1999).

[6] K. Kask and R. Dechter, 'A general scheme for automatic search heuristics from specification dependencies', *Artificial Intelligence*, 91–131, (2001).

[7] Radu Marinescu and Rina Dechter, 'Memory intensive and/or search for combinatorial optimization in graphical models', *Artif. Intell.*, **173**(16-17), 1492–1524, (2009).

[8] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[9] Lars Otten and Rina Dechter, 'Anytime and/or depth-first search for combinatorial optimization', *AI Communications*, **25**(3), 211–227, (2012).

[10] J. Pearl, *Heuristics: Intelligent Search Strategies*, Addison-Wesley, 1984.

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

[12] Bastian Wemmenhove, JorisM. Mooij, Wim Wiegerinck, Martijn Leisink, HilbertJ. Kappen, and JanP. Neijt, 'Inference in the promedas medical expert system', in *Artificial Intelligence in Medicine*, volume 4594 of *Lecture Notes in Computer Science*, 456–460, Springer Berlin Heidelberg, (2007).