Journal of Artificial Intelligence Research 60 (2017) 287-346 Submitted 02/17; published 10/17

Residual-Guided Look-Ahead in AND/OR Search for Graphical Models

William Lam WILLMLAM@UCI.EDU
Kalev Kask KKASK@ICS.UCI.EDU
Dept. of Computer Science, Univ. of California, Irvine
Irvine, CA 92697, USA

Javier Larrosa LARROSA@CS.UPC.EDU
Dept. of Computer Science, UPC Barcelona Tech, Spain

Rina Dechter DECHTER@ICS.UCI.EDU
Dept. of Computer Science, Univ. of California, Irvine
Irvine, CA 92697, USA

Abstract

We introduce the concept of *local bucket error* for the mini-bucket heuristics and show how it can be used to improve the power of AND/OR search for combinatorial optimization tasks in graphical models (e.g. MAP/MPE or weighted CSPs). The local bucket error illuminates how the heuristic errors are distributed in the search space, guided by the mini-bucket heuristic. We present and analyze methods for compiling the local bucket-errors (exactly and approximately) and show that they can be used to yield an effective tool for balancing *look-ahead* overhead during search. This can be especially instrumental when memory is restricted, accommodating the generation of only weak compiled heuristics. We illustrate the impact of the proposed schemes in an extensive empirical evaluation for both finding exact solutions and anytime suboptimal solutions.

1. Introduction

We address the *min-sum problem* over *graphical models*, which includes the *most probable* and the *maximum a posteriori inference* in probabilistic graphical models (Pearl, 1988; Darwiche, 2009; Dechter, 2013). This problem has many applications in areas such as *protein side chain prediction, genetic linkage analysis, and scheduling* (Yanover, Schueler-Furman, & Weiss, 2008; Fishelson & Geiger, 2004; Bensana, Lemaitre, & Verfaillie, 1999).

A well-known solving approach is *depth-first branch and bound over an AND/OR search space* (AOBB) (Marinescu & Dechter, 2009a, 2009b). Besides computing the optimal solution, the algorithm also features an *anytime* behavior which provides a sequence of near-optimal solutions of improving quality along time (Otten & Dechter, 2012). The performance of AOBB largely depends on the availability of a *heuristic* function $h(n)$ which underestimates the optimal value of extending any search node n. The tightness of the heuristic has an immense impact on the performance of AOBB. A commonly used heuristic is the mini-bucket elimination (MBE) heuristic (Dechter & Rish, 2002). MBE has a control parameter known as the i-bound that trades computation time and memory for accuracy. In particular, computing the heuristic is time and space exponential in the i-bound. In general, MBE gets more accurate as the i-bound approaches a problem's induced width. In problems with high

induced width, the i-bound cannot be made close to the problem's induced width due to memory demands, yielding weak heuristics.

The goal of our research is to improve both the exact and anytime performance of AOBB when it is guided by the MBE heuristic. For that purpose we consider the well-known technique of *look-ahead*, which is known to be useful in the context of *online search algorithms* (e.g. game playing schemes, planning under uncertainty, etc.) (Geffner & Bonet, 2013; Vidal, 2004). Look-ahead improves the heuristic function $h(n)$ of a node by expanding the search tree below it and backing up the $h(n)$ values of descendants (known as a *Bellman update*). Thus, look-ahead can be seen as a secondary search embedded in the primary search. Indeed, it has been used as a way to create a version of the A* algorithm that incorporates depth-first search (Stern, Kulberis, Felner, & Holte, 2010; Bu, Stern, Felner, & Holte, 2014).

A naive implementation of look-ahead is unlikely to be effective in the context of AOBB since it is essentially a transference of the expansion of nodes from the primary search to the secondary (look-ahead) search. In this paper we address the challenge of making look-ahead cost effective. We develop the notion of *local bucket error*, which we show to be equivalent to the *residuals* in depth-1 of look-ahead. We show that local bucket errors can be computed in a pre-process, thus causing no overhead during search. We provide the algorithm and characterize its complexity in terms of a structural parameter called *pseudo-width*. When the pseudo-width indicates that computing local bucket errors is too expensive, we suggest approximation schemes. When bucket errors are computed exactly, they immediately translate to compiled depth-1 look-ahead, yielding an improved heuristic. Besides that, local bucket errors can be consulted to decide on the right depth of look-ahead that is likely to improve the heuristic and thus be more cost-effective. In fact our approach applies look-ahead selectively, only up to the depth where it is likely to improve the heuristic significantly. To facilitate this, we introduce the notion of *look-ahead subtrees* which determines the look-ahead frontier for each variable and *prunes* them individually as a pre-processing step based on the local bucket errors.

We also develop the notion of *look-ahead graphical model* which presents the look-ahead task as a min-sum sub-problem. We show that the structural complexity (i.e., width) of such a task can be characterized and determined as a pre-process. The consequence is that good look-ahead depths can be identified prior to search and an inference algorithm such as Bucket Elimination can be applied.

In most heuristic search literature, the heuristic function is treated as a *black box*. The novelty of our approach lays on a more structural exploitation of the heuristic. Our research was inspired from the observation that, in a wide spectrum of problems, the heuristic errors are not uniformly distributed in the search space. On the contrary, there are localized regions where most of the error accumulates and those regions are just a small fraction of the entire search space. The main implication is that a blind look-ahead will mainly do redundant computations (look-ahead on error-free or near-error-free regions has no effect what-so-ever).

Thus, our main contribution is to *exploit the error function structure and design a scheme that performs look-ahead selectively.* In particular, look-ahead will intensify where the heuristic error is high and decrease where it is locally low. In cases where the heuristic is known to be locally exact, we can even completely skip look-ahead. In our empirical evaluation,

we show improved runtime for finding exact solutions in many cases and more generally, improved anytime behavior compared with current state-of-the-art methods (Otten, Ihler, Kask, & Dechter, 2012).

The rest of the paper is organized as follows. In Section 2, we provide the relevant background on graphical models and AND/OR search. In Section 3, we describe our main contributions. We introduce the notion of local bucket error for MBE, establish its connection with the residuals, and present and analyze an algorithm for computing them. We then show how the local bucket errors can be used to guide look-ahead. In Section 4, we illustrate through several benchmarks how the error is distributed non-homogeneously along the search spaces. In Section 5, we provide an empirical evaluation of our selective look-ahead scheme. Section 6 concludes.

2. Background

$ch(n)$	children of node n
$c(n,m)$	cost of edge (n,m)
$h^*(n)$	optimal cost from n
$h(n)$	heuristic value
$h^d(n)$	depth d heuristic look-ahead
$res^d(n)$	depth d residual

Table 1: Notation on AND/OR search.

2.1 AND/OR Search

While many discrete optimization problems arising in AI can be solved using OR search spaces, there are some important cases for which AND/OR seems more suitable (see Table 1 for a summary of notation). In AND/OR search trees (Nilsson, 1980; Pearl, 1984) there are two types of nodes: OR nodes and AND nodes. OR nodes represent branching points where a decision has to be made, and AND nodes represent sets of (conditionally) independent sub-problems that need to be solved. In this work, we assume that children of OR nodes are AND nodes, and the children of AND nodes are OR nodes. OR nodes are always internal nodes, while AND nodes may be internal nodes, or leaves. There is a cost $c(n,m)$ associated with each edge between an OR node n and an AND node m, which represents the cost of making the corresponding decision at that branching point. Here, we will assume AND/OR trees of bounded height.

A *solution tree* is a sub-tree of the AND/OR tree that (1) includes its root, (2) if an OR node is in the solution tree, 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. The cost of a solution tree (for the min-sum problem) is the sum of costs of all its branches. A solution tree is optimal if there is no solution tree with lower cost. The task is to find an optimal solution tree.

2.1.1 Depth-First Branch and Bound

A standard way of finding optimal solution trees on bounded height AND/OR search trees or graphs is *depth-first branch and bound* (AOBB). Each search node n is associated with the subproblem below it, which is finding the optimal cost of the subproblem rooted by n. Let $h^*(n)$ denote the cost of the optimal solution of its subproblem. AOBB uses a *heuristic value* $h(n)$ which is a *lower bound* of the optimal cost of subproblem below n (that is, $h(n) \leq h^*(n)$ for all nodes n).

Algorithm 1: AOBB. The initial call, BBor$(root, \infty)$, returns the cost of the optimal solution tree. The children of a node n are denoted $ch(n)$, the cost of an edge is noted $c(n, m)$, the heuristic value of node n is denoted $h(n)$.

```
1  Function BBor(n, ub)
2  begin
3      for m ∈ ch(n) do
4          if c(n, m) + h(m) < ub then
5              ub := c(n, m)+BBand(m, ub − c(n, m))
6          end
7      end
8      return ub;
9  end

10 Function BBand(n, ub)
11 begin
12     if ch(n) = ∅ then return 0;
13     foreach m ∈ ch(n) do q(m) := h(m);
14     foreach m ∈ ch(n) do
15         if Σ_{m'∈ch(n)} q(m') ≥ ub then return ub;
16         else q(m) :=BBor(m, ub − Σ_{m'∈ch(n),m'≠m} q(m)) ;
17     end
18     return Σ_{m∈ch(n)} q(m);
19 end
```

Algorithm 1 presents pseudo-code for AOBB. In a call to BBor(n, ub) and BBand(n, ub), n is an OR node and an AND node, respectively. It represents the subproblem currently being solved. In both cases, ub is a bound on the best solution tree found so far minus the path cost and current lower bounds of all the AND ancestors' sibling OR nodes. The behavior of both functions is the same: if $h^*(n) < ub$ then it returns $h^*(n)$, else it returns ub. The initial call is BBor$(root, \infty)$ which returns the cost of the problem's optimal solution tree $h^*(root)$.

The code of both functions is very similar since both solve the current sub-problem n by recursively solving in sequence its children $ch(n)$. The only difference is that the optimal cost of an OR node is the *minimum* among its children, while the optimal cost of an AND node is the *sum* over the optimal costs of its children. In both cases, the result of each

recursive call is used to adjust the ub of the next call (lines 5 and 16, respectively). Note that in BBand(n, ub), we use a local variable $q(m)$ for each m, child of n. It initially stores $h(m)$ (line 12) but it is replaced by $h^*(m)$ as soon as it is known (line 16).

The main role of $h(n)$ is to facilitate *pruning*, which takes place in lines 4 (OR pruning) and 15 (AND pruning). In both cases, the heuristic is used to identify when it is not possible to find solutions that improve over the current threshold ub.

2.2 Background on Graphical Models

X_k, x_k	variable, assigned variable
$f_j(\cdot)$, S_j	function, scope
$\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes)$	graphical model
$G = (V, E)$	primal graph (nodes correspond to variables)
$G^*(o)$	induced graph relative to order o
$w^*(o)$	induced width relative to order o
$\mathcal{T}$	pseudo-tree (nodes correspond to variables)
$\bar{X}_k$	pseudo-tree path from root to X_k
$\mathcal{T}_k$	sub-trees rooted by $ch(X_k)$
$\mathcal{T}_{k,d}$	sub-trees rooted by $ch(X_k)$ 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
B_k, S_{B_k}	bucket associated to pseudo-tree node X_k, scope
B_k^s	mini-bucket associated to pseudo-tree node X_k
$\lambda_{k\to p}^s(\cdot)$	message computed at B_k and sent to B_p
$\Lambda_k(\cdot)$	sum of messages from B_k to $\bar{X}_p$
$h(\bar{x}_p)$	heuristic value of node $\bar{x}_p$

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

In this sub-section we give background and notation on graphical models, which is the context of this research (see Table 2 for a summary). We start by defining the concept of a graphical model and its primal graph, which captures its structure. Then we define the associated pseudo-tree which dictates the AND/OR search spaces (AND/OR tree and AND/OR graph). We also review the Bucket and Mini-Bucket Elimination (BE and MBE) algorithms and show how MBE can be used as heuristic function.

2.2.1 Graphical Models

Consider a finite set of variables $\mathbf{X}$. Let $X_k \in \mathbf{X}$ and D_k denote a variable and its domain. An arbitrary assignment of X_k to one of its domain values is noted x_k. Similarly, an assignment of a set of variables $S \subseteq \mathbf{X}$ is noted s. Thus, s is an element of the cartesian product $\prod_{X_k \in S} D_k$. An assignment of all the variables will be noted $\mathbf{x}$.

We will use f_j to denote a function returning a positive real number, and $S_j \subseteq \mathbf{X}$ its scope (i.e., $f_j : \prod_{X_k \in S_j} D_k \to \mathbb{R}^+$). We will often write $f_j(\cdot)$ (instead of $f_j(S_j)$) to emphasize that f_j is a function when not needing to refer to its scope. Assigning a function $f_j(\cdot)$ with a

tuple s_j returns a constant value. A partially assigned function, noted $f_j(\cdot|\ s)$ (with $S \subset S_j$), is a function whose scope is the set of variables in $S_j \backslash S$. Sometimes we will abuse notation and write $f_j(s)$ with $S_j \subset S$, which denotes the assignment of $f_j(\cdot)$ with the projection of s onto S_j.

A graphical model is a collection of functions over subsets of a common set of variables,

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

1. $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a finite set of variables*
2. $\mathbf{D} = \{D_1, \ldots, D_n\}$ *is a set of finite domains associated with each variable.*
3. $\mathbf{F} = \{f_1, \ldots, f_m\}$ *is a set of valued* local *functions with scopes* $S_j \subseteq \mathbf{X}$ *for all* f_j.
4. $\otimes$ *is a combination operator (typically the sum or product)*

A graphical model represents a *global* function which is the combination of all the local functions, $\otimes_{j=1}^{m} f_j(S_j)$. Graphical models are used to model, in a factorized way, complex systems that can be queried. For the purpose of this paper we will consider the combination operator to be the *sum* and we will focus on the *minimization* query. We will often omit the $\otimes$ and refer to $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$.

Definition 2 (min-sum problem). *Given a graphical model* $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, *the min-sum problem is the task of computing the optimal assignment of its variables with respect to the global function. Namely,*

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

where $\mathbf{x}$ *ranges over the possible assignments of* $\mathbf{X}$.

For instance, when the variables are random variables, the combination operator is the product, and the local functions are conditional probability tables (plus some additional conditions) the graphical model is a *Bayesian network* (Pearl, 1988; Darwiche, 2009). If a negative log transformation is applied to the local functions, the min-sum problem corresponds to the MPE/MAP query (Dechter, 2013). Another well-known example occurs when variables correspond to decisions, the combination operator is the sum, and local functions represent local costs of taking the decisions. Then the graphical model is a constraint optimization problem (or weighted constraint satisfaction problem) (Dechter, 2003).

Each graphical model can be associated with graph which makes explicit conditional independencies,

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* X_k *and* $X_{k'}$ *that appear in the same scope* S_j *of a function* $f_j \in \mathbf{F}$.

Consider a graphical model with variables indexed from A to G and functions $\mathbf{F} = \{f_1(A), f_2(A,B), f_3(A,D), f_4(A,G), f_5(B,C), f_6(B,D), f_7(B,E), f_8(B,F),\ f_9(C,D),$ $f_{10}(C,E), f_{11}(F,G)\}$. Its primal graph is shown in **Figure 1**.

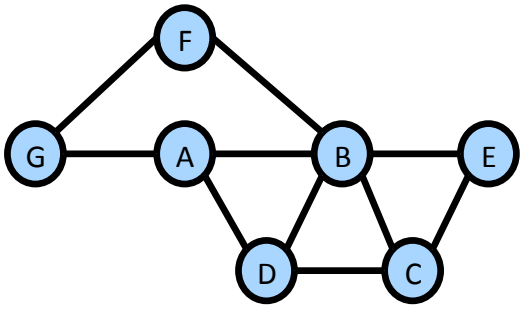


Figure 1: A primal graph of a graphical model with 7 variables.

The complexity of the min-sum problem and several other queries for a given graphical model can be characterized by the *induced width* of its associated primal graph, which is built upon several definitions we include here.

Definition 4 (induced width, Dechter, 2013). *Given a primal graph $G = (V, E)$, an* ordered graph *is a pair (G, o), where o 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, o), *denoted $w(o)$, 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, o) is an ordered graph (G^*, o), where G^* is obtained from G as follows: the nodes of G are processed from last to first along o. When a node X_k is processed, all of its parents are connected.*
- *The* induced width of an ordered graph (G, o), *denoted $w^*(o)$, is the maximum number of parents a node has in the induced ordered graph (G^*, o).*
- *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 (Marinescu & Dechter, 2009a; Dechter, 2013).

2.3 AND/OR Search Spaces for Graphical Models

We show now how the primal graph of a graphical model allows to identify conditionally independent problems and introduce pseudo-trees which make these independencies explicit for a particular (fixed) ordering of assigning the variables.

Definition 5 (pseudo tree $\mathcal{T}$). *(Dechter & Mateescu, 2007) 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 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.*

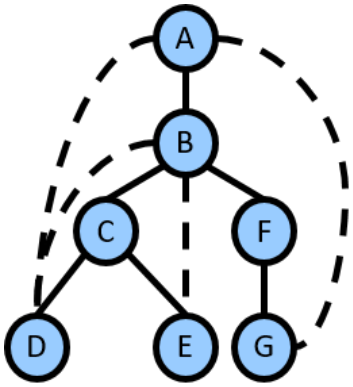


Figure 2: A pseudo tree for the running example. Solid arcs form the main tree structure and dotted arcs the back-arcs.

We say that an ordering o of the variables is valid for a pseudo tree if it is top-down with respect to it.

Solid arcs denote the pseudo tree structure and dotted arcs are the back-arcs. $ch(X_k)$ denotes the children of X_k in the pseudo tree, $\bar{X}_k$ denotes the set of variables in the path from the root to X_k, and $\mathcal{T}_k$ denotes the set of subtrees rooted by the children of X_k. **Figure 2** shows a pseudo tree for our running example. Observe that $\mathcal{T}_k$ represents subproblems that will become independent if the variables in $\bar{X}_k$ are assigned. This independency occurs no matter which values are assigned to the variables. However, each assignment induces different subproblems. Next, we show that the conditional independencies uncovered by the pseudo-tree allow to define AND/OR search spaces.

Definition 6 (Graphical Model AND/OR search tree). *(Dechter & Mateescu, 2007) Given a graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ and a pseudo tree $\mathcal{T}$, its AND/OR search tree $S_{\mathcal{T}}$ is defined as:*

- *The root node is an OR node labeled by the variable at the root of $\mathcal{T}$.*
- *The children of an OR node labeled X_k are AND nodes labeled with the different value assignments $x_k \in D_k$;*
- *The children of an AND node x_k are OR nodes labeled with the children of X_k in the pseudo-tree $\mathcal{T}$*

Each edge from an OR node X_k to an AND node x_k represents a variable assignment. 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 $\bar{x}_k$. Solution trees of $S_{\mathcal{T}}$ correspond to complete assignments of the variables in the graphical model. The size of an AND/OR search tree is exponential in the height of the pseudo-tree $\mathcal{T}$ (Dechter & Mateescu, 2007).

The AND/OR search tree has arc costs for edges from OR to AND nodes,

Definition 7 (arc cost $c(X_k, x_k)$). *The cost $c(X_k, x_k)$ of the arc (X_k, x_k) is the sum of all the functions in the graphical model whose scope includes X_k and is fully assigned by the values specified along the path from the root to node x_k.*

The cost of a solution tree which corresponds to the cost of the assignment as given by the objective function is the sum of the costs of its arcs. Thus, the optimal solution tree corresponds to the solution of the min-sum problem. A direct consequence is that the min-sum problem of a graphical model can be solved using AOBB (**Algorithm 1**).

A more compact search space can be obtained if identical subproblems in the AND/OR tree are merged, producing an AND/OR graph (Dechter & Mateescu, 2007). 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 $E' \cup E$ (i.e. the arcs of the pseudo-tree and the back-arcs).*

A *context-minimal AND/OR search graph* $C_{\mathcal{T}}$ is obtained from the AND/OR tree merging OR nodes having the same context. The size of $C_{\mathcal{T}}$ is bounded exponentially in the induced width of (G, o) for any valid order with respect to $\mathcal{T}$. The AOBB algorithm (**Algorithm 1**) can be adapted to work on AND/OR search graphs (Marinescu & Dechter, 2009a). Then it can be shown that its time and space complexity is exponential in the induced width $w^*(o)$.

Example. **Figure 3a** shows the same pseudo tree as in **Figure 2**, but annotated with contexts. **Figure 3b** shows the corresponding context-minimal AND/OR search graph to this pseudo tree. Since the context of variable E is only over B, C, the OR nodes have been merged with respect to A, which is not in the context. Similarly, the OR nodes of G are merged with respect to B. The *solution tree* for the assignment $(A = 0, B = 1, C = 1, D = 0, E = 0, F = 0, G = 0)$ is highlighted in the same figure.

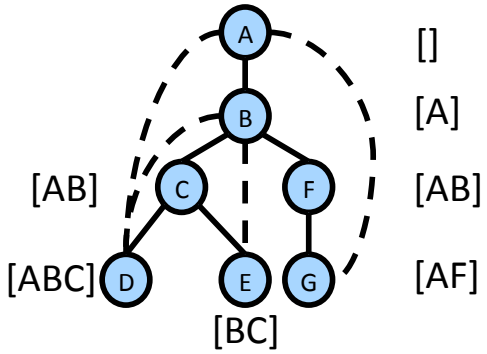
2.4 Mini-Bucket Elimination Heuristics

The most commonly used heuristic for AND/OR search in the literature is the mini-bucket elimination (MBE) heuristic (Dechter & Rish, 2002). It is based on a relaxation of the exact bucket elimination (BE) algorithm (Dechter, 1999) by running BE on a problem with duplicated variables.

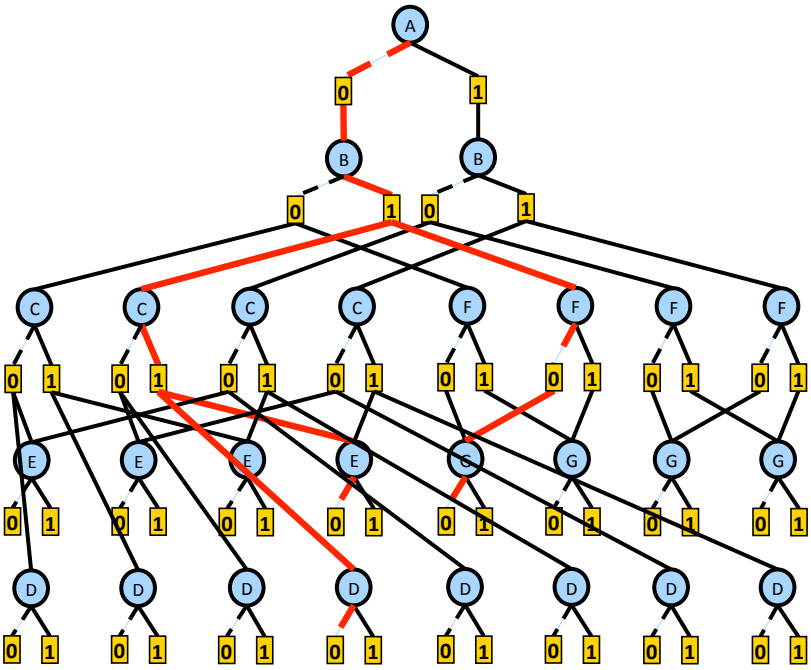
2.4.1 Bucket Elimination

Bucket elimination works relative to the same pseudo tree that defines the AND/OR search graph. Each variable X_k of $\mathcal{T}$ is associated with a bucket B_k which is a set of functions. A function $f_j(\cdot)$ from $\mathbf{F}$ is placed into B_k if X_k is the deepest variable in $\mathcal{T}$ such that $X_k \in S_j$.

Each bucket B_k is then processed, bottom-up, from the leaves of the pseudo tree to the root by computing a new function, known as a *message*, $\lambda_{k\to p}(\cdot) = \min_{x_k}(\sum_{f\in B_k} f(\cdot) + \sum_{X_q\in\mathcal{T}_k} \lambda_{q\to k}(\cdot))$, where p is the parent of k in the pseudo-tree, $f(\cdot)$ denotes original functions and $\lambda_{q\to k}(\cdot))$ denotes messages received in the bucket. This message is then added to bucket B_p. The scope of the message is a subset of $\bar{X}_p$. The scope of a bucket B_p, noted $Scope(B_p)$, is the union of the scopes of its functions. It can be seen that the size of $Scope(B_p)$ corresponds to the induced width of X_p with any ordering o valid for the pseudo-tree (i.e, top-down). Due to the bottom-up processing schedule, a bucket is never processed until it receives messages from all of its children. At the end of processing, the message



(a) A pseudo tree for the running example, annotated with contexts.



(b) Example AND/OR search graph. A solution tree is highlighted.

Figure 3

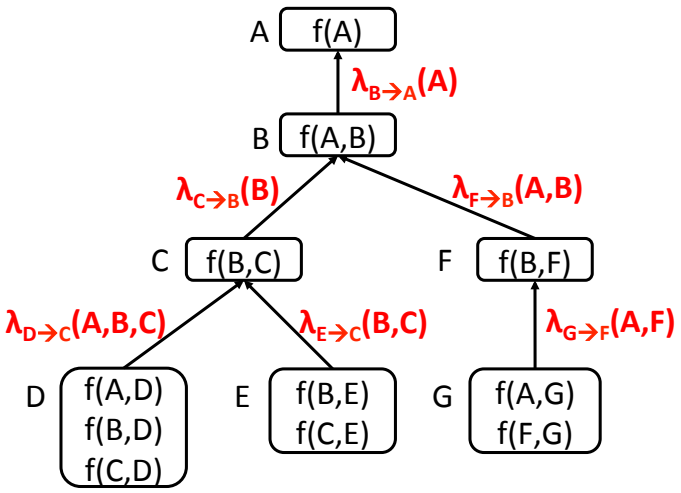


Figure 4: Illustration of bucket elimination.

generated by the root bucket is a constant which is the optimal min-sum value. Message $\lambda_{k\to p}(\cdot)$ compiles the optimal solution for all subproblems on $\mathcal{T}_k$ variables, so BE in fact provides exact heuristics in the context of search.

The BE algorithm's time and space complexity is exponential in the problem's induced width $w^*(o)$ relative to any ordering o that is top-down with respect to $\mathcal{T}$. BE is a variant of non-serial dynamic programming (Bertele & Brioschi, 1972).

We illustrate the computation on our example problem in **Figure 4**. It can clearly be seen here that bucket elimination breaks the entire optimization problem of the graphical model into smaller subproblems, then combines results by sending messages to parents. This has been shown to be equivalent to exploring the context-minimal AND/OR search graph in a *bottom-up* fashion, given certain conditions (Mateescu & Dechter, 2005).

2.4.2 Mini-Bucket Elimination

If $w^*(o)$ is very large, then using BE to solve the *min-sum* problem is infeasible. Mini-Bucket Elimination (MBE) is a relaxation of BE that bounds the induced width of the problem via a parameter known as the i-bound (Dechter & Rish, 2002). The main difference is in how functions are processed inside buckets. MBE relaxes the problem by partitioning buckets into *mini-buckets* $B_k = B_k^1 \cup \ldots \cup B_k^{r_k}$ whose scope sizes do not exceed the i-bound. Each mini-bucket then generates its own message that is sent to its closest ancestor bucket B_p such that X_p is in the scope of the message. We denote these messages as $\lambda_{k\to p}^s(\cdot)$, where $s \in \{1, \ldots, r_k\}$ is the mini-bucket index. The scope of a $\lambda_{k\to p}^s(\cdot)$ message is a subset of $\bar{X}_p$ of size at most i. The partitioning process can be interpreted as a process of duplicating variables in the problem and optimizing over the copies independently. Therefore, MBE generates lower bounds on the min-sum problem.

The scope's size of a bucket B_k after having received all the messages is what we will define as pseudo-width in Section 3. As we will see, it will be important to characterize the complexity of computing local bucket errors

Algorithm 2: Mini-Bucket Elimination (Dechter & Rish, 2002)

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 $\lambda^s_{q\to p}(\cdot)$

1 **foreach** $X_p \in \mathbf{X}$ in bottom up order according to $\mathcal{T}$ **do**
2 $\quad B_p := \{f_j(\cdot) \in \mathbf{F} \mid X_p \in S_j\}$
3 $\quad \mathbf{F} := \mathbf{F} - B_p$
4 $\quad$ Put all generated messages $\lambda^s_{q\to p}(\cdot)$ in B_p
5 $\quad$ Partition bucket B_p into mini-buckets $B^1_p, \ldots, B^{r_p}_p$ with scope bounded by the i-bound
6 $\quad$ **foreach** $s = 1..r_p$ **do**
7 $\quad\quad$ Let X_a be closest ancestor variable of X_p in B^s_p
8 $\quad\quad$ Generate message $\lambda^s_{p\to a}(\cdot) := \min_{x_p}(\sum_{f_j \in B^s_p} f_j(\cdot) + \sum_{\lambda^{s'}_{q\to p} \in B^s_p} \lambda^{s'}_{q\to p}(\cdot))$
9 $\quad$ **end**
10 **end**
11 **return** All λ-messages generated (root message is the min-sum lower bound)

We provide details in **Algorithm 2**. The main loop (lines 1-9) partitions the bucket into mini-buckets and generates the λ messages used in the above expressions. The message computed by the root variable is a lower bound on the optimal solution of the graphical model. In general, each message is possibly not exact if the variable it was generated from or its descendants had bucket partitions with more than one mini-bucket (i.e. approximation errors propagate up to ancestors). When the i-bound equals the induced width $w^*(o)$, there is no need to partition, so the algorithm reduces to bucket elimination. In general, a higher i-bound leads to a more accurate approximation, but increases in accuracy are not guaranteed.

MBE's messages can be used to construct a heuristic for search (Kask & Dechter, 1999). Heuristics generated from the messages of mini bucket elimination are called *static* heuristics since they require the execution of MBE to generate all of the messages as a pre-processing step. As a result the heuristic is pre-compiled and the search only needs to make table look-ups.

Definition 9 (MBE heuristic). *Let $\bar{x}_p$ be a partial assignment and $\bar{X}_p$ be the set of corresponding instantiated variables. $\Lambda_{(k,p)}$ denotes the sum of the messages sent from bucket B_k to B_p or its ancestors.*

$$\Lambda_{(k,p)}(\cdot) = \sum_{X_q \in \bar{X}_p} \sum_{s=1..r_k} \lambda^s_{k\to q}(\cdot) \tag{1}$$

Note that $\Lambda_{(k,p)}(\cdot)$ is a function whose scope is a subset of $\bar{X}_p$. The heuristic value for $\bar{x}_p$ is based on messages sent from buckets bellow $\bar{X}_p$ to its ancestors. Formally,

$$h(\bar{x}_p) = \sum_{X_k \in \mathcal{T}_p} \Lambda_{(k,p)}(\bar{x}_p) \tag{2}$$

where $\mathcal{T}_p$ denotes the set of variables in the pseudo subtree rooted by X_p, excluding X_p.

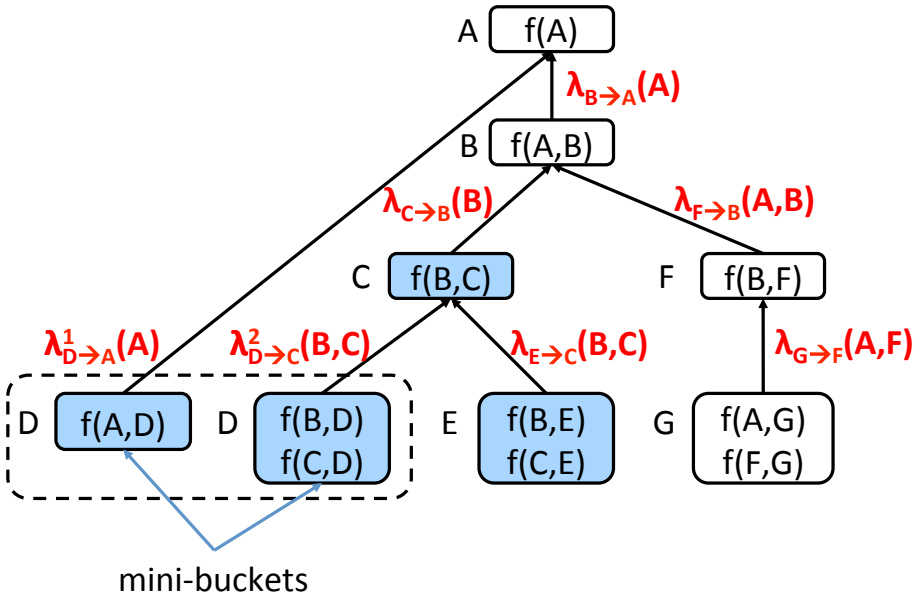


Figure 5: Example of mini-bucket elimination on the running example using an i-bound of 3. The index of mini-buckets is omitted when there is no partitioning.

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

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 the functions $f(A, D), f(B, D), f(C, D)$ which all contain that variable. However, the total scope size here is 4, which exceeds the i-bound of 3. Therefore, we partition it into two mini-buckets and each generates a separate λ 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.

3. Look-Ahead for AND/OR Search in Graphical Models

$\mathcal{T}_{p,d}$	depth d look-ahead subtree of X_p
$\mathcal{M}(\bar{x}_p, d)$	depth d look-ahead graphical model relative to assignment $\bar{x}_p$
$w_{p,d}$	induced width of look-ahead graphical model $\mathcal{M}(\bar{x}_p, d)$
$E_k(\cdot)$	local bucket error
$\tilde{E}_k$	average local bucket error
$\hat{E}_k$	sampled local bucket error
$\mathcal{T}^{\epsilon}_{p,d}$	pruned look-ahead subtree

Table 3: Notation on look-ahead.

This section contains the main contributions of our work. We present and analyze the look-ahead principle for AND/OR search in graphical models when using the MBE heuristic. In the first subsection we rephrase look-ahead as a min-sum problem over a graphical (sub)

problem. In the second subsection we perform a residual analysis and present a method that identifies the look-ahead relevant regions of the search space that can be used to skip redundant look-ahead. A summary of the notation introduced in this section appears in **Table 3**.

3.1 Look-Ahead

As mentioned in the background, the AOBB algorithm's performance may improve by having more accurate heuristic values. One way to achieve this improvement is by look-ahead, which is especially attractive because it does not increase the space complexity of MBE. The idea is to replace the $h(n)$ value of a node by the minimum value among all successors to a certain depth d. Look-ahead has been defined in the OR case in various contexts such as games or planning (Russell & Norvig, 2009; Vidal, 2004). A natural generalization to the AND/OR case follows. In our definition we take into account that only OR nodes represent branching points (i.e., alternatives). Therefore, the notion of depth is in terms of OR nodes, only. Formally,

Definition 10 (AND/OR look-ahead). *The depth d look-ahead of an AND node n is*

$$h^d(n) = \begin{cases} \sum_{m \in ch(n)} \min_{b \in ch(m)} \{c(m,b) + h^{d-1}(b)\} & d > 0 \\ h(n) & d = 0 \end{cases}$$

A related notion that we will be using later is that of *residual*. The residual measures the gain produced by look-ahead.

Definition 11 (residual). *The depth d* residual *of node n is*

$$res^d(n) = h^d(n) - h(n)$$

3.2 The Look-Ahead Graphical Model

We pointed out in Section 2 that looking ahead is like performing a secondary search inside of the primary search and backing up heuristic values of the expanded nodes. Next, we show that in the context of graphical models, looking-ahead corresponds to solving a min-sum problem over a graphical sub-model. Consequently, it is possible to characterize the induced width (and therefore the complexity) of such sub-models. The analysis depends only on the node's depth and the look-ahead depth, but it does not depend on the actual assignment. Therefore this induced width can be computed for each variable before search.

As a first step, we define the depth d *look-ahead subtree* for variable X_p.

Definition 12 (look-ahead subtree). *Consider pseudo tree $\mathcal{T}$ over a graphical model $\mathcal{M}$ and a variable X_p. The depth d* look-ahead subtree *for variable X_p, noted $\mathcal{T}_{p,d}$, is the subtree formed by the descendants of X_p in $\mathcal{T}$ that are no more than depth d away from X_p.*

The look-ahead subtree shows which variables in the AND/OR search space are considered in the look-ahead computation. In our running example pseudo tree (**Figure 6**) $\mathcal{T}_{B,1}$ is the shaded region, meaning that if B is the last variable assigned in a node, the depth-1 look-ahead will minimize over a search space with respect to variables C and F. Next we define the *look-ahead graphical model*, which captures the look-ahead computation.

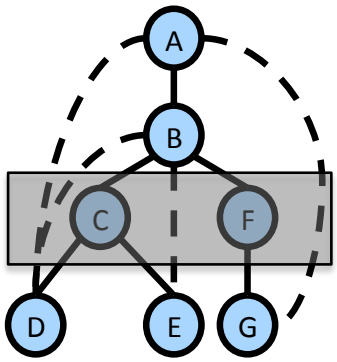


Figure 6: Look-ahead subtree example for $\mathcal{T}_{B,1}$ (shaded region)

Definition 13 (look-ahead graphical model at node $\bar{x}_p$). *Consider a graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ with pseudo-tree $\mathcal{T}$, the set of messages $\lambda^s_{j\to k}$ generated by MBE(i) along the pseudo tree. Given a partial assignment $\bar{x}_p$, the depth d look-ahead graphical model $\mathcal{M}(\bar{x}_p, d) = (\mathbf{X}_{p,d}, \mathbf{D}_{p,d}, \mathbf{F}_{p,d})$ is defined by,*

- *Variables $\mathbf{X}_{p,d}$: variables in the look-ahead tree $\{X_k |\ X_k \in \mathcal{T}_{p,d}\}$,*
- *Domains $\mathbf{D}_{p,d}$: original domains $\{D_k |\ X_k \in \mathcal{T}_{p,d}\}$,*
- *Functions $\mathbf{F}_{p,d}$:*
  - *Original functions that were originally placed in the buckets of $\mathcal{T}_{p,d}$, possibly partially assigned by $\bar{x}_p$,*
$$\{f(\cdot|\bar{x}_p) |\ X_k \in \mathcal{T}_{p,d}, f \in B_k\}$$
  - *Messages sent from buckets below $\mathcal{T}_{p,d}$ to buckets in $\mathcal{T}_{p,d}$ by the MBE algorithm, possibly partially assigned by $\bar{x}_p$,*
$$\{\lambda^s_{j\to k}(\cdot|\bar{x}_p) |\ X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}, s \in 1, \ldots, r_j, X_k \in \mathcal{T}_{p,d}\}$$

Clearly, $\mathcal{T}_{p,d}$ is a valid pseudo tree for $\mathcal{M}(\bar{x}_p, d)$. Note that the induced width of $\mathcal{M}(\bar{x}_p, d)$ along $\mathcal{T}_{p,d}$, denoted $w_{p,d}$, does not depend on the partial assignment $\bar{x}_p$. It only depends on the path of assigned variable $\bar{X}_p$ and the look-ahead depth d. Therefore, it can be identified prior to search.

The min-sum problem of $\mathcal{M}(\bar{x}_p, d)$ is therefore

$$L^d(\bar{x}_p) = \min_{\bar{x}_{p,d}} \left\{ \sum_{\substack{X_k \in \mathcal{T}_{p,d} \\ f \in B_k}} f(\bar{x}_{p,d}|\bar{x}_p) + \sum_{\substack{X_k \in \mathcal{T}_{p,d}, X_j \in \mathcal{T}_p - \mathcal{T}_{p,d} \\ s \in 1,\ldots,r_j}} \lambda^s_{j\to k}(\bar{x}_{p,d}|\bar{x}_p) \right\} \tag{3}$$

where $\bar{x}_{p,d}$ denotes an arbitrary assignment of all the variables in $\mathcal{T}_{p,d}$.

Next we show that $L^d(\bar{x}_p)$ is the task required to compute look-ahead when the MBE heuristic is used,

Proposition 1 (look-ahead value for MBE heuristic in graphical models). *Consider a graphical model $\mathcal{M}$, a pseudo-tree $\mathcal{T}_{p,d}$ and its associated AND/OR search graph. If the MBE heuristic guides the search, the depth d look-ahead value of partial assignment $\bar{x}_p$ (Definition 10) satisfies,*

$$h^d(\bar{x}_p) = L^d(\bar{x}_p) + \sum_{X_k \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_{(k,p)}(\bar{x}_p) \tag{4}$$

Proof. See Appendix A. □

Note that the second term in Equation 4 contains all messages sent from buckets below $\mathcal{T}_{p,d}$ to buckets in $\bar{X}_p$. Note as well that all these messages are constant values, since they are completely assigned by $\bar{x}_p$, and therefore irrelevant in terms of the optimization task. Therefore, computing the look-ahead at node $\bar{x}_p$ is equivalent to computing $L^d(\bar{x}_p)$, and it can be done with BE in time and space exponential on $w_{p,d}$. In those levels of X_p where the width is smaller than the depth (i.e., $w_{p,d} < d$) exact inference (i.e., BE) is more efficient than search.

Computing depth-d AND/OR look-ahead, even with bucket elimination, can be computationally expensive. Clearly, look-ahead is worthless if it does not increase and thus improve the accuracy of the heuristic value. Recall that the gain produced by the look-ahead is the so-called *residual* (Definition 11). We next analyze depth 1 residuals and show how they can be used to approximate residuals of higher depth.

We start by relating the residual's expression to $L^d(\bar{x}_p)$,

Proposition 2 (AND/OR depth-d residual for MBE). *Consider a graphical model $\mathcal{M}$, a pseudo-tree $\mathcal{T}_{p,d}$ and its associated AND/OR search graph. If the MBE heuristic (Definition 9) guides the search, the depth d residual at $\bar{x}_p$ (Definition 11) satisfies*

$$res^d(\bar{x}_p) = L^d(\bar{x}_p) - \sum_{X_k \in \mathcal{T}_{p,d}} \Lambda_{(k,p)}(\bar{x}_p) \tag{5}$$

Proof. From Definition 11, $res^d(n) = h^d(n) - h(n)$. Replacing h^d and h and using Proposition 1 and Equation 2 respectively, we obtain the expresion above. □

Note that the subtracted expression in Equation 5 is a constant. Therefore, as could be expected, computing the residual requires computing $L^d(\bar{x}_p)$. We therefore propose to approximate depth-d residuals using a sum of depth-1 residuals.

Proposition 3. *Given a node n, let N_k denote all nodes that are k-levels away from n in the search graph. Then we have*

$$res^d(n) \geq \sum_{k=0}^{d-1} \min_{n_k \in N_k} res^1(n_k)$$

Proof. See Appendix B. □

Corollary 1. *For a given level j in a depth-d residual, if $res^1(n_j) = 0$ for all nodes $n_j \in N_j$, clearly,*

$$res^d(n) \geq \sum_{k=0}^{d-1} \min_{n_k \in N_k} res^1(n_k) = \sum_{k=0, k\neq j}^{d-1} \min_{n_k \in N_k} res^1(n_k)$$

This suggests that depth-1 residuals can be informative when they are large, pointing out which levels are likely to contribute to a depth-d look-ahead. Furthermore, if the depth-1 residuals for all nodes at a particular level are 0, they have a null contribution. Since depth-1 residuals can informative for depth-d look-ahead, we next analyze depth-1 residuals for MBE heuristics. We show that it corresponds to a notion of *local bucket error* of MBE, to be defined next.

3.3 Local Bucket Error

We start by comparing the message that a particular bucket would have computed without partitioning (called the *exact bucket message* $\mu_k^*(\cdot)$) to that of the sum of the messages computed by the mini-buckets of the bucket (called the *combined mini-bucket message* $\mu_k(\cdot)$). We define these notions below.

Definition 14 (combined bucket and mini-bucket messages). *Given a mini-bucket partition $B_k = B_k^1 \cup \ldots \cup B_k^{r_k}$, we define the combined mini-bucket message at B_k,*

$$\mu_k(\cdot) = \sum_{s=1}^{r_k} \left(\min_{x_k} \left(\sum_{f \in B_k^s} f(\cdot) + \sum_{\lambda_{p\to k}^s \in B_k^s} \lambda_{p\to k}^s(\cdot) \right) \right) \tag{6}$$

where f and λ denote original functions and messages, respectively. In contrast, the exact bucket message without partitioning at B_k is

$$\mu_k^*(\cdot) = \min_{x_k} \left(\sum_{f \in B_k} f(\cdot) + \sum_{\lambda_{p\to k}^s \in B_k} \lambda_{p\to k}^s(\cdot) \right) \tag{7}$$

Note that although we say that $\mu_k^*(\cdot)$ 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 15 (local bucket error of MBE). *Given a completed run of MBE, the local bucket error function at B_k denoted $E_k(\cdot)$ is*

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

The scope of $E_k(\cdot)$ is the set of variables in bucket B_k excluding X_k.

Next, we show that depth-1 residual corresonds to the sum over children variables of local bucket errors of MBE.

Theorem 1 (equivalence of residuals and local bucket errors). *Assume an execution of MBE(i) along $\mathcal{T}$ yielding heuristic $h(\bar{x}_p)$, then for every $\bar{x}_p$,*

$$res^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} E_k(\bar{x}_p) \tag{8}$$

We first present the following lemmas which relates μ_k (Equation 6) to the MBE heuristic at the parent of X_x, and relates μ_k^* (Equation 7) to the depth 1 look-ahead also at the parent of X_k.

Lemma 1. *If X_k is a child of variable X_p, then $\Lambda_{(k,p)}(\cdot) = \mu_k(\cdot)$.*

Proof. $\Lambda_{(k,p)}(\cdot)$ is the sum of messages that MBE(i) sends from B_k to the buckets of variables in $\bar{X}_p$. Since X_p is the parent of X_k, $\Lambda_{(k,p)}(\cdot)$ is the sum of *all* the messages sent from B_k, which is the definition of $\mu_k(\cdot)$. □

Lemma 2. $L^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} \mu_k^*(\bar{x}_p)$.

Proof. Given the definition of $L^1(\bar{x}_p)$ (Equation 3, with $d = 1$), we can push the minimization into the summation, yielding:

$$L^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} \min_{x_k} \left\{ \sum_{f \in B_k} f(x_k|\bar{x}_p) + \sum_{X_j \in \mathcal{T}_p - ch(X_p)} \lambda^s_{j \to k}(x_k|\bar{x}_p) \right\} \tag{9}$$

The set of functions inside each $\min_{x_k}$ are, original functions or messages having X_k in their scope and possibly having ancestors of X_k. This is the definition of the exact bucket message (Equation 7) so we obtain

$$L^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} \mu_k^*(\bar{x}_p)$$

□

Proof of Theorem 1. From **Proposition 2** given $d = 1$, we have

$$res^1(\bar{x}_p) = L^1(\bar{x}_p) - \sum_{X_k \in ch(X_p)} \Lambda_{(k,p)}(\bar{x}_p)$$

By applying **Lemma 2** and **Lemma 1** to the first and second terms respectively, we obtain

$$\begin{aligned} res^1(\bar{x}_p) &= \sum_{X_k \in ch(X_p)} \mu_k^*(\bar{x}_p) - \sum_{X_k \in ch(X_p)} \mu_k(\bar{x}_p) \\ &= \sum_{X_k \in ch(X_p)} (\mu_k^*(\bar{x}_p) - \mu_k(\bar{x}_p)) \end{aligned}$$

Yielding (**Definition 15**),

$$res^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} E_k(\bar{x}_p) \tag{10}$$

□

Corollary 2. *When a bucket is not partitioned into mini-buckets, its local bucket error is 0, therefore it does not contribute to the look-ahead value of its parent.*

Establishing this equivalence between the depth-1 residuals and local bucket error is useful as each bucket corresponds to a particular variable and look-ahead is based on the look-ahead subtree (**Definition 12**), which is defined in terms of these variables.

3.4 Computing Local Bucket Errors

Now that we established that local bucket errors can be used to assess the impact of depth-1 look-ahead at a particular variable, we present an algorithm for computing them in a pre-processing step before search begins and analyze it.

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 $E_k(\cdot)$ for each bucket B_k

Initialization: Run $MBE(i)$ w.r.t. $\mathcal{T}$.
foreach $X_k \in \mathbf{X}$ in bottom-up order w.r.t. $\mathcal{T}$ **do**
  Let $B_k = B_k^1 \cup \ldots \cup B_k^{r_k}$ be the partition used by $MBE(i)$
  $\mu_k(\cdot) = \sum_{s=1..r_k} \min_{x_k} (\sum_{f \in B_k^s} f(\cdot) + \sum_{\lambda_{p\to k}^s \in B_k^s} \lambda_{p\to k}^s(\cdot))$
  $\mu_k^*(\cdot) = \min_{x_k} (\sum_{f \in B_k} f(\cdot) + \sum_{\lambda_{p\to k}^s \in B_k} \lambda_{p\to k}^s(\cdot))$
  $E_k(\cdot) = \mu_k^*(\cdot) - \mu_k(\cdot)$
end
return E functions

Algorithm 3 (LBEE) computes the local bucket error $E_k(\cdot)$ for each bucket B_k. Following the execution of $MBE(i)$, 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 $\mu_k(\cdot)$, the exact bucket message $\mu_k^*(\cdot)$, and the error function $E_k(\cdot)$. The complexity of processing each bucket is exponential in the scope of the bucket following the execution of $MBE(i)$. The total complexity is therefore dominated by the largest scope of the output buckets. We call this number the *pseudo-width*.

Definition 16 (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 2 (complexity of LBEE). *The time and space complexity of LBEE is $O(nz^{psw(i)})$, where n is the number of variables, z bounds the domain size, and $psw(i)$ is the pseudo-width along $\mathcal{T}$ relative to $MBE(i)$.*

The pseudo-width lies between the width $w(o)$ and the induced width $w^*(o)$ of a pseudo-tree ordering o and grows with the i-bound. When the i-bound of $MBE(i)$ is large, computing the local errors may be intractable.

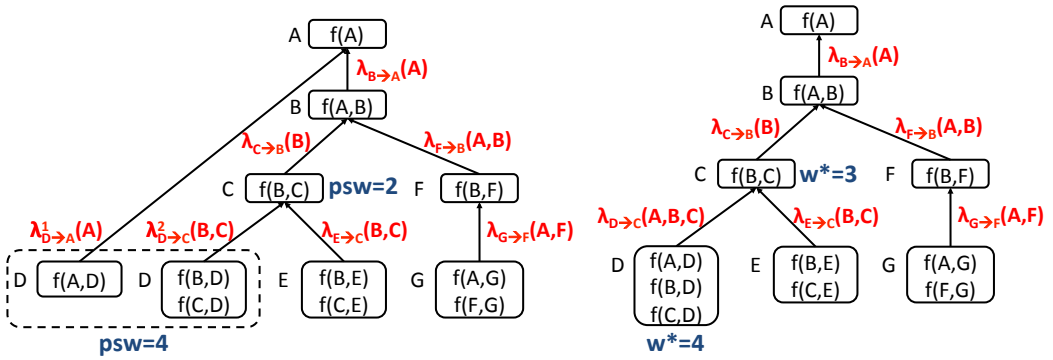


Figure 7: Example to illustrate the concept of pseudo-width. Left: the mini-bucket tree, right: the exact bucket tree. Buckets C and D in both are annotated with their pseudo-width for MBE and induced width for BE.

Example. In **Figure 7**, we illustrate the concept of pseudo-width and also relate it to LBEE with our running example. We consider the buckets for variable C and D in this example. On the left-hand side, we show the mini-bucket tree for an i-bound of 3 and on the right-hand side, we show the bucket tree of exact bucket elimination. In the mini-bucket tree, we annotate C and D with their pseudo-width. For the bucket tree, since the pseudo-width is equal to the induced width, C and D are annotated with their induced width. Starting with processing bucket D, computing the exact message μ^*_D is like treating the bucket as if it was not partitioned (e.g., the form of bucket D on the right-hand side figure). Therefore, its pseudo-width is 4, which is greater than the i-bound of 3. However, when moving to process bucket C, we still use the message obtained by MBE ($\lambda_{D\to C}(B,C)$). Thus, the pseudo-width of bucket C is 2 rather than being equal to the induced width in the bucket tree 3. The distinction here is that unlike in exact bucket elimination, the complexity of LBEE stays local. Overall, we illustrate here that the pseudo-width lies between the i-bound and induced width.

3.4.1 Approximating Local Bucket Errors

Indeed, since the time and space complexity of *LBEE* may be higher than that of *MBE* itself, sometimes it may not be practical. Therefore, we consider *sampling* and subsequently *aggregating* the local bucket error functions for each variable as an approximation. The goal here is to obtain an efficiently computable metric for each variable, which we will then use to inform us about the impact of look-ahead at each variable. For the sake of simplicity in the analysis, we will assume that all variable domains have size z.

We first address the space complexity with the following:

Definition 17 (average local error). *Consider bucket B_k, with scope S_{B_k}, its average local error $\tilde{E}_k$ is the average value of the error function,*

$$\tilde{E}_k = \frac{1}{z^{|S_{B_k}|}} \sum_{s_{B_k}} E_k(s_{B_k}) \tag{11}$$

Computing $\tilde{E}_k$ takes $O(z^{|S_{B_k}|})$ time per variable, but only $O(1)$ space to store. Clearly, we also do not lose any information if $\tilde{E}_k$ turns out to be 0, so it is sufficient to conclude in this case that performing look-ahead on variable X_k yields no benefit. Otherwise, we have an approximation for all assignments to $\bar{X}_k$.

An alternative measure is the *average relative local error*, computed by dividing each $E_k(s_{B_k})$ term by the exact bucket message $\mu^*_k(s_{B_k})$. 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.

Sampling Local Errors The average local error may still require significant time to compute, because we would still need to enumerate over all the possible assignments of the variables in the scope, as mentioned above. To address this, we can sample rather than enumerate. We can draw samples from a uniform distribution over the domain of the error function's scope and finally average over the samples to approximate the average local error $\tilde{E}_k$.

Definition 18 (sampled average local error). *Consider bucket B_k, with scope S_{B_k}, its sampled average local error $\hat{E}_k$ is the average value over a uniformly generated sample of its entries,*

$$\hat{E}_k = \frac{1}{\#samples} \sum_{s_{B_k}} E_k(s_{B_k}) \quad s_{B_k} \sim U(S_{B_k}) \tag{12}$$

3.5 Look-Ahead Subtree Pruning

With efficient methods to approximate the bucket errors, we now present our main scheme for selective look-ahead through a method of choosing a look-ahead subtree for each variable that balances time and accuracy. From **Proposition 3**, we have a lower-bound on depth-d residual using summation of depth-1 residuals (which is exact when $d = 1$). We will use the local bucket error as a measure of *relevance* for including a particular variable when looking ahead.

Definition 19 (ϵ-relevant variable). *A variable X_k is ϵ-relevant if $\tilde{E}_k > \epsilon$.*

We will include paths in the look-ahead subtree only if they reach relevant variables.

Definition 20 (ϵ-pruned look-ahead subtree). *An ϵ-pruned look-ahead subtree $\mathcal{T}^{\epsilon}_{p,d}$ is a subtree of $\mathcal{T}_{p,d}$ containing only the nodes of $\mathcal{T}_{p,d}$ that are ϵ-relevant or on a path to an ϵ-relevant node.*

We show in **Figure 8** the look-ahead subtree $\mathcal{T}_{B,2}$. Since D is the only relevant variable due to its mini-bucket partitioning (see Figure 5), only the path from C to D remains in the ϵ-pruned look-ahead subtree $\mathcal{T}^{\epsilon}_{B,2}$ for $\epsilon = 0$ (circled).

Algorithm 4 (CompilePLS(ϵ)) generates the ϵ-pruned look-ahead subtree for each variable. Its complexity is linear in the size of the look-ahead subtree $\mathcal{T}_{p,d}$.

The ϵ-pruned look-ahead subtrees are computed prior to search. This suggests a static approach for deciding where look-ahead before search begins by consulting the readily available look-ahead subtrees during search. When $\epsilon = 0$, the look-ahead subtrees guide the computation to only compute as much as necessary for a given depth-d look-ahead, since

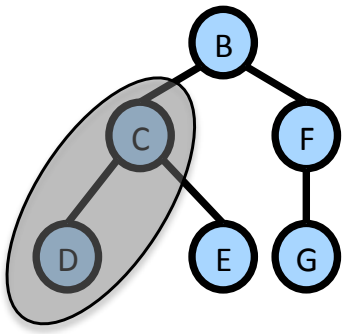


Figure 8: Look-ahead subtree $\mathcal{T}_{B,2}$. (B is shown for reference to root the multiple subtrees that make up $\mathcal{T}_{B,2}$.) Circled: the ϵ-pruned look-ahead subtree $\mathcal{T}^{\epsilon}_{B,2}$ for $\epsilon = 0$.

Algorithm 4: Compile ϵ-pruned Look-ahead Subtrees (CompilePLS(ϵ))

Input: A Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, a pseudo tree $\mathcal{T}$, i-bound, threshold ϵ, depth d
Output: ϵ-pruned look-ahead subtree for each $X_p \in \mathbf{X}$

```
1 Compute average (relative) local error Ẽ_k for each X_k ∈ X
2 X' ← all nodes in T that are ε-relevant (X_k ∈ X s.t. Ẽ_k > ε)
3 foreach X_p ∈ X do
4 |   Initialize T^ε_{p,d} to T_{p,d}
5 |   while T^ε_{p,d} has leaves ∉ X' do
6 |   |   Remove leaf X_j ∉ X' from T^ε_{p,d}
7 |   end
8 end
9 return T^ε_{p,d} for each X_p
```

buckets with zero error do not contribute to any look-ahead. As ϵ increases, we get less look-ahead across the search space, targeting regions with higher error only. Finally, at $\epsilon = \infty$, our scheme reduces to using no look-ahead at all.

We present in **Algorithm 5** pseudo-code for our MBE look-ahead heuristic. Before search begins, we initialize the regular MBE heuristics, which generates the λ messages used for the heuristics (Definition 9). We also execute CompilePLS(ϵ) for some depth d and ϵ to generate the ϵ-pruned look-ahead subtrees. Then, AND/OR depth-first search begins (AOBB, **Algorithm 1**) where the two references to the heuristic function $h(\cdot)$ in lines 4 and 15 are replaced by MBE look-ahead. As can be seen in the look-ahead algorithm, if the ϵ-pruned look-ahead subtree $\mathcal{T}^{\epsilon}_{p,d}$ is empty, it returns the MBE heuristic value (look-ahead is skipped). Otherwise, we construct the look-ahead graphical model $\mathcal{M}^{\epsilon}(\bar{x}_p, d)$ with respect to $\mathcal{T}^{\epsilon}_{p,d}$ and solve for its min-sum value with BE and return that value plus the MBE heuristic value without the contribution from messages generated from variables within the look-ahead subtree (Definition 1). The overall complexity of this algorithm is $O(nz^{w_{p,d}})$, where $w_{p,d}$ is the width of the look-ahead graphical model $\mathcal{M}^{\epsilon}(\bar{x}_p, d)$.

Algorithm 5: Look-ahead Heuristic for MBE (MBE-Look-ahead) of a partial assignment $\bar{x}_p$.

Input: A Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, a pseudo tree $\mathcal{T}$, messages from $MBE(i)$, pruned look-ahead trees $\mathcal{T}^{\epsilon}_{p,d}$ for each $X_p \in \mathbf{X}$, assignment $\bar{x}_p$
Output: Lower bound on partial assignment $\bar{x}_p$ to $\mathcal{M}$

```
if T^ε_{p,d} is empty then
    return Σ_{X_k ∈ T_p} Λ_{(k,p)}(x̄_p)
end
else
    Construct look-ahead graphical model M^ε(x̄_p, d) w.r.t. T^ε_{p,d}
    L^d(x̄_p) := min-sum Bucket Elimination on M^ε(x̄_p, d)
    return L^d(x̄_p) + Σ_{X_k ∈ T_p − T^ε_{p,d}} Λ_{(k,p)}(x̄_p)
end
```

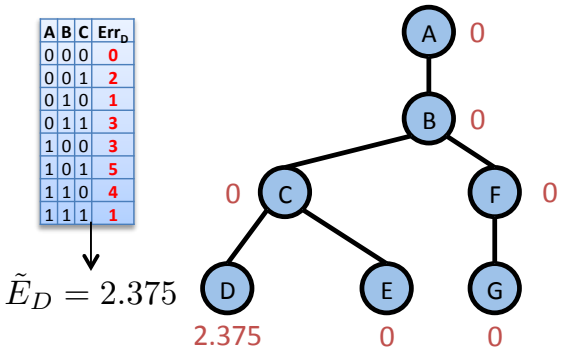


Figure 9: A pseudo-tree annotated with bucket errors, given that the error function of D shown here.

4. Analysis of Local Errors

To better understand the practical use of look-ahead subtrees, we show the pseudo-tree of selected problem instances. We annotate their respective variables with the average local error. As an example, **Figure 9** shows the annotated pseudo-tree for our running example problem. Only variable D has mini-bucket partitioning and a non-zero bucket error. On this example, for a particular depth d, we can construct a look-ahead subtree for each variable. For example, if $d = 2$ and $\epsilon = 0$, then we can extract from this figure the ϵ-pruned look-ahead subtree shown earlier in **Figure 6** by observing that D is the only relevant variable (**Figure 9**). For every other variable excluding C, the ϵ-pruned look-ahead subtrees are empty, so look-ahead would be completely skipped.

We now show annotated pseudo-trees of one problem instance from the **pedigree**, **grid**, **promedas**, and **dbn** classes. These classes make up some of the benchmarks that are used in the main experimental evaluation in Section 5. We also annotate the pseudo-trees with the number of mini-buckets (mb), the pseudo-width (psw). Each node is color coded on a spectrum of pale yellow to dark red to indicate its relative degree of error. Within each

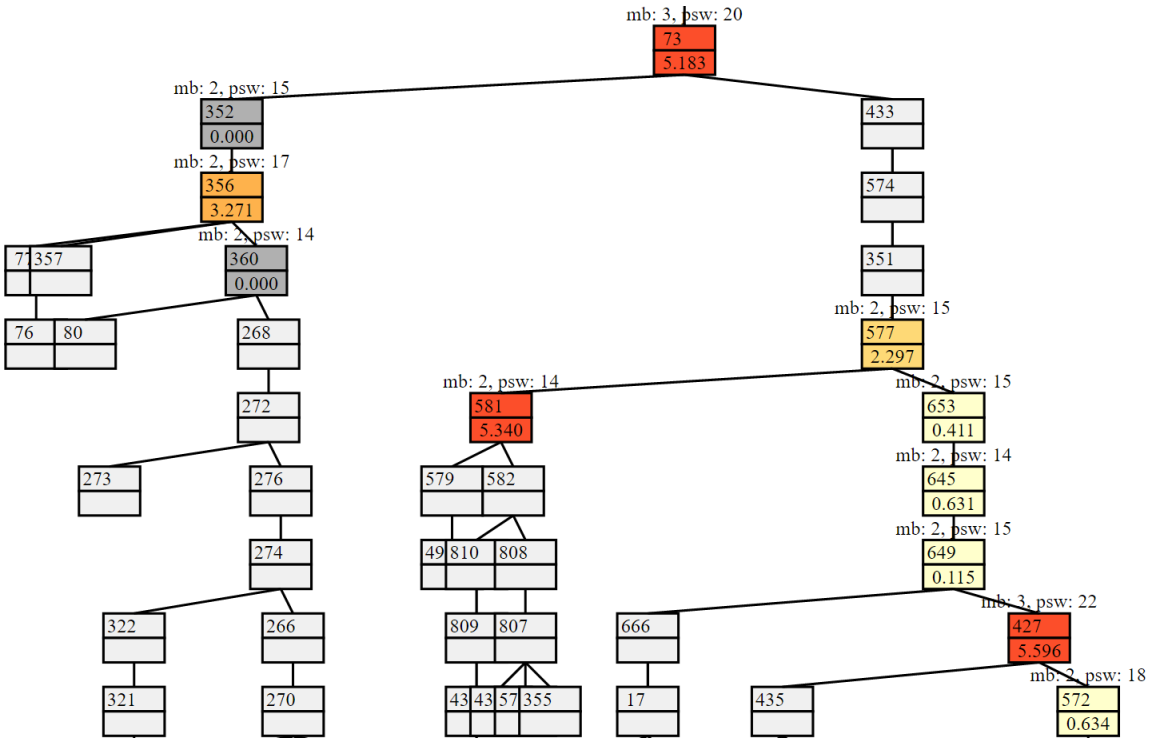


Figure 10: Extracted structure from pseudo-tree showing errors for *pedigree40* (n=842, z=7, w=27, h=111) with an i-bound of 12. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error.

node, we indicate its variable index on top and its average relative local bucket error on the bottom. Nodes with partitioning but zero error are colored dark gray. Finally, nodes with no partitioning are colored light gray and the mini-bucket and pseudo-width annotations are omitted. We also provide problem statistics in the caption: n - the number of variables, z - the maximum domain size, w - the induced width, and h - the pseudo-tree height. We only show portions of the tree since the full pseudo-trees are too large to show. The full pseudo-trees can be viewed online (Lam, 2017c).

For each instance, we also provide two additional plots. First, we show variables ordered by their average relative local errors in order to see the frequency of different error values. Second, we plot for each variable the number of mini-buckets in order to show how mini-bucket partitioning is related to error. We also note the number of buckets with zero average relative error, the average across all variables, and the average across all variables with non-zero error. The quadruplet in the title of each plot indicates the same problem statistics (n, z, w, h) mentioned in each pseudo-tree.

4.1 Case Study: Pedigree

We show in **Figure 10** an extracted portion of the pseudo-tree of *pedigree40* annotated with local bucket error information when the i-bound of MBE applied is 12. We see that the

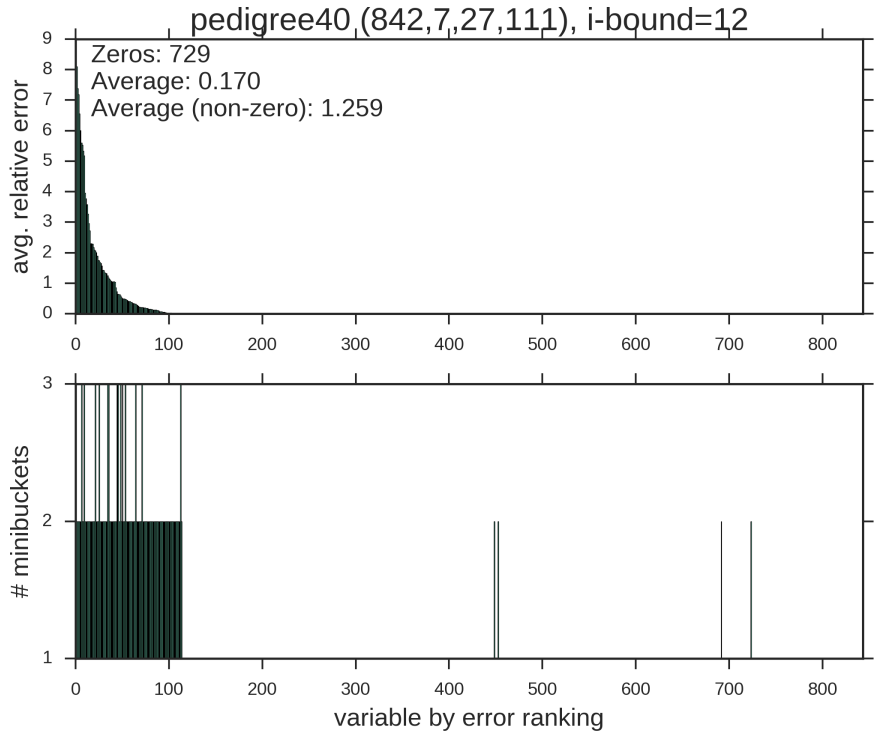


Figure 11: Distribution of errors and mini-buckets for *pedigree40*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

errors tend to appear along several paths. On the left side, the decomposed subproblems that sit near the leaves of the pseudo-tree have zero partitioning, and therefore, zero error. It means that messages in the leaves are small and no partitioning is required. Notably, the magnitude of the errors differ, as we notice that the error on nodes 73, 581, and 427 (the red nodes) have much higher error than the rest. We also see that nodes 352 and 360 (dark gray) have zero error, despite having partitioning.

In **Figure 11**, we plot each variable with its error (top) and mini-bucket partitioning (bottom). The variables are sorted in descending order based on their error. In the plot on the top, we observe that most of the variables (729 out of 842) have zero error, and that very few variables have a high error. The average error is 0.17 (1.259 excluding zeros). In the plot on the bottome we see that most, but not all, the variables that have zero error is because there is no mini-bucket partitioning. We also see that the number of mini-buckets does not seem to be correlated to the error value (left side of the bottom plot).

4.2 Case Study: Grid

Figure 12 provides another example, showing a part of the pseudotree for *grid80x80.f15* when the i-bound of the MBE applied is 14. This instance is difficult, having an induced width of 112. The pseudo-tree contains long chains with errors (see top plot) as well as very arboreous regions (see bottom plot) which indicate high amounts of decomposition.

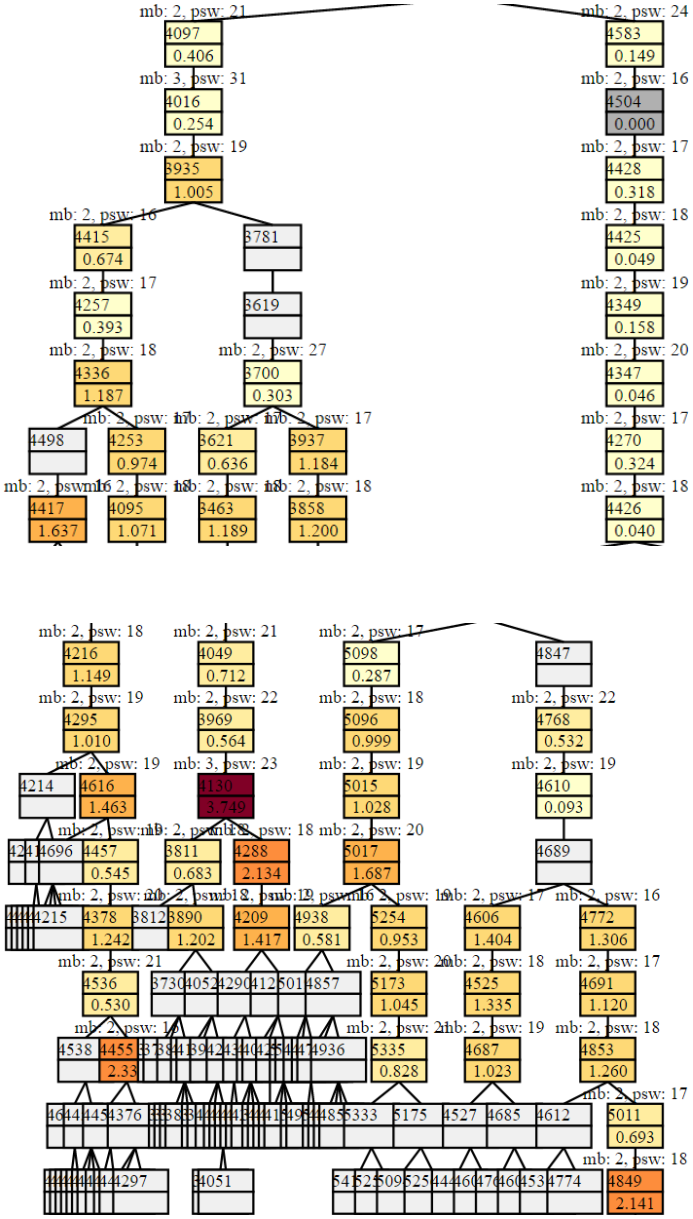


Figure 12: Extracted structures from pseudotree showing errors for grid instance grid80x80.f15 (n=6400, z=2, w=112, h=296) with an i-bound of 14. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error.

The first observation is that decomposition regions seems to be where larger errors occur. Despite most buckets only needing to partition into two mini-buckets, we can observe that the errors can vary, from as low as 0.09 (variable 4610, near the top right) to 2.33 (variable 4455, near the bottom left).

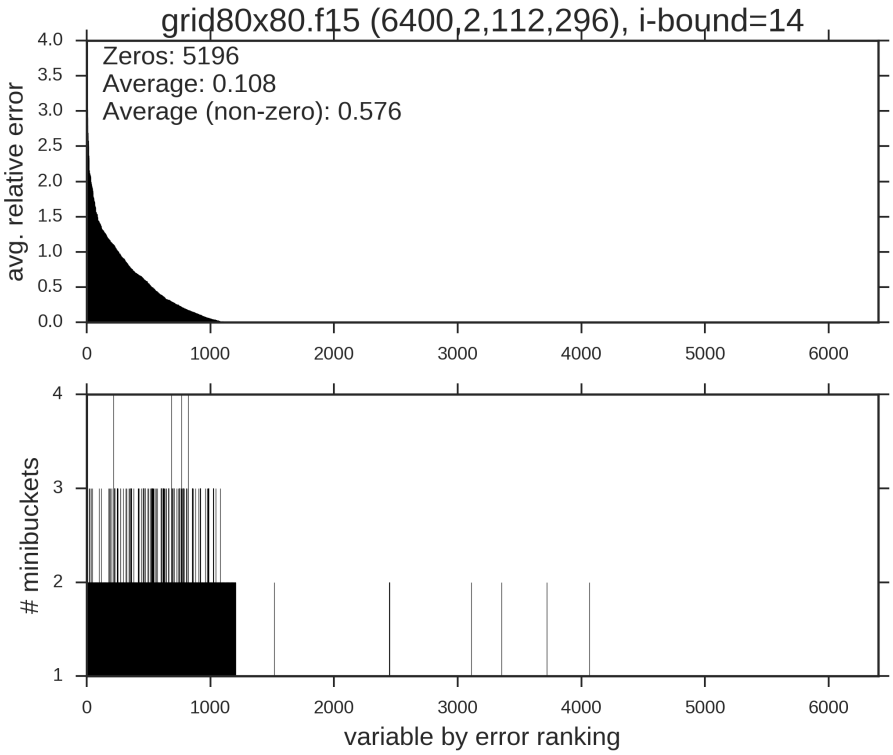


Figure 13: Distribution of errors and mini-buckets for grid instance *grid80x80.f15*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

In the top plot of **Figure 13** we see again that most variables have zero error (5196 out of 6400) and very few have large error. The average error is 0.576 and the maximum is 3.5. In the plot at the bottom we see again that several buckets where partition is needed still have zero error. As in the previous example we observe that the number of mini-buckets does not correlate with the magnitude of error. There are many variables with 3 mini-buckets that are spread out roughly uniformly over the range of errors.

An interesting final observation is that, when a bucket needs to partition into mini-buckets, the partition rarely needs more than 3 minibuckets and never more than 4. That means that in this instance the pseudo-width (bounded above by $4 \times 11 + 1 = 45$) is clearly smaller than the induced width (112) and in most of the buckets much smaller.

4.3 Case Study: Promedas

Another example of an instance is *or_chain_140.fg* in **Figure 14**. Here, the MBE i-bound is 10. Despite the many places here where there is mini-bucket partitioning, a fair number of them have zero error. For example, notice on the right branch that there are three variables in a row with zero error. Also, like in the pedigree instance, the errors appear along paths.

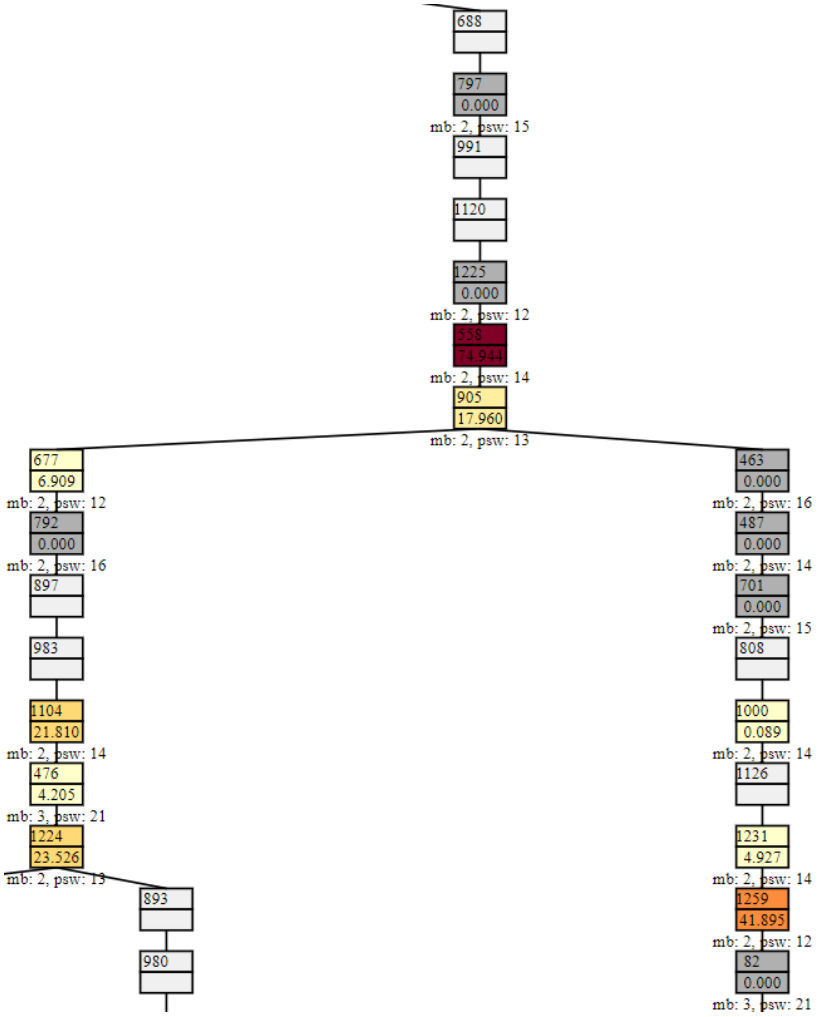


Figure 14: Extracted structure from pseudo-tree showing errors for *or_chain_140.fg* (n=1260, z=2, w=32, h=79) with an i-bound of 10. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error.

The magnitude of the errors also differ as we see for variable 558 in red with an error of 74.944.

In **Figure 15** we can see that only 122 variables have error, yet there are more variables with mini-bucket partitioning. The average error excluding everything with zero error is 10.361. As with the examples we have seen so far, most variables have zero error due to the lack of mini-bucket partitioning. Also, when comparing the magnitude of the errors to the number of mini-buckets, there is no correlation once again.

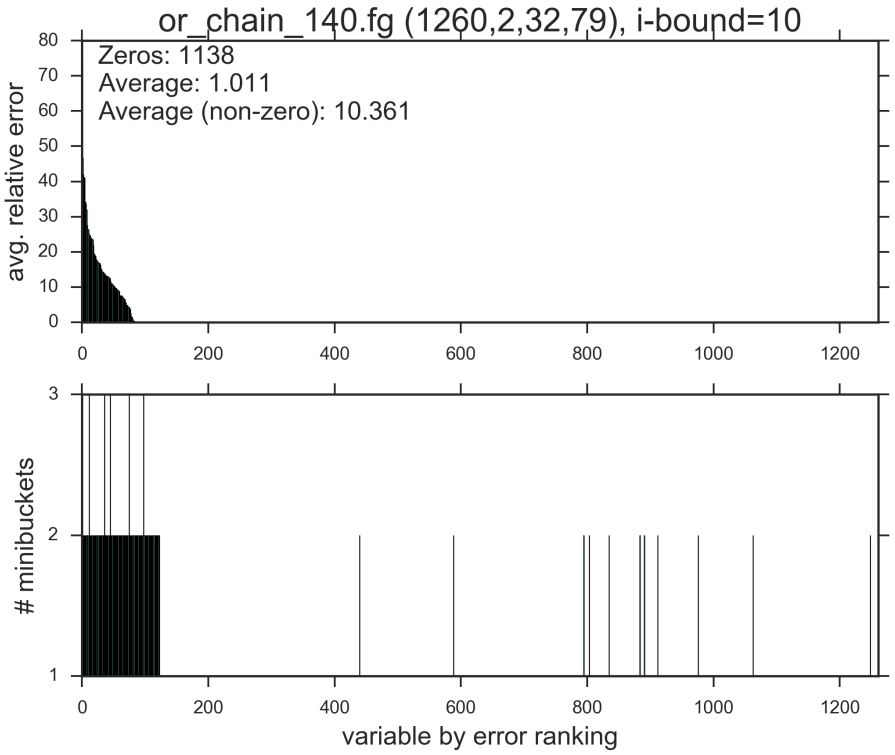


Figure 15: Distribution of errors and mini-buckets for *or_chain_140.fg*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

4.4 Case Study: DBN

In the last example pseudo-tree of the instance *rus2_50_100_3_2* from the DBN benchmark in **Figure 16** the i-bound is 14. The problem instances of this benchmark typically have the same structure where the pseudo-tree starts with a chain (the first 60 variables, in this example) until one point where the rest of variables branch out from it (100 variables, in the example). There is partitioning only at the leaves with 5 mini-buckets per variable. The average relative bucket errors are very large, suggesting that the residuals can be highly informative in guiding search. Observe that there is a single node having far higher error than the rest (variable 57, colored in red in the figure).

Figure 17 is consistent with the annotated pseudo-tree. The chain of 60 variables is error free. In the rest of the variables, unlike previous examples, errors are high. In every bucket where there is partition, there are errors (with an average of 7966).

Because of the very special structure of these instances, the pseudo-width is the same as the induced width, meaning that it is equally expensive to pre-compute local bucket errors exactly as to solve the problem. However, the analysis is not completely useless, since it let's know that our approach will produce empty pruned look-ahead trees along the chains, thus preventing the algorithm from doing any useless look-ahead. Interestingly, only nodes at the end of the chain, where look-ahead can be advantageous, will have non-empty look-ahead

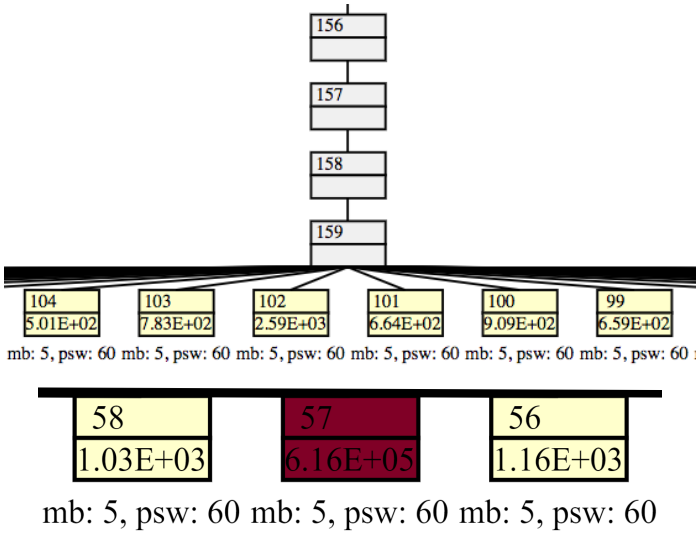


Figure 16: Extracted structure from pseudotree showing errors for DBN instance rus2_50_100_3_2 (n=160, z=2, w=59, h=59) with an i-bound of 14. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error. We also include here another portion of the leaf level of the tree with an outlier node.

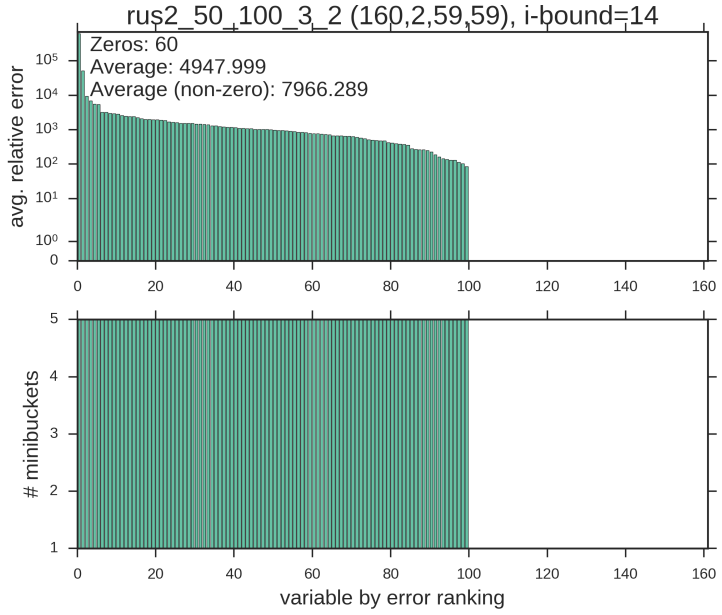


Figure 17: Distribution of errors and mini-buckets for DBN instance *rus2_50_100_3_2*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

Benchmark	d=1	d=2	d=3	d=4	d=5	d=6
Pedigree	0.89	0.87	0.85	0.85	0.85	0.85
LargeFam3	0.87	0.86	0.85	0.85	0.85	0.85
Promedas	0.90	0.87	0.86	0.86	0.86	0.86
Type4	0.86	0.83	0.81	0.81	0.81	0.81
DBN	0.99	0.98	0.97	0.97	0.96	0.95
Grid	0.82	0.77	0.75	0.74	0.73	0.73

Table 4: For each benchmark, the average ratio of variables with near empty look-ahead subtrees over for various look-ahead depths with a fixed i-bound of 10.

trees. In these nodes, due to the extremely high decomposition level that the pseudo-tree uncovers, the induced width of the look-ahead graphical model $w_{p,d}$ is much smaller than d, so exact inference (BE) is a more efficient method to compute the look-ahead than search.

4.4.1 Discussion

We considered a handful of differently structured instances in this section that illustrated different structures of error distributed along the pseudo-trees. As expected, the bucket error grows when the i-bound is low relative to the induced width (consider the grid instance compared with the pedigree instance). We also observed that there are a fair number of nodes within the pseudo-trees having no error, which is useful for controlling the look-ahead. Indeed we systematically see that there are nodes that appear to have error when considering the mini-bucket partitioning alone, yet evaluating their bucket errors tells otherwise. We showed here that the local bucket error provides information beyond the presence of mini-bucket partitioning.

Most importantly, we observe that problems typically have zero error in a majority of its variables, meaning that look-ahead during search is most often to be redundant and therefore counter-effective. To demonstrate this beyond the 4 previous examples, we compiled the look-ahead subtrees for every problem instance across the 6 benchmarks that will be used in our experimental evaluation in the following section. The results are summarized in **Table 4** by averaging the ratio of variables which have a look-ahead subtree that is nearly empty (defined by being at most 10% of the unpruned look-ahead subtree's size) for an i-bound of 10 and look-ahead depths ranging from 1 to 6 with an error threshold ϵ of 0.01. Indeed, most variables have nearly empty look-ahead subtrees, with the ratio decreasing relatively slowly as depth increases. Clearly, this would yield a positive impact on dealing with the overhead of look-ahead.

5. Experimental Evaluation

We now evaluate empirically the impact of our look-ahead scheme in branch and bound depth first for different look-ahead depths. In Subsection 5.1 we consider the problem of finding the optimal solution and proving its optimality which is important in not-so-hard instances. The main efficiency measure here is cpu time to complete the execution. In Subsection 5.2 we consider the problem of obtaining near-optimal solutions in an anytime

manner which is important when dealing with hard instances that can not be solved within reasonable time limits. In this context one algorithm is better than another if it obtains better solutions sooner.

Our baseline algorithm is AOBB with context-based caching, so the search is done on the AND/OR search graph (Marinescu & Dechter, 2009b). We augmented it with breadth rotation (BRAOBB) (Otten & Dechter, 2012) in order to enhance the any-time performance. As heuristic, we use Mini-Bucket Elimination with Moment-Matching, which adds a step that shifts costs between mini-buckets to tighten the approximation (Ihler, Flerova, Dechter, & Otten, 2012). Together, they form one of the best algorithms for optimization in graphical models, which won the PASCAL Inference Competition in 2011 (Otten et al., 2012).

We compare the baseline with an algorithm that starts computing CompilePLS(ϵ) (**Algorithm 4**) for compiling the pruned look-ahead subtrees. The average error was computed exactly if the local bucket error function had no more than 10^5 entries. Otherwise, we approximated by sampling 10^5 of the entries and averaging over the samples. Lastly, we have our two parameters of depth d and the error threshold ϵ that control the amount of look-ahead performed. Using ϵ, we can in principle control look-ahead without using the depth d at all (namely we can always use full look-ahead subtree and pruning that), this parameter is highly dependent on the particular problem instance and heuristic strength. On the other hand, controlling with the depth d as usual provides much tighter control of the overhead. We generally found that using a fixed ϵ of 0.01 tended to be the best when controlling look-ahead primarily when using the depth d. We provide an overview of the experiments performed in this area in Section 5.3. (Also see Lam, 2017a, for a full account of the experiments.) In the following two sections evaluating the exact and anytime performance, we vary the look-ahead depth from 1 to 6 and used a fixed ϵ of 0.01.

Both the baseline and our approach were based on a branch of the DAOOPT code which is implemented in C++ (64-bit) (Otten, 2013; Lam, 2017b). Experiments were run on an Intel Xeon X5650 2.66GHz processor, with a 4GB memory limit for each job. The time limit for every experiment was bounded to 2 hours (7200 seconds).

Benchmarks. We used benchmarks from the UAI and PASCAL2 competitions. In particular we considered instances from genetic linkage analysis (**Pedigree, LargeFam3, Type4**) (Fishelson & Geiger, 2004), medical diagnosis (**Promedas**) (Wemmenhove, Mooij, Wiegerinck, Leisink, Kappen, & Neijt, 2007), deep belief networks (**DBN**), and binary grids (**Grids**). Altogether, we report results on 221 instances. For each problem, we used a fixed pseudo-tree. We provide additional details on instance selection criteria and benchmark statistics at the beginning of Sections 5.1 and 5.2, which focus on evaluating for exact solutions and anytime behavior, respectively.

5.1 Evaluating Look-Ahead for Exact Solutions

In order to experiment on non-trivial, yet solvable instances, we selected a subset of the benchmark instances. Instances that could be solved with the baseline in less than 30 seconds with a weak heuristics (i-bound = 6) were discarded for being too easy. Instances that could not be solved with the baseline in less than 7200 seconds with the highest i-bound fitting in memory (4GB) were discarded for bein too hard. 95 instances, having induced widths ranging from 19 to 69 passed the filter. See **Table 5** for additional statistics.

Benchmark	# inst	n	z	w	h	$\|F\|$	a
Pedigree	12	581	3	19	79	794	4
		1006	7	39	143	1185	5
LargeFam3	13	874	3	21	44	1321	4
		1712	3	39	77	2720	4
Promedas	31	615	2	28	65	625	3
		1911	2	69	128	1427	3
DBN	30	70	2	29	29	16167	2
		70	2	29	29	16167	2
Grids	9	400	2	24	62	1161	2
		1600	2	52	157	4721	2

Table 5: Benchmark statistics for exact solution evalution. # inst - number of instances, n - number of variables, z - 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.

For each instance we conducted experiments with 3 different i-bounds: the highest one fitting in memory, the lowest one that allowed to solve the instance with the baseline in less than 7200 seconds, and another one in between.

Tables 6, 7, 8, 9, and 10, present results on the amount of time spent (in seconds) and nodes expanded (in millions of nodes) for selected representative instances. Next to each time, we also provide the relative speedup over the baseline. Similarly, next to each node count, we provide a "compression" ratio of nodes expanded relative to the baseline. In cases were the baseline fails to find the exact solution, we give a lower bound on the speedup, assuming the baseline is 7200 seconds. For the number of nodes expanded, the ratio is an upper bound obtained by counting the number of nodes expanded by the timeout. Within an instance, each column corresponds to a different i-bound and each row corresponds to a different look-ahead depth.

To account for the full set of instances solved within the time limit, we provide in **Figures 18, 19, 20, 21, and 22** scatter plots of the speedups of the runtime of each look-ahead depth against the baseline. The differently colored points represent the different problem instances in the benchmark. Each depth is annotated by the number of instances which performed better than the baseline. We separate these into the same weak to strong heuristic groupings as done in the tables.

The **Type4** benchmark is not included in the results for finding exact solutions since there were no instances which were solved within the time limit.

5.1.1 Pedigree

Our first benchmark consists of genetic linkage analysis problems. **Table 6** shows the results in terms of time spent and nodes expanded to find the exact solution on selected i-bounds of representative instances. We observe that look-ahead improves the performance, especially for lower i-bounds. For instance, on *pedigree18* with an i-bound of 5, we see a runtime of 675 seconds with a look-ahead depth of 4, which is 2.16 times faster than the baseline time

instance (n, z, w^*, h)	depth	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)
		i=11		i=16		i=21	
	d=0	2078 (1.00)	385.79 (1.00)	74 (1.00)	13.32 (1.00)	142 (1.00)	4.66 (1.00)
	d=1	1905 (1.09)	280.04 (0.73)	75 (0.99)	10.59 (0.80)	146 (0.97)	3.95 (0.85)
	d=2	1846 (1.13)	226.18 (0.59)	72 (1.02)	8.91 (0.67)	147 (0.96)	3.45 (0.74)
pedigree7	d=3	1972 (1.05)	177.49 (0.46)	76 (0.97)	7.43 (0.56)	148 (0.95)	3.08 (0.66)
(867,4,28,123)	d=4	2423 (0.86)	135.83 (0.35)	86 (0.86)	6.18 (0.46)	154 (0.92)	2.64 (0.57)
	d=5	3204 (0.65)	103.89 (0.27)	104 (0.71)	5.11 (0.38)	169 (0.84)	2.44 (0.52)
	d=6	4976 (0.42)	81.77 (0.21)	146 (0.50)	4.29 (0.32)	194 (0.73)	2.22 (0.48)
		i=5		i=8		i=23	
	d=0	5207 (1.00)	974.63 (1.00)	386 (1.00)	79.94 (1.00)	85 (1.00)	0.01 (1.00)
	d=1	4249 (1.23)	668.92 (0.69)	372 (1.04)	65.51 (0.82)	85 (0.99)	0.01 (0.95)
	d=2	4061 (1.28)	564.38 (0.58)	325 (1.19)	50.21 (0.63)	87 (0.97)	0.01 (0.90)
pedigree9	d=3	3594 (1.45)	371.50 (0.38)	312 (1.24)	38.44 (0.48)	85 (0.99)	0.01 (0.88)
(935,7,25,137)	d=4	3548 (1.47)	245.15 (0.25)	398 (0.97)	32.54 (0.41)	85 (0.99)	0.01 (0.85)
	d=5	3654 (1.43)	159.41 (0.16)	457 (0.84)	22.58 (0.28)	85 (0.99)	0.01 (0.83)
	d=6	5523 (0.94)	128.85 (0.13)	680 (0.57)	19.37 (0.24)	85 (1.00)	0.01 (0.75)
		i=5		i=7		i=10	
	d=0	1464 (1.00)	327.19 (1.00)	66 (1.00)	17.60 (1.00)	7 (1.00)	1.93 (1.00)
	d=1	1245 (1.18)	244.45 (0.75)	55 (1.19)	12.88 (0.73)	6 (1.12)	1.36 (0.70)
	d=2	1147 (1.28)	200.20 (0.61)	48 (1.35)	10.08 (0.57)	5 (1.39)	0.94 (0.49)
pedigree18	d=3	887 (1.65)	136.37 (0.42)	36 (1.83)	6.64 (0.38)	4 (1.53)	0.69 (0.36)
(931,5,19,102)	d=4	675 (2.17)	84.72 (0.26)	29 (2.30)	4.17 (0.24)	5 (1.34)	0.48 (0.25)
	d=5	755 (1.94)	64.45 (0.20)	30 (2.22)	3.14 (0.18)	6 (1.23)	0.39 (0.20)
	d=6	777 (1.88)	46.28 (0.14)	42 (1.55)	2.32 (0.13)	8 (0.87)	0.28 (0.14)
		i=13		i=15		i=18	
	d=0	3193 (1.00)	544.11 (1.00)	1470 (1.00)	246.25 (1.00)	108 (1.00)	2.63 (1.00)
	d=1	3125 (1.02)	457.74 (0.84)	1458 (1.01)	210.39 (0.85)	114 (0.95)	2.26 (0.86)
	d=2	2705 (1.18)	334.54 (0.61)	1497 (0.98)	188.04 (0.76)	114 (0.95)	2.02 (0.77)
pedigree34	d=3	3118 (1.02)	284.64 (0.52)	1736 (0.85)	168.68 (0.69)	106 (1.02)	1.87 (0.71)
(922,5,28,143)	d=4	4086 (0.78)	249.13 (0.46)	2119 (0.69)	147.71 (0.60)	123 (0.88)	1.61 (0.61)
	d=5	6205 (0.51)	212.84 (0.39)	3022 (0.49)	128.82 (0.52)	136 (0.80)	1.39 (0.53)
	d=6	oot	-	4992 (0.29)	116.30 (0.47)	166 (0.65)	1.27 (0.48)
		i=8		i=12		i=23	
	d=0	2256 (1.00)	492.98 (1.00)	345 (1.00)	81.92 (1.00)	76 (1.00)	0.00 (1.00)
	d=1	2066 (1.09)	389.99 (0.79)	351 (0.98)	68.86 (0.84)	77 (0.99)	0.00 (1.00)
	d=2	2027 (1.11)	324.45 (0.66)	313 (1.10)	53.82 (0.66)	77 (0.99)	0.00 (1.00)
pedigree44	d=3	1787 (1.26)	220.62 (0.45)	303 (1.14)	38.10 (0.47)	76 (1.00)	0.00 (1.00)
(644,4,24,79)	d=4	1847 (1.22)	166.69 (0.34)	330 (1.05)	28.83 (0.35)	77 (0.99)	0.00 (1.00)
	d=5	2167 (1.04)	114.99 (0.23)	382 (0.90)	20.07 (0.24)	77 (0.99)	0.00 (1.00)
	d=6	3027 (0.75)	91.94 (0.19)	494 (0.70)	13.80 (0.17)	77 (0.99)	0.00 (1.00)
		i=16		i=19		i=22	
	d=0	4075 (1.00)	917.81 (1.00)	2545 (1.00)	599.16 (1.00)	502 (1.00)	82.85 (1.00)
	d=1	4168 (0.98)	782.74 (0.85)	2673 (0.95)	508.52 (0.85)	514 (0.98)	71.01 (0.86)
	d=2	4108 (0.99)	670.59 (0.73)	2759 (0.92)	442.76 (0.74)	536 (0.94)	64.13 (0.77)
pedigree51	d=3	3892 (1.05)	512.25 (0.56)	2590 (0.98)	335.50 (0.56)	542 (0.93)	53.31 (0.64)
(871,5,39,98)	d=4	4871 (0.84)	437.29 (0.48)	3004 (0.85)	278.58 (0.46)	614 (0.82)	45.33 (0.55)
	d=5	5896 (0.69)	351.41 (0.38)	3697 (0.69)	224.62 (0.37)	759 (0.66)	39.73 (0.48)
	d=6	oot	-	5708 (0.45)	194.39 (0.32)	1091 (0.46)	35.33 (0.43)

Table 6: Selected **pedigree** instances: "time" indicates the CPU time in seconds (speedup over baseline) and "nodes" indicates the number of OR nodes expanded in *millions* of nodes (ratio relative to baseline) In a time column, 'oot' that the time limit of 2 hours was exceeded. The problem parameters are also provided for each instance (n: number of variables, z: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time is boxed.

of 1464 seconds. Indeed, the number of nodes expanded here decreases by 74%. However, on higher i-bounds, lookahead is less cost effective.

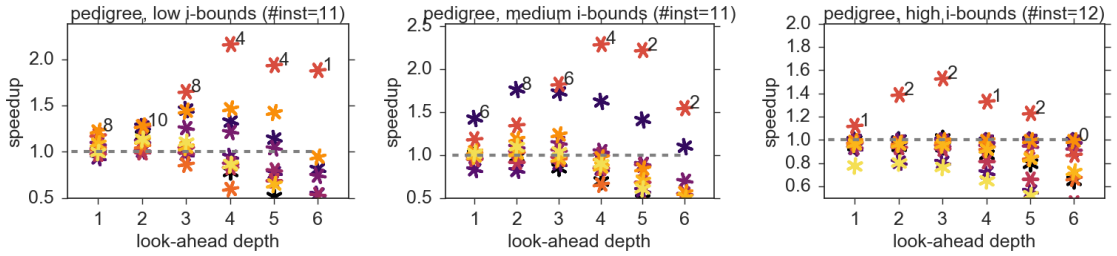


Figure 18: Solved **pedigree** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

Figure 18 shows the distribution of speedups across all the instances of this benchmark that were solved within the time limit. For low i-bounds of modest look-ahead depths (less than 3), we observe that look-ahead improves over the baseline for most of the instances. However, as the look-ahead depths increase, it is often the case that it is not cost-effective. Also, when the heuristic is more accurate, look-ahead has less of an impact on the reduction of the number of nodes, and is consequently less effective.

In summary, due to the relatively easy nature of this benchmark, the bucket errors tend to be very low for the higher i-bounds. Thus, there are fewer opportunities for look-ahead to improve the pre-compiled mini-bucket heuristic.

5.1.2 LargeFam3

Here we have another benchmark based on genetic linkage analysis, but in contrast to the **pedigree** benchmark, these instances are more difficult as seen by the relatively higher induced width and therefore higher i-bounds required to find exact solutions. **Table 7** shows the detailed results for representative instances in this benchmark. We observe that on weaker heuristics, look-ahead obtains some speedups. For instance *lf3-10-52*, we see a runtime of 4915 seconds for a depth of 3 compared to 6560 seconds for the baseline, close to the timeout. At a depth of 4, the ratio of the number of nodes only changes by 10%, thus making look-ahead less cost-effective. When moving to higher i-bounds, we see a shift towards lower depths being cost effective, but with relatively small improvements over the baseline. For example, on the same instance, for i=20, a depth of 1 reduces the runtime only marginally. Still, in *lf3-13-58*, there is more payoff with a look-ahead depth of 2 giving a 1.2 speedup over the baseline, thanks to a 50% reduction in the number of nodes expanded.

Figure 19 shows the distributions of speedups across all the instances of the benchmark that were solved within the time limit. Here we observe that for the weaker heuristics, only a very small number of instances improve over the baseline for the various look-ahead depths. We see that for depths of 1 and 2, 5 of the instances performed better than the baseline, but as the depth increases, the number of instances that perform better decreases. For stronger heuristics (medium and high i-bounds), a slightly larger proportion of instances improve

instance (n, z, w^*, h)	depth	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)
		i=16		i=18		i=20	
	d=0	6560 (1.00)	1306.25 (1.00)	2180 (1.00)	471.67 (1.00)	387 (1.00)	75.13 (1.00)
	d=1	6058 (1.08)	991.91 (0.76)	1999 (1.09)	343.13 (0.73)	371 (1.04)	54.20 (0.72)
	d=2	5669 (1.16)	744.66 (0.57)	1978 (1.10)	263.95 (0.56)	380 (1.02)	42.71 (0.57)
lf3-10-52	d=3	4915 (1.33)	431.36 (0.33)	2545 (0.86)	214.29 (0.45)	467 (0.83)	33.51 (0.45)
(959,3,39,68)	d=4	6068 (1.08)	301.99 (0.23)	3809 (0.57)	165.31 (0.35)	651 (0.59)	26.56 (0.35)
	d=5	oot	-	6627 (0.33)	131.89 (0.28)	1087 (0.36)	21.56 (0.29)
	d=6	oot	-	oot	-	1974 (0.20)	17.54 (0.23)
		i=14		i=16		i=18	
	d=0	oot	-	5319 (1.00)	1041.65 (1.00)	471 (1.00)	70.24 (1.00)
	d=1	oot	-	4879 (1.09)	752.73 (0.72)	433 (1.09)	49.08 (0.70)
	d=2	oot	-	4418 (1.20)	529.90 (0.51)	390 (1.21)	35.29 (0.50)
lf3-13-58	d=3	oot	-	3858 (1.38)	340.03 (0.33)	462 (1.02)	28.11 (0.40)
(1272,3,32,76)	d=4	oot	-	5222 (1.02)	261.37 (0.25)	575 (0.82)	21.10 (0.30)
	d=5	oot	-	oot	-	865 (0.55)	16.66 (0.24)
	d=6	oot	-	oot	-	1447 (0.33)	12.20 (0.17)
		i=14		i=16		i=18	
	d=0	3971 (1.00)	821.75 (1.00)	644 (1.00)	154.40 (1.00)	56 (1.00)	10.94 (1.00)
	d=1	3579 (1.11)	609.38 (0.74)	499 (1.29)	99.26 (0.64)	51 (1.10)	7.59 (0.69)
	d=2	3071 (1.29)	464.28 (0.56)	455 (1.41)	75.97 (0.49)	48 (1.16)	5.97 (0.55)
lf3-15-59	d=3	3057 (1.30)	346.87 (0.42)	480 (1.34)	61.42 (0.40)	50 (1.11)	4.79 (0.44)
(1574,3,33,71)	d=4	3896 (1.02)	284.40 (0.35)	614 (1.05)	47.89 (0.31)	61 (0.92)	3.65 (0.33)
	d=5	4740 (0.84)	219.70 (0.27)	972 (0.66)	38.95 (0.25)	98 (0.57)	3.09 (0.28)
	d=6	oot	-	1655 (0.39)	30.63 (0.20)	178 (0.31)	2.50 (0.23)
		i=14		i=16		i=18	
	d=0	1760 (1.00)	367.61 (1.00)	381 (1.00)	77.79 (1.00)	104 (1.00)	9.30 (1.00)
	d=1	1954 (0.90)	337.70 (0.92)	400 (0.95)	66.06 (0.85)	112 (0.93)	8.11 (0.87)
	d=2	1862 (0.95)	281.86 (0.77)	376 (1.01)	53.07 (0.68)	107 (0.97)	6.28 (0.67)
lf3-16-56	d=3	1926 (0.91)	227.79 (0.62)	366 (1.04)	40.25 (0.52)	104 (1.00)	4.55 (0.49)
(1688,3,38,77)	d=4	2232 (0.79)	175.98 (0.48)	430 (0.89)	31.17 (0.40)	112 (0.92)	3.65 (0.39)
	d=5	2183 (0.81)	104.73 (0.28)	513 (0.74)	20.77 (0.27)	115 (0.91)	2.31 (0.25)
	d=6	2803 (0.63)	77.90 (0.21)	732 (0.52)	15.65 (0.20)	131 (0.79)	1.60 (0.17)
		i=12		i=14		i=16	
	d=0	1386 (1.00)	263.46 (1.00)	476 (1.00)	93.62 (1.00)	20 (1.00)	2.28 (1.00)
	d=1	1401 (0.99)	208.27 (0.79)	463 (1.03)	70.44 (0.75)	22 (0.90)	1.53 (0.67)
	d=2	1212 (1.14)	161.49 (0.61)	436 (1.09)	53.60 (0.57)	21 (0.92)	1.12 (0.49)
lf3-17-58	d=3	1468 (0.94)	117.04 (0.44)	579 (0.82)	43.52 (0.46)	22 (0.91)	0.74 (0.33)
(1712,3,31,75)	d=4	1988 (0.70)	88.02 (0.33)	1039 (0.46)	35.35 (0.38)	27 (0.73)	0.57 (0.25)
	d=5	2129 (0.65)	33.84 (0.13)	2770 (0.17)	23.36 (0.25)	36 (0.54)	0.41 (0.18)
	d=6	3968 (0.35)	24.37 (0.09)	5428 (0.09)	10.40 (0.11)	63 (0.31)	0.32 (0.14)

Table 7: Selected **LargeFam3** instances: "time" indicates the CPU time in seconds (speedup over baseline) and "nodes" indicates the number of OR nodes expanded in *millions* of nodes (ratio relative to baseline) In a time column, 'oot' that the time limit of 2 hours was exceeded. The problem parameters are also provided for each instance (n: number of variables, z: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time is boxed.

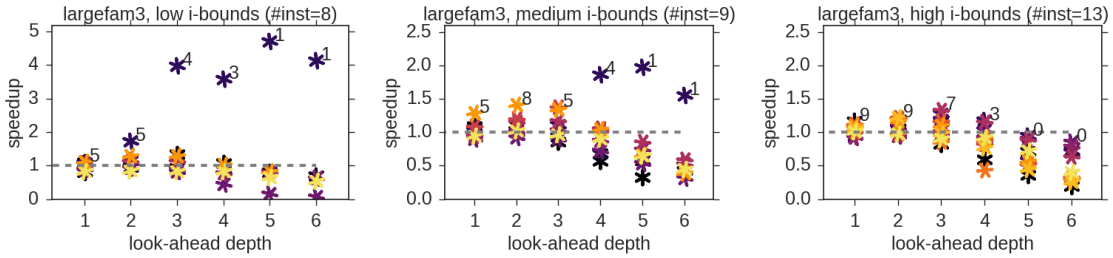


Figure 19: Solved **LargeFam3** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

over the baseline since it also includes more difficult instances that could not be solved with weaker heuristics. However, increasing depth past 2 or 3 results in fewer improvements.

In summary, for this benchmark, the impact of look-ahead is somewhat similar to what we saw for the **pedigree** benchmark. While the instances that we could solve were more difficult, the bucket errors behave similarly.

5.1.3 Promedas

We now move onto a benchmark of problems based on medical diagnosis. **Table 8** shows the detailed results for representative instances. We see a significant speedup when using weak heuristics. For example on *or-chain-140.fg*, a depth 6 look-ahead completed in 1156 seconds where the baseline required 4555 seconds, a 3.94 speedup. Also worth noting in this benchmark is *or-chain-108.fg*, which is a fairly hard instance with an induced width of 67. Here, even the highest i-bound of 22 which we could use resulted in the baseline timing out at 7200 seconds. Thus, with a depth of 4, we achieved a time which is at least twice as fast.

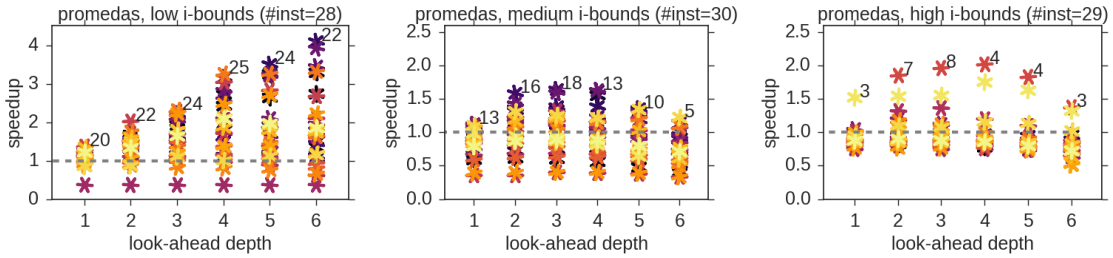


Figure 20: Solved **promedas** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

Figure 20 shows the distribution of speedups across all instances that were solved within the time limit. For low i-bounds, we observe a general trend of deeper look-ahead improving performance. In particular, the number of instances for which look-ahead improved performance increases monotonically until a depth of 4. Moving to medium i-bounds, look-ahead improves over the baseline on only about half of the 30 solved instances from depths 1 to

instance (n, z, w^*, h)	depth	time	nodes	time	nodes	time	nodes
		i=18		i=20		i=22	
	d=0	oot	-	6685 (1.00)	1538.31 (1.00)	oot	-
	d=1	oot	-	6519 (1.03)	1311.00 (0.85)	oot	-
	d=2	oot	-	6216 (1.08)	1135.75 (0.74)	3892 (>1.85)	746.99 (<0.66)
or-chain-108.fg	d=3	6782 (>1.06)	1083.13 (<0.69)	5741 (1.16)	928.40 (0.60)	3664 (>1.97)	633.30 (<0.56)
(1263,2,67,117)	d=4	6221 (>1.16)	836.41 (<0.54)	oot	-	3554 (>2.03)	526.19 (<0.47)
	d=5	6741 (>1.07)	723.05 (<0.46)	oot	-	3931 (>1.83)	450.86 (<0.40)
	d=6	oot	-	oot	-	5224 (>1.38)	361.61 (<0.32)
		i=8		i=14		i=22	
	d=0	4048 (1.00)	922.01 (1.00)	940 (1.00)	250.21 (1.00)	40 (1.00)	7.13 (1.00)
	d=1	3225 (1.26)	677.95 (0.74)	897 (1.05)	200.63 (0.80)	45 (0.91)	6.14 (0.86)
	d=2	2981 (1.36)	588.24 (0.64)	687 (1.37)	151.52 (0.61)	42 (0.96)	4.92 (0.69)
or-chain-113.fg	d=3	2454 (1.65)	486.12 (0.53)	704 (1.34)	125.95 (0.50)	40 (1.01)	3.54 (0.50)
(1416,2,40,83)	d=4	2327 (1.74)	395.83 (0.43)	763 (1.23)	95.09 (0.38)	42 (0.95)	2.81 (0.39)
	d=5	2442 (1.66)	296.26 (0.32)	797 (1.18)	76.51 (0.31)	47 (0.86)	2.26 (0.32)
	d=6	2790 (1.45)	227.63 (0.25)	1024 (0.92)	60.46 (0.24)	55 (0.73)	1.77 (0.25)
		i=6		i=14		i=22	
	d=0	4555 (1.00)	989.30 (1.00)	485 (1.00)	123.22 (1.00)	23 (1.00)	1.96 (1.00)
	d=1	3769 (1.21)	724.32 (0.73)	432 (1.12)	96.52 (0.78)	25 (0.89)	1.76 (0.90)
	d=2	3005 (1.52)	550.05 (0.56)	337 (1.44)	67.55 (0.55)	25 (0.92)	1.40 (0.72)
or-chain-140.fg	d=3	2132 (2.14)	370.58 (0.37)	304 (1.60)	50.38 (0.41)	24 (0.95)	1.12 (0.57)
(1260,2,32,79)	d=4	1604 (2.84)	226.90 (0.23)	297 (1.63)	39.68 (0.32)	24 (0.93)	1.04 (0.53)
	d=5	1403 (3.25)	152.70 (0.15)	370 (1.31)	33.30 (0.27)	26 (0.87)	0.94 (0.48)
	d=6	1156 (3.94)	93.02 (0.09)	514 (0.94)	26.82 (0.22)	28 (0.82)	0.76 (0.39)
		i=16		i=18		i=22	
	d=0	3392 (1.00)	776.00 (1.00)	1347 (1.00)	332.59 (1.00)	590 (1.00)	135.98 (1.00)
	d=1	3445 (0.98)	672.30 (0.87)	1521 (0.89)	292.67 (0.88)	583 (1.01)	115.44 (0.85)
	d=2	3049 (1.11)	531.20 (0.68)	1094 (1.23)	200.99 (0.60)	446 (1.32)	74.57 (0.55)
or-chain-202.fg	d=3	3037 (1.12)	443.17 (0.57)	1016 (1.33)	159.28 (0.48)	433 (1.36)	62.09 (0.46)
(1138,2,57,99)	d=4	3465 (0.98)	385.27 (0.50)	1163 (1.16)	138.94 (0.42)	493 (1.20)	54.06 (0.40)
	d=5	3833 (0.88)	286.23 (0.37)	1304 (1.03)	115.07 (0.35)	536 (1.10)	44.13 (0.32)
	d=6	5345 (0.63)	247.75 (0.32)	1815 (0.74)	102.03 (0.31)	801 (0.74)	37.78 (0.28)
		i=14		i=16		i=20	
	d=0	5360 (1.00)	1051.18 (1.00)	3179 (1.00)	641.34 (1.00)	1860 (1.00)	381.76 (1.00)
	d=1	4292 (1.25)	736.64 (0.70)	3309 (0.96)	558.36 (0.87)	1829 (1.02)	323.42 (0.85)
	d=2	3554 (1.51)	553.47 (0.53)	2420 (1.31)	382.38 (0.60)	1661 (1.12)	271.88 (0.71)
or-chain-230.fg	d=3	3325 (1.61)	463.20 (0.44)	2501 (1.27)	340.48 (0.53)	1764 (1.05)	245.41 (0.64)
(1338,2,61,109)	d=4	3583 (1.50)	406.36 (0.39)	2518 (1.26)	271.47 (0.42)	1853 (1.00)	202.43 (0.53)
	d=5	4723 (1.13)	323.11 (0.31)	2814 (1.13)	212.02 (0.33)	2202 (0.84)	166.51 (0.44)
	d=6	4779 (1.12)	200.85 (0.19)	3360 (0.95)	173.98 (0.27)	2832 (0.66)	143.76 (0.38)
		i=8		i=14		i=22	
	d=0	1936 (1.00)	473.14 (1.00)	698 (1.00)	174.41 (1.00)	34 (1.00)	4.99 (1.00)
	d=1	1455 (1.33)	318.79 (0.67)	654 (1.07)	138.89 (0.80)	39 (0.86)	4.43 (0.89)
	d=2	1221 (1.59)	240.71 (0.51)	593 (1.18)	112.57 (0.65)	39 (0.87)	3.79 (0.76)
or-chain-8.fg	d=3	1072 (1.81)	192.22 (0.41)	574 (1.21)	93.09 (0.53)	36 (0.94)	2.53 (0.51)
(1195,2,42,80)	d=4	1034 (1.87)	151.75 (0.32)	577 (1.21)	75.34 (0.43)	38 (0.88)	2.09 (0.42)
	d=5	1094 (1.77)	127.13 (0.27)	682 (1.02)	62.81 (0.36)	43 (0.79)	1.72 (0.34)
	d=6	1117 (1.73)	87.01 (0.18)	904 (0.77)	56.84 (0.33)	53 (0.64)	1.50 (0.30)

Table 8: Selected **promedas** instances: "time" indicates the CPU time in seconds (speedup over baseline) and "nodes" indicates the number of OR nodes expanded in *millions* of nodes (ratio relative to baseline) In a time column, 'oot' that the time limit of 2 hours was exceeded. The problem parameters are also provided for each instance (n: number of variables, z: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time is boxed.

4. This is due to how a large number of the instances are trivial to solve (less than 30 seconds of runtime) once using a stronger heuristic (typically an i-bound of 14). Increasing the heuristic strength further, most of the instances are solved easily, except for a few of the hardest ones such as *or-chain-108* in **Table 8**, which still exhibit significant speedup.

In this benchmark, which contains some hard instances, we see for the first time the power of look-ahead when memory restrictions allow only relatively weak heuristics. Indeed, here we see more than before that look-ahead improves with depths even when we have the strongest heuristics that we can compile under the memory constraints.

5.1.4 DBN

This benchmark contains problems derived from deep belief networks. **Table 9** shows the detailed results for representative instances. On the hardest instance we were able to solve (*rus2-20-40-9-3*), we see a significant improvement using the lowest i-bound of 8, and a look-ahead depth of 4 resulting in a runtime of 1744 seconds compared with the baseline yielding 6171 seconds, a speedup of 3.54. Indeed, the number of nodes expanded is reduced by about 90.5%. We observe improved performance for many instances as we increase the i-bound. The baseline time is generally at least twice that of the look-ahead depth of 1.

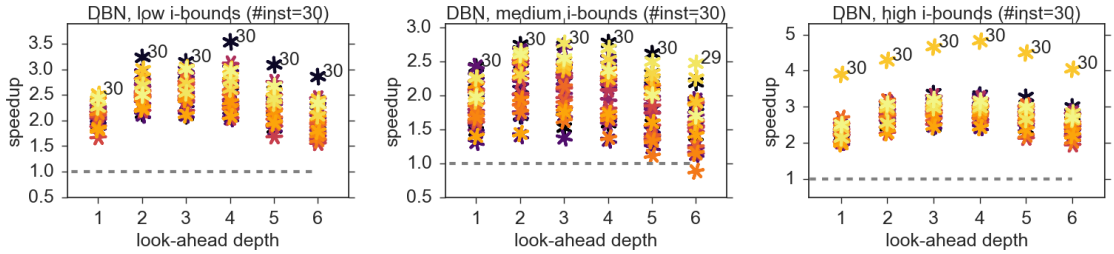


Figure 21: Solved **DBN** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

In **Figure 21**, we see that look-ahead nearly always improves over the baseline. For low i-bounds, the speedups range between 1.5 to 3.5 for all instances. This range decreases for medium i-bounds, but in nearly all cases look-ahead produces gains over the baseline. Note that the number of nodes expanded at an i-bound of 12 is greater than the number at an i-bound of 10. This is an example of a case where higher i-bounds may result in weaker heuristics, due to the unpredictable behavior of partitioning given that all of the functions in this benchmark are binary, yet having high induced width in the model as a whole. Partitioning has been shown to be an important factor in the quality of MBE heuristics (Rollon, Larrosa, & Dechter, 2013). At the same time, the minimum speedup increases for the high i-bounds, suggesting more errors in the heuristic, which look-ahead manages to exploit.

In summary, all of the instances in this benchmark have structure where all of the partitioning occurs at the leaves of the pseudo-tree. As a result, it is easy to identify where look-ahead should be performed to be cost-effective (near the leaves of the search space). Furthermore, the relative errors are extremely high for this benchmark, which can

instance (n, z, w^*, h)	depth	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)
		i=8		i=10		i=12	
	d=0	271 (1.00)	3.50 (1.00)	47 (1.00)	0.64 (1.00)	498 (1.00)	5.57 (1.00)
	d=1	126 (2.16)	2.02 (0.58)	28 (1.68)	0.39 (0.61)	197 (2.53)	3.04 (0.55)
	d=2	111 (2.44)	1.19 (0.34)	24 (1.95)	0.24 (0.38)	166 (3.01)	1.68 (0.30)
rus2-20-40-1-1	d=3	110 (2.46)	0.71 (0.20)	26 (1.81)	0.15 (0.24)	166 (2.99)	0.94 (0.17)
(70,2,29,29)	d=4	119 (2.28)	0.43 (0.12)	28 (1.66)	0.09 (0.15)	164 (3.03)	0.53 (0.10)
	d=5	137 (1.97)	0.26 (0.07)	35 (1.33)	0.06 (0.09)	189 (2.64)	0.30 (0.05)
	d=6	158 (1.72)	0.16 (0.04)	41 (1.14)	0.04 (0.06)	210 (2.37)	0.17 (0.03)
		i=8		i=10		i=12	
	d=0	264 (1.00)	3.09 (1.00)	104 (1.00)	1.21 (1.00)	386 (1.00)	4.28 (1.00)
	d=1	120 (2.21)	1.70 (0.55)	53 (1.97)	0.67 (0.56)	169 (2.28)	2.27 (0.53)
	d=2	97 (2.73)	0.95 (0.31)	42 (2.47)	0.39 (0.32)	129 (3.00)	1.22 (0.29)
rus2-20-40-3-3	d=3	95 (2.79)	0.54 (0.17)	42 (2.49)	0.22 (0.18)	122 (3.17)	0.66 (0.15)
(70,2,29,29)	d=4	99 (2.68)	0.31 (0.10)	48 (2.19)	0.13 (0.11)	131 (2.95)	0.36 (0.09)
	d=5	108 (2.45)	0.18 (0.06)	48 (2.15)	0.08 (0.06)	142 (2.71)	0.20 (0.05)
	d=6	119 (2.21)	0.11 (0.03)	57 (1.84)	0.04 (0.04)	147 (2.62)	0.11 (0.03)
		i=8		i=10		i=12	
	d=0	307 (1.00)	3.89 (1.00)	53 (1.00)	0.76 (1.00)	539 (1.00)	6.41 (1.00)
	d=1	150 (2.05)	2.27 (0.58)	30 (1.79)	0.47 (0.61)	240 (2.25)	3.52 (0.55)
	d=2	123 (2.50)	1.35 (0.35)	26 (2.01)	0.29 (0.38)	206 (2.61)	1.97 (0.31)
rus2-20-40-4-1	d=3	124 (2.48)	0.81 (0.21)	29 (1.83)	0.18 (0.24)	188 (2.86)	1.11 (0.17)
(70,2,29,29)	d=4	134 (2.30)	0.49 (0.13)	32 (1.67)	0.11 (0.15)	201 (2.68)	0.64 (0.10)
	d=5	160 (1.92)	0.30 (0.08)	39 (1.36)	0.07 (0.09)	211 (2.55)	0.37 (0.06)
	d=6	187 (1.64)	0.19 (0.05)	46 (1.16)	0.05 (0.06)	246 (2.20)	0.22 (0.03)
		i=8		i=10		i=12	
	d=0	1517 (1.00)	17.09 (1.00)	569 (1.00)	6.76 (1.00)	2156 (1.00)	22.32 (1.00)
	d=1	698 (2.17)	9.47 (0.55)	234 (2.43)	3.83 (0.57)	861 (2.50)	11.89 (0.53)
	d=2	562 (2.70)	5.32 (0.31)	213 (2.67)	2.21 (0.33)	691 (3.12)	6.41 (0.29)
rus2-20-40-5-2	d=3	523 (2.90)	3.03 (0.18)	209 (2.72)	1.29 (0.19)	680 (3.17)	3.48 (0.16)
(70,2,29,29)	d=4	544 (2.79)	1.75 (0.10)	231 (2.47)	0.76 (0.11)	661 (3.26)	1.92 (0.09)
	d=5	621 (2.44)	1.02 (0.06)	269 (2.11)	0.45 (0.07)	750 (2.88)	1.06 (0.05)
	d=6	659 (2.30)	0.61 (0.04)	292 (1.95)	0.27 (0.04)	746 (2.89)	0.60 (0.03)
		i=8		i=10		i=12	
	d=0	350 (1.00)	4.21 (1.00)	161 (1.00)	1.80 (1.00)	564 (1.00)	6.49 (1.00)
	d=1	154 (2.27)	2.36 (0.56)	81 (2.00)	1.02 (0.57)	232 (2.43)	3.48 (0.54)
	d=2	130 (2.70)	1.34 (0.32)	70 (2.30)	0.59 (0.33)	189 (2.98)	1.89 (0.29)
rus2-20-40-8-2	d=3	128 (2.73)	0.78 (0.18)	64 (2.53)	0.34 (0.19)	183 (3.08)	1.04 (0.16)
(70,2,29,29)	d=4	134 (2.62)	0.46 (0.11)	64 (2.53)	0.20 (0.11)	183 (3.09)	0.58 (0.09)
	d=5	149 (2.35)	0.27 (0.06)	69 (2.33)	0.12 (0.07)	193 (2.92)	0.33 (0.05)
	d=6	167 (2.10)	0.17 (0.04)	71 (2.26)	0.07 (0.04)	212 (2.66)	0.19 (0.03)
		i=8		i=10		i=12	
	d=0	6171 (1.00)	58.00 (1.00)	1906 (1.00)	23.11 (1.00)	oot	-
	d=1	2620 (2.36)	31.70 (0.55)	903 (2.11)	13.03 (0.56)	2913 (>2.47)	40.58 (<0.53)
	d=2	1905 (3.24)	17.52 (0.30)	713 (2.67)	7.48 (0.32)	2435 (>2.96)	21.53 (<0.28)
rus2-20-40-9-3	d=3	1956 (3.15)	9.74 (0.17)	696 (2.74)	4.30 (0.19)	2128 (>3.38)	11.47 (<0.15)
(70,2,29,29)	d=4	1744 (3.54)	5.48 (0.09)	748 (2.55)	2.50 (0.11)	2158 (>3.34)	6.17 (<0.08)
	d=5	1995 (3.09)	3.07 (0.05)	820 (2.32)	1.44 (0.06)	2198 (>3.28)	3.33 (<0.04)
	d=6	2156 (2.86)	1.75 (0.03)	1035 (1.84)	0.83 (0.04)	2403 (>3.00)	1.81 (<0.02)

Table 9: Selected **DBN** instances: "time" indicates the CPU time in seconds (speedup over baseline) and "nodes" indicates the number of OR nodes expanded in *millions* of nodes (ratio relative to baseline) In a time column, 'oot' that the time limit of 2 hours was exceeded. The problem parameters are also provided for each instance (n: number of variables, z: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time is boxed.

be exploited by look-ahead to a great extent. Overall, across all i-bounds shown here, we see that a look-ahead depth of 1 leads to most of the improvement, with higher depths having incremental positive impact to a depth of 4.

instance (n, z, w^*, h)	depth	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)	time (speedup)	nodes (ratio)
		i=12		i=14		i=16	
	d=0	2566 (1.00)	598.29 (1.00)	1876 (1.00)	433.11 (1.00)	14 (1.00)	3.22 (1.00)
	d=1	2522 (1.02)	492.63 (0.82)	1915 (0.98)	372.87 (0.86)	18 (0.77)	2.94 (0.91)
	d=2	2308 (1.11)	402.01 (0.67)	1559 (1.20)	276.33 (0.64)	17 (0.81)	2.42 (0.75)
grid20x20.f10.wrap	d=3	2427 (1.06)	355.08 (0.59)	1667 (1.13)	246.56 (0.57)	18 (0.78)	2.16 (0.67)
(400,2,44,68)	d=4	2622 (0.98)	303.87 (0.51)	1963 (0.96)	221.55 (0.51)	20 (0.68)	1.94 (0.60)
	d=5	2689 (0.95)	232.46 (0.39)	2127 (0.88)	175.57 (0.41)	26 (0.54)	1.78 (0.55)
	d=6	3186 (0.81)	198.89 (0.33)	2242 (0.84)	132.04 (0.30)	33 (0.42)	1.54 (0.48)
		i=10		i=12		i=14	
	d=0	148 (1.00)	33.63 (1.00)	126 (1.00)	32.05 (1.00)	95 (1.00)	24.37 (1.00)
	d=1	138 (1.08)	27.24 (0.81)	132 (0.95)	27.81 (0.87)	97 (0.98)	19.94 (0.82)
	d=2	136 (1.09)	23.77 (0.71)	126 (1.00)	24.01 (0.75)	86 (1.10)	16.59 (0.68)
grid20x20.f5.wrap	d=3	151 (0.99)	21.60 (0.64)	95 (1.32)	15.93 (0.50)	90 (1.05)	14.48 (0.59)
(400,2,45,69)	d=4	159 (0.94)	18.11 (0.54)	95 (1.32)	13.22 (0.41)	92 (1.04)	11.90 (0.49)
	d=5	190 (0.78)	15.34 (0.46)	95 (1.32)	10.30 (0.32)	95 (1.00)	9.74 (0.40)
	d=6	238 (0.62)	12.18 (0.36)	100 (1.26)	8.16 (0.25)	109 (0.87)	8.05 (0.33)
		i=18		i=20		i=22	
	d=0	oot	-	2907 (1.00)	562.40 (1.00)	845 (1.00)	156.13 (1.00)
	d=1	oot	-	2934 (0.99)	504.87 (0.90)	851 (0.99)	141.49 (0.91)
	d=2	oot	-	2732 (1.06)	430.81 (0.77)	667 (1.27)	104.68 (0.67)
grid40x40.f10	d=3	oot	-	2923 (0.99)	397.71 (0.71)	665 (1.27)	93.53 (0.60)
(1600,2,52,148)	d=4	oot	-	3276 (0.89)	361.41 (0.64)	695 (1.22)	83.73 (0.54)
	d=5	oot	-	4211 (0.69)	337.03 (0.60)	767 (1.10)	73.28 (0.47)
	d=6	oot	-	6094 (0.48)	305.26 (0.54)	972 (0.87)	68.07 (0.44)
		i=16		i=18		i=20	
	d=0	4924 (1.00)	947.79 (1.00)	373 (1.00)	68.89 (1.00)	1177 (1.00)	213.90 (1.00)
	d=1	4908 (1.00)	877.94 (0.93)	386 (0.97)	65.21 (0.95)	932 (1.26)	151.49 (0.71)
	d=2	5049 (0.98)	845.76 (0.89)	369 (1.01)	57.45 (0.83)	859 (1.37)	135.76 (0.63)
grid40x40.f2	d=3	4782 (1.03)	727.57 (0.77)	392 (0.95)	54.90 (0.80)	859 (1.37)	123.10 (0.58)
(1600,2,52,157)	d=4	4873 (1.01)	591.59 (0.62)	451 (0.83)	53.48 (0.78)	911 (1.29)	111.25 (0.52)
	d=5	6389 (0.77)	554.56 (0.59)	567 (0.66)	52.05 (0.76)	1132 (1.04)	105.15 (0.49)
	d=6	oot	-	811 (0.46)	49.68 (0.72)	1596 (0.74)	105.20 (0.49)
		i=18		i=20		i=22	
	d=0	oot	-	oot	-	543 (1.00)	92.97 (1.00)
	d=1	oot	-	oot	-	393 (1.38)	57.67 (0.62)
	d=2	oot	-	6231 (>1.16)	1068.42 (<0.79)	383 (1.42)	50.90 (0.55)
grid40x40.f5	d=3	oot	-	7147 (>1.01)	975.40 (<0.72)	421 (1.29)	49.22 (0.53)
(1600,2,52,136)	d=4	oot	-	oot	-	504 (1.08)	45.44 (0.49)
	d=5	oot	-	oot	-	730 (0.74)	48.60 (0.52)
	d=6	oot	-	oot	-	868 (0.63)	37.27 (0.40)

Table 10: Selected **grid** instances: "time" indicates the CPU time in seconds (speedup over baseline) and "nodes" indicates the number of OR nodes expanded in *millions* of nodes (ratio relative to baseline) In a time column, 'oot' that the time limit of 2 hours was exceeded. The problem parameters are also provided for each instance (n: number of variables, z: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time is boxed.

5.1.5 Grids

This benchmark is based on binary grid structured networks. **Table 10** shows the detailed results for representative instances. We observe generally modest speedups for this bench-

mark. For instance, on *grid40x40.f2* with an i-bound of 16, the baseline achieved a runtime of 4924 seconds where the best setting of the depth of 3 only reduced this runtime to 4782. Indeed, we only observe roughly a 23.2% reduction in the number of nodes expanded in this case. Though deeper depths lead to additional reduction, it is not cost-effective.

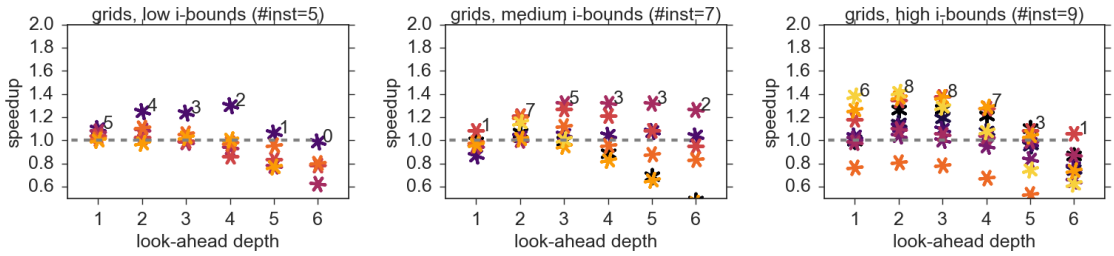


Figure 22: Solved **grid** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

In **Figure 22**, considering the instances that were solved with the low i-bounds, we do see that a look-ahead depth of 1 always has a positive impact, though only marginally, with the best speedup at around 1.1. As the depth increases, the number of instances that are improved decreases, though there is one instance that exhibits additional speedup with more look-ahead. At higher i-bounds, many instances benefit from look-ahead, but with modest speedups up to 1.4.

In summary, we see on this benchmark that using high i-bounds yields accurate heuristics and therefore little search. As such, there are few cases where look-ahead can exploit any error. Overall, look-ahead improves performance on this benchmark, but the improvements are modest.

5.1.6 Summary

We conclude our evaluation of exact solutions with the following takeaways.

1. **Look-ahead improves more for weak heuristics**: the purpose of look-ahead is to *correct* the error in heuristics and this conclusion is supported by our evaluation. We observed that lower i-bounds tended to benefit more in benchmarks where high i-bounds were fairly accurate without look-ahead (e.g. **pedigree** and **LargeFam3**). For harder instances where the heuristic was relatively weak even the highest i-bound under our memory constraints, look-ahead was also beneficial.

2. **Depth is a significant control parameter that should be used for the best balancing**: Across the benchmarks, the best depth tended to range between 2 and 3, suggesting that a modest depth of look-ahead is best.

3. **Look-ahead is a method that enables trading memory for time**: In cases where even the highest i-bound that memory allows is still weak (instances having runtimes in hundreds of seconds for the baseline), spending time to perform look-ahead is a cost-effective way improve the heuristic without spending more memory.

4. **Bucket error is a useful metric for enabling cost-effective look-ahead:** We see that when the heuristic is quite accurate the look-ahead often does not improve, but also does not deteriorate the performance by much.

5.2 Evaluating the Anytime Behavior

Benchmark	# inst	n	z	w	h	$\|F\|$	a
LargeFam3	39	950	3	34	66	1457	4
		2180	3	67	153	3772	4
Type4	31	3907	5	21	300	5749	4
		8984	5	48	925	13585	4
DBN	74	70	2	29	29	16167	2
		310	2	109	109	99927	2
Grids	12	1600	2	95	153	4801	2
		6400	2	196	341	19201	2

Table 11: Benchmark statistics for anytime evaluation. # inst - number of instances, n - number of variables, z - 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.

We next show results on the impact of our look-ahead scheme on anytime behavior. For this evaluation, we selected those instances selection focused on instances that could not be solved with the baseline within 7200 seconds. Thus, the **pedigree** and **promedas** benchmarks are omitted from this part of the evaluation, but we include the type4 benchmark, which has no instances we were able solve exactly. Additionally, we included instances only if we were able to generate at least one anytime solution. Overall, this resulted in a total of 156 instances with induced widths ranging from 21 to 196. **Table 11** shows problem statistics on the selected instances across each benchmark.

In this experiment we chose the highest i-bound fitting in memory, plus another lower one to demonstrate the effects of varying the heuristic strength.

We show results in **Figures 23, 25, 27, and 29**. In these figures, we plot the cost of the best solution found as a function of time on selected instances.

Lastly, to summarize over each benchmark, we plot, for each instance, the normalized relative accuracy for selected i-bounds and look-ahead depths at different time points (60, 1800, 3600, and 7200 seconds) compared with no look-ahead. For a given i-bound, we define the normalized relative accuracy as $\frac{C_w - C}{C_w - C_b}$, where C_w and C_b are the worst and best solutions obtained at any time over any look-ahead depth. Thus, an algorithm is better if it obtains a higher relative accuracy. The differently colored points represent the different problem instances over the benchmark. We summarize this by annotating each plot with a tuple *(#wins for look-ahead/#wins for baseline/#ties)*. For clarity, we exclude instances from a plot if no solutions were found by both the baseline and look-ahead method by any time point. These are shown in **Figures 24, 26, 28, and 30**.

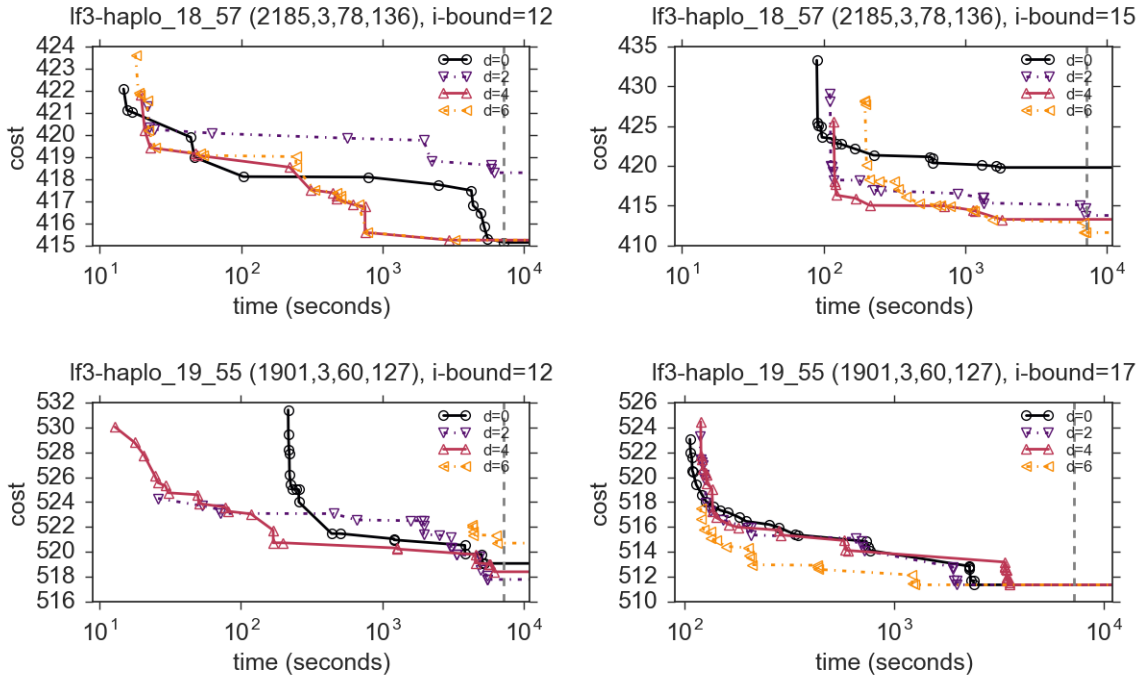


Figure 23: **LargeFam3** instances: Anytime plots across 2 different i-bounds for two selected instances. As usual, the tuple next to the problem instance name indicates (n: number of variables, z: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower plots early on are better.

5.2.1 LargeFam3

In **Figure 23**, we see that for *lf3-haplo_18_57* with an i-bound of 12 and with look-ahead depths of 3 and higher, we quickly obtain a better solution than the baseline at the start of the time period. It is not until near the timeout that the baseline manages to obtain a solution of the same quality as look-ahead with a depth of 4 or higher. Moving to an i-bound of 15, all of the look-ahead schemes shown outperform the baseline. Furthermore, the best solution is obtained near timeout with a look-ahead depth of 6.

Figure 24 summarizes instances for depths of 2 and 5. Starting with the lowest i-bound of 12 and a depth of 2, we observe at 60 seconds that look-ahead has a slight edge over the baseline, with 6 instances where it improves and 4 instances where it is outperformed by the baseline. However, we see that look-ahead performs better moving forward in time.

Increasing the look-ahead depth to 5, the results are similar, but puts look-ahead at an advantage on more instances. Increasing the i-bound to 16, look-ahead can still help when the depth is lower, but at a higher depth, it tends to be less cost-effective. Across all the plots at anytime point, it is worth nothing that there are a number of cases where the relative accuracy of the baseline is zero while look-ahead obtains non-zero relative accuracies,

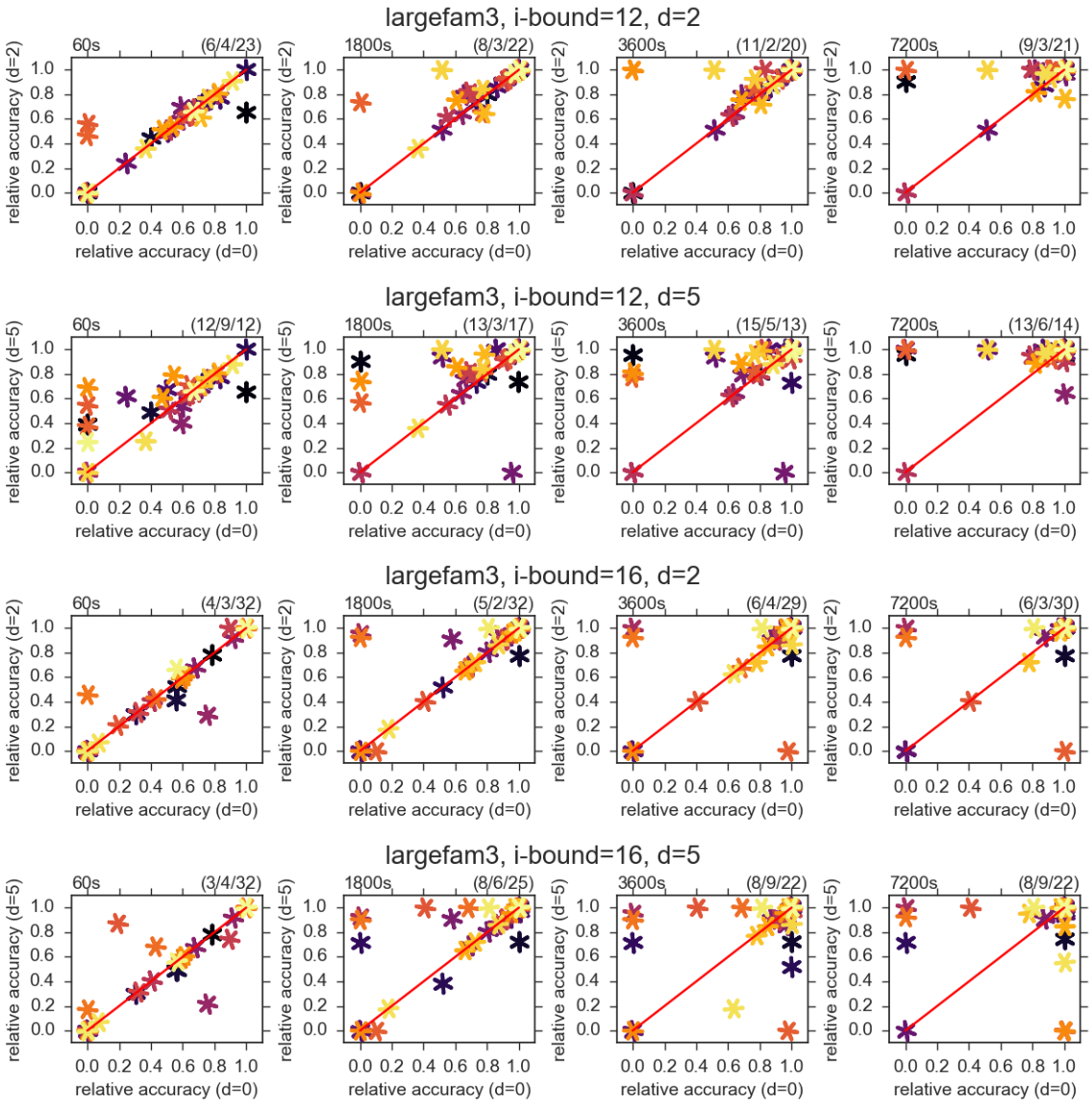


Figure 24: **LargeFam3.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular i-bound/depth and each column corresponds to a time point. We provide summary statistics for each plot with a tuple that counts the numbers of (#wins for look-ahead/#wins for baseline/#ties). Instances above the diagonal line indicate better accuracy for the look-ahead scheme.

indicating that there are a number of instances where look-ahead manages to produce much better solutions all the time.

In summary, on this benchmark, a high look-ahead depth is quite useful when the i-bound is lower. On the other hand, a lower look-ahead depth is preferable when the heuristic is stronger.

5.2.2 Type4

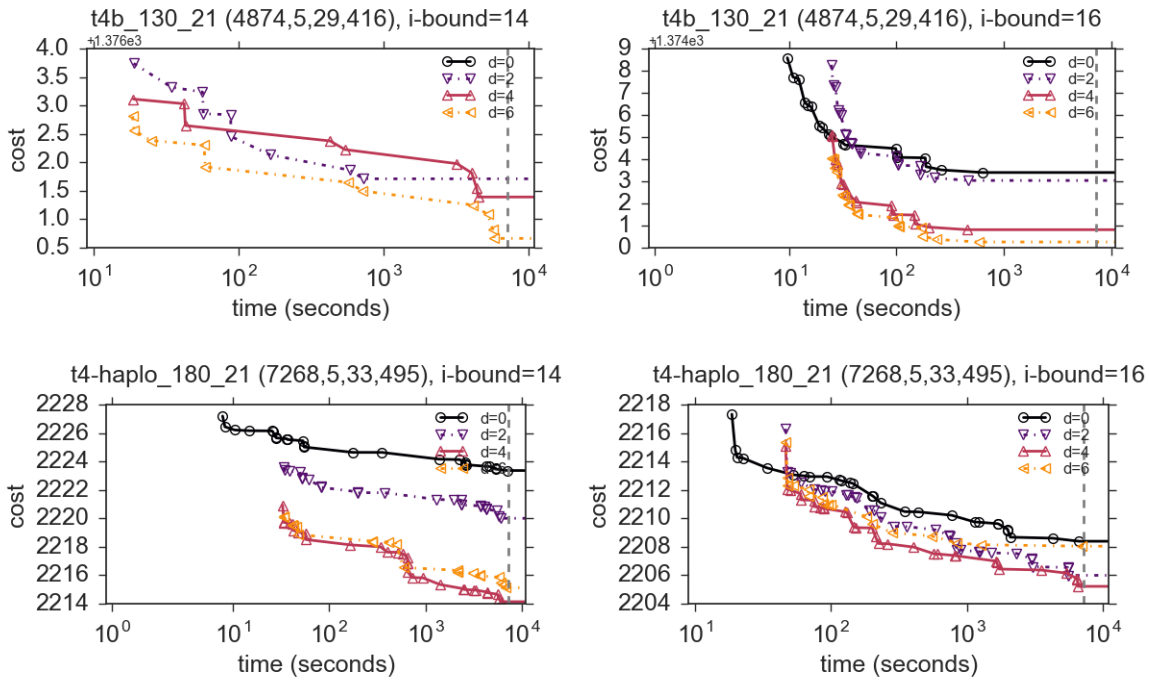


Figure 25: **Type4** instances: Anytime plots across 2 different i-bounds for two selected instances. As usual, the tuple next to the problem instance name indicates (n: number of variables, z: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower plots early on are better.

This is another benchmark based on genetic linkage analysis, but contains problems that are harder than those in **LargeFam3**. One factor contributing to the difficulty is a large domain size of 5. In **Figure 25**, for instance *t4b_130_21* with an i-bound of 14, the baseline does not produce any solution during the entire time period (namely, the corresponding line is not present). Comparing the look-ahead depths against each other, a depths of 6 is superior. Increasing the i-bound to 16, the heuristic becomes strong enough so that the baseline produces solutions and it is now also the first to do so. However, it is outperformed by look-ahead of all depths in under 100 seconds, with depths of 4 and higher producing considerably better solutions. Overall, look-ahead is usually superior to the baseline, with a bit of a preference for deeper depth regardless of heuristic strength.

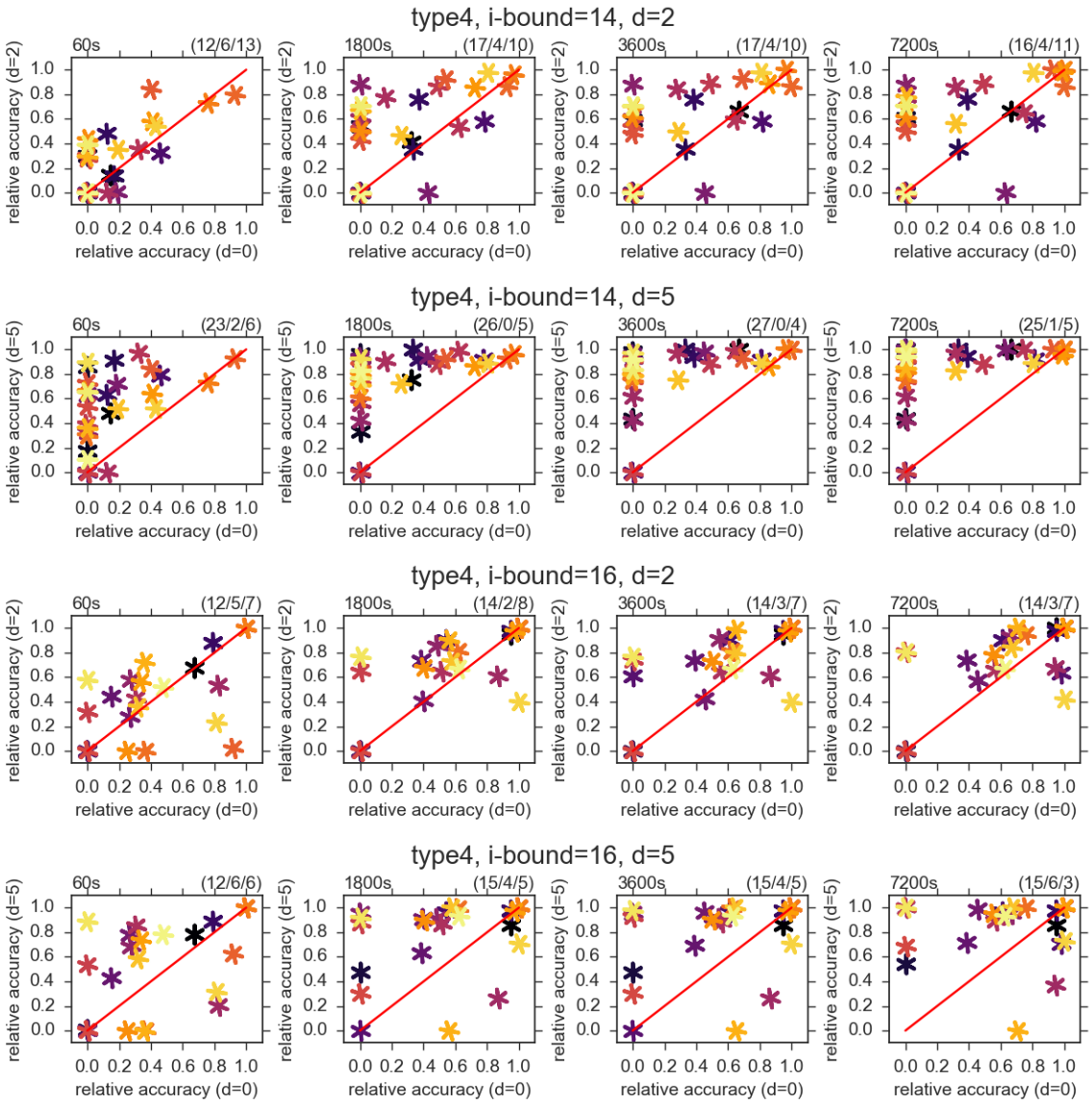


Figure 26: **Type4.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular i-bound/depth and each column corresponds to a time point. We provide summary statistics for each plot with a tuple that counts the numbers of (#wins for look-ahead/#wins for baseline/#ties). Instances above the diagonal line indicate better accuracy for the look-ahead scheme.

In **Figure 26**, we summarize over the benchmark for depths 2 and 5. Starting with the lower i-bound of 14 and lower look-ahead depth of 2, we observe that look-ahead produces better solutions early on many instances at the 60 second mark. As time advances, additional instances also benefit from look-ahead. Increasing the depth to 5, look-ahead dominates the baseline. Increasing the i-bound to 16, we can observe that look-ahead remains dominant over the baseline regardless of depth. Additionally, we see that there are a number of instances where look-ahead manages to produce solutions of non-zero relative accuracy while the baseline remains at zero, indicating a clear dominance over the baseline in solution quality by look-ahead.

In summary, while we were not able to find any exact solutions within the time limit for instances in this benchmark, look-ahead clearly has a positive impact when considering anytime solutions, even under high i-bounds.

5.2.3 DBN

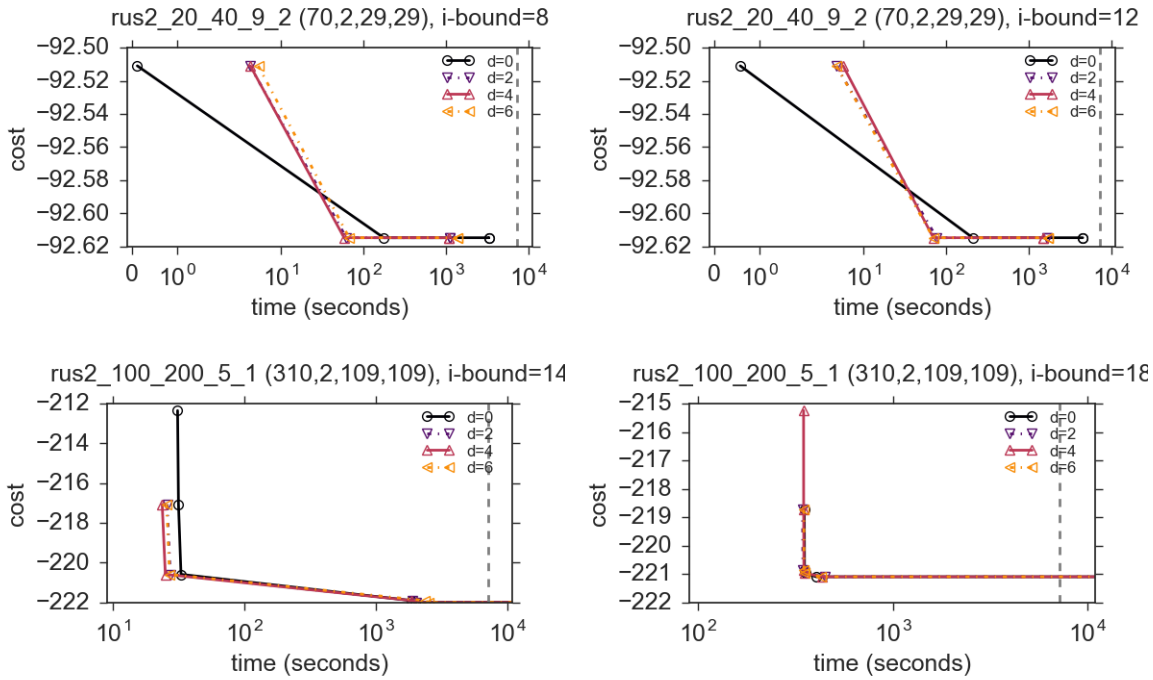


Figure 27: **DBN** instances: Anytime plots across 2 different i-bounds for two selected instances. As usual, the tuple next to the problem instance name indicates (n: number of variables, z: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower plots early on are better.

In **Figure 27**, we observe little difference between look-ahead and the baseline. Indeed, across all the instances (including the 30 instances where exact solutions were achieved), we see in **Figure 28**, we see that this behavior is systematic for this benchmark. Although

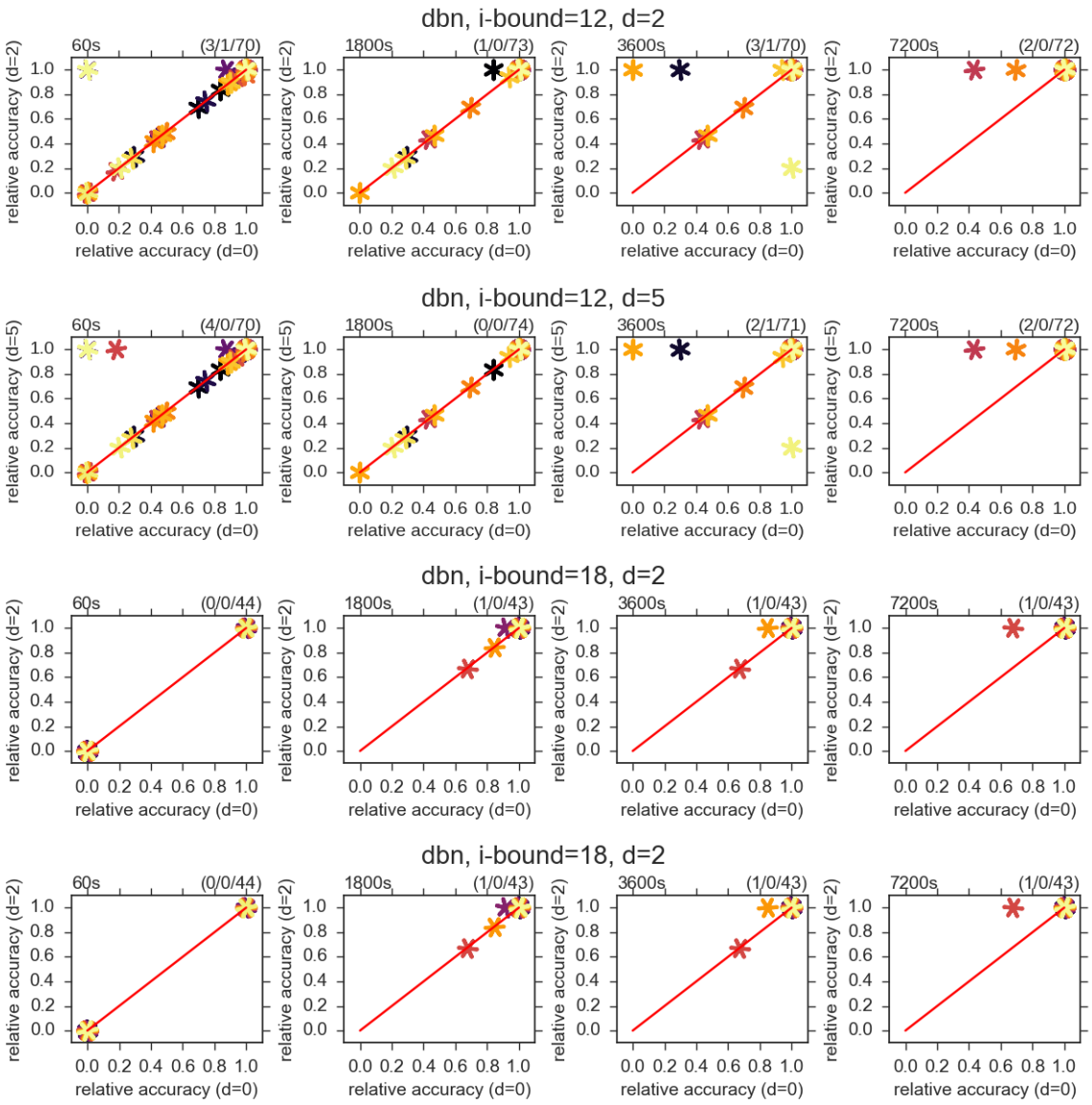


Figure 28: **DBN.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular i-bound/depth and each column corresponds to a time point. We provide summary statistics for each plot with a tuple that counts the numbers of (#wins for look-ahead/#wins for baseline/#ties). Instances above the diagonal line indicate better accuracy for the look-ahead scheme.

we saw impressive speedups for look-ahead when finding exact solutions, we see here that the exact solution is actually obtained with a less significant margin of time between the two competing schemes. For example, in the anytime plot for *rus-2-20-40-9-2* (**Figure 27**, top), the exact solution is found by all look-ahead depths in less than 100 seconds, while the base line took about 200 seconds. The rest of the time is spent proving that the solution is optimal. However, look-ahead methods achieve this 2 to 3 times faster than the base line.

In summary, in the context of anytime behavior, though look-ahead here results in a speedup for reaching the exact solution, there is little to no variance in the solution quality over most the time period since the baseline also manages to reach the exact solution relatively early.

5.2.4 Grids

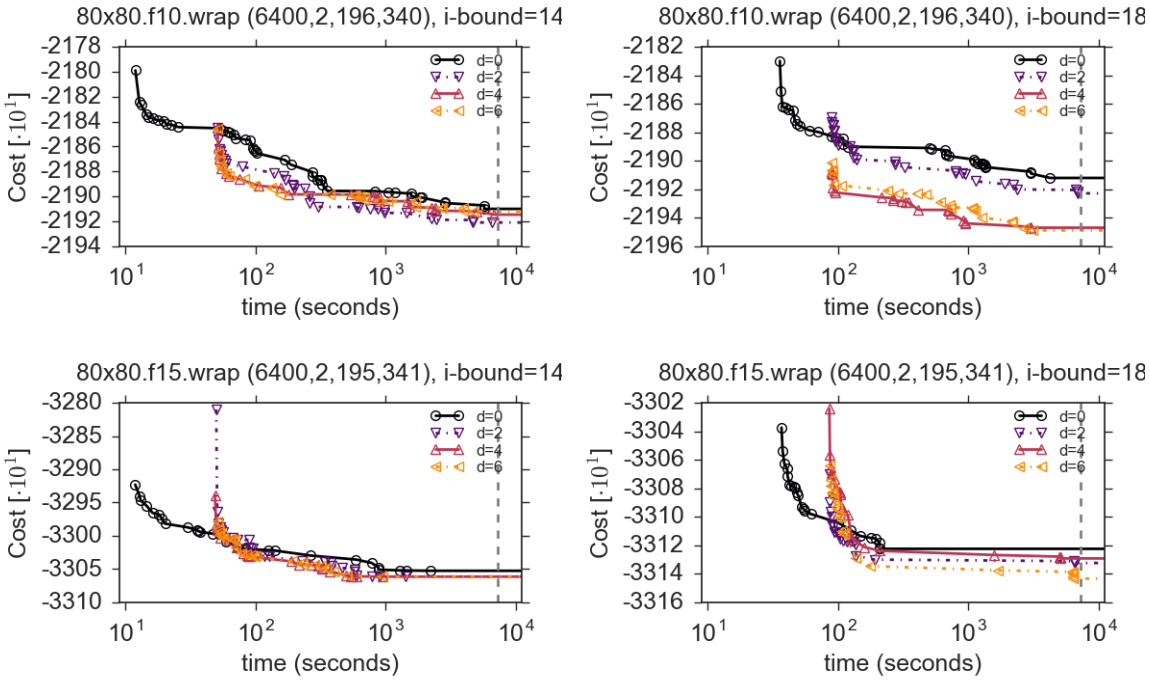


Figure 29: **Grid** instances: Anytime plots across 2 different i-bounds for two selected instances. As usual, the tuple next to the problem instance name indicates (n: number of variables, z: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower plots early on are better.

In **Figure 29**, for *80x80.f10.wrap* using an i-bound of 14, the baseline generates a solution earlier than look-ahead, but all look-ahead depths of 2 and higher produce better solutions by 100 seconds. The solution qualities converge towards the end, but all look-ahead depths manage to maintain leads over the baseline, with a depth of 2 performing the best. Moving to a higher i-bound of 18, the behavior at the start is similar. However, there is more

variance in the solutions between each setting, with depths of 4 and higher performing the best. The results for *80x80.f15.wrap* are similar, with look-ahead still outperforming the baseline, though there is less variance between the solutions.

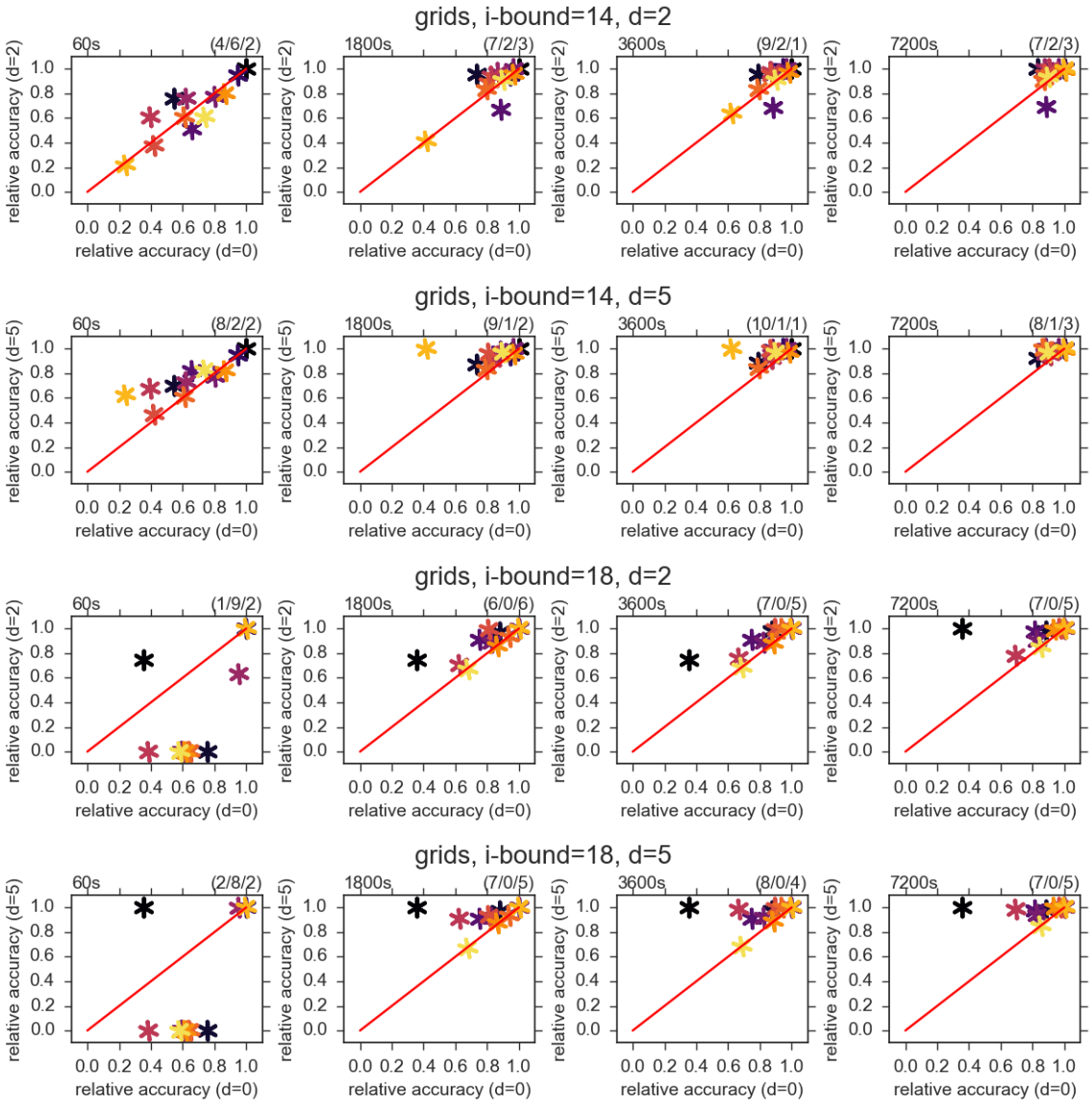


Figure 30: **Grids.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular i-bound/depth and each column corresponds to a time point. We provide summary statistics for each plot with a tuple that counts the numbers of (#wins for look-ahead/#wins for baseline/#ties). Instances above the diagonal line indicate better accuracy for the look-ahead scheme.

In **Figure 30**, for an i-bound of 14, at 60 seconds that a depth of 2 falls a bit short compared with the baseline. However, moving forward in time, look-ahead establishes a clear advantage. Increasing the depth to 5, the advantage starts at 60 seconds and this is maintained to the end. Increasing the i-bound to 18, the baseline outperforms look-ahead regardless of the depth at 60 seconds. Indeed, the relative accuracy of the solution for a number of instances is zero for look-ahead. However, past this, look-ahead establishes itself as the better performer regardless of depth, having better solutions on about half of the instances and matching the baseline on the other.

In summary, look-ahead always has a positive impact on this benchmark.

5.2.5 Summary

In shifting our focus to hard instances where we could not evaluate for exact solution within the time bound, our takeaways from Section 5.1 carry over to this evaluation of the anytime behavior. We re-iterate the first two points with discussion specific to this section.

First, on **look-ahead's impact on weak heuristics**, this evaluation further enforces its positive impact as the best heuristics are *relatively weaker* due to the difficulty of the problems. We see that the baseline tended to be outperformed by look-ahead on many instances for all of the benchmarks.

Next, on the **depth as a control parameter**, one difference is that the best depth tended to be *deeper* for anytime solutions. On each of the benchmarks, a depth of 5 tended to produce better solutions earlier on more instances compared with a depth of 2. Many of the instance-specific plots also show higher depths resulting in higher quality solutions being found earlier in general. It is worth noting that in many cases that as the depth increases, the *first* solution found improves. Thus, this suggests that deep look-ahead is particularly effective for guiding search early on to more promising parts of the search space.

5.3 Impact of the ϵ Parameter

As all of the experiments in the previous two sections used a fixed ϵ of 0.01 for generating the ϵ-pruned look-ahead subtrees, the question of the impact of the ϵ parameter remains. As discussed earlier in Section 3.5, less look-ahead is performed as ϵ increases since the look-ahead subtrees are pruned more aggressively. This opens up the opportunity for more focused look-ahead at parts of the search space with more significant errors. Clearly, as $\epsilon \to \infty$, the look-ahead scheme reduces to the baseline. Thus, adjusting ϵ is an alternative way to control the computational trade-off of look-ahead.

Figure 31 plots the anytime performance on two representative **largefam3** instance for two different i-bounds. For the look-ahead control parameters, we vary the depth on $\{2, 5\}$ and vary ϵ on $\{0, 0.01, 1, \infty\}$. These plots illustrate how strong look-ahead can be especially useful when the heuristics is weaker, as a depth of 5 with no pruning of the look-ahead subtrees ($\epsilon = 0$) tends to be the best performing when $i = 12$. Increasing ϵ here degrades the performance. On the other hand, we see that higher ϵ can be useful to mitigate the effects of excessive look-ahead at a depth of 5 when the heuristic is strong, as seen in the plot for *lf3-haplo-19-55* with $i = 17$.

We refer the reader to the work of Lam (2017a) for a full account of the experiments on the **largefam3** and **grid** benchmarks. Overall, we found that moving from $\epsilon = 0$ to

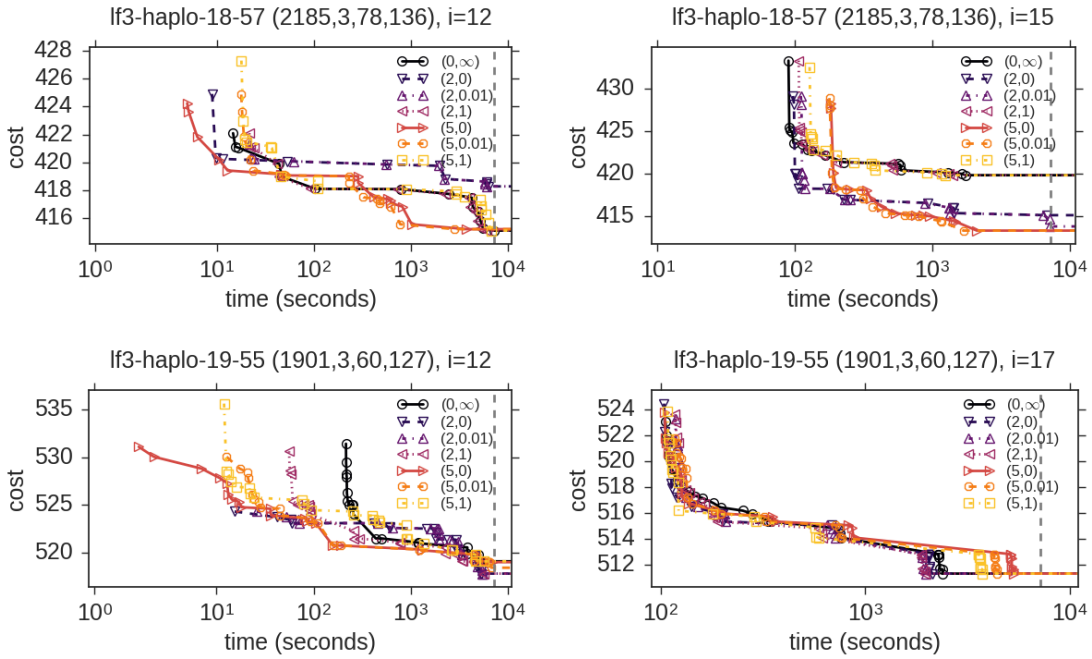


Figure 31: **LargeFam3** instances: Solutions obtained over time for different ϵ thresholds with look-ahead depths of 2 and 5. The information shown is similar to figures presented in Section 5.2.

$\epsilon = 0.01$ yielded the largest positive change in the speedup. Therefore, we used $\epsilon = 0.01$ as our default setting. Increasing ϵ further generally helps in making look-ahead more cost-effective for a fixed high depth, but this tended to approach the baseline using no look-ahead for lower depths. A slight caveat for $\epsilon = 0$ is that since no subtree pruning is required here, we can skip pre-processing, which allows search to commence sooner. This is especially relevant when evaluating for anytime solutions on extremely hard problems. On many of these, we found that strong look-ahead allows for a superior early solution, so for very low time bounds, subtree pruning is not cost-effective.

6. Conclusion

Look-ahead, known as the Bellman update, is a well-known technique to improve search when dealing with weak heuristics. In this paper we addressed the topic of making look-ahead cost effective in the context of AND/OR search for graphical models when using the MBE heuristic.

While most heuristic search literature takes the heuristic function as a black box, we try to gain insight and exploit some structural aspects in order to make look-ahead cost-effective. We have observed that in many benchmarks the heuristic function behaves in a non-monotone manner. Namely, as we expand more and more deep nodes, the evaluation function (which is a lower bound on the exact cost) grows along a path and yields potentially more accurate lower bounds. However, the change in accuracy (namely the nodes where the

Bellman update, or look-ahead, can improve the heuristic) does not happen evenly across the search space. In large regions there is no increase along many paths, while in others we observe changes occurring in well localized contiguous regions. Since look-ahead will have an impact only where the evaluation function increases, our goal was to predict where changes can occur in order to focus look-ahead in only the necessary places to improve accuracy of the heuristic.

Our paper introduced the concept of *local bucket-error* in order to predict look-ahead cost-effective regions. We showed that the local bucket error is equivalent to depth-1 residual (i.e, the gain of depth-1 look-ahead w.r.t. no look-ahead) and developed algorithm LBEE for its computation whose time and space complexity is exponential in a structural parameter called pseudo-width. LBEE can compile the depth-1 look-ahead in a pre-processing manner which can be consulted during search as lookup table. We also proposed two approximation schemes, average local error and sampled average local error, that mitigate the complexity when the full computation of the bucket-errors it is too high.

To go beyond depth 1, we aggregated depth-1 residuals to estimate higher depth residuals. We defined look-ahead tree and show that computing the look-ahead up to depth d is equivalent to computing a min-sum problem over a graphical sub-problem. Using information from the local bucket errors (or their estimates) we prune the tree in order to avoid useless look-ahead.

From our experimental evaluation we can conclude that look-ahead is potentially useful when dealing with weak heuristics, namely when the i-bound is not near the induced width of the problem. We observed that, in many cases our selective look-ahead allows solving instances faster with relative small look-ahead depths (2 or 3). For anytime solving we observed that higher depths (4 to 6) were more effective.

In future work we would like to change the algorithm parameters ϵ and d dynamically during the executions, since there do not seem to be universally good values for them. Also, we would like to move from the current variable-based look-ahead subtrees to context-dependent trees based on the current instantiation. Another promising idea is to determine the amount of look-ahead in terms of *look-ahead width* rather than depth (i.e, w_{pd} instead of d), which is a more faithful estimator of the look-ahead overhead when it is computed with inference algorithms.

Acknowledgments

This work was sponsored in part by NSF grants IIS-1065618, IIS-1254071, and IIS-1526842, the United States Air Force under Contract No. FA8750-14-C-0011 under the DARPA PPAML program, and by the Spanish MINECO under project TIN2015-69175-C4-3-R.

Appendix A. Proof of Proposition 1

Proof. We rewrite the $h^d(\bar{x}_p)$ term by using **Definition 10** and unrolling the recursive $h^{d-1}(\bar{x}_q)$ term. The unrolling produces an expression that alternates summations and minimizations d times. All the summations can be pushed inside the expression resulting in the following,

$$h^d(\bar{x}_p) = \min_{\bar{x}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} c(X_k, x_k) + \sum_{X_k \in \mathcal{T}_{p,d} - \mathcal{T}_{p,d-1}} h(\bar{x}_p, \bar{x}_{p,d}) \right\}$$

Next, we replace the cost $c(X_k, x_k)$ and the heuristic $h(\bar{x}_p, \bar{x}_{p,d})$ with the appropriate terms (**Definition 7** and **Equation 2**), yielding

$$h^d(\bar{x}_p) = \min_{\bar{x}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} \sum_{f \in B_k} f(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{X_k \in \mathcal{T}_{p,d} - \mathcal{T}_{p,d-1}} \Lambda_{(j,k)}(\bar{x}_p, \bar{x}_{p,d}) \right\} \quad (13)$$

We then further expand Λ terms using **Equation 1**,

$$h^d(\bar{x}_p) = \min_{\bar{x}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} \sum_{f \in B_k} f(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{s=1..r_j} \sum_{X_q \in \mathcal{T}_{p,d} \cup \bar{X}_p} \lambda^s_{j \to q}(\bar{x}_p, \bar{x}_{p,d}) \right\}$$

Next, the term's inner sum

$$\sum_{X_q \in \mathcal{T}_{p,d} \cup \bar{X}_p} \lambda^s_{j \to q}(\bar{x}_p, \bar{x}_{p,d})$$

can be broken down into

$$\sum_{X_q \in \mathcal{T}_{p,d}} \lambda^s_{j \to q}(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_q \in \bar{X}_p} \lambda^s_{j \to q}(\bar{x}_p)$$

where we drop the $\bar{x}_{p,d}$ argument in the second term since those messages do not contain any variables in $\mathcal{T}_{p,d}$.

$$h^d(\bar{x}_p) = \min_{\bar{x}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} \sum_{f \in B_k} f(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{s=1..r_j} \left[\sum_{X_q \in \mathcal{T}_{p,d}} \lambda^s_{j \to q}(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_q \in \bar{X}_p} \lambda^s_{j \to q}(\bar{x}_p) \right] \right\}$$

Factoring out the terms that do not depend on the minimization over $\bar{x}_{p,d}$ and applying **Equation 1** on the factored out terms,

$$\begin{aligned} h^d(\bar{x}_p) &= \min_{\bar{x}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} \sum_{f \in B_k} f(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{s=1..r_j} \sum_{X_q \in \mathcal{T}_{p,d}} \lambda^s_{j \to q}(\bar{x}_p, \bar{x}_{p,d}) \right\} \\ &\quad + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{s=1..r_j} \sum_{X_q \in \bar{X}_p} \lambda^s_{j \to q}(\bar{x}_p) \\ &= \min_{\bar{x}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} \sum_{f \in B_k} f(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{s=1..r_j} \sum_{X_q \in \mathcal{T}_{p,d}} \lambda^s_{j \to q}(\bar{x}_p, \bar{x}_{p,d}) \right\} \\ &\quad + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_{(j,p)}(\bar{x}_p) \end{aligned}$$

Finally, redistributing and renaming indexes (from q to k) the summation of the λ terms, we obtain

$$\begin{aligned} h^d(\bar{x}_p) &= \min_{\bar{x}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} \left[\sum_{f \in B_k} f(\bar{x}_p, \bar{x}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{s=1..r_j} \lambda^s_{j \to k}(\bar{x}_p, \bar{x}_{p,d}) \right] \right\} \\ &\quad + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_{(j,p)}(\bar{x}_p) \end{aligned}$$

By **Definition 13**, we replace the first term with $L^d(\bar{x}_p)$, therefore showing that

$$h^d(\bar{x}_p) = L^d(\bar{x}_p) + \sum_{X_k \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_{(k,p)}(\bar{x}_p)$$

□

Appendix B. Proof of Proposition 3

Given a node n, let N_k denote all nodes that are k-levels away from n in the search graph. Then we have

$$res^d(n) \geq \sum_{k=0}^{d-1} \min_{n_k \in N_k} res^1(n_k)$$

We start by assuming that we have the optimal depth-d look-ahead path $\{n_k^{opt(d)} \in N_k | 0 \leq k \leq d\}$. We derive the following to relate the depth-1 look-ahead heuristic for each level k to the path costs and base heuristic under this assumption. Given the definition of the look-ahead heuristic (**Definition 10**) for $d = 1$ and some node $n_k^{opt(d)}$ on the optimal path, we have

$$h^1(n_k^{opt(d)}) = \min_{n_{k+1} \in ch(n_k^{opt(d)})} \left\{ c(n_k^{opt(d)}, n_{k+1}) + h(n_{k+1}) \right\}$$

With the optimal depth-d look-ahead path, setting $n_{k+1} = n_{k+1}^{opt(d)}$, this yields an upper-bound on the minimization.

$$h^1(n_k^{opt(d)}) \leq c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + h(n_{k+1}^{opt(d)})$$

Subsequently, from the definition of the depth-1 residual (**Definition 11**), we can derive the following:

$$\begin{aligned} res^1(n_k^{opt(d)}) &= h^1(n_k^{opt(d)}) - h(n_k^{opt(d)}) \\ &\leq c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + h(n_{k+1}^{opt(d)}) - h(n_k^{opt(d)}) \end{aligned} \tag{14}$$

We will refer to Equation 14, which is an upper-bound on the depth-1 residual as $res^1_{\leq}(n_k^{opt(d)})$ in the following lemma which establishes that the summation of these upper-bounds is equivalent to the depth-d residual $res^d(n)$.

Lemma 3. *If $res^d(n)$ is the depth-d residual from node n and $\{n_k^{opt(d)} \in n_k|_{0\leq k\leq d}\}$ is the set of nodes on the optimal depth-d look-ahead path (where $n_0^{opt(d)}$ is trivially n), then the following holds:*

$$res^d(n) = \sum_{k=0}^{d-1} res^1_{\leq}(n_k^{opt(d)})$$

Proof. Starting with the definition of the depth-d residual, we have

$$res^d(n) = h^d(n) - h(n)$$

Rewriting the look-ahead heuristic h^d, we obtain

$$res^d(n) = \min_{n_1 \in n} \Big\{ c(n, n_1) + h^{d-1}(n_1) \Big\} - h(n)$$

Without loss of generality, we substitute n with n_0 in the following. By unrolling the recursive h^{d-1} look-ahead term completely, we obtain a min-sum problem over a path.

$$res^d(n_0) = \min_{n_1,\ldots,n_d} \left\{ \sum_{k=0}^{d-1} (c(n_k, n_{i+k})) + h(n_d) \right\} - h(n_0)$$

Since we are given the optimal path, we remove the minimization and substitute each n_k with $n_k^{opt(d)}$, obtaining

$$\begin{aligned} res^d(n_0^{opt(d)}) &= \sum_{k=0}^{d-1} c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + h(n_d^{opt(d)}) - h(n_0^{opt(d)}) \\ &= \sum_{k=0}^{d-1} c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + h(n_d^{opt(d)}) + \sum_{k=1}^{d-1} h(n_k^{opt(d)}) - \sum_{k=1}^{d-1} h(n_k^{opt(d)}) - h(n_0^{opt(d)}) \\ &= \sum_{k=0}^{d-1} c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + \sum_{k=1}^{d} h(n_k^{opt(d)}) - \sum_{k=0}^{d-1} h(n_k^{opt(d)}) \\ &= \sum_{k=0}^{d-1} c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + \sum_{k=0}^{d-d} h(n_{k+1}^{opt(d)}) - \sum_{k=0}^{d-1} h(n_k^{opt(d)}) \\ &= \sum_{k=0}^{d-1} c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + h(n_{k+1}^{opt(d)}) - h(n_k^{opt(d)}) \end{aligned}$$

We can see that we obtain a summation over k for Equation 14, so we prove our claim (substituting $n_0^{opt(d)}$ with n):

$$res^d(n) = \sum_{k=0}^{d-1} res^1_{\leq}(n_k^{opt(d)})$$

□

Proof of Proposition 3. Since $res^1_{\leq}(n_k^{opt(d)})$ is an upper-bound for $res^1(n_k^{opt(d)})$ for every k, it follows that their summation is an lower-bound on the depth-d residual.

$$res^d(n) = \sum_{k=0}^{d-1} res^1_{\leq}(n_k^{opt(d)}) \geq \sum_{k=0}^{d-1} res^1(n_k^{opt(d)}) \tag{15}$$

Taking the minimization of a depth-1 residual with respect to all nodes for a given level k, we obtain

$$res^1(n_k^{opt(d)}) \geq \min_{n_k \in N_k} res^1(n_k) \tag{16}$$

Applying Equations 15 and 16 together, we obtain our proposed statement.

$$res^d(n) \geq \sum_{k=0}^{d-1} \min_{n_k \in N_k} res^1(n_k)$$

□

References

Bensana, E., Lemaitre, M., & Verfaillie, G. (1999). Earth observation satellite management. *Constraints*, *4*(3), 293–299.

Bertele, U., & Brioschi, F. (1972). *Nonserial Dynamic Programming*. Academic Press.

Bu, Z., Stern, R., Felner, A., & Holte, R. C. (2014). A* with lookahead re-evaluated. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*.

Darwiche, A. (2009). *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press.

Dechter, R., & Rish, I. (2002). Mini-buckets: A general scheme for approximating inference. *Journal of the ACM*, *50*(2), 107–153.

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

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.

Dechter, R. (2013). *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Dechter, R., & Mateescu, R. (2007). And/or search spaces for graphical models. *Artif. Intell.*, *171*(2-3), 73–106.

Fishelson, M., & Geiger, D. (2004). Optimizing exact genetic linkage computations. *Journal of Computational Biology*, *11*(2-3), 263–275.

Geffner, H., & Bonet, B. (2013). *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Ihler, A. T., Flerova, N., Dechter, R., & Otten, L. (2012). Join-graph based cost-shifting schemes. In *Proceedings of the 28th Conference on Uncertainty of Artificial Intelligence (UAI 2012)*.

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

Lam, W. (2017a). Advancing heuristics for search over graphical models. Tech. rep., Ph.D. Thesis, University of California, Irvine, California.

Lam, W. (2017b). https://github.com/willmlam/daoopt-exp..

Lam, W. (2017c). https://willmlam.github.io/supp/lookahead/index.html..

Marinescu, R., & Dechter, R. (2009a). And/or branch-and-bound search for combinatorial optimization in graphical models. *Artif. Intell.*, *173*(16-17), 1457–1491.

Marinescu, R., & Dechter, R. (2009b). Memory intensive and/or search for combinatorial optimization in graphical models. *Artif. Intell.*, *173*(16-17), 1492–1524.

Mateescu, R., & Dechter, R. (2005). The relationship between and/or search spaces and variable elimination. In *Proceeding of Uncertainty in Artificial Intelligence (UAI2005)*.

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

Otten, L., Ihler, A., Kask, K., & Dechter, R. (2012). Winning the pascal 2011 map challenge with enhanced and/or branch-and-bound. In *Workshop on DISCML 2012 (a workshop of NIPS 2012)*.

Otten, L. (2013). https://github.com/lotten/daoopt..

Otten, L., & Dechter, R. (2012). Anytime and/or depth-first search for combinatorial optimization. *AI Communications*, *25*(3), 211–227.

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

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

Rollon, E., Larrosa, J., & Dechter, R. (2013). Semiring-based mini-bucket partitioning schemes.. In *IJCAI*, pp. 3–9. Citeseer.

Russell, S. J., & Norvig, P. (2009). Artificial intelligence: a modern approach (3rd edition)..

Stern, R., Kulberis, T., Felner, A., & Holte, R. (2010). Using lookaheads with optimal best-first search..

Vidal, V. (2004). A lookahead strategy for heuristic search planning. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pp. 150–160.

Wemmenhove, B., Mooij, J. M., Wiegerinck, W., Leisink, M., Kappen, H. J., & Neijt, J. P. (2007). Inference in the promedas medical expert system. In *Artificial Intelligence in Medicine*, Vol. 4594 of *Lecture Notes in Computer Science*, pp. 456–460. Springer Berlin Heidelberg.

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