UNIVERSITY OF CALIFORNIA,
IRVINE

Advancing Heuristics for Search over Graphical Models

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

William Ming Lam

Dissertation Committee:
Professor Rina Dechter, Chair
Professor Alexander Ihler
Professor Sameer Singh

2017

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

ACKNOWLEDGMENTS

This thesis has been one of the biggest challenges I have ever faced, and it would not have come together without the support of various people.

First, I would like to thank my advisor, Professor Rina Dechter, for the many years she has guided me through graduate school. Her meticulous nature in all aspects of research has been a very good model for me to follow. In particular, her extreme attention to detail has always served to greatly improve the quality of my writing. She has also been extremely patient with me, which allowed me to improve myself at my own pace. It was definitely an honor to be able to work with her all these years.

I would also like to thank my committee members, Professors Alexander Ihler and Sameer Singh for their time. In particular, I appreciate the research discussions I had with Alex during one of my projects, where he provided plenty of useful suggestions and advice.

The members of my research group, past and present were also instrumental in making my experience in graduate school enjoyable. For the many interesting conversations on research and otherwise, I thank Dr. Lars Otten, Dr. Andrew Gelfand, Dr. Natasha Flerova, Junkyu Lee, Silu Yang, Filjor Broka, and Zhengli Zhao. I especially thank Dr. Kalev Kask, with whom I collaborated with on much of the work presented in this thesis. I also enjoyed my collaboration with Professor Javier Larrosa while he was on a sabbatical at UC Irvine and appreciate his continued help afterward with parts of this thesis.

I also thank Professor Christian Shelton at UC Riverside for introducing me to the world of graphical models and machine learning when I was an undergraduate, giving me my first taste of research. This experience ultimately led me to make a decision to apply to graudate school, of which he was an extremely valuable resource in the process.

My parents have also been important in supporting in this journey. When I was living on campus during my first four years, they would take the hour commute to come visit me for lunch and help with other things every single weekend. For the rest of my time in graduate school, I moved back home and they have continued to make my life easier, which enabled me to focus on my work.

I would likely have never taken the path of graduate school if it were not for my brother, Professor Antony Lam. His influence was a key factor in getting me interested in AI, machine learning, and probably computer science in general. Additionally, he has always made himself available to talk to on just about anything.

Finally, I would like to acknowledge the assistance I have received during my graudate studies from NSF grants IIS-1065618 and IIS-1254071, from the United States Air Force under Contract No. FA8750-14-C-0011 under the DARPA PPAML program and the Donald Bren School of Information and Computer Sciences at University of California, Irvine.

CURRICULUM VITAE

William Ming Lam

EDUCATION

Doctor of Philosophy in Computer Science — **2017**
University of California, Irvine — *Irvine, CA*

Master of Science in Computer Science — **2012**
University of California, Irvine — *Irvine, CA*

Bachelor of Science in Computer Science — **2010**
University of California, Riverside — *Riverside, CA*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2010–2017**
University of California, Irvine — *Irvine, CA*

TEACHING EXPERIENCE

Teaching Assistant — **2010–2012**
University of California, Irvine — *Irvine, CA*

PROFESSIONAL EXPERIENCE

Software Engineering Intern — **2015**
Google — *New York, NY*

Software Engineering Intern — **2014**
Google — *Los Angeles, CA*

PUBLICATIONS

On the Impact of Subproblem Orderings on Anytime AND/OR Best-First Search for Lower Bounds **2016**
European Conference on Artificial Intelligence

Look-Ahead with Mini-Bucket Heuristics for MPE **2016**
AAAI Conference on Artificial Intelligence

Empowering Mini-Bucket in Anytime Heuristic Search with Look-Ahead: Preliminary Evaluation **2015**
Symposium on Combinatorial Search

Beyond Static Mini-Bucket: Towards Integrating with Iterative Cost-Shifting Based Dynamic Heuristics **2014**
Symposium on Combinatorial Search

Benchmark on DAOOPT and GUROBI with the PASCAL2 Inference Challenge Problems **2013**
NIPS Workshop on Discrete Optimization in Machine Learning

Empirical Evaluation of AND/OR Multivalued Decision Diagrams for Inference **2012**
CP Doctoral Programme

Factored Filtering of Continuous-Time Systems **2011**
Conference on Uncertainty in Artificial Intelligence

Continuous Time Bayesian Network Reasoning and Learning Engine **2010**
Journal of Machine Learning Research

SOFTWARE

DAOOPT-EXP `http://github.com/willmlam/daoopt-exp`
A branch of DAOOPT containing implementations of dynamic heuristics. This inlcudes look-ahead, subproblem ordering for AOBF, and cost-shifting based heuristics.

AOMDD `http://github.com/willmlam/aomdd`
An implementation of AOMDDs as a black-box representation of discrete functions. Implements a version of the bucket elimination algorithm using AOMDDs.

ABSTRACT OF THE DISSERTATION

Advancing Heuristics for Search over Graphical Models

By

William Ming Lam

Doctor of Philosophy in Computer Science

University of California, Irvine, 2017

Professor Rina Dechter, Chair

Graphical models are widely used to model complex interactions between variables. A graphical model contains many functions over these variables which have an underlying graph structure and represents an overall joint function composed of these functions. Examples include Bayesian networks, Markov networks, and weighted constraint satisfaction problems. In these, a common query is the optimization query, which is to find a joint configuration of variables that optimizes the objective function defined by the joint function.

One of the best frameworks for these queries is the AND/OR search framework, which casts this optimization problem as a heuristic search problem. AND/OR Branch-and-Bound (AOBB) and AND/OR Best-First (AOBF) search are both algorithms that search the AND/OR search space, which exploits conditional independencies in a problem. The performance of these algorithms are highly dependent on the strength of its heuristics, which is the main focus of this thesis.

We first investigate the well-known technique of look-ahead, commonly used in search applications such as games, adapting it to the optimization task in graphical models. In particular, we analyze the typically used MBE heuristic within the AND/OR search framework, establishing a connection between the "bucket error" of MBE and the residual. This connection enables a cost-effective scheme that allows look-ahead to be selectively performed only

where it is likely to have a positive impact on the heuristic. An extensive empirical evaluation demonstrates that we are able to improve the power of AND/OR Branch-and-Bound.

Second, in seeking an anytime scheme for providing lower bounds on the optimal solution, we explore the impact of subproblem ordering on AOBF. Our analysis demonstrates that this can significantly impact the performance of AOBF for both exact and anytime solutions. We propose heuristics based on the bucket error of MBE and demonstrate their potential.

Third, to tackle problems where MBE is known to perform poorly on, we explore dynamic heuristics that are computed during search rather than static heuristics which are pre-computed such as with MBE. We adapt the FGLP algorithm, a coordinate-descent method for a linear programming relaxation-based bound on the objective, for use a a heuristic generator for AOBB. Through a re-derived update method and a defined update schedule, we demonstrate the advantage of FGLP heuristics on problems known to be difficult when using MBE.

Additionally, we extend AND/OR Multi-valued Decision Diagrams, a framework for the compact representation of functions in graphical models. We provide a previously missing empirical evaluation of previously introduced methods. We also extend the framework to allow for exact bucket elimination to be implemented using AOMDDs, thus pushing the feasibility of bucket elimination on problems that normally require an infeasible amount of memory.

Chapter 1

Introduction

Graphical models are a widely used framework for representing structure in complex systems such as probability distributions over a large number of variables. They allow us to model interactions between variables using graphs, where nodes represent variables and cliques represent local functions over the connected variables.

Examples of graphical models include Bayesian networks, Markov networks, factor graphs, and weighted constraint satisfaction problems [11, 42, 13]. One of the most common tasks over graphical models is the optimization query, which is to find a joint configuration of the variables that optimizes an objective function represented by the graphical model. When the graphical model represents a probability distribution, a maximization corresponds to finding the most likely configuration of the graphical model, which is known as the Most Probable Explanation (MPE) or Maximum A Posteriori (MAP) query. In other literature such as in constraint satisfaction, the local functions are viewed as costs and thus a minimization corresponds to finding the least cost configuration of the model. The optimization task is known to be NP-hard. Thus, given limited resources in time and/or memory, we often turn to approximation algorithms. One desirable property for approximation algorithms is to be anytime, which provide monotonically improving suboptimal configurations over time and are guaranteed to return the optimal configuration with enough time. Casting inference as a heuristic search problem is a natural step, which systematically enumerates

the configurations of a model. Depending on the design of the heuristic search algorithm, it can be an effective anytime approximation algorithm. Indeed, one of the most effective frameworks for performing these tasks is AND/OR search [16], including algorithms such as AND/OR Branch-and-Bound (AOBB) and AND/OR Best-First (AOBF), along with the Mini-Bucket with Moment Matching (MBE-MM) heuristic. This was demonstrated by winning recent UAI inference competitions [40]. The most significant strength of AND/OR search lies in its ability to take advantage of problem decomposition in graphical models [16].

1.1 Dissertation Outline and Contributions

In this dissertation, we focus on improving the state-of-the-art heuristic search, and the heuristics available to guide the AND/OR search framework to yield faster exact and anytime inference (both upper and lower bounds) for optimization queries, thus extending their reach to a wider variety of problems. In particular, we focus on look-ahead during search and on dynamic heuristic computation. Within these, balancing a time and space trade-off is an inherent problem which we address in our work. We also improve the AND/OR Best-First search algorithm's performance by addressing the generally overlooked aspect of subproblem ordering.

Lastly, we extend a framework, AND/OR Decision Diagrams, for the compact representation of graphical models to allow exact inference on summation queries on problems that cannot be solved due to memory constraints.

We provide an overview of each chapter in the following subsections. Subsequently, we provide background relevant to all of the work. Additionally, more specific background is deferred to each of the chapters.

1.1.1 Cost-Directed Look-ahead in AND/OR Search via Residual Analysis

Chapter 2 focuses on applying the classic technique of look-ahead to improve the mini-bucket elimination (MBE) heuristic used in AND/OR Branch-and-Bound (AOBB). The look-ahead technique is known to be useful in the context of *online search algorithms* (e.g. game playing schemes, planning under uncertainty, etc.) [4, 22, 47]. It improves the heuristic function $h(\cdot)$ of a node by expanding the search tree below it and backing up the $h(\cdot)$ values of descendants (known as a *Bellman update*). Thus, look-ahead can be seen as a secondary search embedded in the primary search. In these, the heuristic is commonly treated as a black-box, and thus look-ahead is used in an arbitrary way. As such, it may not always be cost-effective because there are no guarantees that the heuristic will improve enough to make sufficient impact that counter balances the computational overhead. However, in the context of MBE heuristics which are pre-compiled prior to search, there is information that can be exploited to inform of the impact before attempting a look-ahead. Thus, the goal of the research here is whether look-ahead can be made into a cost-effective scheme given this information. This motivates the work in this chapter:

Contributions

- We define the concept of *bucket error* of the mini-bucket heuristic and show their connection with the residual.
- We present bucket error as a guide for controlling look-ahead to avoid frivolous look-ahead.
- We show how look-ahead can be accomplished as a conditioned variable elimination task, which enables it to be carried out more efficiently.
- In an extensive empirical evaluation, we demonstrate that look-ahead can be a cost-effective scheme for both finding an exact solution and approximate anytime solutions.

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

Chapter 3 focuses on the *subproblem ordering* in AND/OR Best-First Search (AOBF). Unlike traditional best-first search in an OR search space, the current best partial solution contains many "best" nodes corresponding to different subproblems, while only one node is chosen for expansion at a time. This problem has already been posed by Pearl in [41], where AND/OR Best-First search is presented as having two heuristics: one for estimating the quality of the current partial solution graph (as typical in heuristic search), and one for guiding which node expansion to this partial solution graph will lead to the largest refinement in the estimate. Our work aims for an algorithm that provides *anytime lower bounds* on the optimal solution. Therefore, a secondary heuristic should aim to increase the lower bound the most. Indeed, the generation of anytime lower bounds using best-first search has been a recent topic of interest in the context of graphical models [1, 34, 31]. However, the specific question on the impact subproblem ordering in best-first search in AND/OR search spaces has not been explored, and is the main focus on this chapter. The following outlines the contributions.

Contributions

- We illustrate that subproblem ordering can have a significant impact on the performance of AOBF. In particular, exploring one subproblem before another can increase the lower bound on the solution far more significantly.
- We show that look-ahead can be helpful for identifying the best subproblem to explore amongst the current partial solution tree frontier nodes. Thus, we show that the *bucket errors* of the MBE heuristic presented in Chapter 2 can be instrumental for guiding subproblem ordering.
- Due to the overhead associated with full look-ahead, we propose several schemes that approximate the bucket errors which are used to guide look-ahead.

- We conducted an extensive empirical evaluation through which we demonstrate that subproblem ordering can impact significantly the runtime of AOBF in a large number of problems, both when it runs to completion, providing an exact solution, and when it provides lower bounds in an anytime manner.

1.1.3 Dynamic FGLP Heuristics

Chapter 4 explores the use of dynamic heuristics in branch-and-bound using the Factor Graph Linear Programming (FGLP) algorithm [25]. The quality of the MBE heuristic depends greatly on the available memory for the compilation of its functions. On problems with high domain size and/or densely structured graphs, the amount of memory needed for accurate MBE increases greatly, leading to poor bounds. A common alternative is based on Linear Programming relaxation [23] and FGLP is a particular coordinate descent algorithm for this class of algorithms [25]. The algorithm works by *re-parameterizing* the problem to tighten the relaxation. A similar approach to in the WCSP literature is known as *soft arc-consistency*, which generates bounds by a set of transformations that re-parameterize the problem [30, 10]. FGLP, however, was not designed to be used repetitively during search as a heuristic generator. The main question of this chapter is whether using FGLP as a heuristic can allow us to boost the performance of branch-and-bound on problems where MBE fails to provide strong heuristics. Our contributions are outlined below.

Contributions

- We present a new version of FGLP that tightens the bound with fewer coordinate descent steps in the context of search, which is important in making it a cost-effective heuristic.
- In order to generate heuristics that can be strong for a wider variety of problem classes,

we explore the potential of a hybrid algorithm that combines the strengths of FGLP and MBE heuristics.

- We demonstrate empirically that FGLP heuristics are more effective than MBE heuristics on specific problem classes known to be challenging when using MBE heuristics.

1.1.4 AND/OR Multivalued Decision Diagrams for Inference

Chapter 5 builds on the framework of AND/OR Multivalued Decision Diagrams (AOMDD) [37]. Although graphical models are established as a compact representation of complex systems, they only capture the structural relationships between variables and are oblivious to the structure of the values defining these relationships. In many cases, there is a more compact representation using AOMDDs [37], which are canonical representations of functions. This can allow models that may have extremely high induced width to be tractable, which would otherwise be intractable due to memory constraints. The empirical evaluations in earlier work on this AOMDD framework [37] was based on constructing AOMDDs representing the global function of a graphical model via tracing a regular AND/OR search and pruning nodes from the trace to satisfy the properties of AOMDDs. If there are sufficient resources to compute and store the AOMDD, various queries can be performed on the model in time linear in the size of the AOMDD. An algorithm for this same task based on constructing AOMDDs bottom-up was defined [37], but was not evaluated empirically. Furthermore, in other decision diagram literature, it is typical to treat decision diagrams as complete replacements to tabular representations of functions in the implementation of any algorithm that performs operations on functions [5, 2]. AOMDDs have not be used this way because its APPLY algorithm for performing operations was not previously implemented. Taking this step can have the potential of improving a variety of inference algorithms that operate on large functions within graphical models, such as Bucket Elimination, which can still be important if resources restrict the feasibility of compiling the entire model to an

AOMDD. Thus, the main questions in this work are 1) whether the performance of the bottom-up compilation algorithm is better than the existing compilation algorithms, and 2) the potential for pushing the feasibility of exact inference by using AOMDDs.

Contributions

- We provide a first implementation of the previously introduced bottom-up BE-AOMDD [37] algorithm for compiling the global function of a graphical model and empirically evaluated it by comparing it to previous work based on top-down search algorithms.
- We define an elimination operator for AOMDDs, thus completing (relative to work in [37]) the AOMDD framework as an alternative way to represent discrete real-valued functions such as those within in a graphical model.
- We empirically evaluate AOMDDs as an alternative representation to tables for functions by applying it within the exact BE algorithm, demonstrating its potential in pushing the boundaries of exact inference to problems of high induced width.

1.2 Background

We start by providing a formal definition of a graphical model. Next, we provide an overview of heuristic search and review the search algorithms and heuristics used for inference in graphical models.

1.2.1 Graphical Models

We will use X_i to denote a variable and D_i its domain. A generic domain value will be noted x_i. A variable assignment will be denoted (X_i, x_i) or, when the context is clear, just x_i. We

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

Table 1.1: Notation on graphical models.

will use f_j to denote a function and S_{f_j} its scope. The scope S_{f_j} is the set of variables for which the function is defined (i.e., $f_j : \prod_{X_i \in S_{f_j}} D_i \to \mathbb{R}$). A graphical model is a collection of functions over subsets of variables,

Definition 1.1 (**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 scope* $S_{f_j} \subseteq \mathbf{X}$ *for all* f_j.
4. $\otimes$ *is a combination operator (typically the sum or product)*

Each graphical model has an associated graph which makes explicit some of the conditional independencies that exists in the model

Definition 1.2 (**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* $(X_i, X_{i'}) \in E$ *for each pair of variables that appear in the same scope* S_{f_j} *of some local function* $f_j \in \mathbf{X}$.

Figure 1.1 is a primal graph of a graphical model with variables indexed from A to G with binary functions over pairs of variables connected by an edge. In this particular example we have $\mathbf{F} = \{f(A), f(A,B), f(A,D), f(A,G), f(B,C), f(B,D), f(B,E), f(B,F),$

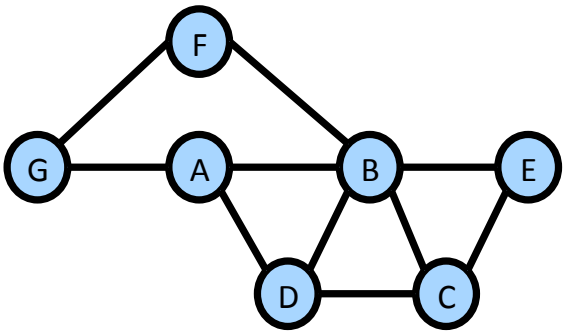


Figure 1.1: A primal graph of a graphical model with 7 variables.

$f(C,D), f(C,E), f(F,G)\}$, for a total of 11 functions. Note that it is possible for unary functions to exist which are not apparent from looking at the primal graph (as seen here with $f(A)$).

A graphical model represents a *global* function which is the combination of all the local functions, denoted $\otimes_{j=1}^{m} f_j(\cdot)$. Graphical models are used to model complex systems and their main virtue is allowing compact representation and their structure can often allow efficient query processing. Given a set of variables $\mathbf{Y}$, queries are defined by a *marginalization* operator $\Downarrow_{\mathbf{Y}}$, which eliminates variables in $\mathbf{Y}$ from a function $f(\cdot)$.

In this thesis (excluding Chapter 5), the combination operator is *summation* and the marginalization operator is min.

Definition 1.3 (min-sum problem). *Given a graphical model* $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F}, \sum)$*, the min-sum problem is the optimal assignment of its variables with respect to the global function,*

$$\min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot)$$

The framework is general and can include various queries of interest. For instance, when 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 [11]. If a negative log transformation is applied to the local functions, the min-sum problem corresponds to the MPE/MAP inference query [14]. 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. This is the graphical model used in constraint optimization problems (or weighted constraint satisfaction problems) [13]. This query is known to be *NP-hard*.

In Chapter 5, we also focus on the *summation* query.

Definition 1.4 (**weighted counting problem).** *Given a graphical model* $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F}, \sum)$*, the weighted counting problem is the sum of the weight of all full assignments to the global function.*

$$\sum_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot)$$

Namely, the combination operator is the same as before, but the marginalization operator is now *summation*. This query also shows up in various contexts. In a *Markov network*, the local functions are unnormalized probability distributions, thus also making the global probability function unnormalized. Thus, computing the normalizing constant, which can be achieved by solving the weighted counting problem, is necessary to compute the probability of a particular assignment to the system. This is commonly as computing the *partition function*. Alternatively, to compute the probability of a partial assignment in a Bayesian network, we condition on the partial assignment and sum out the rest of the variables. This corresponds to the weighted counting task over the conditioned model.

The complexity of both the optimization and the summation queries for a given graphical model can be bounded by the *induced width* w^* of its associated primal graph.

Definition 1.5 (**induced width** [14])**.** *Given a primal graph* $G = (V, E)$*, an* ordered graph *is a pair* (G, d)*, where* $d = (X_1, ..., X_n)$ *is an ordering of the nodes. The nodes adjacent to*

X_k that precede it in the ordering are called its parents. *The* width of a node *in an ordered graph is its number of parents. The* width of an ordered graph (G, d), *denoted* $w(d)$, *is the maximum width over all nodes. The* width of a graph *is the minimum width over all orderings of the graph. The* induced graph *of an ordered graph* (G, d) *is an ordered graph* (G^*, d), *where* G^* *is obtained from* G *as follows: the nodes of* G *are processed from last to first along* d. *When a node* X_k *is processed, all of its parents are connected. The* induced width of an ordered graph (G, d), *denoted* $w^*(d)$, *is the maximum number of parents a node has in the induced ordered graph* (G^*, d). *The* induced width of a graph w^*, *is the minimum induced width over all its orderings.*

1.2.2 Heuristic Search

n_0, n	initial state, state
$c(n, n')$	arc cost between n and n'
$g(n)$	cost of path from initial state n_0 to n
$h(n)$	heuristic function of n
$h(n)$	evaluation function of n
$succ(n)$	successors of n in the search space

Table 1.2: Notation on heuristic search.

Search is a general framework that is used in a variety of applications. As mentioned earlier, we apply it to our work on optimization queries over graphical models. For completeness, we provide some background here in a general setting. **Table 1.2** lists notation used in this section.

A general search problem is defined by an (implicit) search tree (or graph) where its nodes correspond to *states*, *actions* which map states to states, and *costs* associated with the edges of the graph. Within the set of states, one state is defined as the *initial state* n_0 and a subset of them are defined as *goal states*. We also have a *successor function* whose input is a single state n, returns the set of possible actions that can be taken from that state, which can also be viewed as the set of children of n. A node *expansion* is defined as applying the successor

function to a node n and *generating* its children. We say that a part of a search space has been *explored* if its states have been expanded. Search is a process that repeatedly selects a generated node, but unexpanded node, and expands it to explore the search graph. The task is typically to find a minimum cost path from n_0 to one of the goal states, where the path cost is defined by a combination of the arc costs, typically the summation [44].

There are different search strategies that determine the order that nodes are selected for expansion. In the most general sense, they can be categorized as either *blind (uninformed)* search or *heuristic (informed)* search. The former only considers information in the generated search graph, such as the current best path cost from the initial state n_0 to some generated node n, denoted $g(n)$. The latter guides the expansions based on (heuristic) information about the unexplored parts of the search space.

The information is defined based on a *heuristic function*, denoted $h(n)$, which maps states to an estimate of the cost of the minimum cost path from the state to any of the goal states. Given the actual minimum cost, denoted $h^*(n)$, $h(n)$ is *admissible* if $\forall n\ h(n) \leq h^*(n)$, that is, it is an optimistic estimate of the actual cost. Next, given the cost of an edge (n, n') in the search graph, denoted $c(n, n')$, $h(n)$ is *consistent* if $\forall n \forall n'\ h(n) \leq c(n, n') + h(n')$.

Within heuristic search, we can categorize different strategies under *depth-first branch-and-bound search* or *best-first search*. In either case, they use an *evaluation function* $f(n)$, which is typically chosen to be $g(n) + h(n)$. This uses both the current best path cost from the initial state n_0 to n and the heuristic in order to estimate the minimum cost path from n_0 to a goal that includes n.

1.2.2.1 Depth-First Branch-And-Bound Search

Depth-First Branch-and-Bound (DFBB), is a popular search strategy based on expanding nodes in a *depth-first* ordering. The main strength of a depth-first expansion ordering is

that, provided the search depth is bounded, it can be carried in linear memory since paths that either do not reach a goal state or are suboptimal can be deleted from memory. If the search space is a graph, then we can also elect to cache nodes to avoid its re-expansion in order to improve performance, thus providing a way to trade space for time.

In general, it maintains an upper bound UB on the current minimum cost path to a goal node from the initial state. At the start, UB is set to infinity. For a given node n, with path cost $g(n)$ and an admissible heuristic $h(n)$, if the value of its evaluation function $f(n) = g(n) + h(n) > UB$, then it is possible to *prune* the node, forgoing its expansion and thus never exploring its successors. In the worst case when no pruning is possible, DFBB explores the entire search space. In addition, for infinite search spaces with unbounded depth, the algorithm may never terminate.

Algorithm 1: Depth-First Branch-and-Bound (DFBB)

Input: Node n, current upper bound UB
Output: Cost of minimum cost path from n to a goal state, or UB if the goal is unreachable

```
1 if n is a goal state then return 0;
2 foreach n' ∈ succ(n) do
3     if c(n, n') + h(n) ≤ UB then
4         UB := min(UB, c(n, n') + DFBB(n', UB − c(n, n')))
5 return UB
```

We present pseudocode for DFBB in **Algorithm 1**, defined recursively. We leave out the details of path generation here for simplicity. It assumes that a successor function $succ(n)$ is defined for all nodes in the search space that returns the set of successors. Given an input node n and current upper bound UB, it proceeds as follows. Line 1 captures the trivial case when n is already a goal state and thus has a cost of 0. Otherwise, we expand n by generating its successors in line 2. Inside the loop, line 3 is the pruning check and line 4 recursively calls DFBB on n' to find the minimum cost from n' to a goal state. On the same line, UB is updated if a better cost path has been found, which is returned in line 5 after

either expanding or pruning each successor.

1.2.2.2 Best-First Search

Best-First Search is an alternative strategy which expands nodes based on the evaluation function $f(n)$ directly. In contrast to depth-first search, it maintains all of the explored paths, thus generally requiring exponential memory. It maintains an $OPEN$ list which contains all generated, but unexpanded nodes and a $CLOSED$ list containing all expanded nodes. At each step, it expands a node $n \in OPEN$ having the minimum $f(n)$ value. The variants of best-first search are defined by how their $f(n)$ is defined. In the case of $f(n) = g(n) + h(n)$, this yields the A* algorithm, which is the most popular variant. Given that $h(n)$ is consistent, the $f(n)$ values along a path are monotonically non-decreasing. If this is the case A* is known to be optimally efficient for that given $h(n)$ [44]. When a goal node is reached, its evaluation function is guaranteed to be equal to the cost of the minimum cost path from n_0. Thus, A* is the best algorithm in terms of the number of nodes expanded, but does so at the cost of memory.

Algorithm 2: Best-First Search (A*)

Input: Initial state n_0
Output: Cost of minimum cost path from n_0 to a goal state, or ∞ if the goal is unreachable

```
1 OPEN := {n_0}
2 while OPEN ≠ ∅ do
3 |   n := arg min_{n∈OPEN} f(n)
4 |   if n is a goal state then return f(n);
5 |   else OPEN := OPEN − {n} + succ(n);
6 return ∞
```

Algorithm 2 presents pseudocode for the A* algorithm under the assumptions about the evaluation and heuristic stated above, along with a $succ(n)$ function as defined for DFBB. Since we do not specify path generation here, the $CLOSED$ list is not required.

1.2.2.3 Look-ahead

In cases where the heuristic is poor, a technique that can be used to improve any heuristic is known as *look-ahead*, commonly used on applications of search such as game-playing [44]. It works by replacing the $h(n)$ value of node n with information based on its successors through fully expanding a part of the search space below n to a specified depth d. We can formally define the d-level look-ahead recursively as follows.

Definition 1.6 (look-ahead). *The depth d look-ahead of node n is*

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

1.2.3 Heuristic Search for Graphical Models

We now turn back to graphical models and show how heuristic search is applied to performing queries. As a systematic method of enumeration, we can use it to enumerate the possible assignments to a graphical model with the appropriate search space and use heuristic search algorithm to find the optimal cost path.

The simplest variant is known as an OR search space, where each level corresponds to a variable in the graphical model. The nodes correspond to individual variable assignments while the arc costs are determined from the function values. This search space is clearly bounded, where all the paths correspond to full assignments to the graphical model, thus making each leaf a goal state. The size of the search tree is $O(k^n)$, where n is the number of variables and k is the maximum domain size.

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

Table 1.3: Notation on AND/OR search for graphical models.

1.2.3.1 AND/OR Search for Graphical Models

In the context of graphical models, the conditional independencies in the model can be exploited via the AND structures in AND/OR search spaces.

In AND/OR search trees [38, 41], 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 subproblems that need to be solved. In AND/OR search over graphical models, the 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 associated with each edge between an OR node and its child AND node, which represents the cost of making the corresponding decision at that branching point.

Thus, to adapt AND/OR search spaces for graphical models, they are defined relative to a *pseudo tree* of the primal graph [16].

Definition 1.7 (**pseudo tree** [16]). *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.*

Definition 1.8 (**AND/OR search tree** [16]). *Given a graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes)$ and a pseudo tree $\mathcal{T}$, its AND/OR search tree consists of alternating levels of OR and AND*

nodes. OR nodes are labeled with a variable $X_i \in \mathbf{X}$. Its children are AND nodes, each labeled with an instantiation x_i of X_i. Children of AND nodes are OR nodes, labeled with the children of X_i in $\mathcal{T}$. Each child represents a conditionally independent subproblem given assignments to their ancestors. The root of the AND/OR search tree is an OR node labeled by the variable at the root of $\mathcal{T}$.

Each edge from an OR node X_i to an AND child node x_i represents a variable assignment. The path from the root to an AND node x_i represents a unique assignment to the variables in $\bar{X}_i$, that will be denoted $\bar{x}_i$ (see **Table 1.3**)

The path from the root to an AND node x_i represents a unique assignment to the variables in $\bar{X}_i$, that will be denoted $\bar{x}_i$ (see **Table 1.3**). In the AND/OR search tree, the costs of the OR-to-AND arcs denoted $c(X_i, x_i)$ (abusing notation, since they are dependent on the path to x_i) are defined as follows:

Definition 1.9 (arc cost $c(X_i, x_i)$). *The cost $c(X_i, x_i)$ of the arc (X_i, x_i) is the combination of all the functions in the graphical model whose scope includes X_i and that are fully assigned by the values specified along the path $\bar{x}_i$ from the root to node x_i.*

For completeness, we define the costs of edges from AND nodes to OR nodes to be the identity element of the chosen combination operator $\otimes$ (i.e. 0 for $\sum$ and 1 for $\prod$).

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

Definition 1.10 (OR context). *The* context *of a variable X_i in a pseudo tree $\mathcal{T} = (V, E')$ is the set of ancestor variables connected to X_i or its descendants by arcs in E'.*

Definition 1.11 (context-minimal AND/OR search graph [16]**).** *Identical subproblems can be merged based on the* OR context. *We can merge two nodes if they have the same*

assignment to the context variables. This yields a context-minimal AND/OR search graph $C_{\mathcal{T}}$ *whose size can be shown to be bounded exponentially in the induced width of* G *along the pseudo-tree* $\mathcal{T}$.

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

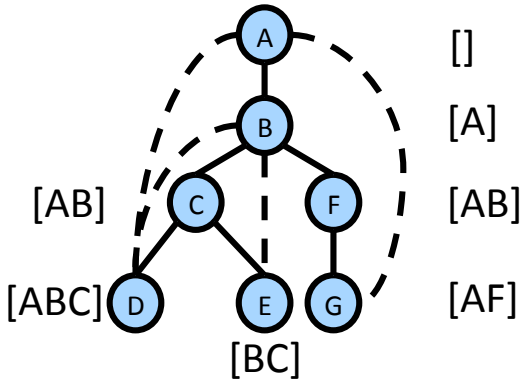
Definition 1.12 (**solution tree**). *A solution tree* T *is a subtree of the AND/OR search graph* $C_{\mathcal{T}}$ *such that:*

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

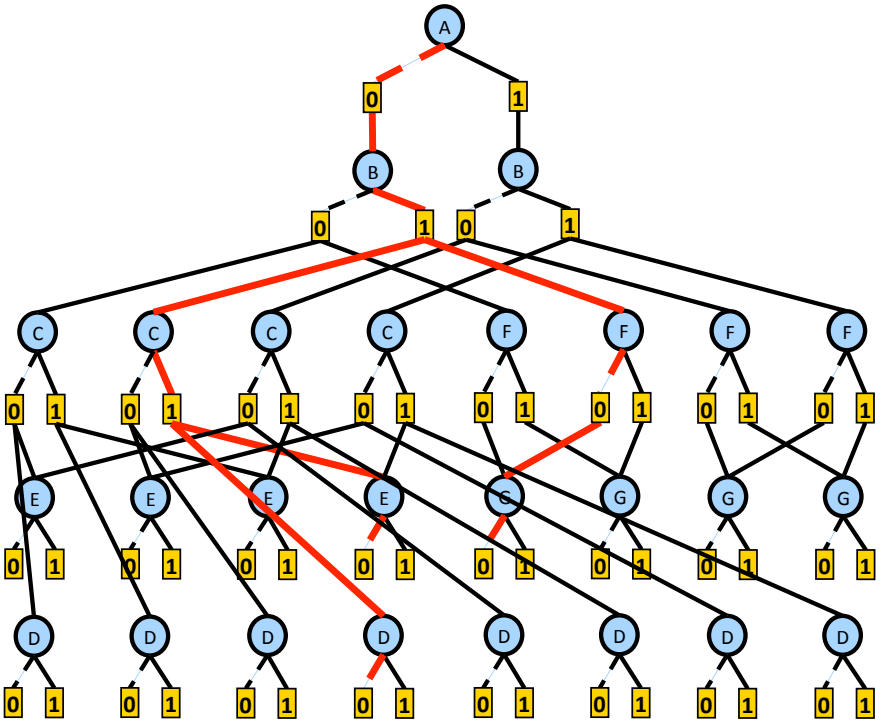
The cost *of a solution tree is the combination of the arc weights associated with the arcs of* $C_{\mathcal{T}}$.

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

Example. **Figure 1.2a** shows a pseudo tree for our running example. We indicate, with solid arcs, the arcs that form the main tree structure and indicate with dotted arcs the back-arcs. Each variable is annotated with its context. **Figure 1.2b** shows the corresponding context-minimal AND/OR search graph guided by this pseudo tree. Since the context of variable E is only over B, C, the corresponding OR nodes have been merged with respect to



(a) A pseudo tree for the running example.



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

A, which is an ancestor not in the context. Similarly, the OR nodes of G are merged with respect to B. The *solution tree* corresponding to the assignment $(A = 0, B = 1, C = 1, D = 0, E = 0, F = 0, G = 0)$ is highlighted.

$u(n)$	current upper bound of node n
$h(n)$	heuristic (lower bound) of node n
$OPEN$	stack of created but unexpanded nodes
$succ(n)$	successors of n in the search space
$T(n), T^*(n)$	current (best) partial solution tree rooted by n

Table 1.4: Notation on AND/OR search algorithms.

1.2.3.2 Depth-First AND/OR Branch-and-Bound

We now present the one of the most used AND/OR search algorithms used for inference in graphical models, depth-first AND/OR Branch-and-Bound (AOBB), which is used in both Chapters 2 and 4. We defer the presentation of AND/OR Best-First search (AOBF) to Chapter 3 where we use it. In **Table 1.4**, we introduce some notation used in the pseudocode and assume the min-sum query.

The depth-first AOBB (AND/OR Branch and Bound) algorithm traverses the AND/OR search space in a *depth-first* manner [32]. It keeps track of the best solution it encounters, yielding *upper bounds* on the optimal solution whenever terminated.

We present the pseudocode in **Algorithm 3**, for simplicity, showing the AND/OR search tree without context-minimal merging. The two main activities are 1) node expansion (lines 5-27) and 2) a bound propagation step (lines 28-38). The *node expansion* step expands nodes and puts them on the $OPEN$ list. It also prunes subtrees if the heuristic evaluation function for a partial solution tree is worse than the current best solution (lines 12-13). To compute the heuristic evaluation function for a given partial solution tree, we recursively evaluate it with **Algorithm 4**. The *bound propagation* step propagates the upper bounds and best partial solution trees bottom-up. It is carried out whenever a node n has no successors and repeats up until one of its ancestors has successors in OPEN. Thus, if the root node is reached in this step, the upper bound and solution tree recorded at the root is the optimal solution.

Algorithm 3: AND/OR Branch-and-Bound (AOBB) [32]

Input: A graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, pseudo-tree $\mathcal{T}$, heuristic function $h(\cdot)$
Output: Optimal solution to $\mathcal{M}$

Create root OR node r labeled corresponding to the root X_1 of pseudo-tree $\mathcal{T}$
$OPEN \leftarrow \{r\}$
$u(r) \leftarrow \infty$; $T^*(r) \leftarrow \emptyset$
while $OPEN \neq \emptyset$ **do**
    `// Node Expansion`
    $n \leftarrow top(OPEN)$; $OPEN \leftarrow OPEN - \{n\}$
    $succ(n) \leftarrow \emptyset$
    if *n is an OR node (labeled X_i)* **then**
        foreach $x_i \in D_i$ **do**
            Create AND node n' labeled x_i
            $u(n') \leftarrow \infty$; $T^*(n') \leftarrow \emptyset$
            $succ(n) \leftarrow succ(n) \cup \{n'\}$
    else if *n is an AND node (labeled x_i)* **then**
        $canBePruned \leftarrow false$
        foreach *OR ancestor a of n* **do**
            $LB \leftarrow$ evalPST$(T^*(a), a)$ `// See Algorithm 4`
            if $LB \geq u(a)$ **then**
                $canBePruned \leftarrow true$
                break
        if $canBePruned = false$ **then**
            foreach *child X_j of $X_i \in \mathcal{T}$* **do**
                Create an OR node n' labeled by X_j
                $u(n') \leftarrow \infty$; $T^*(n') \leftarrow \emptyset$
                $succ(n) \leftarrow succ(n) \cup \{n'\}$
        else
            $p \leftarrow parent(n)$
            $succ(p) \leftarrow succ(p) - \{n\}$
    Add $succ(n)$ on top of $OPEN$
    `// Bound Propagation`
    while $succ(n) = \emptyset$ **do**
        $p \leftarrow parent(n)$
        if *n is an OR node (labeled X_i)* **then**
            if *X_i is the root* **then**
                return $\langle u(n), T^*(n) \rangle$
            $u(p) \leftarrow \sum_{m \in succ(p)} u(m)$; $T^*(p) \leftarrow \bigcup_{m \in succ(p)} T^*(m) \cup \{p\}$
        else if *n is an AND node (labeled x_i)* **then**
            if $u(p) > c(p, n) + u(n)$ **then**
                $u(p) \leftarrow c(p, n) + u(n)$; $T^*(p) \leftarrow T^*(n) \cup \{p\}$
        $succ(p) \leftarrow succ(p) - \{n\}$
        $n \leftarrow p$

Algorithm 4: Recursive Computation of Heuristic Evaluation Function (evalPST)

Input: Partial solution subtree $T(n)$ rooted at node n, heuristic function $h(\cdot)$
Output: Heuristic evaluation function value $f(T(n))$

```
1  if succ(n) = ∅ then
2      if n is an AND node then
3          return 0
4      else
5          return h(n)
6  else
7      if n is an AND node then
8          Let k_1 ... k_l be the OR children of n
9          return Σ_{i=1}^{l} evalPST(T(k_i), h(·))
10     else if n is an OR node then
11         Let k be the AND child of n
12         return c(n, k)+ evalPST(T(k), h(·))
```

The time complexity of AOBB is bounded by $O(nk^{w^*})$, which is when it explores every node in the search space. As a depth-first search algorithm, its space complexity is linear in the depth of the pseudo tree when we search a tree. When it searches the context-minimal graph, the space complexity is $O(nk^{w^*})$.

The next 2 sections cover two types approximation algorithms for graphical models that can be used as heuristics for AOBB and other AND/OR search algorithms for graphical models. Since each search node can be viewed as rooting a problem in itself, we can use any approximation algorithm. However, the following two function as admissible heuristics because they give upper bounds.

1.2.3.3 Mini-Bucket Elimination

The most commonly used heuristic guiding AND/OR search is the MBE (mini-bucket elimination) heuristic [17]. It is based on a relaxation of the exact BE (bucket elimination) algorithm [12]. Thus, MBE is an approximation algorithm that provides lower bounds on

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

Table 1.5: Notation on bucket elimination for graphical models.

the optimal solution. We present both algorithms next.

Bucket Elimination Bucket elimination [14] works relative to the same pseudo tree that defines the AND/OR search graph. Each variable X_i of $\mathcal{T}$ is associated with a bucket B_i that includes a subset of functions from $\mathbf{F}$. A function f_j is placed into a bucket B_i if X_i is the deepest variable in $\mathcal{T}$ such that $X_i \in S_{f_j}$. The scope of a bucket B_i is the union of the scopes of its functions. Each bucket B_i is then processed, bottom-up, from the leaves of the pseudo tree to the root by computing a new function, known as a *message*, $\lambda_{i\to p} = min_{x_i} B_i$, where p is the parent of i in the pseudo tree. This message is then placed in bucket B_p. 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 generated by the root bucket (a constant) is the optimal C^* value. The messages generated by this process represent a "cost to go" in the order from root to leaves, so BE in fact provides exact heuristics in the context of search. In particular, the optimal assignment can be computed in linear time by a greedy search using BE as a heuristic. The BE algorithm's time and space complexity is bounded by $O(nk^{w^*})$ [12].

We illustrate the algorithm on our example problem in **Figure 1.3**. We see that bucket elimination breaks the entire optimization problem of the graphical model into smaller subproblems. BE has been shown to be equivalent to exploring the context-minimal AND/OR search graph in a *bottom-up* fashion, given certain conditions [35].

If $w^*(d)$ (d is a DFS order of the pseudo-tree) is very large, then BE is not feasible. Mini-

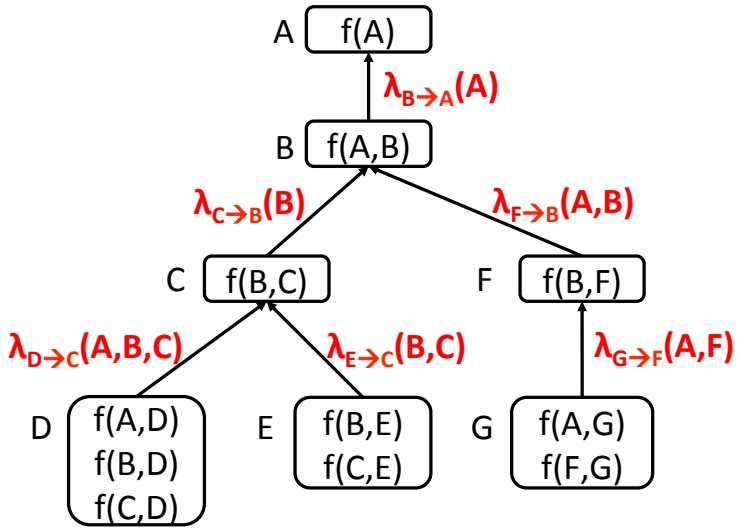


Figure 1.3: Illustration of bucket elimination.

Bucket Elimination (MBE) is a relaxation of BE that bounds the induced width of the problem via a parameter known as the i-bound [17]. The main difference is in how functions are processed inside buckets. MBE relaxes the problem by partitioning buckets into *mini-buckets* $B_k^1, \ldots, B_k^r$ 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 $\lambda_{k\to p}^s$, where $s \in 1, \ldots, r$ is the mini-bucket index. (The index s is omitted when there is no partitioning.) 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.

We provide details in **Algorithm 5**. The main loop (lines 1-9) partitions and generates the λ messages used in the above expressions. The message computed by the root variable is a lower bound to the optimal solution of the graphical model. When the i-bound equals the induced width w^*, each bucket partition will consist of a single mini-bucket, so the algorithm reduces to bucket elimination. The time and space complexity is $O(nrk^i)$, where r is the maximum number of mini-buckets for any variable [14].

Algorithm 5: Mini-Bucket Elimination [17]

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 $\lambda^s_{q\to p}$

foreach $X_p \in \mathbf{X}$ *in bottom up order according to* $\mathcal{T}$ **do**
  $B_p \leftarrow \{f_j \in \mathbf{F} \mid X_p \in S_j\}$
  $\mathbf{F} \leftarrow \mathbf{F} - \mathbf{F}_p$
  Put all generated messages $\lambda^s_{q\to p}$ in B_p
  Partition the B_p into mini-buckets $B^1_p, \ldots, B^r_p$ with scope bounded by the i-bound
  foreach $B^s_p \in B^1_p, \ldots, B^r_p$ **do**
    Let X_a be closest ancestor variable of X_p in the mini-bucket
    Generate message: $\lambda^s_{p\to a} \leftarrow \min_{X_p} \sum_{f_j \in B^s_p} f_j$
return *All* λ*-messages generated (root message is the min-sum lower bound)*

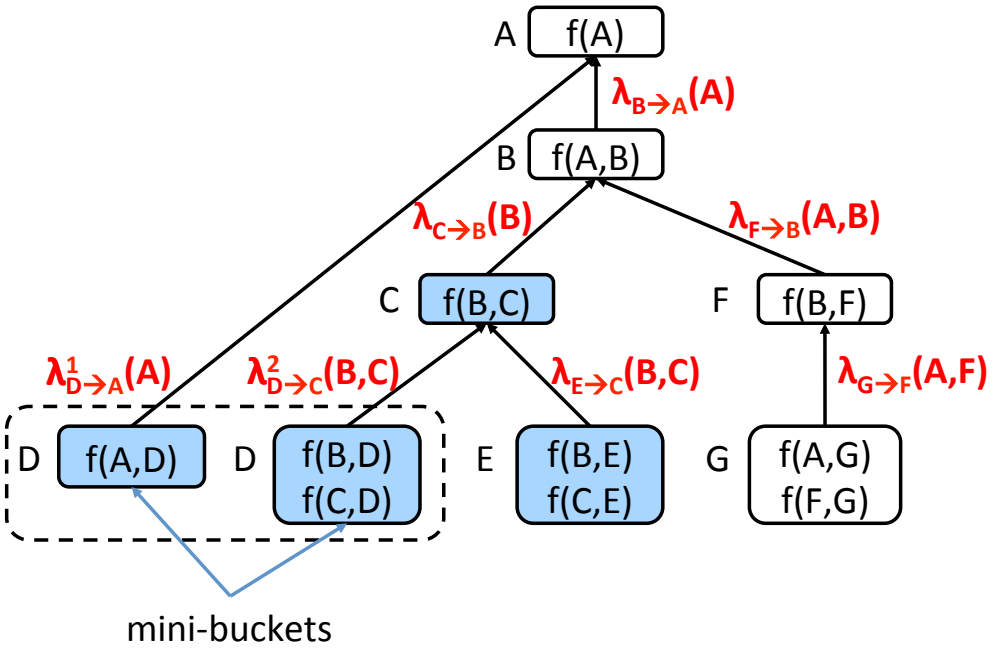


Figure 1.4: Example of mini-bucket elimination on the running example using an i-bound of 3.

We provide an example in **Figure 1.4** 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.

MBE messages can be used to construct a heuristic for search in the same spirit as BE [26]. Heuristics generated from the messages of mini bucket elimination are called *static* heuristics since they are pre-compiled before search starts. During search, they can be extracted efficiently by table lookups.

Definition 1.13 (**MBE heuristic).** *Let $\bar{x}_p$ be a partial assignment and $\bar{X}_p$ be the set of corresponding instantiated variables. Λ_k denotes the sum of the messages sent from bucket B_k to all of the instantiated ancestor variables, which includes only the subset of messages which are sent to a variable in $\bar{X}_p$.*

$$\Lambda_k(\bar{x}_p) = \sum_{s \in 1,\ldots,r_k | X_q \in \bar{X}_p} \lambda^s_{k \to q}(\bar{x}_p) \tag{1.1}$$

where r_k denotes the number of mini-buckets for variable X_k. The heuristic value for $\bar{x}_p$ is given by:

$$h(\bar{x}_p) = \sum_{X_k \in \mathcal{T}_p} \Lambda_k(\bar{x}_p) \tag{1.2}$$

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

Example. In the example, (see **Figure 1.4**), the heuristic function of the partial assignment $(A = 0, B = 1)$ is $h(A = 0, B = 1) = \lambda_{D \to A}(A = 0) + \lambda_{C \to B}(B = 1) + \lambda_{F \to B}(A = 0, B = 1)$.

$\mathbf{F}_i$	collection of functions, $\{f_j \in \mathbf{F} \vert X_i \in S_{f_j}\}$
$\lambda_{f_j}(X_i)$	auxiliary unary function associated with f_j over X_i
$\gamma_{f_j}(X_i)$	"min-marginal" of function f_j

Table 1.6: Notation on re-parameterization.

1.2.3.4 Cost Shifting Methods

Another class of approximations work by *re-parameterizing* the original problem. Assuming the min-sum task, we can approximate it by exchanging the min and sum operators, yielding

$$\min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot) \geq \sum_{f_j \in \mathbf{F}} \min_{\mathbf{x}} f_j(\cdot) \tag{1.3}$$

In other words, the bound is based on optimizing each function independently. This bound can be tightened by re-parameterizing the functions.

In the following we use the notation in **Table 1.6**. For each variable X_i, we introduce a collection of functions $\{\lambda_{f_j}(X_i) \vert f_j \in \mathbf{F}_i\}$ which we will add to the graphical model. Therefore, we require

$$\forall X_i, \quad \sum_{f_j \in \mathbf{F}_i} \lambda_{f_j}(X_i) = 0 \tag{1.4}$$

which enforces that there is no change to the global function represented by the graphical model.

This yields

$$
\begin{aligned}
C^* &= \min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot) \\
&= \min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot) + \sum_{X_i \in \mathbf{X}} \sum_{f_j \in \mathbf{F}_i} \lambda_{f_j}(X_i) \\
&= \min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot) + \sum_{f_j \in \mathbf{F}} \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i) \\
&= \min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} \left(f_j(\cdot) + \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i) \right) \\
&\geq \sum_{f_j \in \mathbf{F}} \min_{\mathbf{x}} \left(f_j(\cdot) + \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i) \right) \qquad (1.5)
\end{aligned}
$$

Let $\mathbf{\Lambda} = \{\lambda_{f_j}(X_i) | f_j \in \mathbf{F}_i, X_i \in \mathbf{X}\}$. The objective is then to find an optimal $\mathbf{\Lambda}$ such that we maximize the bound in (1.5). The new functions $f'_j(\cdot) = f_j(\cdot) + \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i)$ define a re-parameterization of the original model such that the bound presented in Equation 1.5 is optimal.

Depending on the literature, these λ functions are viewed as *equivalence-preserving transformations (EPTs)*, in the soft arc consistency literature [8], or as Lagrange multipliers enforcing consistency between the variable copies in the LP relaxation [50]. In the latter view, the re-parameterized functions are computed by coordinate descent updates that can be viewed as message passing [23].

In [25], a scheme known as LP-tightening was introduced to maximize (1.5) over $\mathbf{\Lambda} = \{\lambda_{f_j}(X_i) | f_j \in \mathbf{F}_i, X_i \in \mathbf{X}\}$.[1] First, we initialize each $\lambda_{f_j}(X_i) = 0$. As a coordinate descent algorithm (ascent in our case), the maximization proceeds by iteratively optimizing the objective over each $\lambda_{f_j}(X_i)$, fixing all other $\lambda_{f_j}(X_k)$ $s.t. X_k \neq X_i$. This yields the follow-

[1] The original work presents this material for bounding the max-sum problem, thus minimizing over $\mathbf{\Lambda}$ instead for a reversed version of the objective.

ing,

$$\begin{aligned}
&\max_{\lambda_{f_j}(X_i)} \left(\sum_{f_j \in \mathbf{F}_i} \min_{S_{f_j}} f_j(\cdot) + \lambda_{f_j}(X_i) \right) \\
&= \max_{\lambda_{f_j}(X_i)} \left(\sum_{f_j \in \mathbf{F}_i} \min_{X_i} \left[\min_{S_{f_j} \setminus X_i} f_j(\cdot) + \lambda_{f_j}(X_i) \right] \right)
\end{aligned}$$

We can define $\gamma_{f_j}(X_i) = \min_{S_{f_j} \setminus X_i} f_j(\cdot)$, which we refer to as *min-marginals*. Rearranging the above, we obtain:

$$\begin{aligned}
&\max_{\lambda_{f_j}(X_i)} \left(\sum_{f_j \in \mathbf{F}_i} \min_{X_i} \left[\min_{S_{f_j} \setminus X_i} f_j(\cdot) + \lambda_{f_j}(X_i) \right] \right) \\
&= \max_{\lambda_{f_j}(X_i)} \left(\sum_{f_j \in \mathbf{F}_i} \min_{X_i} \left[\gamma_{f_j}(X_i) + \lambda_{f_j}(X_i) \right] \right) \\
&\leq \max_{\lambda_{f_j}(X_i)} \left(\min_{X_i} \sum_{f_j \in \mathbf{F}_i} \left[\gamma_{f_j}(X_i) + \lambda_{f_j}(X_i) \right] \right)
\end{aligned} \tag{1.6}$$

One choice of $\lambda_{f_j}(X_i)$ that maximizes (1.6) is making all min-marginals of the updated functions equal. First we define the *average min-marginal* over $f_j \in \mathbf{F}_i$,

$$\bar{\gamma}_{\mathbf{F}_i}(X_i) = \frac{1}{|\mathbf{F}_i|} \sum_{f_j \in \mathbf{F}_i} \gamma_{f_j}(X_i)$$

Then we choose

$$\lambda_{f_j}(X_i) = \bar{\gamma}_{\mathbf{F}_i}(X_i) - \gamma_{f_j}(X_i) \tag{1.7}$$

Using this, we iterate over all the functions $f_j \in \mathbf{F}$, updating them with $f_j(\cdot) \leftarrow f_j(\cdot) + \lambda_{f_j}(X_i)$, then recalculate the updates $\lambda_{f_j}(X_i)$ and applying them again until convergence.

This update step is the building block for several algorithms such as factor graph linear

programming (FGLP), join graph linear programming (JGLP), and mini-bucket elimination with moment-matching (MBE-MM) [25].

Throughout this thesis, we use MBE-MM as a heuristic [25]. By taking the functions formed by the combination of functions within each mini-bucket, it applies the same update described above between the mini-buckets in order to tighten their approximation before generating messages. We present it in **Algorithm 6**, which is identical to **Algorithm 5** presented earlier, but adds lines 7-9 to compute the min-marginal for each mini-bucket along with the average min-marginal. It then proceeds to also update the function of each mini-bucket in line 12 before generating the messages as before in line 13.

Algorithm 6: Mini-Bucket Elimination with Moment Matching [25]

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 $\lambda^s_{q \to p}$

1 **foreach** $X_p \in \mathbf{X}$ *in bottom up order according to* $\mathcal{T}$ **do**
2 $\quad B_p \leftarrow \{f_j \in \mathbf{F} \mid X_p \in S_j\}$
3 $\quad \mathbf{F} \leftarrow \mathbf{F} - \mathbf{F}_p$
4 $\quad$ Put all generated messages $\lambda^s_{q \to p}$ in B_p
5 $\quad$ Partition the B_p into mini-buckets $B^1_p, \ldots, B^r_p$ with scope bounded by the i-bound
6 $\quad$ Let $f_{B^s_p} \leftarrow \sum_{f_j \in B^s_p} f_j$ denote the function of each mini-bucket.
7 $\quad$ **foreach** $B^s_p \in B^1_p, \ldots, B^r_p$ **do**
8 $\quad\quad$ Compute min-marginal: $\gamma_{B^s_p} \min_{S_{B^s_p} \, X_p} f_{B^s_p}$
9 $\quad$ Compute average min-marginal: $\bar{\gamma}_{B_p} \leftarrow \frac{1}{r} \sum_{B^s_p \in B^1_p, \ldots, B^r_p} \gamma_{B^s_p}$
10 $\quad$ **foreach** $B^s_p \in B^1_p, \ldots, B^r_p$ **do**
11 $\quad\quad$ Let X_a be closest ancestor variable of X_p in the mini-bucket
12 $\quad\quad$ Update function: $f_{B^s_p} \leftarrow f_{B^s_p} - \gamma_{B^s_p} + \bar{\gamma}_{B_p}$
13 $\quad\quad$ Generate message: $\lambda^s_{p \to a} \leftarrow \min_{X_p} f_{B^s_p}$
14 **return** *All* λ*-messages generated (root message is the min-sum lower bound)*

The complexity of MBE-MM is still dominated by the step of computing the functions of each mini-bucket, which is required to generate the mini-bucket messages, and is $O(rk^i)$ time for each bucket, where r is the number of mini-buckets. Thus, its time complexity is equivalent to that of MBE. Clearly, since the messages are also the same size, the space

complexity is also equivalent. This gives MBE-MM a time and space complexity of $O(nrk^i)$ [21].

Chapter 2

Cost-Directed Look-ahead in AND/OR Search via Residual Analysis

2.1 Introduction

The goal of our research is to improve both the exact and anytime performance of AOBB using the MBE heuristic. It has repeatedly observed that MBE gets tighter as the i-bound approaches the problem induced width. In problems with high induced width, this approach is likely to fail, since i-bound cannot be made close to the problem's induced width. 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.) [22, 47]. Look-ahead improves the heuristic function $h(\cdot)$ of a node by expanding the search tree below it and backing up the $h(\cdot)$ values of descendants (known as a *Bellman update*). Thus, look-ahead can be seen as a secondary search embedded in the primary depth-first search.

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 search. In this paper we address the challenge of making it cost effective. First, we

develop the notion of *look-ahead graphical model* which presents the look-ahead task as min-sum task over a graphical 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 full inference algorithms can be used for look-ahead computation. Second, we develop the notion of *local bucket error*, which we show to be equivalent to the *residuals* in a single level 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 compute approximations. Then, we improve AOBB exploiting local bucket errors in a number of ways:

- At some nodes, they provide overhead-free depth one look-ahead, which translates into a heuristic improvement. This leads to better pruning and value ordering.
- At each node, they give advice on which is the right level of look-ahead that may improve the heuristic function. In fact, AOBB makes a negative use of the information. It looks-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 dictates the look-ahead computation for each variable and *prune* them individually as a preprocessing step based on the local bucket errors.

In most literature, the heuristic function is treated as a *black box* and no assumption is made about its topology. The originality 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 local 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 of this observation

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, the main contribution of this work 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 locally 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 and more generally, better anytime profiles over the current methods of AOBB with static MBE heuristics.

The rest of the chapter is organized as follows. In section 2.2, we describe our main contributions. We introduce the notion of local bucket error in MBE, establish its connection to the look-ahead residuals, and present an algorithm for computing them. Next, we show how the local bucket errors can be used to guide look-ahead. In section 2.3, we show in several benchmarks how error is distributed non-homogeneously along the search spaces. In section 2.4, we provide an empirical evaluation our selective look-ahead scheme in the context of AOBB with MBE heuristics for both exact solutions and anytime performance. Section 2.5 concludes.

2.2 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 $\mathcal{M}(\bar{x}_p, d)$
$E_k(\bar{x}_p)$	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 2.1: 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.

2.2.1 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 $h(\cdot)$ value of a node by the best alternative 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 [44, 47]. 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. We extend Definition 1.6 based on OR search spaces to AND/OR search spaces.

Definition 2.1 (**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 the look-ahead.

Definition 2.2 (**residual**). *The depth d* residual *of node n is*

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

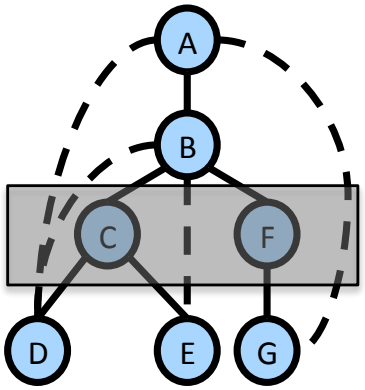


Figure 2.1: Look-ahead subtree example for $\mathcal{T}_{B,1}$ (shaded region)

2.2.2 The Look-ahead Graphical Model

We pointed out in the introduction 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 graphical sub-models. 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 it 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 2.3 (**look-ahead subtree**). *Given a 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 pruned below depth d. Note that X_p is excluded from the look-ahead subtree since the look-ahead computation does not depend on X_p.*

Figure 2.1. In our running example pseudo tree $\mathcal{T}$, $\mathcal{T}_{B,1}$ is the shaded region. It illustrates which variables in the AND/OR search space are minimized over in the look-ahead computation. We will use the look-ahead subtree $\mathcal{T}_{p,d}$ to refer its set of variables.

Next we define the *look-ahead graphical model*, which captures the problem *micro*-structure,

Definition 2.4 (**look-ahead graphical model**). *Consider a graphical model* $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$ *with pseudo-tree* $\mathcal{T}$*, the set of messages generated by MBE(i) along the pseudo tree, and 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* $\mathbf{X}_{p,d}$*:* $\mathcal{M}(\bar{x}_p, d)$*, is defined by the variables* $\{X_k | X_k \in \mathcal{T}_{p,d}\}$*,*
- *Domains* $\mathbf{D}_{p,d}$*: original domains.*
- *Functions* $\mathbf{F}_{p,d}$*:*
  - *Possibly partially assigned original functions that were originally placed in the buckets of* $\mathcal{T}_{p,d}$*,*

$$\{f(\bar{x}_p) | X_k \in \mathcal{T}_{p,d}, f \in B_k\}$$

  - *Possibly partially assigned messages sent from buckets below* $\mathcal{T}_{p,d}$ *to buckets in* $\mathcal{T}_{p,d}$ *by the MBE algorithm. Thus, for each* $X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}$*,*

$$\{\lambda^s_{j\to k}(\bar{x}_p) | s \in 1, \ldots, r_j, X_k \in \mathcal{T}_{p,d}\}$$

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 $\bar{x}_p$. Therefore, for a given d, they can easily be computed prior to search.

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

$$L^d(\bar{x}_p) = \min_{\bar{x}_{p,d}} \left\{ \sum_{\substack{X_k \in \mathcal{T}_{p,d} \\ f \in B_k}} f(\bar{x}_p, \bar{x}_{p,d}) + \sum_{\substack{X_k \in \mathcal{T}_{p,d} \\ s \in 1,\ldots,r, X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}}} \lambda^s_{j\to k}(\bar{x}_p, \bar{x}_{p,d}) \right\} \tag{2.1}$$

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

The following property shows 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 the MBE heuristic in graphical models).** *Consider a graphical model $\mathcal{M}$, a pseudo-tree $\mathcal{T}_{p,d}$ and its associated AND/OR search tree. If the MBE heuristic (Definition 1.13) is used, the depth d look-ahead value of partial assignment $\bar{x}_p$ (Definition 2.1) satisfies,*

$$h^d(\bar{x}_p) = L^d(\bar{x}_p) + \sum_{X_k \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_k(\bar{x}_p) \tag{2.2}$$

where $\mathcal{T}_{p,d}$ is the corresponding look-ahead tree and the expression $X_k \in \mathcal{T}_p - \mathcal{T}_{p,d}$ denotes the variables below the look-ahead tree.

Proof. See Appendix A.1. □

Note that the second term in Equation 2.2 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 completely assigned by $\bar{x}_p$, so the expression is a constant 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 X_p where the width is smaller than the depth (i.e., $w_{p,d} < d$) exact inference by BE is likely to be carried out more efficiently than with search. Though the space complexity of look-ahead is no longer linear when carried out in this manner, as long as $w_{p,d}$ is less than the i-bound, MBE itself would still be the dominant factor in memory usage.

Computing AND/OR d-level look-ahead requires solving a min-sum problem dictated by the *look-ahead subtree*. Even with the use of bucket elimination, it can still 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 2.2). We analyze depth 1 residuals and show how they can be used to estimate residuals of higher depth.

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

PROPOSITION **2** (**AND/OR d-level residual for MBE).** *Consider a graphical model $\mathcal{M}$, a pseudo-tree $\mathcal{T}_{p,d}$ and its associated AND/OR search tree. If the MBE heuristic (Definition 1.13) guides the search, the depth d residual at $\bar{x}_p$ (Definition 2.2) satisfies*

$$res^d(\bar{x}_p) = L^d(\bar{x}_p) - \sum_{X_k \in \mathcal{T}_{p,d}} \Lambda_k(\bar{x}_p) \tag{2.3}$$

Proof. From Definition 2.2, $res^d(n) = h^d(n) - h(n)$ Replacing h^d and h with Proposition 1 and Equation 1.2 respectively, we obtain the equation above. □

Note that the subtracted expression in Equation 2.3 is a constant. Therefore, as could be expected, computing the residual requires to computing $L^d(\bar{x}_p)$. We therefore propose to approximate d-level residuals using a sum of 1-level 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 A.2. □

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

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

Therefore, this suggests that 1-level residuals can be informative when they are large, pointing out which levels are likely to contribute to a d-level look-ahead. Furthermore, if the 1-level residuals for all nodes at a particular level are 0, they contribute nothing. Since 1-level residuals can be generally informative for d-level look-ahead, we next analyze 1-level residuals for MBE heuristics. We show that it corresponds to a notion of *local bucket error* of MBE, to be defined next.

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

Definition 2.5 (**combined bucket and mini-bucket messages).** *Given a mini-bucket partition $B_k = \cup_k B_k^r$, we define the combined mini-bucket message at B_k,*

$$\mu_k(\cdot) = \sum_r \left(\min_{x_k} \sum_{f \in B_k^r} f(\cdot) \right) \tag{2.4}$$

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

$$\mu_k^*(\cdot) = \min_{x_k} \sum_{f \in B_k} f(\cdot) \tag{2.5}$$

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

Definition 2.6 (**local bucket error of MBE).** *Given a completed run of MBE, the local bucket error function at B_k denoted E_k is*

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

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

Next, we show that the local bucket error of MBE equals the 1-level residual.

Theorem 2.1 (**equivalence of residuals and local bucket errors**). *Assume an execution of MBE(i) along $\mathcal{T}$ yielding heuristic h, then for every $\bar{x}_p$,*

$$res^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} E_k(\bar{x}_p) \tag{2.6}$$

We first present the following lemmas which relates μ_k (Equation 2.4) to the MBE heuristic of its parent X_p, and relates μ_k^* (Equation 2.5) to $L^d(\bar{x}_p)$.

Lemma 1. *If X_k is a child of variable X_p, then $\Lambda_k(\bar{X}_p) = \mu_k(\bar{X}_p)$.*

Proof. $\Lambda_k(\bar{X}_p)$ is the sum of messages that MBE(i) sends from B_k to the buckets of variables in $\bar{X}_p$, defined in Equation 1.1 as

$$\Lambda_k(\bar{x}_p) = \sum_{s \in 1,\ldots,r, X_q \in \bar{X}_p} \lambda^s_{k \to q}(\bar{x}_p)$$

Since X_p is the parent of X_k, $\Lambda_k(\bar{X}_p)$ is the sum of all the messages sent from X_k, which is the definition of $\mu_k(\bar{X}_p)$. □

Lemma 2. *If X_k is a child of variable $X_p \in \mathcal{T}$, then $L^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} \mu_k^*(\bar{x}_p)$.*

Proof. Given the expression of $L^1(\bar{x}_p)$ (Equation 2.1, substituting $d = 1$), we can push the minimization into the summation, yielding:

$$L^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} \min_{x_k} \left\{ f_k(\bar{x}_p, x_k) + \sum_{X_j \in \mathcal{T}_p - ch(X_p)} \lambda_{j \to k}(\bar{x}_p, x_k) \right\} \tag{2.7}$$

The set of functions inside each $\min_{x_k}$ are, by definition, the set of functions in B_k placed there either originally or are messages received from its descendants, therefore we can rewrite

$$L^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} \min_{x_k} \sum_{f \in B_k} f(\bar{x}_p)$$

By the definition of the exact bucket message (Equation 2.5), we obtain

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

□

Proof of Theorem 2.1. From **Proposition 2** given $d = 1$, we have

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

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 2.6**),

$$res^1(\bar{x}_p) = \sum_{X_k \in ch(X_p)} E_k(\bar{x}_p) \tag{2.8}$$

□

Corollary 2. *When a bucket is not partitioned, its local bucket error is 0, contributing 0 to the residual and to the look-ahead value of its parent.*

Establishing this equivalence between the 1-level 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 2.3**), which is defined in terms of these variables.

2.2.3 Computing Local Bucket Errors

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

Algorithm 7: Local Bucket Error Evaluation (LBEE)

Input: A Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, a pseudo tree $\mathcal{T}$, i-bound
Output: Error function E_k for each bucket B_k
Initialization: Run $MBE(i)$ w.r.t. $\mathcal{T}$.
foreach $B_k, X_k \in \mathbf{X}$ **do**
  Let $B_k = \cup_r B_k^r$ be the partition used by $MBE(i)$
  $\mu_k = \sum_r (\min_{x_k} \sum_{f \in B_k} f)$
  $\mu_k^* = \min_{x_k} \sum_{f \in B_k} f$
  $E_k \leftarrow \mu_k^* - \mu_k$
return E *functions*

Algorithm 7 (LBEE) computes the local bucket error for each bucket. 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 μ_k, the exact bucket message μ_k^*, and the error function E_k. 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 2.7 (**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 the output bucket following MBE(i). The pseudo-width of $\mathcal{T}$ relative to $MBE(i)$ is $psw(i) = \max_k \{psw_k^{(i)}\}$*

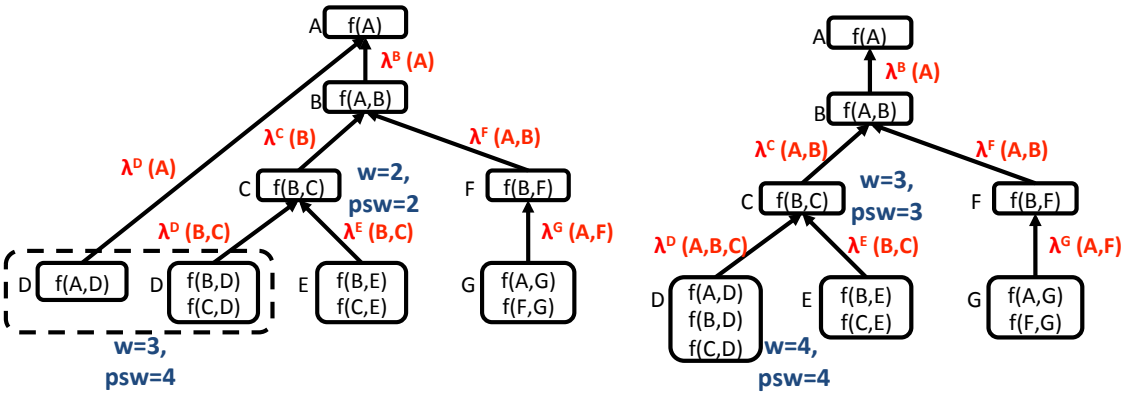


Figure 2.2: Example to illustrate the concept of pseudo-width. Left: the mini-bucket tree, right: the exact bucket tree. Buckets C and D in both are annotated with their induced width and pseudo-width.

Theorem 2.2 (complexity of LBEE). *The time and space complexity of LBEE is* $O(nk^{psw(i)})$, *where* n *is the number of variables,* k *bounds the domain size, and* $psw(i)$ *is the pseudo-width along* $\mathcal{T}$ *relative to* $MBE(i)$.

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

Example. In **Figure 2.2**, 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. Buckets C and D are annotated with their induced width and pseudo-width, where the induced width of a bucket is the number of variables in its largest mini-bucket. 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. However, when moving to process bucket C, we still use the message obtained by MBE ($\lambda D \rightarrow C(B, C)$). Thus, the pseudo-width of bucket C is 2 rather than 3. The distinction

here is that unlike in exact bucket elimination, the complexity of LBEE stays local.

2.2.3.1 Approximating Local Bucket Errors

Indeed, since the time and space complexity of $LBEE$ is at least as high as that of MBE itself, it is not advisable to use it in practice when the i-bound is high. Therefore, we consider *sampling* and subsequently *aggregating* the local bucket error functions for each variable. 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.

We first address the space complexity with the following:

Definition 2.8 (**average local error).** *The average local error of* B_k *given a run of* $MBE(i)$ *is*

$$\tilde{E}_k = \frac{1}{|\mathbf{D}_{Scope(B_k)}|} \sum_{\bar{x}_k} E_k(\bar{x}_k) \tag{2.9}$$

Computing $\tilde{E}_k$ takes $O(k^{psw(i)})$ time per variable, but $O(1)$ 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 by dividing the $E_k(\bar{x}_k)$ term by the exact bucket message $\mu^*_k(\bar{x}_k)$. 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 $O(k^{psw(i)})$ values per variable, 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 2.9 (**sampled average local error).** *The sampled average local error of* B_k *given a run of* $MBE(i)$ *is*

$$\hat{E}_k = \frac{1}{\#samples} \sum_{\bar{x}_k} E_k(\bar{x}_k) \quad \bar{x}_k \sim U(Scope(B_k)) \tag{2.10}$$

2.2.4 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 heuristic accuracy. From **Proposition 3**, we also have a lower-bound on d-level residual using summation of 1-level 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 2.10 (**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 2.11 (**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 2.3** the look-ahead subtree $\mathcal{T}_{B,2}$. Since D is the only relevant variable due to its mini-bucket partitioning (see **Figure 1.4**), 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 8 (CompilePLS(ϵ)) generates the ϵ-pruned look-ahead subtree for each variable. Its complexity is linear in the size of the look-ahead subtree $\mathcal{T}_{p,d}$.

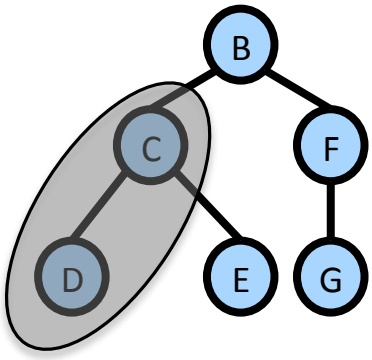


Figure 2.3: Look-ahead subtree $\mathcal{T}_{B,2}$. (B is shown for reference to root the multiple subtrees that make up $\mathcal{T}_{B,2}$.) Circled: the ϵ-pruned look-ahead subtree $\mathcal{T}^{\epsilon}_{B,2}$ for $\epsilon = 0$.

Algorithm 8: 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 $\tilde{E}_k$ for each $X_k \in \mathbf{X}$
2 $\mathbf{X}' \leftarrow$ all nodes in $\mathcal{T}$ that are ϵ-relevant ($X_k \in \mathbf{X}$ s.t. $\tilde{E}_k > \epsilon$)
3 **foreach** $X_p \in \mathbf{X}$ **do**
4 Initialize $\mathcal{T}^{\epsilon}_{p,d}$ to $\mathcal{T}_{p,d}$
5 **while** $\mathcal{T}^{\epsilon}_{p,d}$ *has leaves* $\notin \mathbf{X}'$ **do**
6 Remove leaf $X_j \notin \mathbf{X}'$ from $\mathcal{T}^{\epsilon}_{p,d}$
7 **return** $\mathcal{T}^{\epsilon}_{p,d}$ for each X_p

The ϵ-pruned look-ahead subtrees are used during search for performing d-level look-ahead. This suggests a static approach for deciding where look-ahead before search begins. When $\epsilon = 0$, the look-ahead subtrees guide the computation to only compute as much as necessary for a given d-level look-ahead, since 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 9** 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 1.13). These are also used by CompilePLS(ϵ) to generate the ϵ-pruned look-ahead subtrees. Lines 2-7 describe the main execution of the algorithm for each input

Algorithm 9: Look-ahead Heuristic for MBE (MBE-Look-ahead)

Input: A Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, a pseudo tree $\mathcal{T}$, i-bound, threshold ϵ, depth d, partial assignment $\bar{x}_p$

Output: Lower bound on partial assignment $\bar{x}_p$ to $\mathcal{M}$

1 **Initialization:** Run $MBE(i)$ w.r.t. $\mathcal{T}$ and run CompilePLS(ϵ) to generate $\mathcal{T}^{\epsilon}_{p,d}$ for each $X_p \in \mathbf{X}$

2 **if** $\mathcal{T}^{\epsilon}_{p,d}$ *is empty* **then**

3 **return** $\sum_{X_k \in \mathcal{T}_p} \Lambda_k(\bar{x}_p)$

4 **else**

5 Construct look-ahead graphical model $\mathcal{M}^{\epsilon}(\bar{x}_p, d)$ w.r.t. $\mathcal{T}^{\epsilon}_{p,d}$

6 $L^d(\bar{x}_p) :=$ min-sum Bucket Elimination on $\mathcal{M}^{\epsilon}(\bar{x}_p, d)$

7 **return** $L^d(\bar{x}_p) + \sum_{X_k \in \mathcal{T}_p - \mathcal{T}^{\epsilon}_{p,d}} \Lambda_k(\bar{x}_p)$

partial assignment $\bar{x}_p$ encountered. If the ϵ-pruned look-ahead subtree $\mathcal{T}^{\epsilon}_{p,d}$ is empty, we return the MBE heuristic value. 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).

2.3 Analysis of Local Errors

To illustrate how look-ahead subtree pruning behaves on selected problem instances, we annotate their respective variables in the pseudo-trees with the average local error. We will describe the behavior of the errors on 4 problem instances: one of each from the **pedigree**, **grid**, **spot5**, and **dbn** classes.

Figure 2.4 shows an annotated pseudo-tree for our running example problem. Only variable D has partitioning and a non-zero bucket error. On this particular example, for a particular level 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 2.1** by observing that D is the only relevant variable (**Figure 2.4**). For every other

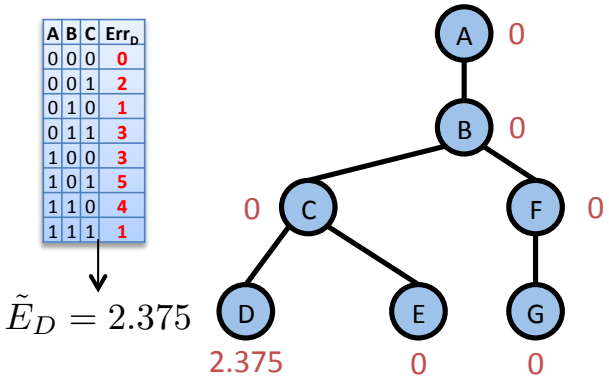


Figure 2.4: A pseudo-tree annotated with bucket errors, given that the error function of D shown here.

variable excluding C, the ϵ-pruned look-ahead subtrees are empty, thus correctly avoiding look-ahead computation.

The rest of this section illustrates this analysis on several pseudo-trees of instances from different problem classes. We annotated the pseudo-trees with the number of mini-buckets (mb), the pseudo-width (psw), and the average relative local bucket errors. Each node is color coded on a spectrum of pale yellow to dark red to indicate its relative degree of error. Within each 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 only show portions of the tree since the full pseudo-trees are too large to show [1].

We also plot the average relative local errors of each variable by its error ranking along with a plot that shows the number of mini-buckets in order to show the relation between the errors and the mini-bucket partitioning. 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. We next show several case studies.

[1] The full pseudo-trees can be viewed at `http://graphmod.ics.uci.edu/~wlam/pseudotree-errors-1609/`.

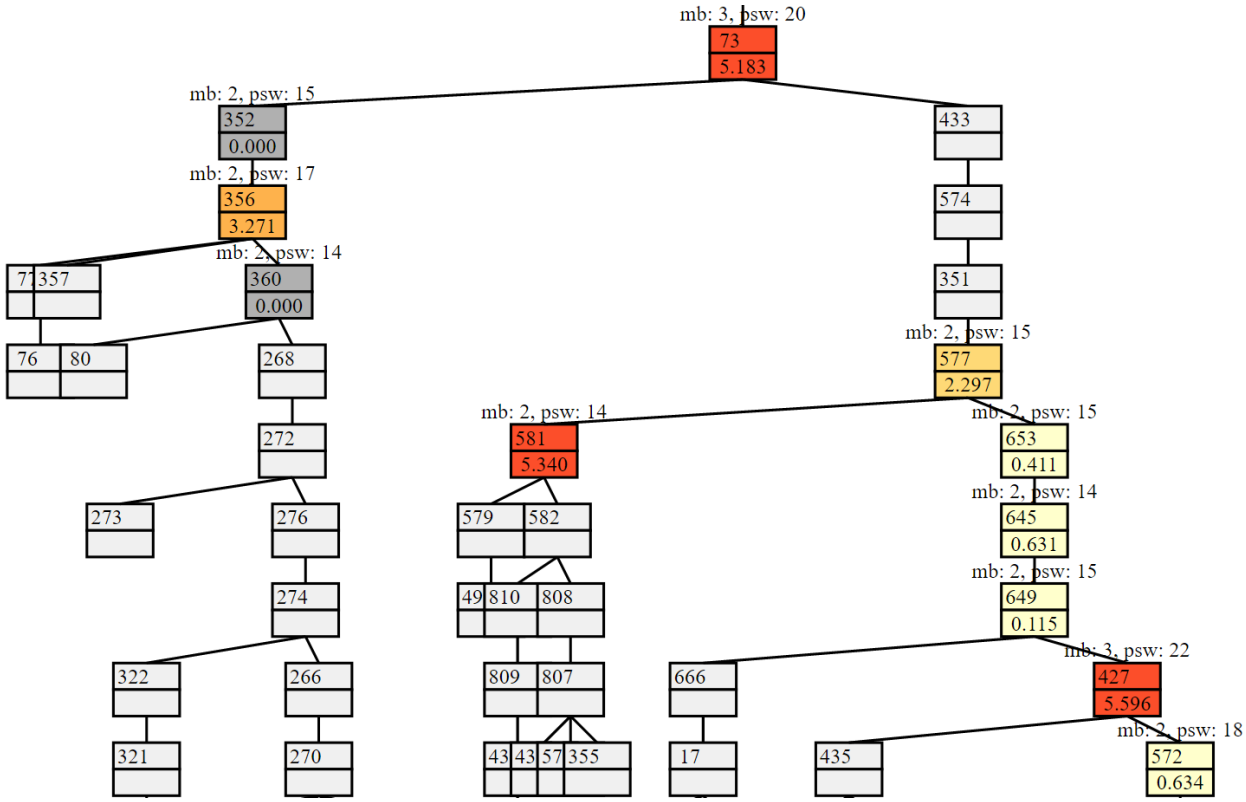


Figure 2.5: Extracted structure from pseudo-tree showing errors for *pedigree40* (n=842, k=7, w=27, h=111) with an i-bound of 12. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error.

2.3.1 Case Study: Pedigree

We show in **Figure 2.5** 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 errors tend to accumulate 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 2.6**, we plot the each variable with its error (top) and mini-bucket partitioning

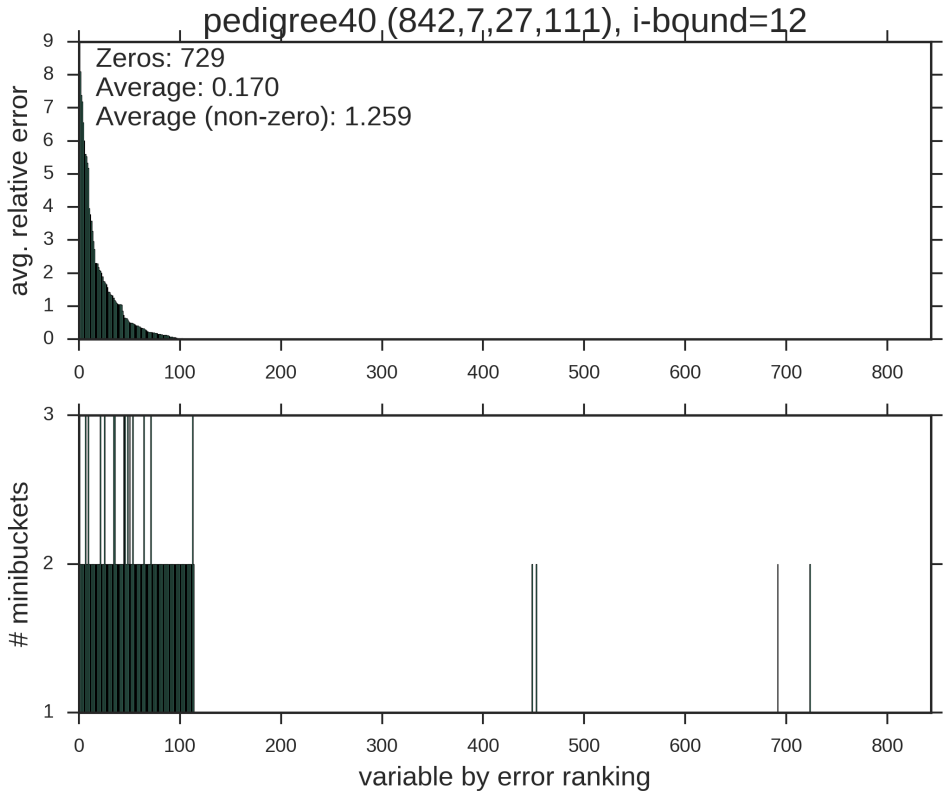


Figure 2.6: Distribution of errors and mini-buckets for *pedigree40*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

(bottom). The variables are sorted in descending order based on their error. We observe a range of errors across the variables, averaging at 0.17 (1.259 excluding zeros). Out of the 842 variables, 729 of them have zero error. We also see that bucket errors provide a much finer-grained measure of the error compared with counting the number of mini-buckets. More importantly, we see a few variables having 2 mini-buckets, yet have zero bucket error. These patterns were often observed in other benchmarks having a fair amount of subproblem decomposition (namely, when the height h of the pseudo-tree is small relative to n).

2.3.2 Case Study: Grid

Figure 2.7 provides another example, showing a part of the pseudo-tree for *grid80x80.f15* when the i-bound of the MBE applied is 14. This instance is difficult, having an induced

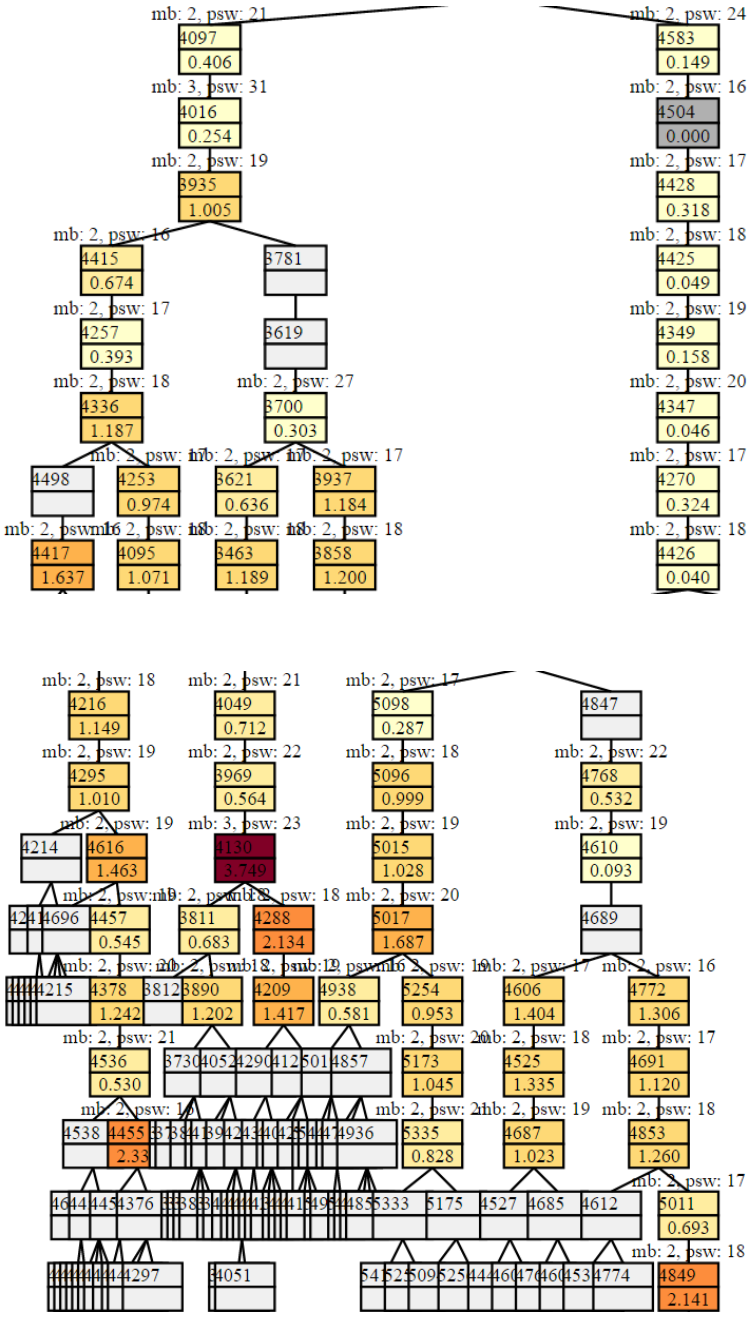


Figure 2.7: Extracted structures from pseudo-tree showing errors for grid instance grid80x80.f15 (n=6400, k=2, w=112, h=296) with an i-bound of 14. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error.

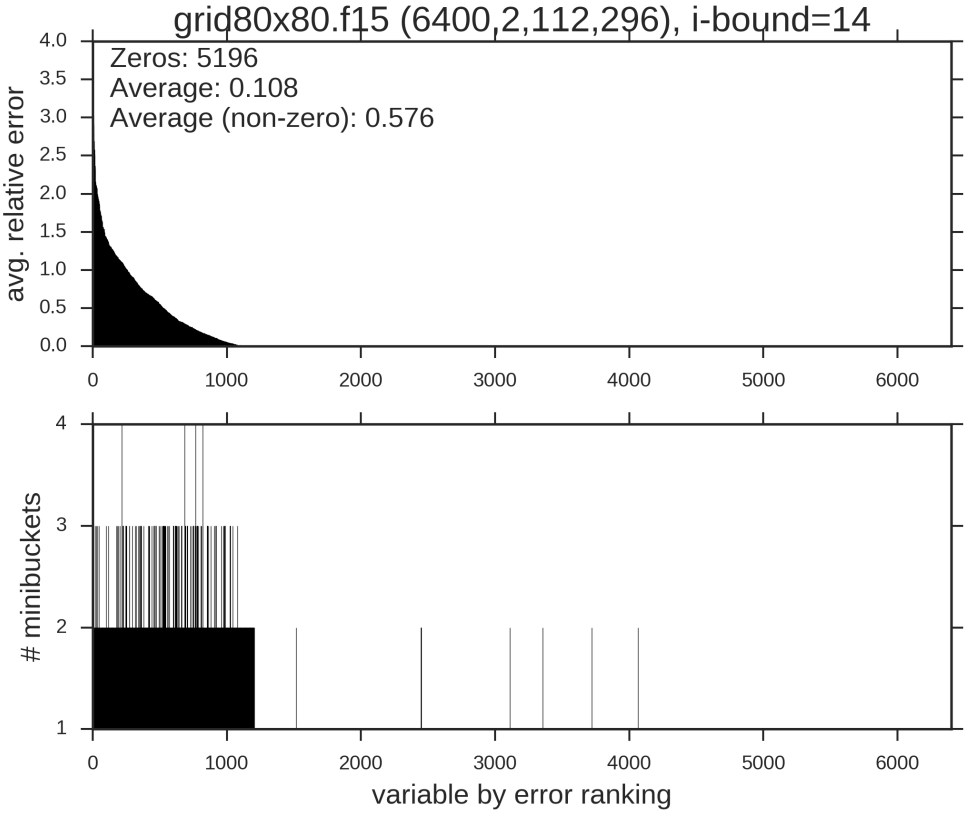


Figure 2.8: Distribution of errors and mini-buckets for grid instance *grid80x80.f15*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

width of 112. It indeed contains long chains with errors (see top plot). However, the nodes in the decomposed subproblems (see bottom plot) typically have higher error than those along the chains. Within decomposed subproblems, despite nearly all having 2 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).

In **Figure 2.8**, we see that most variables have zero error, while the rest vary within a relatively small range up to around 3.5, averaging only 0.576. 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. Once again, due to the high number of variables with zero error, we can have pruned look-ahead

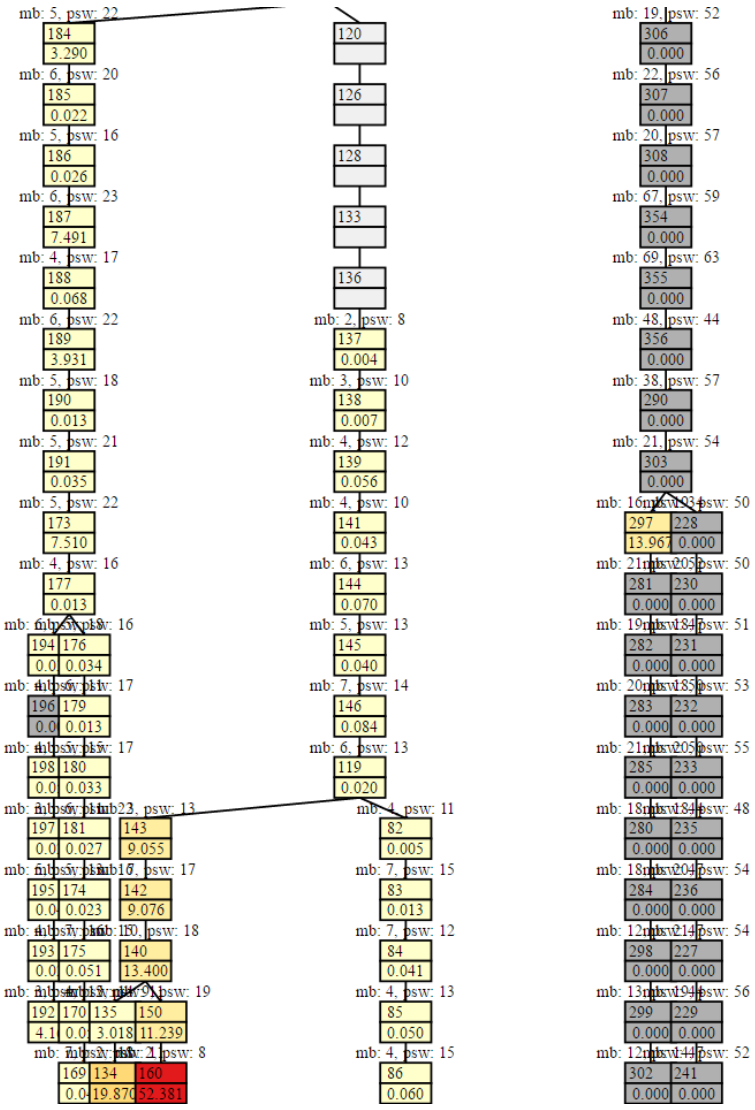


Figure 2.9: Extracted structure from pseudo-tree showing errors for SPOT5-414 (n=364, k=4, w=79, h=226) with an i-bound of 5. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error.

subtrees.

2.3.3 Case Study: SPOT5

Another example of an instance is SPOT5-414 in **Figure 2.9**. Here, the MBE i-bound is 5. We see here that there is a subset of the nodes, having a relatively large number of

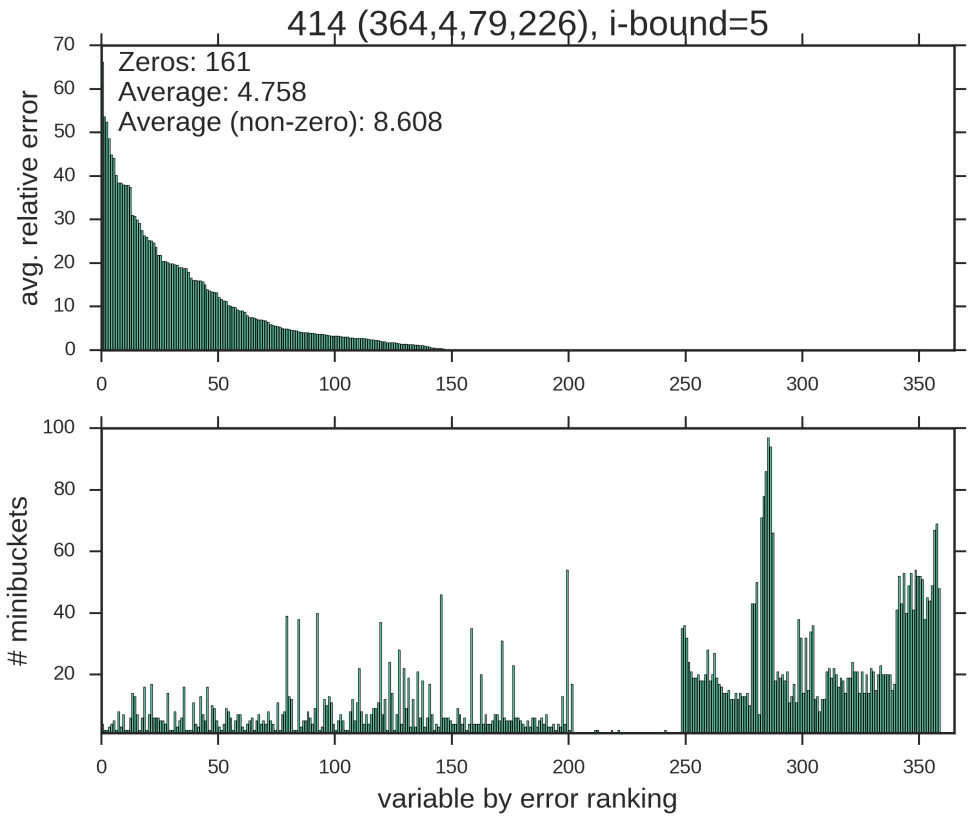


Figure 2.10: Distribution of errors and mini-buckets for *SPOT5:414*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

mini-buckets in the neighborhood of 20-30, while having no error. Considering that the pseudo-width of these nodes are also high (50-70), the fact that these nodes (variables) have zero bucket error will translate to avoiding intensive, yet frivolous look-ahead computation. Most other errors are relatively low (< 0.1). We observe these low errors despite the i-bound of 5 being much lower than the induced width of 79. Therefore, this tells us that the mini-bucket heuristic actually works fairly well on this benchmark in spite of the significant partitioning applied by MBE.

In **Figure 2.10**, we count that 161 variables have zero error, while there are a few places with relatively higher errors (we see errors up to near 70, but the average across the non-zero error variables is only 8.608). When comparing the magnitude of errors with the number of

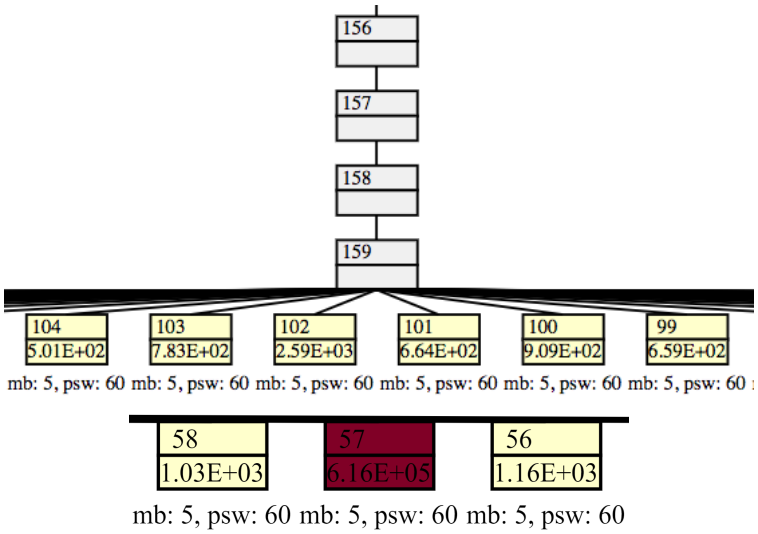


Figure 2.11: Extracted structure from pseudo-tree showing errors for DBN instance rus2_50_100_3_2 (n=160, k=2, w=59, h=59) with an i-bound of 14. *mb*: the number of mini-buckets, *psw*: the pseudo-width of that node. The top number in each box is the variable index and the bottom is the average relative local bucket error. We also include here another portion of the leaf level of the tree with an outlier node.

mini-buckets per variable, there is little correlation. Again, the high number of zero error buckets, will help to have a more effective control on the magnitude of look-ahead during search.

2.3.4 Case Study: DBN

In the last example pseudo-tree of the instance *rus2_50_100_3_2* from the DBN benchmark in **Figure 2.11** the i-bound is 14. The problem instances of this benchmark typically have a structure where there is a single point of decomposition at the bottom of the pseudo-tree. In particular, this instance has 100 of its 160 variables branching out from a single node (the actual branching node is not shown in the figure). There is partitioning only at the leaves with 5 mini-buckets per variable. The average relative bucket errors are extremely 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).

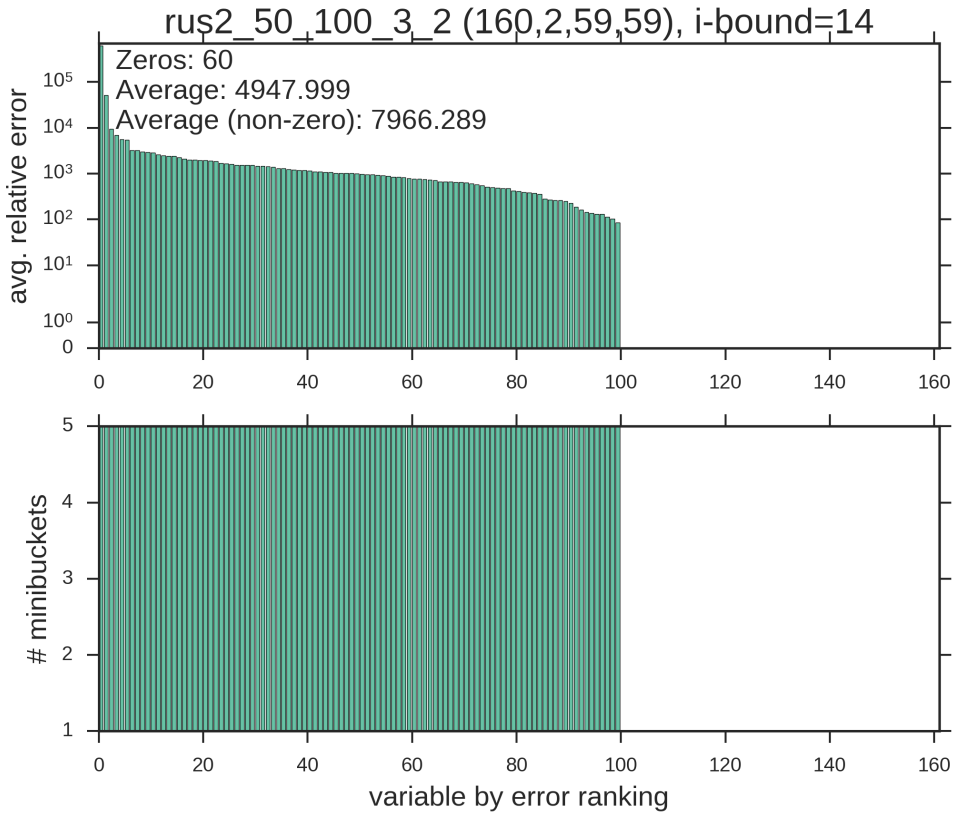


Figure 2.12: Distribution of errors and mini-buckets for DBN instance *rus2_50_100_3_2*. The variables (x-axis) are plotted in descending order based on their average relative local bucket error.

In **Figure 2.12**, we see that the errors are all very high for variables which do have error, with an average of 7966. In particular, we can observe that the errors' range is high. The other 60 variables which sit along the chain of the pseudo tree have no partitioning and thus zero bucket error. Consequently, for look-ahead, many variables along the chain would have empty look-ahead subtrees, thus avoiding unnecessary look-ahead.

2.3.4.1 Discussion

We considered a handful of differently structured instances in this section and 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

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 2.2: 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.

compared with the pedigree instance). We find that there are a fair number of nodes within the pseudo-trees having no error, which is useful for controlling the look-ahead. For example, in the SPOT5 instance, the majority of nodes appear to have high error when considering the mini-bucket partitioning alone, yet evaluating their bucket errors suggest 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 for most of its variables, having significant implications for look-ahead during search. To demonstrate this, we compiled the look-ahead subtrees for every problem instance across the 6 benchmarks used in our experimental evaluation in the following section. The results are summarized in **Table 2.2** 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.

2.4 Experimental Evaluation

We evaluate the impact of our look-ahead scheme on the performance of both finding exact solutions and anytime behavior. We use depth as a control parameter for adjusting the strength of look-ahead using a fixed small non-zero ϵ, and also demonstrate the impact of ϵ given fixed depths.

2.4.1 Overview and Methodology

We augmented breadth-rotating AND/OR branch-and-bound (BRAOBB) guided by $MBE(i)$ with moment-matching (MBE-MM) with our look-ahead scheme. BRAOBB is a variant of the AOBB algorithm that has improved the anytime performance through a method of rotating through subproblems in the AND/OR search space [39]. The MBE-MM heuristic is one of the best versions of the MBE heuristic, which adds a step that shifts costs between mini-buckets to tighten the approximation [25]. Together, they form one of the best algorithms for optimization in graphical models, which won the PASCAL inference competition in 2011 [40]. The pseudocode for our look-ahead algorithm is given in **Algorithm 9**. For the first step of CompilePLS(ϵ) (**Algorithm 8**) for compiling pruned look-ahead subtrees, the average was computed exactly if the local bucket error function had no more than 10^5 entries. Otherwise, we approximated this by sampling 10^5 of the entries and averaging over the samples. We compare it to the baseline heuristic which uses no look-ahead, enforced in our code using empty look-ahead subtrees. All algorithms were implemented in C++ (64-bit) and the 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 bound to 2 hours (7200 seconds).

Benchmarks. We evaluated the look-ahead scheme using the ϵ-pruned subtrees on benchmarks from the UAI and PASCAL2 competitions. This includes instances from genetic

Benchmark	# inst	n	k	w	h	$\|F\|$	a
Pedigree	12	581	3	19	79	794	4
		1006	7	39	143	1185	5
LargeFam3	69	874	3	21	44	1321	4
		2489	3	78	174	3772	4
Type4	38	3907	5	21	300	5749	4
		8473	5	66	971	13570	4
Promedas	36	615	2	28	65	625	3
		1911	2	94	165	1928	3
DBN	74	70	2	29	29	16167	2
		310	2	109	109	99927	2
Grids	21	400	2	24	62	1161	2
		6400	2	196	341	19201	2

Table 2.3: Benchmark statistics for. # inst - number of instances, n - number of variables, $w*$ - induced width, h - pseudo-tree height, k - maximum domain size, $|F|$ - number of functions, a - maximum arity. The top value is the minimum and the bottom value is the maximum for that statistic.

linkage analysis (**Pedigree, LargeFam3, Type4**) [20] and medical diagnosis (**Promedas**) [49], deep belief networks (**DBN**), and binary grids (**Grids**). For each problem, we used a fixed pseudo-tree for the algorithms applied with all levels of look-ahead, thus yielding the induced widths and heights reported in **Table 2.3**. Within each of these benchmarks, we selected a subset of instances that were *not* trivial to solve with weak heuristics (namely solved in less than 30 seconds with an i-bound of 6 or lower). Overall, we evaluated on 250 problem instances of varying difficulty, having induced widths ranging from 19 to 196.

Heuristics. The i-bounds selected for each problem instance yields a range of heuristic strengths from relatively weak to strong. The lowest i-bound for each instance is based on whether it was possible with that i-bound to solve the problem in the 2 hour time limit, while the highest i-bound shown is selected to fit within the 4GB memory limit. We also show an i-bound between these two.

The next two sections (2.4.2 and 2.4.3) use depth as a control parameter for look-ahead and describe the results on exact solutions and anytime behavior, respectively. In order to show the impact of depth clearly, we fix the ϵ parameter to 0.01 to ensure that all error-free

variables are pruned from the look-ahead subtrees and pruning only small non-zero errors. We vary the look-ahead depth from 0 to 6, where 0 is the baseline with no look-ahead. Afterwards, in section 2.4.4, we show the impact of the ϵ parameter, where we vary it from 0 to 10, plus ∞ and show its impact on look-ahead with depths of 2 and 5.

2.4.2 Evaluating Look-ahead for Exact Solutions

Tables 2.4, 2.5, 2.6, 2.7, and **2.8**, 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 the 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 2.13, 2.14, 2.15, 2.16,** and **2.17** 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.

instance (n, k, 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 2.4: 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, k: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time and fewest nodes are boxed.

2.4.2.1 Pedigree

Table 2.4 shows the results in terms of time spent and nodes expanded to find the exact solution on selected i-bounds of representative instances in the benchmark and demonstrates how the look-ahead scheme performs. 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 of 1464 seconds. Indeed, the number of nodes expanded here decrease by 74%. However, on higher i-bounds, lookahead is less cost effective. For instance, we see on *pedigree51* with an i-bound of 22, the baseline is the best performer. The number of nodes here decreases only by 57%, even with the deepest look-ahead.

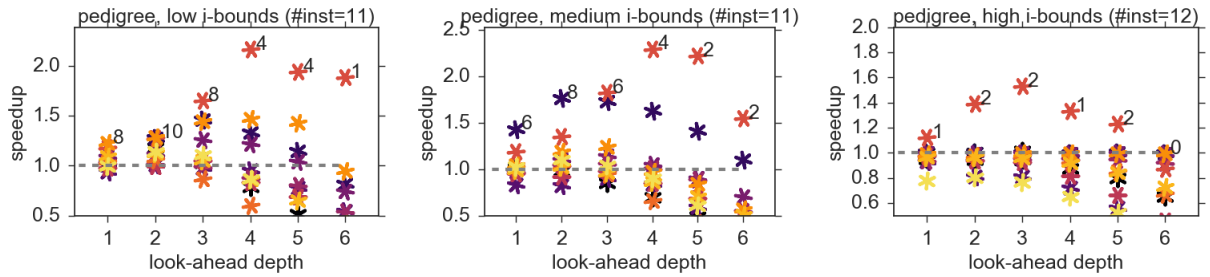


Figure 2.13: Solved **pedigree** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

Figure 2.13 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, though there are a few instances that still show benefit.

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.

instance (n, k, 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 2.5: 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, k: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time and fewest nodes are boxed.

2.4.2.2 LargeFam3

Table 2.5 shows the detailed results for representative instances in this benchmark. 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. 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.

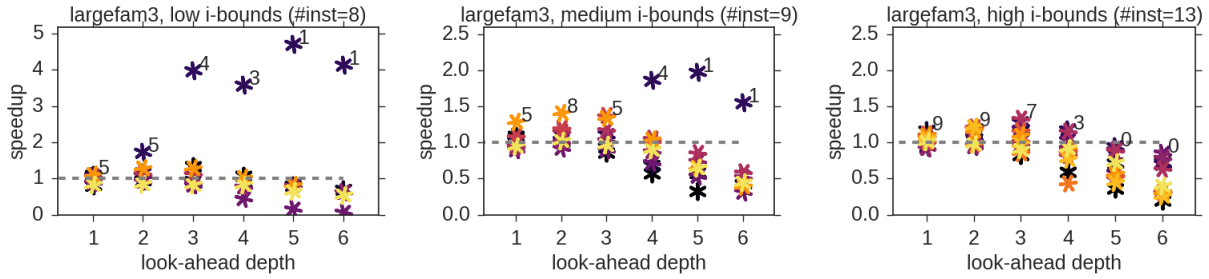


Figure 2.14: Solved **largefam3** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

Figure 2.14 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 over the baseline since it also includes more difficult instances that could not be solved with

weaker heuristics. Still, 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.

2.4.2.3 Promedas

Table 2.6 shows the detailed results for representative instances in the *promedas* benchmark. 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 with 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.

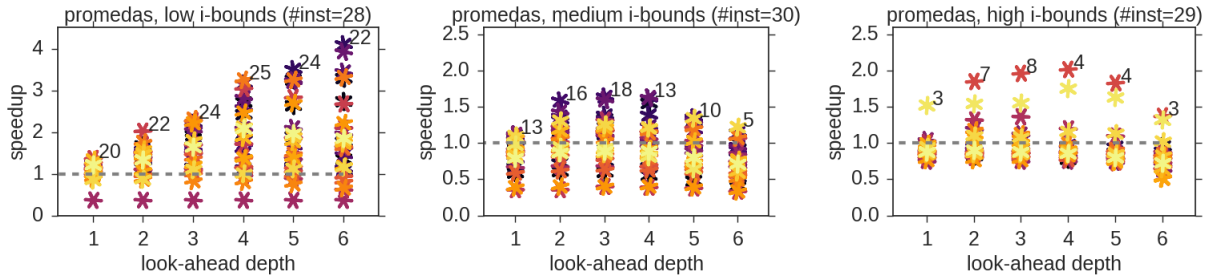


Figure 2.15: Solved **promedas** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

Figure 2.15 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. A few instances continue to improve thereafter. Moving to medium i-bounds, look-ahead improves over the baseline on only

instance (n, k, 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 2.6: 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, k: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time and fewest nodes are boxed.

about half of the 30 solved instances from depths 1 to 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 2.6**, 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 at depths even when we have the strongest heuristics that we can compile under the memory constraints.

2.4.2.4 DBN

Table 2.7 shows the detailed results for representative instances in the *DBN* benchmark. 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.

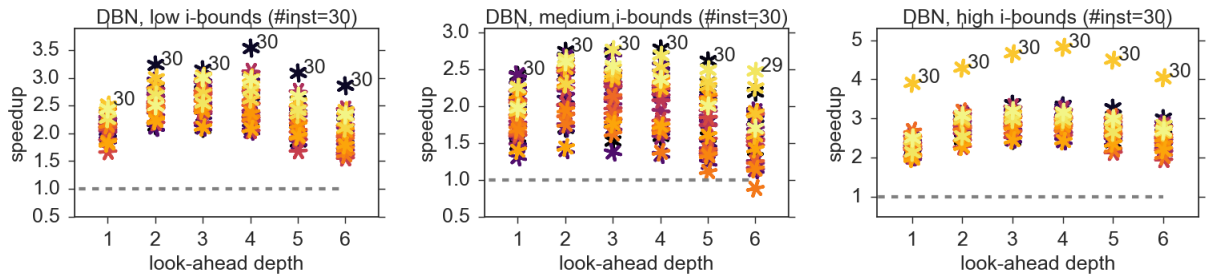


Figure 2.16: Solved **DBN** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

instance (n, k, 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 2.7: 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, k: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time and fewest nodes are boxed.

In **Figure 2.16**, 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. This benchmark is also 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 [43]. Indeed, the number of nodes expanded at an i-bound of 12 is greater than the number at an i-bound of 10. Thus, 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 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 until depths greater than 4.

2.4.2.5 Grids

Table 2.8 shows the detailed results for representative instances in this benchmark. We observe generally modest speedups for this benchmark. 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.

instance (n, k, 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 2.8: 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, k: maximum domain size, w^*: induced width, and h: height) Within each instance and i-bound, the best time and fewest nodes are boxed.

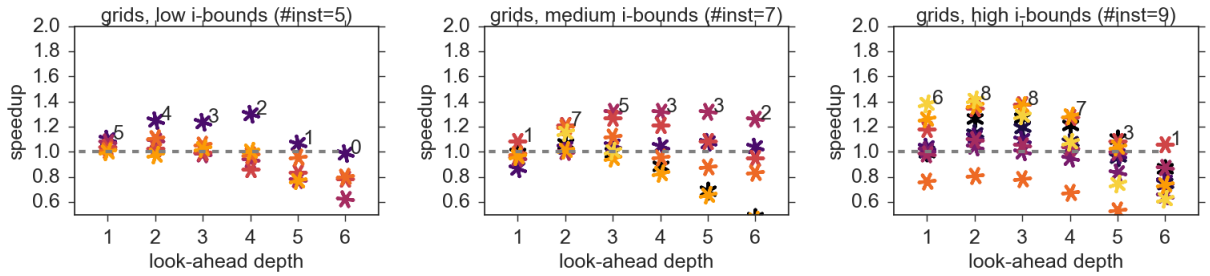


Figure 2.17: Solved **grid** instances: plot of speedups on instances by look-ahead depth. The number on top of each depth group is the number of instances that had speedup over 1. *#inst* indicates the number of instances in the benchmark that are shown in each plot.

In **Figure 2.17**, 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.

2.4.2.6 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 controlled for the best balance**: Across the benchmarks, the best depth tended to range between 2 and 3, suggesting that a modest level 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.

2.4.3 Evaluating the Anytime Behavior

We next show results investigating the impact of our look-ahead scheme on anytime behavior. This evaluation is especially important since based on the previous section, we observed that look-ahead tends to work well when the heuristic is weak. However, it is not possible to evaluate the most difficult instances in those benchmarks without a detailed look at all of the solutions it produces over time. Thus, we provide anytime results on representative instances that were the most difficult in the benchmark for each i-bound by plotting the cost as a function of time. We show two i-bounds for each instance: the highest we could use with our memory limit and another lower i-bound that was still able to produce solutions on a good number of instances in the benchmark. We show results in **Figures 2.18, 2.20, 2.22,** and **2.24**.

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. We define the normalized relative accuracy as $\frac{|C-C_w|}{|C_b-C_w|}$, where C_w and C_b are the worst and best solutions obtained at any time

amongst all setting of the algorithms. 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 2.19, 2.21, 2.23,** and **2.25**.

We omit the **pedigree** and **promedas** benchmarks from this part of the evaluation since they did not contain instances where were unable to find an exact solution.

2.4.3.1 LargeFam3

In **Figure 2.18**, we show the anytime plots for two representative instances from this benchmark over 2 different i-bounds. 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. The baseline manages to find a better solution at 100 seconds, but is eventually dominated once again by these same look-ahead settings. 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 3 or higher. Moving to an i-bound of 15, all of the look-ahead schemes outperform the baseline except for a depth of 1. Furthermore, the best solution is obtained near timeout with a look-ahead depth of 6. On *lf3-haplo_19_55* when using an i-bound of 12, look-ahead obtains solutions sooner than the baseline and maintains higher quality solutions. For the higher i-bound of 17, performance is similar across the different look-ahead depths, except for a depth of 6, look-ahead has a better solution early on and almost until timeout, where all other schemes catch up.

Figure 2.19 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

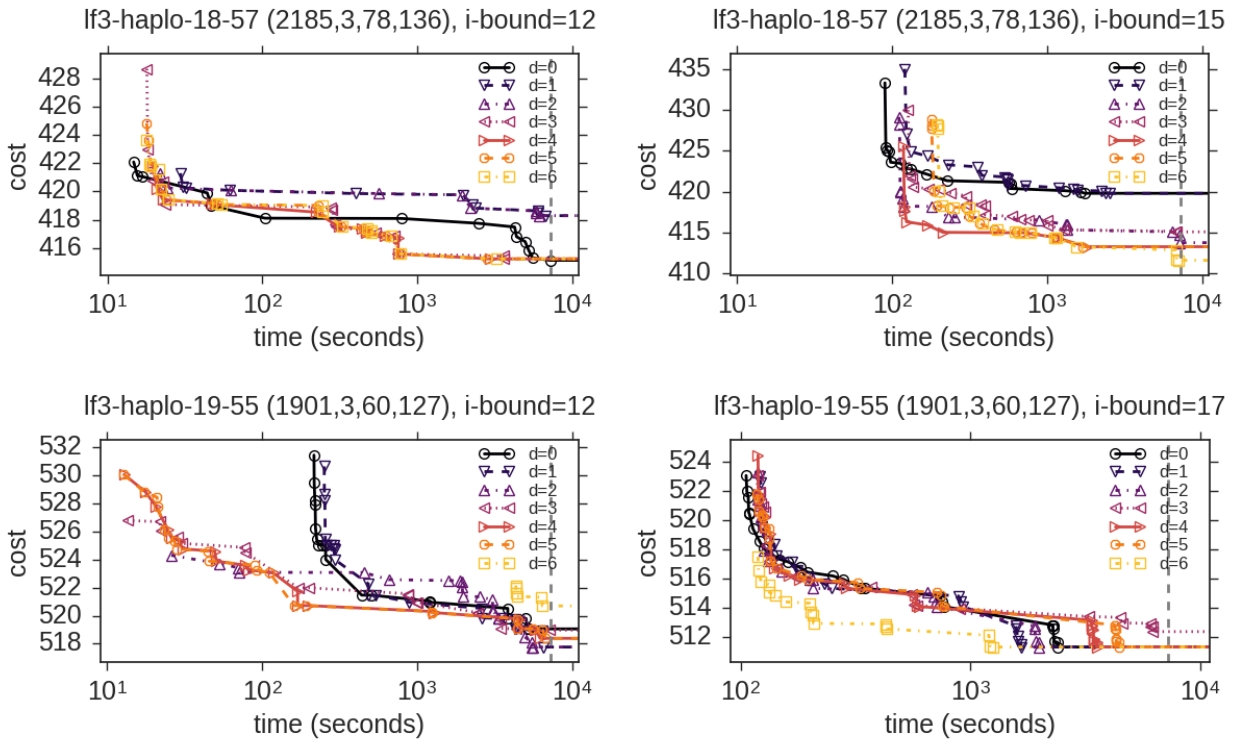


Figure 2.18: **largefam3** instances: Anytime plots across 2 different i-bounds for two selected instances. The tuple next to the problem instance name indicates (n: number of variables, k: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower early on is better.

baseline, with 6 instances where it improves and 4 instances where it is outperformed by the baseline. Moving forward in time at 1800 and 3600 seconds, look-ahead performs even better. Moving to 7200 seconds, the number of instances where look-ahead is superior decreases, but still maintains an advantage. Increasing the look-ahead depth to 5, the results are similar, but puts look-ahead at an advantage on more instances. However, we can also see that the baseline manages to outperform look-ahead on some instances where it was performing identically before. 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, indicating that there

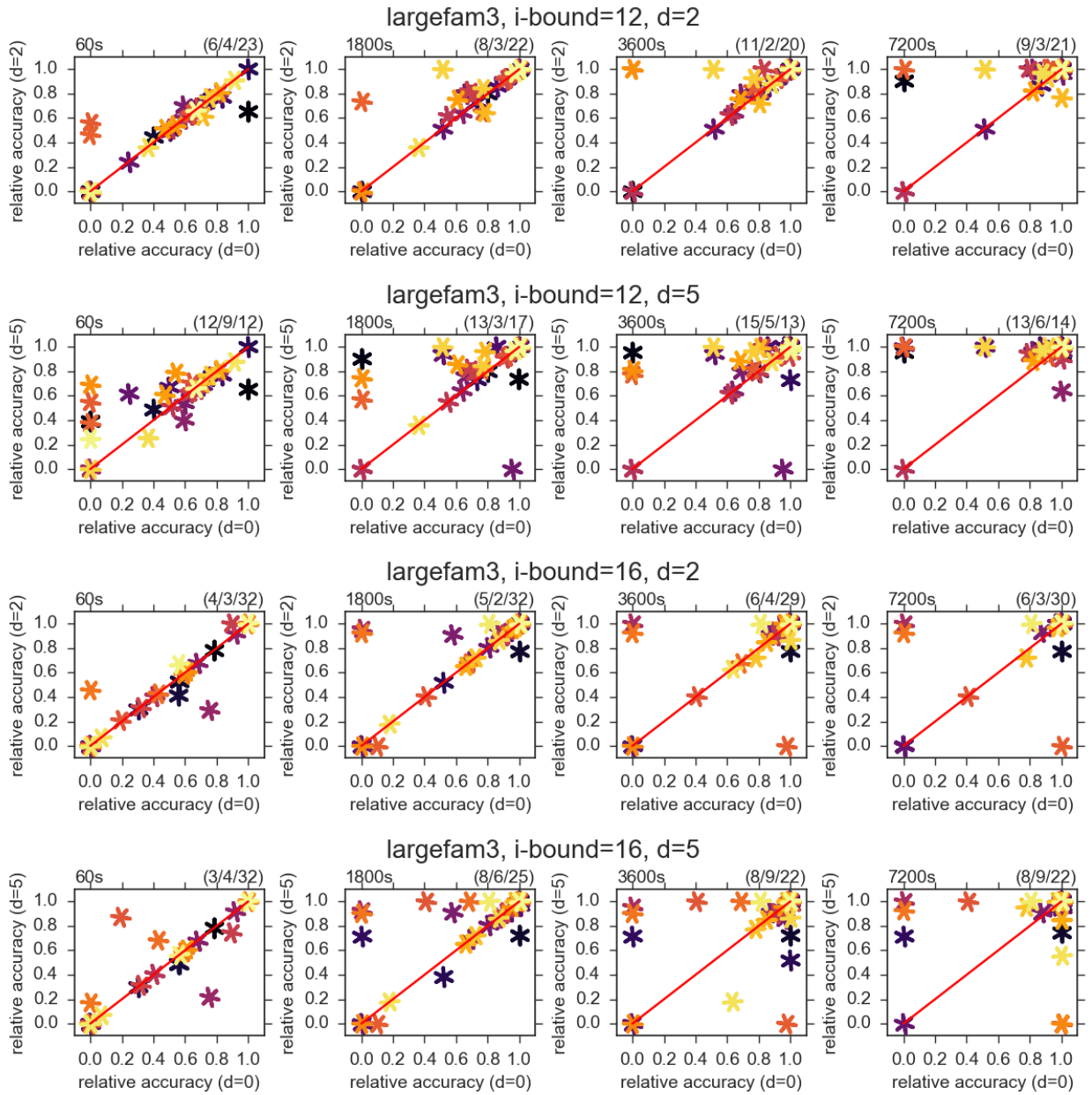


Figure 2.19: **largefam3.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular 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).

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

In summary, we observe on this benchmark that 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.

2.4.3.2 Type4

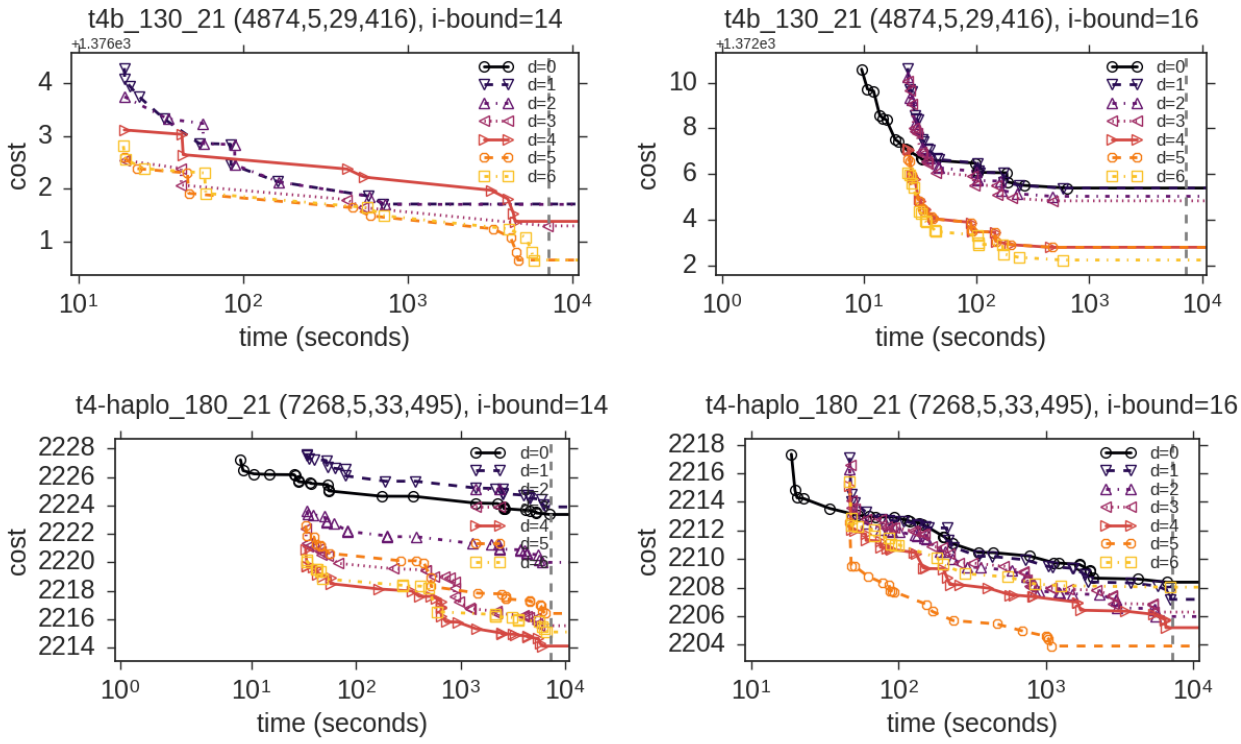


Figure 2.20: **type4** instances: Anytime plots across 2 different i-bounds for two selected instances. The tuple next to the problem instance name indicates (n: number of variables, k: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower early on is better.

In **Figure 2.20**, 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).

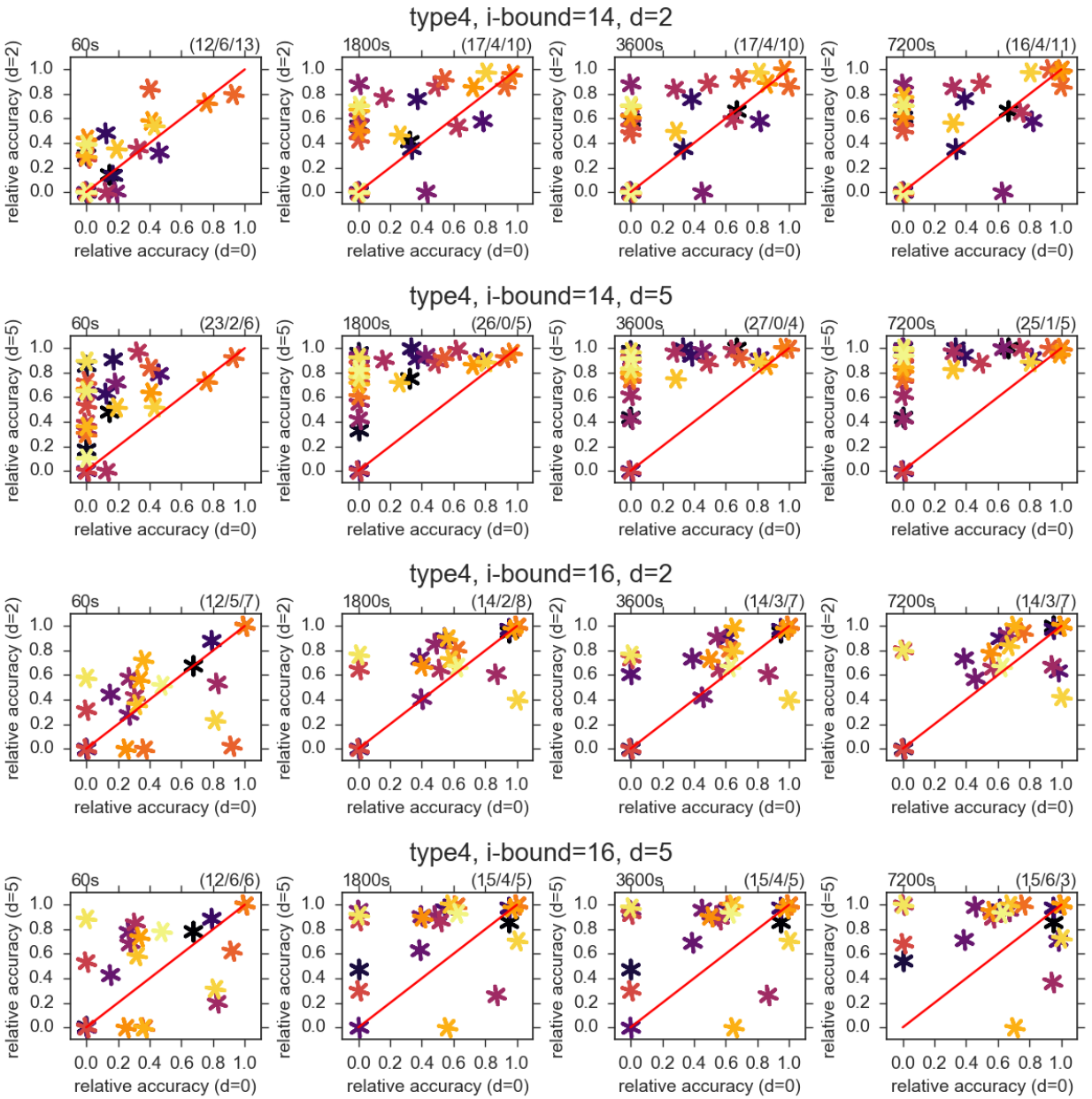


Figure 2.21: **type4.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular 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).

Comparing the look-ahead depths against each other, depths of 5 and 6 are superior, with 5 being slightly more cost-effective. 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 all look-aheads in under 100 seconds, with depths of 4 and higher producing considerably better solutions. For instances *t4-haplo_180_21* using an i-bound of 14, look-ahead with depths of 2 and higher outperforms the baseline for most time periods. Increasing the i-bound to 16, the baseline is quickest to produce a solution, but it is outperformed by all of the look-ahead depths. A depth of 5 yields the best solutions over the time period. Overall, look-ahead is usually superior to the baseline, with a bit of a preference for deeper depth regardless of heuristic strength.

In **Figure 2.21**, 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 the depth. Additionally, we see that there are a number of instances where look-ahead manages 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.

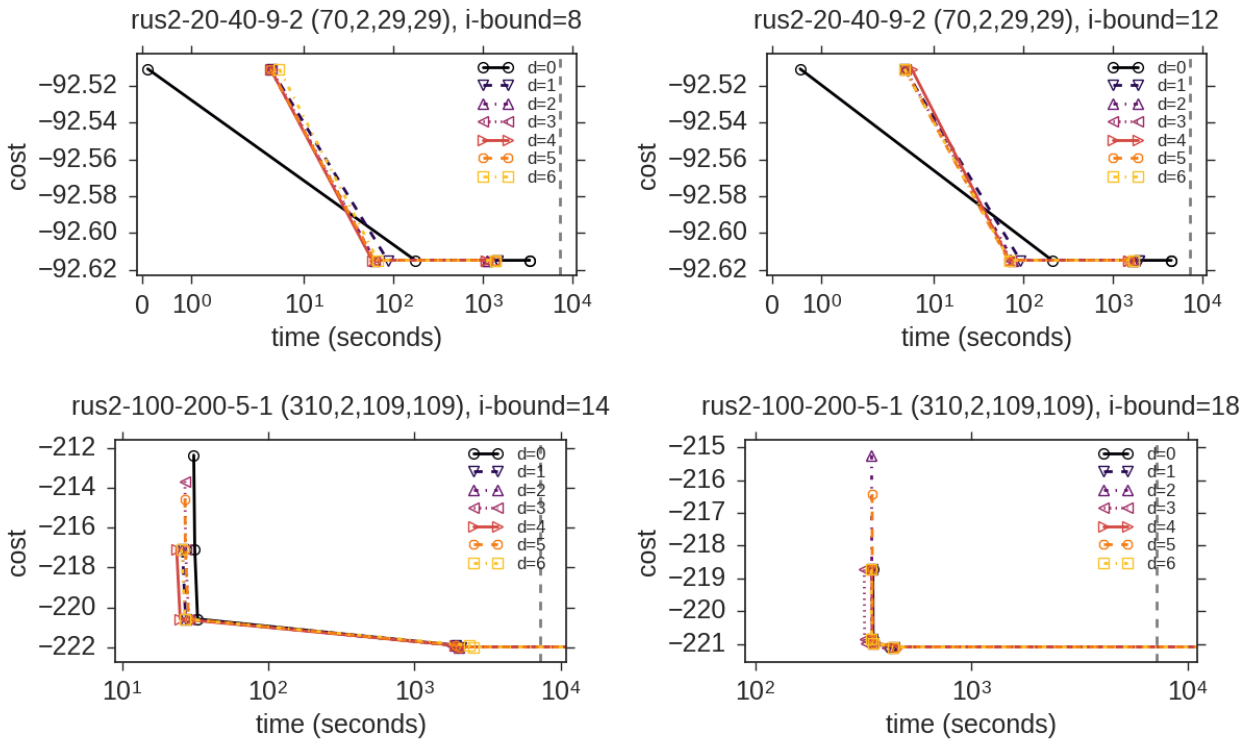


Figure 2.22: **DBN** instances: Anytime plots across 2 different i-bounds for two selected instances. The tuple next to the problem instance name indicates (n: number of variables, k: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower early on is better.

2.4.3.3 DBN

In **Figure 2.22**, 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 2.23**, we see that this behavior is systematic for this benchmark. Although we saw impressive speedups for look-ahead when finding exact solutions, we see here that the exact solution is actually obtained by both with a less significant margin of time between the two. For example, in the anytime plot for *rus-2-20-40-9-2* (**Figure 2.22**, 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.

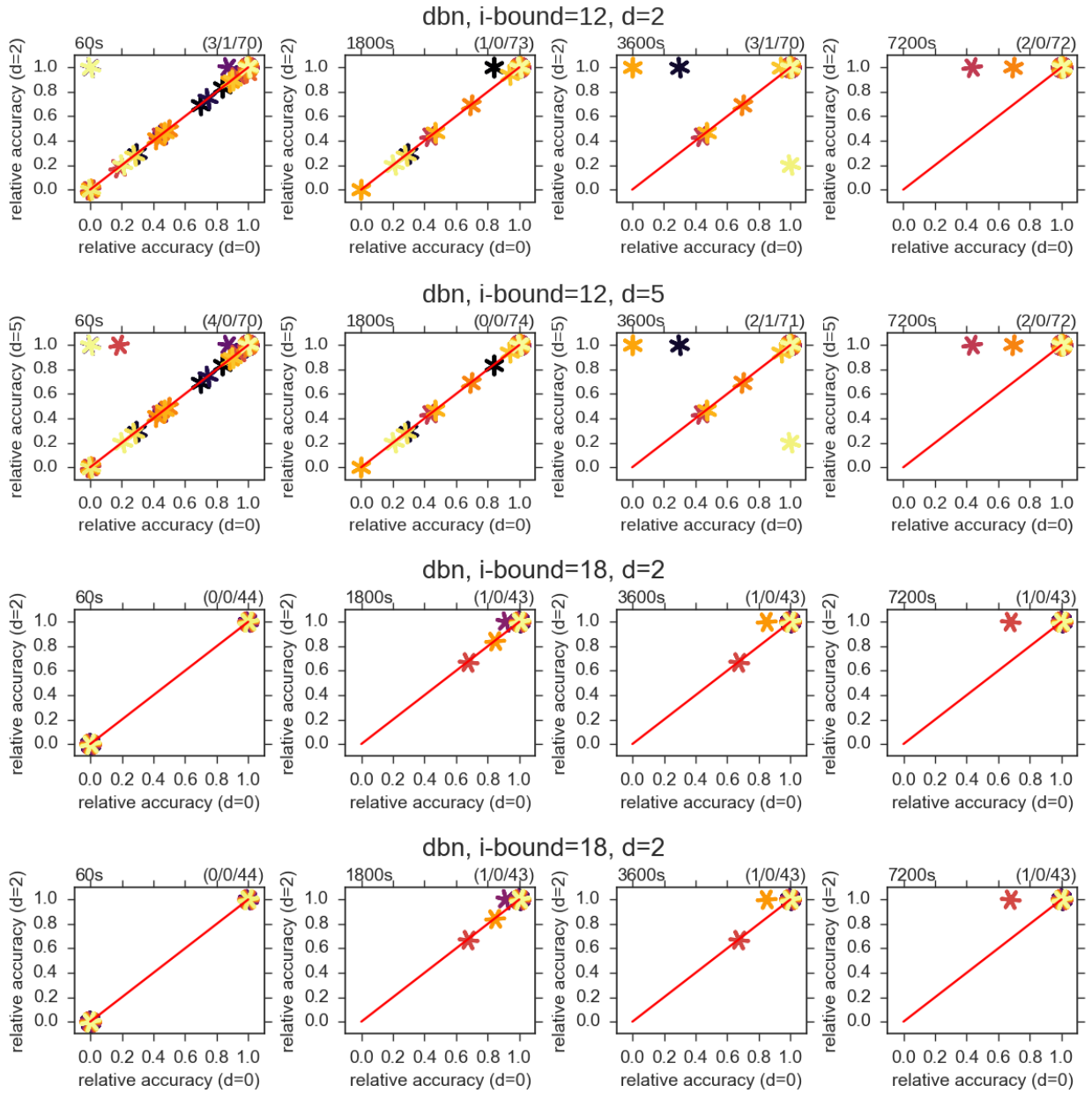


Figure 2.23: **DBN.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular i-bound/depth 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).

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.

2.4.3.4 Grids

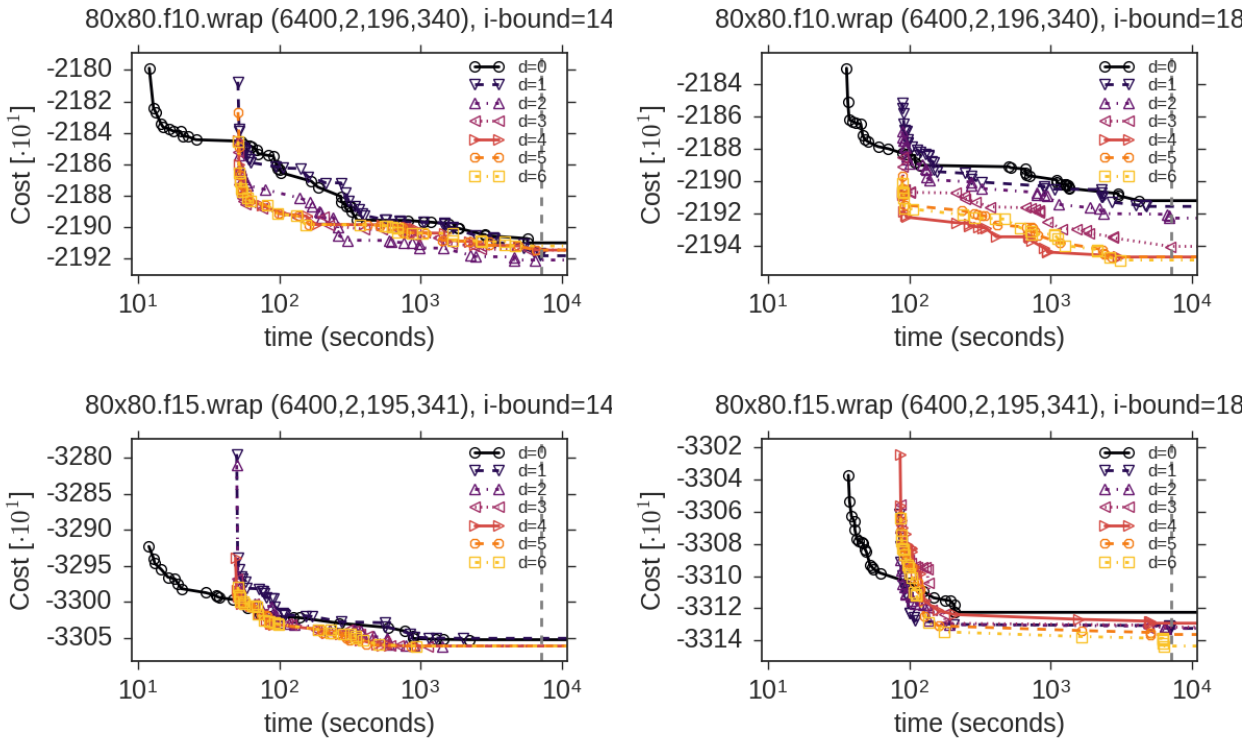


Figure 2.24: **grid** instances: Anytime plots across 2 different i-bounds for two selected instances. The tuple next to the problem instance name indicates (n: number of variables, k: maximum domain size, w: induced width, and h: pseudo-tree height). The solution cost is plotted against the time. The timeout is shown as a vertical dotted line; plots reaching past this line timed out. Lower early on is better.

In **Figure 2.24**, 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.

In **Figure 2.25**, 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.

2.4.3.5 Summary

In shifting our focus to hard instances where we could not evaluate for exact solution within the time bound, our takeaways from section 2.4.2 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

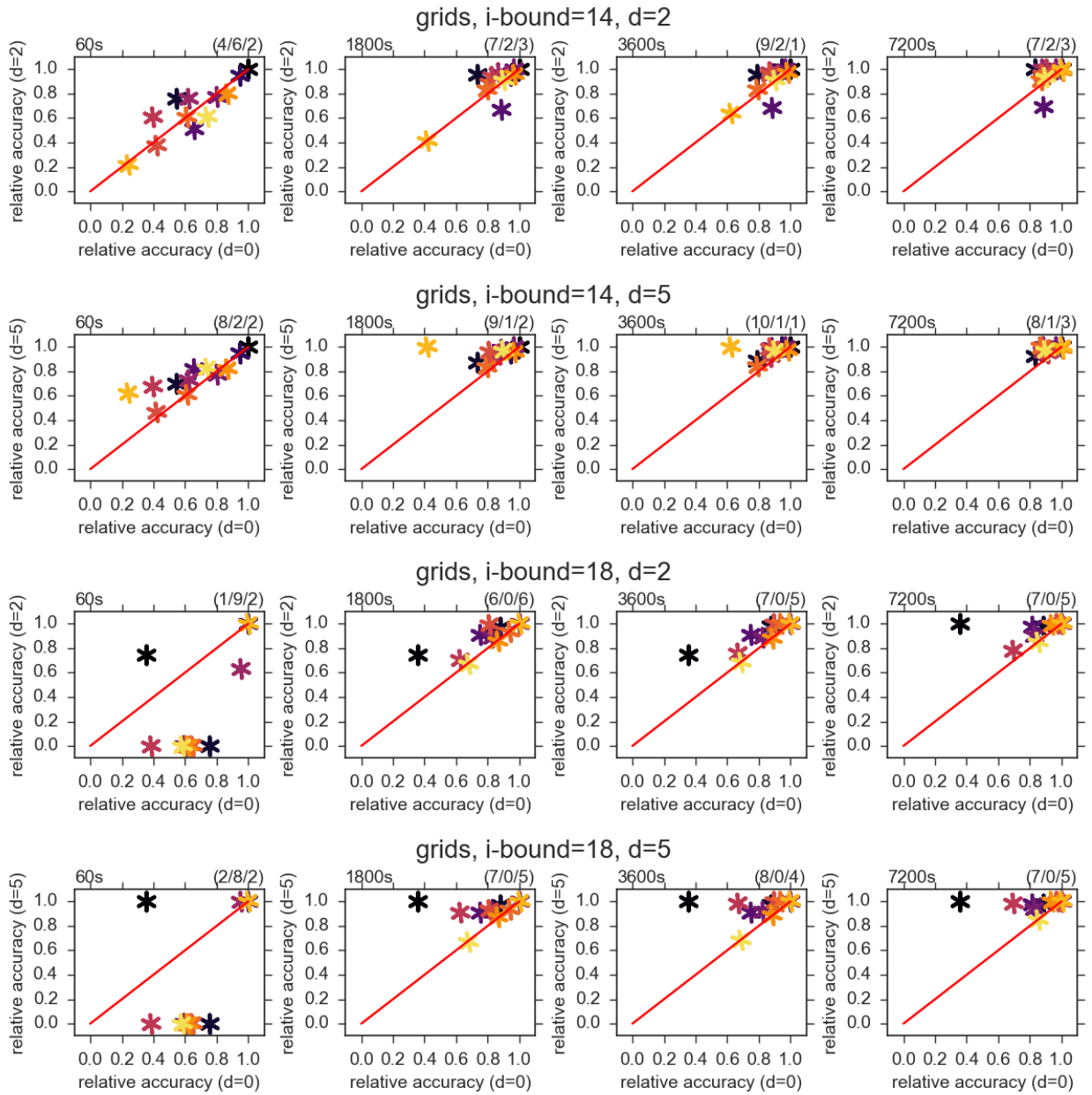


Figure 2.25: **grid.** Normalized relative accuracies for all instances in the benchmark across 2 different i-bounds and 2 different look-ahead depths. On the x-axis is the baseline and on the y-axis is the look-ahead algorithm with the specified depth. Each row of plots corresponds to a particular i-bound/depth 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).

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.

2.4.4 Impact of the ϵ Parameter: Subtree Pruning

All of the experiments in the previous two sections use a fixed ϵ of 0.01 for generating the ϵ-pruned look-ahead subtrees. In this section, we investigate the impact of our second control parameter of ϵ. We ran experiments on the **largefam3** and **grid** benchmarks using the full look-ahead subtree and pruned look-ahead subtrees with ϵ values of $0.01, 0.1, 1, 10, \text{and}, \infty$. We consider look-ahead depths of 2 and 5 in the following experiments in order to evaluate the ϵ parameter's impact on subtree pruning on full look-ahead subtrees of different size.

As discussed earlier in section 2.2.4, 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 and skipping computational overhead otherwise. Clearly, as $\epsilon \to \infty$, the look-ahead scheme reduces to the baseline. Thus, adjusting ϵ is an alternative way to controlling the computational trade-off of look-ahead. An important point to note is that the extreme cases of the full look-ahead subtree ($\epsilon = 0$) and no look-ahead ($\epsilon = \infty$) do not require of the preprocessing of bucket errors. Therefore, search will always begin earlier for those cases. This is especially relevant in an anytime setting as we show later.

In the following, we present plots similar to those seen in the previous 2 sections. All comparison based metrics such as speedup, node reduction, and solution quality are measured

relative the baseline using no look-ahead as before.

2.4.4.1 LargeFam3

instance (n,k,w^*,h)	(d,ϵ)	time	nodes	time	nodes	time	nodes
		i=14		i=16		i=18	
	(2,0)	3757 (1.06)	**464.21 (0.56)**	552 (1.17)	**75.95 (0.49)**	55 (1.02)	**5.97 (0.55)**
	(2,0.01)	3071 (1.29)	464.28 (0.56)	455 (1.41)	75.97 (0.49)	48 (1.16)	5.97 (0.55)
	(2,0.1)	3337 (1.19)	489.43 (0.60)	**452 (1.42)**	78.47 (0.51)	47 (1.19)	6.09 (0.56)
	(2,0.5)	3136 (1.27)	505.07 (0.61)	464 (1.39)	93.14 (0.60)	**45 (1.24)**	6.46 (0.59)
	(2,1)	**2995 (1.33)**	621.75 (0.76)	539 (1.19)	118.10 (0.76)	47 (1.17)	7.68 (0.70)
	(2,10)	4081 (0.97)	821.75 (1.00)	690 (0.93)	154.40 (1.00)	61 (0.91)	10.94 (1.00)
lf3-15-59	(5,0)	oot	-	1350 (0.48)	38.95 (0.25)	152 (0.37)	**3.09 (0.28)**
(1574,3,33,71)	(5,0.01)	4740 (0.84)	**219.70 (0.27)**	972 (0.66)	**38.95 (0.25)**	98 (0.57)	3.09 (0.28)
	(5,0.1)	5604 (0.71)	234.99 (0.29)	972 (0.66)	41.04 (0.27)	90 (0.62)	3.39 (0.31)
	(5,0.5)	4370 (0.91)	251.51 (0.31)	612 (1.05)	50.90 (0.33)	66 (0.84)	3.87 (0.35)
	(5,1)	4652 (0.85)	451.87 (0.55)	**605 (1.06)**	96.35 (0.62)	**48 (1.16)**	5.67 (0.52)
	(5,10)	**4215 (0.94)**	821.75 (1.00)	691 (0.93)	154.40 (1.00)	61 (0.91)	10.94 (1.00)
	(0,∞)	3971	821.75	644	154.40	56	10.94
		i=14		i=16		i=18	
	(2,0)	2151 (0.82)	**281.51 (0.77)**	422 (0.90)	**53.07 (0.68)**	109 (0.95)	**6.23 (0.67)**
	(2,0.01)	1862 (0.95)	281.86 (0.77)	**376 (1.01)**	53.07 (0.68)	**107 (0.97)**	6.28 (0.67)
	(2,0.1)	1951 (0.90)	283.88 (0.77)	383 (1.00)	53.17 (0.68)	108 (0.96)	6.30 (0.68)
	(2,0.5)	1888 (0.93)	285.78 (0.78)	413 (0.92)	63.32 (0.81)	116 (0.89)	8.33 (0.90)
	(2,1)	2149 (0.82)	349.75 (0.95)	437 (0.87)	74.00 (0.95)	124 (0.84)	10.90 (1.17)
	(2,10)	**1842 (0.96)**	367.61 (1.00)	408 (0.93)	77.79 (1.00)	112 (0.93)	9.30 (1.00)
lf3-16-56	(5,0)	3083 (0.57)	104.02 (0.28)	656 (0.58)	**20.76 (0.27)**	138 (0.75)	**2.30 (0.25)**
(1688,3,38,77)	(5,0.01)	2183 (0.81)	104.73 (0.28)	513 (0.74)	20.77 (0.27)	115 (0.91)	2.31 (0.25)
	(5,0.1)	2000 (0.88)	**103.89 (0.28)**	524 (0.73)	20.80 (0.27)	115 (0.91)	2.31 (0.25)
	(5,0.5)	2126 (0.83)	107.91 (0.29)	457 (0.83)	23.59 (0.30)	116 (0.90)	2.99 (0.32)
	(5,1)	2241 (0.79)	144.19 (0.39)	484 (0.79)	39.06 (0.50)	136 (0.76)	7.79 (0.84)
	(5,10)	**1835 (0.96)**	367.61 (1.00)	**413 (0.92)**	77.79 (1.00)	**111 (0.93)**	9.30 (1.00)
	(0,∞)	1760	367.61	381	77.79	104	9.30
		i=12		i=14		i=16	
	(2,0)	1688 (0.82)	161.52 (0.61)	565 (0.84)	**53.59 (0.57)**	**20 (0.98)**	1.12 (0.49)
	(2,0.01)	**1212 (1.14)**	**161.49 (0.61)**	436 (1.09)	53.60 (0.57)	21 (0.92)	1.12 (0.49)
	(2,0.1)	1352 (1.03)	162.99 (0.62)	470 (1.01)	62.15 (0.66)	21 (0.95)	**1.12 (0.49)**
	(2,0.5)	1292 (1.07)	180.75 (0.69)	**434 (1.10)**	66.46 (0.71)	21 (0.93)	1.41 (0.62)
	(2,1)	1336 (1.04)	202.73 (0.77)	454 (1.05)	74.18 (0.79)	21 (0.93)	1.42 (0.62)
	(2,10)	1501 (0.92)	263.23 (1.00)	469 (1.01)	93.62 (1.00)	24 (0.83)	2.28 (1.00)
lf3-17-58	(5,0)	2555 (0.54)	33.84 (0.13)	2905 (0.16)	23.37 (0.25)	40 (0.50)	0.41 (0.18)
(1712,3,31,75)	(5,0.01)	2129 (0.65)	**33.84 (0.13)**	2770 (0.17)	**23.36 (0.25)**	36 (0.54)	**0.41 (0.18)**
	(5,0.1)	1849 (0.75)	34.08 (0.13)	1966 (0.24)	30.27 (0.32)	34 (0.58)	0.42 (0.19)
	(5,0.5)	3629 (0.38)	87.07 (0.33)	1323 (0.36)	34.66 (0.37)	39 (0.50)	0.79 (0.35)
	(5,1)	2294 (0.60)	106.18 (0.40)	1473 (0.32)	51.92 (0.55)	34 (0.57)	0.89 (0.39)
	(5,10)	**1451 (0.96)**	262.29 (1.00)	**512 (0.93)**	93.62 (1.00)	**24 (0.83)**	2.28 (1.00)
	(0,∞)	1386	263.46	476	93.62	20	2.28

Table 2.9: Selected **largefam3** instances for look-ahead depths of 2 and 5. Each row corresponds to a different depth, ϵ pair. The best time and nodes within for each depth are bolded and the best overall time and nodes are boxed. All speedups and reduction are calculated relative to the baseline using no look-ahead $(0,\infty)$.

Table 2.9 shows the impact of the ϵ parameter on look-ahead depths of 2 and 5 on 3 representative instances from the **LargeFam3** benchmark. To account for the entire benchmark,

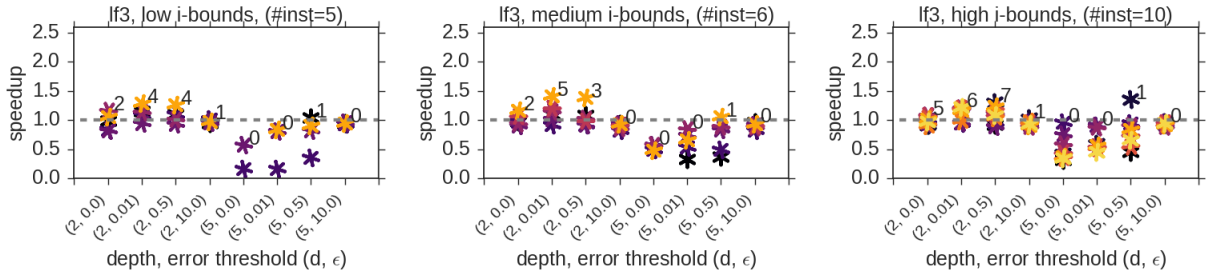


Figure 2.26: Speedups on all solved **largefam3** instances relative to no-lookahead. The information shown is similar to those in section 2.4.2, with both varying depth and ϵ.

we also plot speedups over the baseline for each solved instances in **Figure 2.26**. In this plot, we only show ϵ for 0, 0.01, 0.5, and 10. The first four of each plot correspond to a depth of 2 while the last four are for a depth of 5. In nearly all cases, subtree pruning with any $\epsilon > 0$ has a positive impact on the runtime compared with no pruning ($\epsilon = 0$). As expected, since pruning reduces look-ahead, the number of nodes expanded increases as a function of ϵ in most cases. Looking into the lower i-bounds with a look-ahead depth of 2, ϵ values of 0.01 and 0.5 had the best performance amongst the instances. However, most of the speedup can be attributed by moving to an ϵ of 0.01, which prunes away the error-free variables from look-ahead subtrees. In particular, the speedup increases the most between $\epsilon = 0$ and $\epsilon = 0.01$ in many cases. Under higher i-bounds, the impact is minimal overall since the errors are small. At a depth of 5, higher ϵ is preferred, since the full look-ahead subtrees correspond to expensive look-ahead computation. For example, for *lf3-16-56* and *lf3-17-58*, performing no look-ahead (0,∞) is preferred over a depth 5 look-ahead for all i-bounds here on nearly all instances. In the summary plot, almost no instances benefit at depth 5, which is in contrast to depth 2 where the speedup is over 1.0 for most ϵ. Finally, on this benchmark, since most bucket errors are less than 10, the heuristic strength when $\epsilon = 10$ and when $\epsilon = \infty$ is nearly identical, which we can conclude based on the number of nodes between these two being nearly equivalent in all cases. However, since we do not need to compute bucket errors when $\epsilon = \infty$, it has less preprocessing overhead and therefore performs better on runtime. In summary, a depth of 2 with $\epsilon = 0.01$ or 0.5 was overall best

for evaluating exact solutions on this benchmark, which was also found to be the best in section 2.4.2.

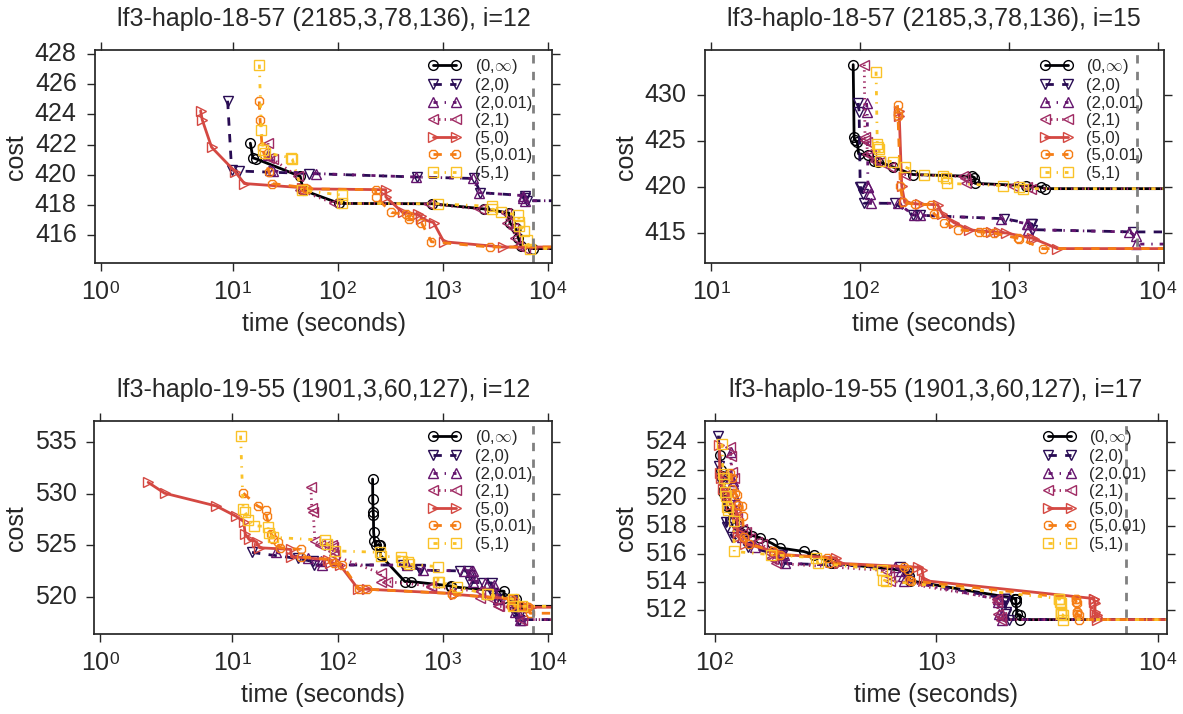


Figure 2.27: **largefam3** instances: Solutions obtained over time for different ϵ thresholds with look-ahead depths of 2 and 5. The information shown is similar to figures presented in Section 2.4.3.

Figure 2.27 shows the anytime performance for 2 representative instances from this benchmark under 2 i-bounds. In all cases, the highest quality solutions tend to be obtained by the full look-ahead subtree ($\epsilon = 0$), especially near the start. For both instances at the lower i-bound, the combination of a depth of 5 and $\epsilon = 0$ produces the best solutions early on and maintains a lead over the other settings for most of the time period. One of the contributing factors to its superior early performance is due to the lack of need to pre-process any bucket errors as mentioned earlier. Still since the strongest look-ahead settings here are best, it reinforces our earlier observations from section 2.4.3. At the higher i-bounds, the best setting is less clear as it varies over time. In *lf3-haplo-18-57* under an i-bound of 15, although the baseline obtains the first solution, the combination of a depth of 2 and $\epsilon = 0$

obtains a much better solution shortly after. Considering a look-ahead depth of 5 on the same instance, the behavior is similar, except that it takes longer for $\epsilon \leq 0.01$ to obtain the same higher quality solutions. There is less variation in the other instance (*lf3-haplo-19-55*) shown. Both $\epsilon = 0$ and $\epsilon = \text{inf}$ obtain solutions earlier due to the lack of need to pre-process bucket errors. The variation is not very significant until the end, where we see that higher ϵ is preferred for both depths.

Figure 2.28 demonstrates the impact of the ϵ parameter on the anytime performance over all of the instances. First, at the 60 second time bound, the difference between ϵ at 0 and 0.01 is small. Performance degrades as ϵ increases, reinforcing that more look-ahead is better for an anytime setting. In terms of the number wins/losses/ties, a depth of 2 with $\epsilon = 0$ is the top performer, given that it had 5 wins and 3 losses, which is the largest number of wins and best win-loss ratio amongst all of the settings. Moving to a 3600 second time bound, for a depth of 2, as ϵ increases, the number of losses decreases without much change in the number of wins, thus providing improving its win-loss ratio. In contrast, for a depth of 5, which has the largest number of wins at $\epsilon = 0$, the win-loss ratio remain negative with increasing ϵ. Thus, a depth of 2 with $\epsilon = 0.5$ is best overall at this time bound.

2.4.4.2 Grids

In **Table 2.10**, we see similar behavior to what we saw with the **LargeFam3** benchmark. Here too, the ϵ parameter has small impact at a look-ahead depth of 2, while the impact is larger at a depth of 5. Much like the previous benchmark, most of the speedup is attributed to moving to $\epsilon = 0.01$ for most of the instances. However, more pruning is not always preferred at a depth of 5. For example, on *grid20x20.f5.wrap*, an ϵ of 0.5 yields the best performance, with higher ϵ resulting in more nodes being expanded. Also, *grid40x40.f2* is an example where a higher i-bound can lead to a weaker heuristic. For both depths, we see that $\epsilon = 0.01$ is best here in these cases under an i-bound of 20, with depth 2 being the overall

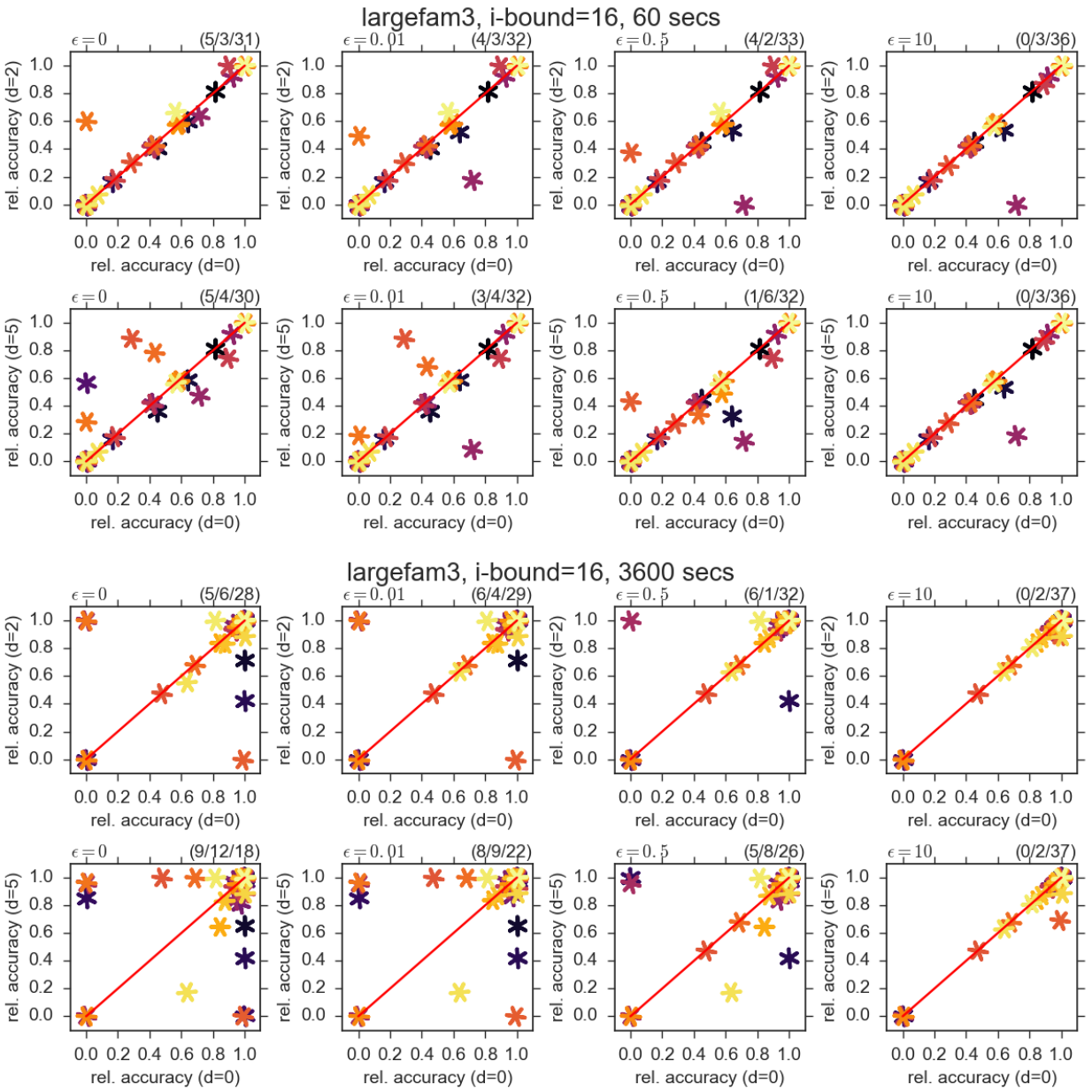


Figure 2.28: **largefam3.** Normalized relative accuracies for all instances in the benchmark. Each 2x4 set of scatter plots corresponds to a different time bound. Within each set, the depth varies along the rows and ϵ varies along the columns.

instance (n,k,w^*,h)	ϵ	time	nodes	time	nodes	time	nodes
		i=12		i=14		i=16	
	(2,0)	2536 (1.01)	402.01 (0.67)	1733 (1.08)	**276.33 (0.64)**	**15 (0.91)**	**2.42 (0.75)**
	(2,0.01)	**2308 (1.11)**	**402.01 (0.67)**	**1559 (1.20)**	276.33 (0.64)	17 (0.81)	2.42 (0.75)
	(2,0.1)	2408 (1.07)	404.57 (0.68)	1581 (1.19)	277.85 (0.64)	17 (0.83)	2.44 (0.76)
	(2,0.5)	2526 (1.02)	487.67 (0.82)	1813 (1.03)	391.17 (0.90)	16 (0.85)	2.69 (0.84)
	(2,1)	2734 (0.94)	577.27 (0.96)	1975 (0.95)	422.48 (0.98)	18 (0.78)	3.21 (1.00)
	(2,10)	2739 (0.94)	598.29 (1.00)	1957 (0.96)	433.11 (1.00)	18 (0.76)	3.22 (1.00)
grid20x20.f10.wrap	(5,0)	3509 (0.73)	232.46 (0.39)	2815 (0.67)	**175.57 (0.41)**	30 (0.46)	**1.78 (0.55)**
(400,2,44,68)	(5,0.01)	**2689 (0.95)**	**232.46 (0.39)**	2127 (0.88)	175.57 (0.41)	26 (0.54)	1.78 (0.55)
	(5,0.1)	2795 (0.92)	234.99 (0.39)	2302 (0.81)	179.01 (0.41)	24 (0.57)	1.80 (0.56)
	(5,0.5)	2875 (0.89)	310.18 (0.52)	2395 (0.78)	296.25 (0.68)	22 (0.61)	2.27 (0.70)
	(5,1)	2880 (0.89)	408.39 (0.68)	2251 (0.83)	347.10 (0.80)	19 (0.73)	3.13 (0.97)
	(5,10)	2731 (0.94)	598.29 (1.00)	**1987 (0.94)**	433.11 (1.00)	**18 (0.78)**	3.22 (1.00)
	(0,∞)	2566	598.29	1876	433.11	14	3.22
		i=10		i=12		i=14	
	(2,0)	148 (1.00)	23.77 (0.71)	139 (0.91)	24.01 (0.75)	95 (0.99)	**16.58 (0.68)**
	(2,0.01)	136 (1.09)	23.77 (0.71)	**126 (1.00)**	**24.01 (0.75)**	86 (1.10)	16.59 (0.68)
	(2,0.1)	**131 (1.13)**	**23.45 (0.70)**	128 (0.98)	24.09 (0.75)	87 (1.09)	16.61 (0.68)
	(2,0.5)	137 (1.08)	27.08 (0.81)	129 (0.98)	26.67 (0.83)	**83 (1.15)**	18.05 (0.74)
	(2,1)	150 (0.99)	31.26 (0.93)	138 (0.91)	31.09 (0.97)	93 (1.02)	20.89 (0.86)
	(2,10)	154 (0.97)	33.63 (1.00)	134 (0.94)	32.05 (1.00)	104 (0.92)	24.37 (1.00)
grid20x20.f5.wrap	(5,0)	236 (0.63)	15.34 (0.46)	113 (1.11)	10.30 (0.32)	120 (0.79)	**9.74 (0.40)**
(400,2,45,69)	(5,0.01)	190 (0.78)	**15.34 (0.46)**	95 (1.32)	**10.30 (0.32)**	95 (1.00)	9.74 (0.40)
	(5,0.1)	198 (0.75)	15.82 (0.47)	100 (1.25)	10.32 (0.32)	95 (1.00)	9.76 (0.40)
	(5,0.5)	174 (0.85)	18.53 (0.55)	**92 (1.37)**	12.19 (0.38)	**72 (1.31)**	11.71 (0.48)
	(5,1)	170 (0.87)	23.81 (0.71)	123 (1.03)	19.99 (0.62)	81 (1.17)	14.25 (0.58)
	(5,10)	**151 (0.98)**	33.63 (1.00)	135 (0.94)	32.05 (1.00)	104 (0.91)	24.37 (1.00)
	(0,∞)	148	33.63	126	32.05	95	24.37
		i=16		i=18		i=20	
	(2,0)	6031 (0.82)	**844.36 (0.89)**	409 (0.91)	**57.34 (0.83)**	1028 (1.14)	**135.69 (0.63)**
	(2,0.01)	5049 (0.98)	845.76 (0.89)	369 (1.01)	57.45 (0.83)	**859 (1.37)**	135.76 (0.63)
	(2,0.1)	4971 (0.99)	854.61 (0.90)	368 (1.01)	59.63 (0.87)	1185 (0.99)	189.89 (0.89)
	(2,0.5)	**4883 (1.01)**	895.64 (0.94)	**359 (1.04)**	61.40 (0.89)	1173 (1.00)	203.87 (0.95)
	(2,1)	5107 (0.96)	942.09 (0.99)	400 (0.93)	68.61 (1.00)	1177 (1.00)	213.90 (1.00)
	(2,10)	5156 (0.96)	947.79 (1.00)	406 (0.92)	68.89 (1.00)	1236 (0.95)	213.90 (1.00)
grid40x40.f2	(5,0)	oot	-	870 (0.43)	**51.94 (0.75)**	1934 (0.61)	**105.02 (0.49)**
(1600,2,52,157)	(5,0.01)	6389 (0.77)	**554.56 (0.59)**	567 (0.66)	52.05 (0.76)	**1132 (1.04)**	105.15 (0.49)
	(5,0.1)	6611 (0.74)	682.91 (0.72)	522 (0.71)	53.94 (0.78)	1540 (0.76)	159.76 (0.75)
	(5,0.5)	5674 (0.87)	833.52 (0.88)	**381 (0.98)**	58.24 (0.85)	1158 (1.02)	194.83 (0.91)
	(5,1)	5507 (0.89)	916.74 (0.97)	413 (0.90)	69.84 (1.01)	1229 (0.96)	213.90 (1.00)
	(5,10)	**5134 (0.96)**	947.79 (1.00)	396 (0.94)	68.89 (1.00)	1226 (0.96)	213.90 (1.00)
	(0,∞)	4924	947.79	373	68.89	1177	213.90

Table 2.10: Selected **grid** instances for look-ahead depths of 2 and 5. Each row corresponds to a different depth, ϵ pair. The best time and nodes within for each depth are bolded and the best overall time and nodes are boxed. All speedups and reduction are calculated relative to the baseline using no look-ahead (0,∞).

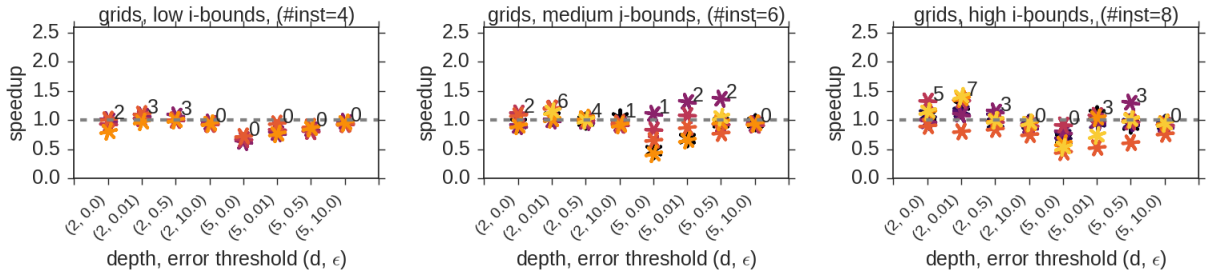


Figure 2.29: Speedups on all solved **grid** instances relative to no-lookahead. The information shown is similar to those in section 2.4.2, with both varying depth and ϵ.

best. Higher levels of ϵ result in considerably worse heuristics here, thus demonstrating look-ahead's impact on improving the heuristic.

Figure 2.29 shows speedups for all solved instances in this benchmark. For low i-bounds, we see minor improvements on nearly all of the solvable instances only for a depth of 2. The impact of ϵ here is also small. Moving to higher i-bounds, we see that increasing ϵ helps when using a depth of 5, though this is unable to outperform the performance using a depth of 2. At a depth of 2, increasing ϵ over 0.01 decreases the overall performance, thus the combination of a depth of 2 with an ϵ of 0.01 was overall the best for all i-bounds on this benchmark for evaluating exact solutions.

Figure 2.30 shows the anytime performance for various ϵ on 2 instances of this benchmark under 2 i-bounds. At an i-bound of 14, a depth of 5 with $\epsilon = 0$ is best at the start for both instances. However, for *80x80.f10.wrap*, a depth of 2 with $\epsilon <= 0.01$ take the lead. For both depths, increasing ϵ to 1 results in worse performance compared with lower ϵ and also the baseline over certain time points. Moving to the highest i-bound of 18, a depth of 5 with $\epsilon <= 0.01$ is generally the best. On *80x80.f10.wrap*, $\epsilon = 0$ produces a solution before $\epsilon = 0.01$ due to the preprocessing overhead of the latter. However, $\epsilon = 0.01$ clearly performs the best thereafter. The same behavior is observed on the other instance, but the difference is less significant. Indeed this is systematic over the benchmark as seen in **Figure 2.31** summarizing over all instances. At a 60 second time bound, we see that $\epsilon = 0$ performs the

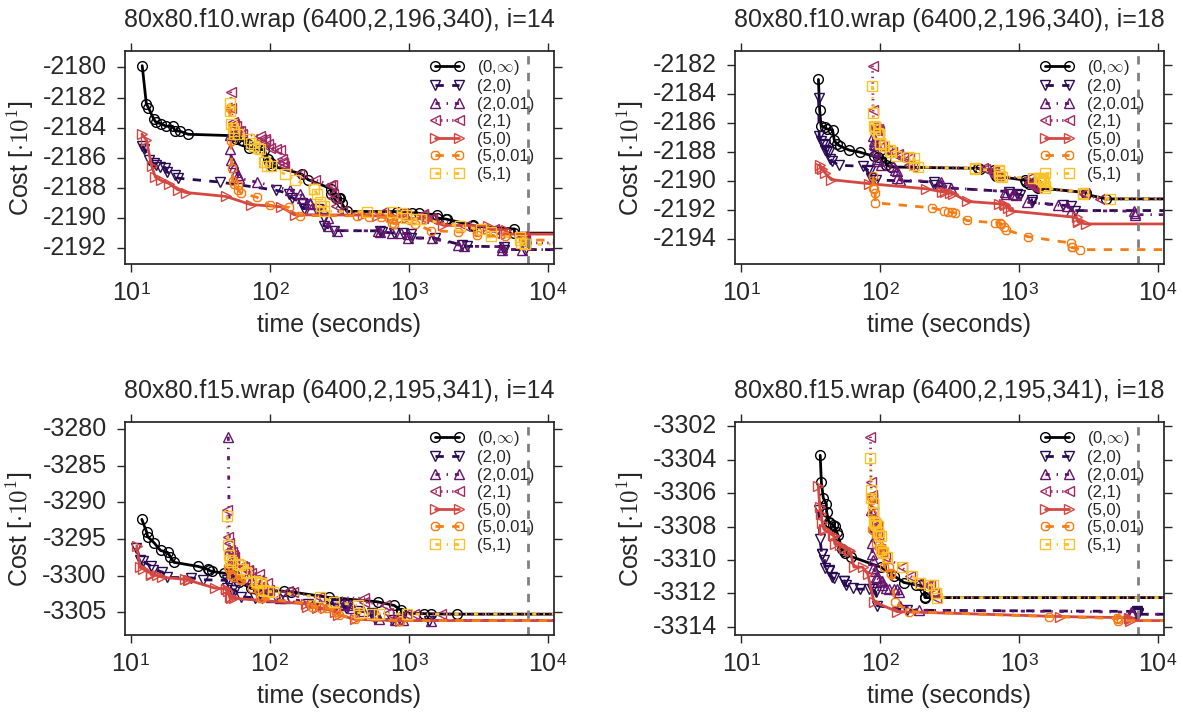


Figure 2.30: **grid** instances: Solutions obtained over time for different ϵ thresholds with look-ahead depths of 2 and 5. The information shown is similar to figures presented in Section 2.4.3.

best at both depths. The best setting here is a depth of 2 with $\epsilon = 0$, having 9 wins and 1 loss over the baseline. However, for $\epsilon = 0.01$ the result is reversed for both depths. Since the losses here are all instances where the look-ahead scheme has zero relative accuracy, this means that a single solution was not yet generated. This is because the preprocessing step in bucket errors for subtree pruning is still in progress at 60 seconds. Moving to a 3600 second time bound, $\epsilon = 0.01$ has nearly identical performance as $\epsilon = 0$, now that preprocessing has been long completed. For either time bound, like on the previous benchmark, higher ϵ severely degrades the anytime performance compared with stronger look-ahead by making its performance approach the baseline.

In summary, this benchmark is more difficult and thus also results in more expensive preprocessing time for the bucket errors. As a result, $\epsilon = 0$ tended to be the best for both depths. Overall, comparing the performance between both depths with $\epsilon = 0$, the performance was

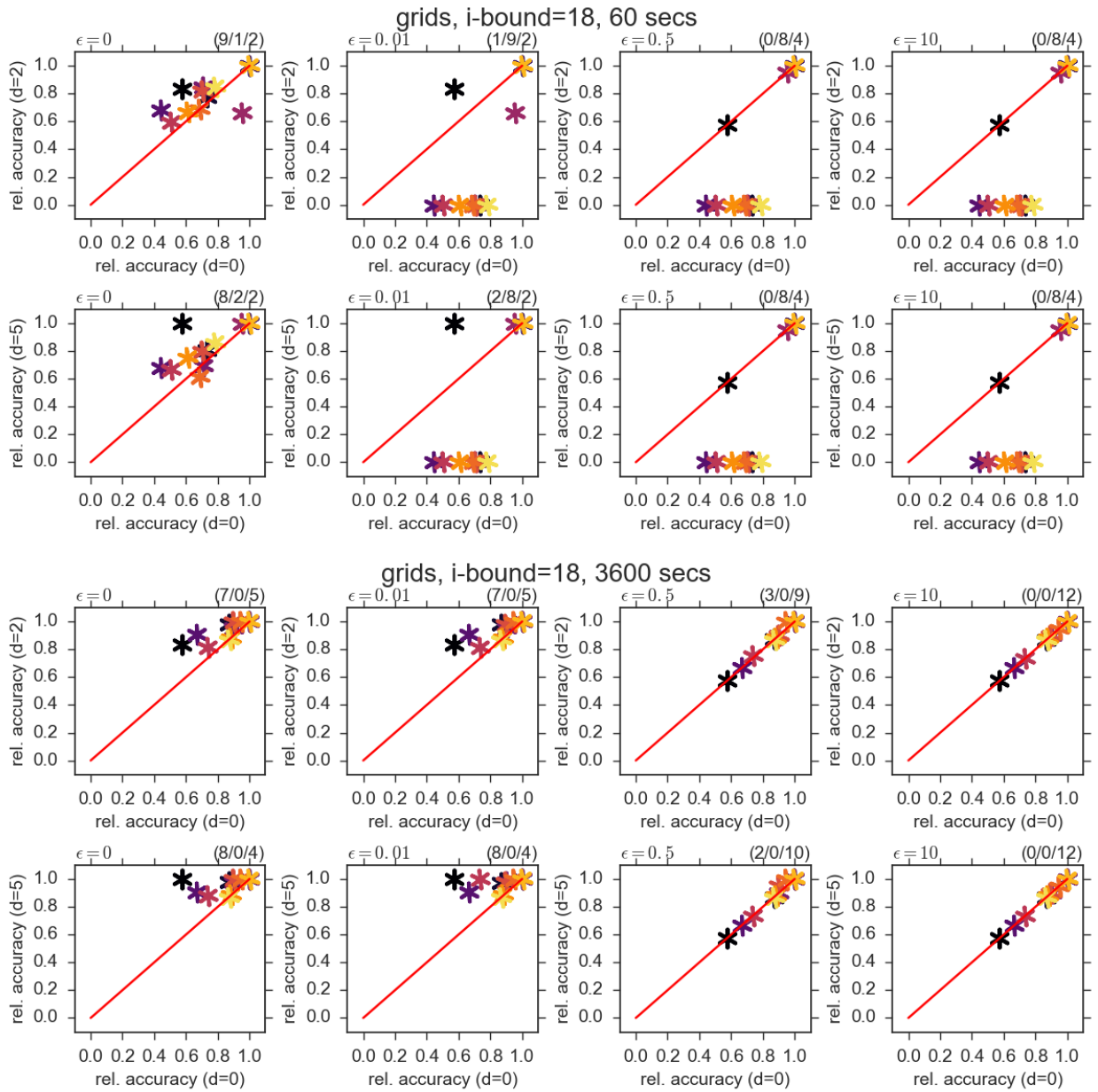


Figure 2.31: **grid.** Normalized relative accuracies for all instances in the benchmark. Each 2x4 set of scatter plots corresponds to a different time bound. Within each set, the depth varies along the rows and ϵ varies along the columns.

similar.

2.4.4.3 Summary

We explored the ϵ parameter in conjunction with the depth parameter in this section. For the purposes *evaluating for exact solutions*, we found that a depth of 2 with $\epsilon = 0.01$ was overall the best. This often gave the best balance of look-ahead power while pruning away only parts of the look-ahead subtree that were unlikely to yield benefits, which we observe in various cases where moving from 0 to 0.01 results in the largest positive change in the speedup. Although, increasing ϵ further helped in some cases, it also tended to weaken look-ahead to the point where the behavior was close to the baseline with no look-ahead. On harder instances where we consider the *anytime behavior*, subtree pruning is less important, especially if we consider a very low time bound. In particular, the overhead of preprocessing for bucket errors can cause any look-ahead scheme with a non-zero ϵ to start search later, thus not generating any solutions in the meantime. Even then, when considering higher time bounds, the quality of solutions obtained between the two do not differ much. On the two benchmarks explored in this section, neither depth 2 or 5 were significantly dominant over the other, but from our earlier results on other benchmarks in section 2.4.3 which all used $\epsilon = 0.01$, stronger look-ahead tended to benefit.

2.5 Conclusion

We provided a framework for performing look-ahead in the context of AND/OR search for graphical models. We analyzed the residuals and introduced a way to approximate them, showing that we can estimate the impact of any d-level look-ahead with local information. Specializing to the MBE heuristic, we introduced the notion of local bucket errors which correspond to local residuals. Using this information, we developed a scheme to control

where look-ahead should be performed in the search space such that it is cost-effective. In our evaluation for exact solving, we found that look-ahead allows us to solve instances more quickly given that we use it sparingly by restricting to a low depth, specifically a depth of 2-3. For our evaluation on anytime performance focusing on hard instances, look-ahead manages to improve the anytime performance, with higher depths performing best in many cases. To generalize, we observed look-ahead is more helpful as the difference between the i-bound and induced width increases. Therefore, depth should be increased to promote strong look-ahead on hard problems, specifically with depths greater than 4. In our exploration of the secondary control parameter of ϵ, we generally found that it should be minimized ($\epsilon = 0.01$) for finding exact solutions. The combination of this with a lower depth of 2 was found to be the best overall. On the other hand, for anytime behavior, there was little difference between using $\epsilon = 0$ and $\epsilon = 0.01$ due to both managing to find the highest quality solutions in nearly the same amount of time. Though it was not clear whether depth 2 or 5 was best here based on the 2 benchmarks we evaluated in that section, strong look-ahead was still shown to be important. This suggests that higher depths with near-zero ϵ are the best for anytime performance. Additionally, care should be taken when the time bound is low, as the preprocessing overhead of look-ahead subtree pruning may exceed the time bound.

A secondary contribution of this paper is in identify a more efficient method of computing look-ahead by using bucket elimination. From this alone, we know that look-ahead can improve the heuristic without increasing the memory usage as long as the look-ahead width is lower than the i-bound.

In future work, ϵ should be tuned to a problem instance automatically, since its impact heavily depends on the distribution of bucket errors on a particular instance. we can also consider finer-grained controls in controlling look-ahead to further improve its cost-effectiveness. In particular, we observed that while deep look-ahead manages to find the best anytime solutions, it tended to be less cost effective when running search to termination to prove the

optimality of an obtained solution. Thus, this suggests a dynamic scheme that performs look-ahead intensively at the beginning, but dials back on it as time passes. Also, the look-ahead subtrees are currently used in a static way for each variable, even though errors can vary depending on the instantiation. Thus, a context-dependent method that can vary the look-ahead subtrees based on the current instantiation could be helpful.

Another direction that comes from our framing of the look-ahead computation as bucket elimination is to use the *look-ahead width* as a control parameter for look-ahead rather than depth. As the look-ahead computation is bound by the look-ahead width, it would be highly beneficial to perform look-ahead in cases where the look-ahead width is much smaller than its depth, thus giving us the benefits of deep look-ahead without incurring heavy computational cost.

Chapter 3

Subproblem Ordering Heuristics for AND/OR Best-First Search

3.1 Introduction

One of our goals in this thesis is to develop anytime algorithms that provide both upper and lower bounds on the min-sum objective [1, 34] as well as for summation queries [31]. As shown in earlier work including that of Chapter 2, AND/OR Branch-and-Bound (AOBB) is one such an anytime algorithm that generates upper bounds through a sequence of improved suboptimal solutions over time. However, there has been little work on the symmetrical problem of **generating lower bounds by search in an anytime manner**.

We turn to best-first search for this task, which explores the search space in frontiers of non-decreasing lower bounds and is thus inherently anytime for producing lower bounds. Specifically, we will explore *AND/OR* Best-First (AOBF) search, guided by the mini-bucket heuristic, which is known to be a state-of-the-art algorithm for the min-sum task, but which has been evaluated mostly on its performance for finding an optimal solution [33]. The scheme can be easily adapted to yield a sequence of lower bounds by simply reporting the best heuristic evaluation it has seen so far. Indeed, this idea was used recently in another

search framework [1].

The focus of this chapter is on the specific aspect of the impact of AND child node ordering on AOBF's ability to generate lower bounds in an anytime manner. AOBF is guided by two heuristic evaluation functions. In the AND/OR search space, the "best" node is represented by a *partial solution graph* with the best potential solution according to a heuristic evaluation function f_1 amongst all partial solution graphs of the currently explored search space. A second heuristic f_2 prioritizes which leaf (known as *tip nodes*) of the current best partial solution graph should be expanded next. We call this the AND subproblem ordering. Quoting Pearl (page 50) [41],

> "These two functions, serving in two different roles, provide two different types of estimates: f_1 estimates some properties of the set of solution graphs that may emanate from a given candidate base, whereas f_2 estimates the amount of information that a given node expansion may provide regarding the alleged superiority of its hosting graph. Most works in search theory focus on the computation of f_1, whereas f_2 is usually chosen in an adhoc manner."

Indeed, in most current implementations of AOBF, f_2 is simply chosen to be equal to f_1. We show in this chapter that the choice of f_2 has a significant impact on the anytime performance of AOBF for finding lower bounds. In our analysis, we show that the *residual*, which captures the accuracy of the heuristic evaluation function, is a natural choice for f_2. In Chapter 2, we showed that residual of the mini-bucket heuristic can be approximated by its *local errors*. We illustrate empirically that the local bucket errors can provide relevant information on the increase of the lower bound due to a given node expansion. To our knowledge, this is a first investigation of subproblem ordering in AOBF and among the first investigations of anytime best-first search for generating lower bounds.

The rest of this chapter is organized as follows: Section 3.2 presents background on the AOBF algorithm. Section 3.3 analyzes the impact of subproblem ordering and illustrates it with an example. Section 3.4 introduces the subproblem ordering heuristic based on residuals and local bucket errors and suggests a way to approximate them. Section 3.5 presents the experiments and section 3.6 concludes.

3.2 Background: AND/OR Best-First Search

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

Table 3.1: Notation on AND/OR Best-First search.

The AO* algorithm is a best-first search algorithm for AND/OR search spaces [38]. Our AOBF (AND/OR Best First) algorithm is a variant of AO* specialized for graphical models [33]. In the following, we use the notation in **Table 3.1**. The algorithm works by gradually expanding a portion of the context-minimal AND/OR search graph $\mathcal{G}$, always identifying the best *partial solution tree* T in $\mathcal{G}$. After node expansion, every node $n \in \mathcal{G}$ needs to be updated with its best lower bound based on the current state of $\mathcal{G}$, denoted $l(n)$. As usual in AO* search, AOBF [33], presented in **Algorithm 10**, interleaves a top-down expansion step and a bottom-up revision step. Once the current best partial solution tree T is determined (using f_1, implicitly in line 28), the top-down expansion step selects a non-terminal tip node of T and generates its children which are appended to $\mathcal{G}$ (lines 4-13). A bottom-up revision step then updates the internal nodes values that represent the current best lower bounds below them (lines 15-26). In this step, the values of newly expanded children are propagated to their parents, and recursively up to the root. During these value updates, the algorithm

Algorithm 10: AND/OR Best-First (AOBF) [33]

Input: A graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, pseudo-tree $\mathcal{T}$, heuristic function h
Output: Lower-bound to solution of $\mathcal{M}$

Create the root OR node r labeled by X_1 and let $\mathcal{G} = \{r\}$
Initialize value $l(r) = h(r)$ and best partial solution tree T to $\mathcal{G}$
while *r is not marked SOLVED and memory is available* **do**
    `// Expand`
    $n \leftarrow$ **Select-Node(**T**)**
    if *n is an OR node (labeled X_i)* **then**
        Create AND node n' for each $x_i \in D_i$
        if *n' is TERMINAL* **then** mark n' as SOLVED;
    else if *n is an AND node (labeled x_i)* **then**
        Create OR node n' for each child X_j of $X_i \in \mathcal{T}$
    foreach *generated n'* **do**
        $succ(n) \leftarrow succ(n) \cup n'$
        $l(n') \leftarrow h(n')$ `// Initial lower bound is the heuristic value`
    $\mathcal{G} \leftarrow \mathcal{G} \cup \{succ(n)\}$
    `// Revise`
    $S \leftarrow \{n\}$
    while $S \neq \emptyset$ **do**
        Select p from S s.t. p has no descendants in $\mathcal{G} \cap S$
        $S \leftarrow S - \{p\}$
        if *p is an OR node* **then**
            $l(p) \leftarrow \min_{m \in succ(p)}(c(p, m) + l(m))$
            Mark $k = \arg\min_m$ as the best successor of p
            if *k is SOLVED* **then** mark p as SOLVED;
            if *$l(p)$ changed and $p = r$* **then**
                Output $l(p)$ `// Report new lower bound`
        else if *p is an AND node* **then**
            $l(p) \leftarrow \sum_{m \in succ(p)} l(m)$
            if *$\forall m \in succ(p)$ are SOLVED* **then** mark p as SOLVED;
        if *$l(p)$ changed or p is SOLVED* **then**
            $S \leftarrow S \cup \{parent(p)\}$
    Update T to new best partial solution tree by following marked best successors from root r
return $\langle l(r), T \rangle$

marks the best child of each OR node. In addition, OR nodes are marked *solved* if their current best child is marked *solved* and AND nodes are marked *solved* if all of their children are marked *solved*. Subsequently, a new best partial solution tree T is identified by following the marked children from the root to leaves of $\mathcal{G}$. When the root node of $\mathcal{G}$ is marked as solved, the best solution tree T is the optimal solution whose cost is the value of the root node $l(r)$ and AOBF terminates.

In earlier literature the algorithm is provided as a purely exact algorithm [38, 33]. However, since lower bound values are constantly updated as the algorithm searches, $l(r)$ provides a *anytime lower bound* on the optimal solution (lines 22-23).

The use of the f_2 heuristic for subproblem ordering occurs implicitly in line 4 of the algorithm which calls the **Select-Node** function that returns any non-terminal tip node of T. The behavior of this function is thus defined by the f_2 heuristic which imposes an ordering of the tip nodes. It is common to choose an f_2 which orders the non-terminal tips using the f_1 heuristic in ascending order. The focus of our work is in proposing a more informed f_2.

3.3 Illustrating the Impact of Subproblem Ordering

We will now show that the f_2 heuristic can have a potentially large impact in AOBF and start with an example. In the following, T_i refers to the best partial solution tree after the i-th node expansion of AOBF. We first define the notion of a profile of f_2 relative to f_1:

Definition 3.1 (profile). *Given a primary and secondary heuristic function f_1 and f_2 respectively for AOBF applied to a graphical model, the sequence $p_{f_2} = \{f_1(T_i)|i = 1 \dots j\}$ produced with a particular f_2 when f_1 is kept fixed, is the* profile *of f_2, under f_1.*

The sequence of f_1 values seen at each step yields lower bounds on C^*, the optimal cost, whose quality increases with steps (for a monotone heuristic function). At termination, we

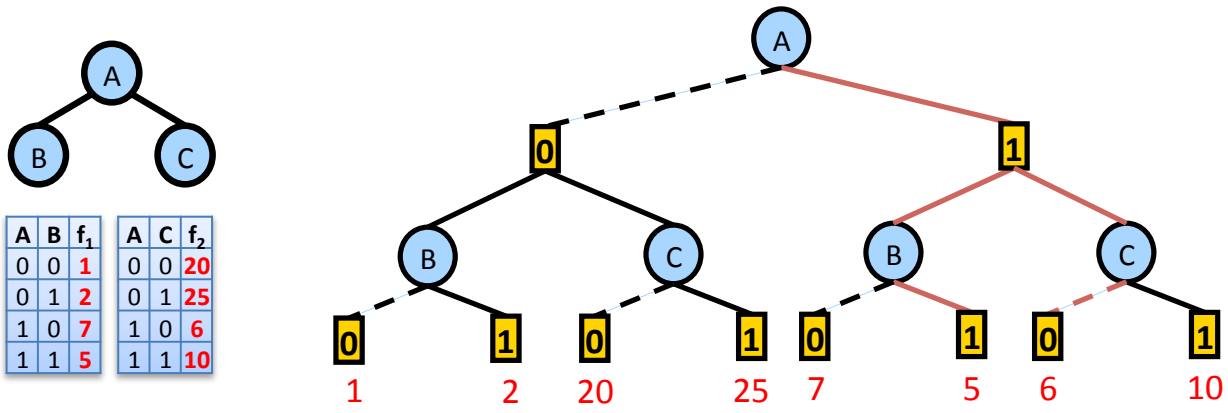


Figure 3.1: A simple graphical model over 3 variables with two functions. Shown above are the primal graph (also a valid pseudo-tree in this case), the function tables, and the associated AND/OR search space with weights labeled. The optimal solution tree is highlighted and has a cost of 11.

get the cost C^*. Yet, one sequence may be superior to another.

PROPOSITION **4.** *Given AOBF using an evaluation function f_1, there exist two tip node evaluation function f_2 and f_2' such that the profiles p_{f_2} and $p_{f_2'}$ under f_1 differ.*

Proof of Proposition 4. We prove by counter-example. **Figure 3.1** depicts a graphical model defined over variables A, B, C having two functions $f(A, B)$ and $f(A, C)$. The full AND/OR search space is shown explicitly and the optimal solution $(A = 1, B = 1, C = 0)$ is marked in red. Assume that our heuristic evaluation function is a constant 0. Assume two f_2 evaluation functions: f_2 which orders the subproblems by from left to right $(B \prec C)$, while f_2' reverses the orderings from right to left. The profile of f_2 under f_1 is: (0,1,5,11), while the profile of f_2' is: (0,6,11). In particular, with f_2 the algorithm explores all solution subtrees while with f_2', it will never expand node B under $A = 0$, since expanding C proves that the $A = 0$ branch has a cost of at least 20. We would never return to the $A = 0$ branch since the $A = 1$ branch never exceeds a cost of 20 at any point. In fact, it yields the optimal solution. Clearly, profile $p_{f_2'}$ dominates that of p_{f_2} in this case. □

In the following we show further that the choice of f_2 can, in the worst case, make an

exponential impact on the number of expansions needed in order to cross a given lower bound threshold L. This means that the impact of f_2 can be quite high for both anytime performance and when searching for an optimal solution.

Theorem 3.1. *Given a weighted AND/OR search graph, a heuristic evaluation function f_1, and two secondary f_2 and f_2' functions, there exists an AND/OR search graph and a threshold L where the profile until reaching L for p_{f_2} is exponentially longer than for $p_{f_2'}$.*

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

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

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

3.4 Local Bucket Errors for AOBF

The overall target for an effective subproblem ordering heuristic should be to select the subproblem which increases the lower bound the most. Assume a greedy scheme where given a frontier of tip nodes, we choose the node which would lead to the largest increase in f_1 in a *single* expansion. We will show that this largest increase in f_1 corresponds to the notion of *residual*, and therefore we can use *look-ahead* as a guide to decide which subproblem to expand next. In Chapter 2, we introduced the concept of local bucket errors and showed its relation to the residual. The bucket error measures the difference between the *exact bucket message* $\mu_k^*(\cdot)$ that would have been generated by a particular bucket B_k without partitioning, to the *mini-bucket message* $\mu_k(\cdot)$ generated with partitioning. The difference between the two messages is defined as the local bucket error function $E_k(\cdot)$. We showed that this quantity is equivalent to the depth-1 residual (see Theorem 2.1). Therefore, the local bucket error functions can be used also to order tip node expansion for the goal of increasing f_1 the most with a single expansion. Furthermore, we showed that this information can be compiled prior to search (see **Algorithm 7**).

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

The actual quantity of interest (which we will have to approximate) is the exact residual defined next.

Definition 3.2 (exact residual). *The* exact residual $res^*(n)$ *is defined as* $h^*(n) - h(n)$.

The exact residual is equivalent to a d-level residual when d is the depth of the search space

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

3.4.1 A Measure for Average Subtree Error

The first approach is to use the *average local bucket error* $\tilde{E}_k$ (Definition 2.8) that was introduced in Chapter 2 as a measure of the error at each bucket. It is the average over the values of the functions, or over a sample of the values if the scope of the function is cost-prohibitive. Computing this quantity is linear in the number of samples and requires a single constant to memorize. The average subtree errors, denoted $\tilde{E}_k^t$, are derived by adding the average bucket error along a subtree t. Namely,

Definition 3.3 (average subtree error). *The average subtree error $\tilde{E}_k^t$ can be defined recursively by:*

$$\tilde{E}_k^t = \tilde{E}_k + \sum_{c \in ch(X_k)} \tilde{E}_c^t \tag{3.1}$$

where $\tilde{E}_k$ is the average local bucket error for X_k.

Clearly, the average subtree error is a constant, so its quality depends on the variance of the bucket error functions. Still, its value is its low memory consumption, which is a critical resource in the context of a best-first algorithm such as AOBF.

3.4.2 A Functional Measure for Subtree Error

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

Definition 3.4 (**average elimination operator** $\Downarrow$). *Let $f(\cdot)$ be a function with scope S_{f_j}, and $S \subseteq S_{f_j}$, and $\mathbf{D}_S$ be the domains of the variables in S. The average elimination $\Downarrow_S$ of $f(\cdot)$ is defined by*

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

Next, we define the subtree error functions, denoted $E_k^t(\cdot)$.

Definition 3.5 (**subtree error function**). *Let $E_k(\cdot)$ be the local bucket error function of X_k The subtree error function relative to a pseudo-tree is defined recursively by:*

$$E_k^t(\cdot) = E_k(\cdot) + \sum_{c \in ch(X_k)} \Downarrow_{(S_{E_c} - S_{E_k})} E_c^t(\cdot) \tag{3.2}$$

We can view each element of summation over the children as a message computed by *averaging* out the variables which are not present in the parent's bucket. Each message computed at c can be interpreted as the expected error of the subproblem rooted by X_c as a function of its ancestor variables.

In the above definition, we treat all the variables as equal contributors to the error. Since one of our goals is to expand fewer nodes to reach a particular threshold L, we need to discount the contribution of errors from variables that are far away from the current variable. Thus, we introduce a variable-dependent discount factor $\gamma_k \leq 1$, yielding

$$E_k^t(\cdot) = E_k(\cdot) + \gamma_k \cdot \sum_{c \in ch(X_k)} \Downarrow_{S_{E_c} - S_{E_k}} E_c^t(\cdot) \tag{3.3}$$

Due to the recursive definition, one can see that the impact of a descendant's message decreases exponentially with distance from X_k.

Algorithm 11 generates the subtree error functions starting from the local bucket error functions as input. We denote the parent of k in $\mathcal{T}$ as $pa(k)$ and $\lambda_{c \to k}^e$ as the message sent

from c to k.

Algorithm 11: Bucket Error Propagation (BEP)

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

Output: Subtree error functions $E_k^t(\cdot)$

Initialize, for all leaves u of $\mathcal{T}$, $E_u^t(\cdot) = E_u(\cdot)$

Compute bottom-up over $\mathcal{T}$, for each variable X_k:

  $E_k^t(\cdot) = E_k(\cdot) + \gamma_k \cdot \sum_{c \in ch(X_k)} \lambda_{c \to k}^e(\cdot)$; `// Incorporate child messages`

  $\lambda_{k \to pa(k)}^e(\cdot) = \downdownarrows_{S_{E_k} - S_{E_{pa(k)}}} E_k^t(\cdot)$; `// Compute message`

return $E_k^t(\cdot)$ for each variable $X_k \in \mathbf{X}$

PROPOSITION 5 (**complexity of BEP**). *Given the local bucket error functions $E_k(\cdot)$ with a maximum scope size of $|S|$ where k bounds the maximum domain size, the time and space complexity of algorithm BEP is $O(nk^{|S|})$.*

Proof. Computing the subtree error message generated by a subtree error function $E_k^t(\cdot)$ requires enumeration over its values which is bounded by $O(k^{|S_{E_k}|})$ time. Clearly, each message used for computing each subtree error function $E_k^t(\cdot)$ is used just once. Incorporating a message into a function $E_k^t(\cdot)$ from a child also enumerates over its values, bounded by $O(k^{|S_{E^k}|})$ time. Since we have n variables, the total time complexity is $O(nk^{|S|})$, where $|S| = max_k |S_{E_k}|$. The space complexity is also $O(nk^{|S|})$, since the scopes of each $E_k^t(\cdot)$ are identical to the scopes $E_k(\cdot)$. □

3.4.3 Approximating the Error Functions

Since the scope of the error functions may be exponential in the pseudo-width, we consider additional simplification.

3.4.3.1 Scope Bounding

One way of bounding further the complexity of the error measure is to quantize by truncating the scopes of the local bucket error functions and aggregating over the eliminated variables.

Definition 3.6 (**scope-bounded bucket error function).** *Given E_k, the* scope-bounded bucket error function *of E_k relative to $S \subseteq S_{E_k}$ is defined by*

$$E_k^b(\cdot) = \downdownarrows_{(S_{E_k}-S)} E_k(\cdot) \tag{3.4}$$

We can select a bounded scope S where $|S|$ is smaller than a specified integer s, which we call the *s-bound*. We typically choose the s-bound to be smaller than or equal to the i-bound of the MBE heuristic. We currently select the bounded scopes in a straightforward fashion by removing variables from S_{E_k} that are closest to X_k in the pseudo-tree $\mathcal{T}$ until the condition of $|S| \leq s$ is satisfied.

Algorithm 12: Scope-Bounded Local Bucket Error Evaluation (SB-LBEE)

Input: A Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, a pseudo-tree $\mathcal{T}$, i-bound, s-bound
Output: Scope-bounded error function $E_k^b(\cdot)$ for each variable X_k

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

Algorithm 12 presents a modified version of LBEE (called SB-LBEE) to account for the bounded bucket errors. It differs from LBEE (**Algorithm 7** in Chapter 2) in having an s-bound parameter (line 4). Lines 5-6 average over the variables not in S, generating bounded versions of the MBE and exact bucket messages. If $s = 0$, the errors reduce to the *average local bucket error*.

PROPOSITION 6 (**complexity of SB-LBEE).** *Given an s-bound, and an i-bound for MBE, the time complexity of SB-LBEE is* $O(nk^{psw(i)})$ *and the space complexity is* $O(nk^s)$*, where n is the number of variables, k bounds the maximum domain size,* $psw(i)$ *is the pseudo-width along* $\mathcal{T}$ *relative to* $MBE(i)$*.*

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

Since the space complexity of BEP (**Algorithm 11**) depends on the maximum scope size of the error functions, we can replace the full local bucket error functions $E_k(\cdot)$ with the scope-bounded versions computed by SB-LBEE to generate *s-bounded subtree error functions*, which are tractable to store in memory with an appropriate choice of s.

3.4.3.2 Sampling

Since the time complexity of SB-LBEE is bounded by $O(nk^{psw(i)})$, it may still be intractable. Since this complexity is due to the computation of the exact bucket message $\mu_k^{b*}(\cdot)$ (Proposition 6), we can further alleviate this by sampling to approximate the $\downdownarrows_{S_{B_k}-S}$ operation. Given $S \subseteq S_{B_k}$, for each instantiation $\bar{x}_S$ of $\mu_k^{b*}(\cdot)$, we define an estimate of $\mu_k^{b*}(\cdot)$ by

$$\hat{\mu}_k^{b*}(\bar{x}_S) = \frac{1}{m} \min_{x_k} \sum_{f \in B_k} f(\bar{x}_S, \bar{x}_i) \quad \bar{x}_i \sim U(\mathbf{D}_{S_{B_k}-S}) \tag{3.5}$$

where m is the number of samples and $U(\mathbf{D}_{S_{B_k}-S})$ is a uniform distribution over its argument $S_{B_k} - S$, the set of variables we need to eliminate. Since we sample m times for each of the k^s instantiations over n variables, the time complexity is $O(nmk^s)$, which carries over to SB-LBEE.

3.4.4 Algorithm: Select-Node-Subtree-Error

Algorithm 13: Subtree Error Compilation (SEC)

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

Algorithm 14: Select-Node-Subtree-Error (Select-Node-STE)

Input: A Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, a pseudo-tree $\mathcal{T}$, i-bound, s-bound, discount factors γ_k, partial solution tree T
Output: Variable X_k
1 **Initialize:** Compile Subtree Errors with given $\mathcal{M}, \mathcal{T}, i$-bound , s-bound, and γ_k. $tips :=$ tip nodes of T
2 $\bar{x}_T :=$ partial assignment corresponding to T
3 **return** $\arg\max_{k \in tips} E_k^t(\bar{x}_T)$

We summarize the approach in **Algorithms 13** and **14**. Before search begins, we compile the subtree errors with SEC (**Algorithm 13**), which includes the standard procedure of compiling the MBE heuristic, then we subsequently use SB-LBEE to generate the scope-bounded bucket error functions. The algorithm then uses BEP (**Algorithm 11**) to generate the subtree error functions based on the output of SB-LBEE. Note that as SB-LBEE reduces to computing average local bucket errors when the s-bound is 0, the subtree error functions are exactly the average subtree errors described in section 3.4.1, but with the discount factors applied.

During search, we can use Select-Node-STE (**Algorithm 14**) in AOBF, which selects a tip node with the largest value according to its corresponding subtree error function and the current assignment based on the current partial solution tree T.

3.5 Experiments

We experimented with 4 variants of our scope-bounded subtree error functions, formed by using either *constant* or *scope-bounded* errors (denoted **Const** and **ScpBnd** respectively), and *absolute* or *relative* errors (denoted **Abs** and **Rel** respectively). Thus, the 4 variants are named **Const-Abs (C-A)**, **Const-Rel (C-R)**, **ScpBnd-Abs (SB-A)**, and **ScpBnd-Rel (SB-R)**.

Using **Algorithm 14**, the **Const** variants set the $s = 0$, while otherwise we use a default $s = 10$ for the **ScpBnd** variants. The absolute error **Abs** computes each error function value exactly as described in the previous section, while the relative error **Rel** divides each error value by the exact bucket message value as a way to normalize for the differing scales of values in a function (see Definition 2.8 in the previous chapter for more details). In all variants, for each variable X_k, we set the discount factor $\gamma_k = \frac{1}{|D_k|}$, the inverse of the domain size of X_k.

We compare against the baseline subproblem ordering that uses the heuristic evaluation function, f_1 (denoted **Heur (H)** in the experiments). We will generally refer to the 4 variants as *error-guided orderings*. Additionally, we break ties using the baseline ordering. For all experiments, we use the mini-bucket elimination with moment-matching (MBE-MM) heuristic [25] and the same pseudo-tree for all settings. We varied the i-bound to show how the results depend on the levels of heuristic error. For i-bounds less than 10, we set the s-bound to be equal to the i-bound. Whenever necessary, we used a maximum sample size

of 10^5 shared across the entire scope-bounded error function. Furthermore, if the number of samples is less than the number of instantiations of a particular scope-bounded error function, we reduce s further. Given these settings, the memory used by the pre-compiled error functions were no greater than that of the MBE heuristics (i.e. MBE dominates the space complexity for anything computed during preprocessing).

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

Table 3.2: Benchmark statistics for. # inst - number of instances, n - number of variables, $w*$ - induced width, h - pseudo-tree height, k - maximum domain size, $|F|$ - number of functions, a - maximum arity. The top value is the minimum and the bottom value is the maximum for that statistic.

We experimented with benchmarks from genetic linkage analysis [19] (*pedigree, type4*), and medical diagnosis networks [49] (*promedas*). and included only instances that had a significant amount of search, in particular, having a number of nodes expanded is larger than 10^5 when using the baseline ordering for the lowest i-bound. Overall we report results on 13 pedigrees, 34 promedas networks, and 65 type4 instances, yielding a total of 112 problem instances. The benchmarks represent a variety of problem difficulties ranging from easy to hard and are presented in that order. **Table 3.2** provides ranges of the various benchmark parameters.

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

3.5.1 Evaluating for Exact Solutions

Tables 3.3, 3.5, and **3.7** report the preprocessing time for compiling the MBE heuristic and subtree error functions (in seconds), total CPU time including preprocessing (in seconds), and number of OR nodes (in thousands of nodes) expanded for a subset of the problems which could be solved exactly by AOBF over the benchmarks. For each instance we also mention the problem's parameters such as the number of variables (n), maximum domain size (k), induced width (w^*), and pseudo-tree height (h). Each column is indexed by the i-bound of the MBE-MM. Each row for each instance shows the various subproblem ordering schemes we experimented with. Additionally, we summarize over all of the problems that could be solved, by i-bound, for each benchmark in **Tables 3.4, 3.6,** and **3.8**.

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

Pedigree. **Table 3.3** shows the results for four different i-bounds for selected instances from the **pedigree** benchmark. At an i-bound of 6, we see that for 2 of the instances (*pedigree9* and *pedigree33*) AOBF runs out of memory when using the baseline ordering. For *pedigree20*, we see when using the **Const-Rel** heuristic, the number of nodes expanded is a about half of that of the baseline. Still, the baseline can be better (e.g. see *pedigree39*). Moving to higher i-bounds, we see that for instances other than *pedigree39*, the error-guided orderings usually result in fewer nodes expanded at termination which usually translates to improved runtime, except at the highest i-bounds. This is due to the small difference in the number of nodes expanded relative to the extra preprocessing time. For example, on *pedigree9* with i =18, although **Const-Abs** expands 60K fewer nodes, preprocessing took 2 seconds longer compared with the baseline, resulting in a total time that is 1 second worse.

Table 3.4 shows the win counts for time and nodes expanded for each of the error-guided orderings relative to the baseline. Examining the number of wins based on nodes expanded,

instance (n, k, w^*, h)	heur	$i=6$ pre	$i=6$ time	$i=6$ nodes	$i=10$ pre	$i=10$ time	$i=10$ nodes	$i=14$ pre	$i=14$ time	$i=14$ nodes	$i=18$ pre	$i=18$ time	$i=18$ nodes
	Heur	0	oom	-	0	154	4137	1	14	512	6	**10**	193
	Const-Abs	0	602	6368	1	73	**1741**	3	12	**350**	8	11	**133**
pedigree9	Const-Rel	0	673	6361	1	**72**	1751	3	11	350	8	11	134
(935,7,25,137)	ScpBnd-Abs	0	oom	-	1	448	9241	3	18	617	8	13	216
	ScpBnd-Rel	0	**588**	**6355**	1	119	3034	2	**10**	350	8	11	133
	Heur	0	85	2678	0	28	1019	0	16	595	6	**6**	5
	Const-Abs	0	78	1933	0	**24**	**812**	2	16	**503**	7	7	5
pedigree20	Const-Rel	0	58	**1523**	1	28	833	1	15	505	7	7	5
(387,5,21,58)	ScpBnd-Abs	0	232	5521	0	43	1354	1	22	831	7	7	4
	ScpBnd-Rel	0	**57**	1579	0	24	818	1	**15**	512	7	7	**4**
	Heur	0	oom	-	0	11	448	1	**4**	163	6	**6**	28
	Const-Abs	0	576	10568	1	13	392	4	8	160	8	9	26
pedigree33	Const-Rel	0	503	**10498**	1	13	434	4	7	156	8	9	26
(581,4,24,116)	ScpBnd-Abs	0	**452**	10527	1	**10**	**380**	3	6	**145**	8	8	26
	ScpBnd-Rel	0	514	10513	1	13	415	3	7	159	8	8	**26**
	Heur	0	**146**	**3869**	0	**11**	**490**	0	**0**	3	6	**6**	1
	Const-Abs	0	230	4964	1	17	604	1	1	2	6	7	1
pedigree39	Const-Rel	0	223	4921	0	15	604	1	1	2	6	7	1
(953,5,20,82)	ScpBnd-Abs	0	217	4974	0	15	605	1	1	2	6	6	1
	ScpBnd-Rel	0	220	4920	0	15	593	1	1	**1**	6	6	**1**
	Heur	0	oom	-	0	oom	-	2	oom	-	34	180	4995
	Const-Abs	0	oom	-	3	oom	-	6	**487**	**13188**	39	**166**	**3942**
pedigree41	Const-Rel	0	oom	-	3	oom	-	6	oom	-	39	168	3942
(885,5,33,100)	ScpBnd-Abs	0	oom	-	2	oom	-	5	oom	-	38	oom	-
	ScpBnd-Rel	0	oom	-	2	oom	-	5	oom	-	38	167	3967
	Heur	0	23	935	0	28	1113	12	**15**	130	170	**170**	20
	Const-Abs	0	20	**671**	2	**22**	780	14	16	83	171	171	14
pedigree42	Const-Rel	0	**20**	679	2	22	**780**	14	16	123	173	174	14
(390,5,21,67)	ScpBnd-Abs	0	40	1109	2	30	986	13	15	**83**	171	171	**12**
	ScpBnd-Rel	0	20	675	2	29	1053	13	16	102	171	171	15
	Heur	0	oom	-	0	664	12209	1	36	1254	4	**9**	223
	Const-Abs	0	oom	-	1	388	9683	3	36	1230	6	11	221
pedigree44	Const-Rel	0	oom	-	1	412	9783	3	35	**1138**	6	11	221
(644,4,24,79)	ScpBnd-Abs	0	oom	-	1	**336**	**9117**	3	**35**	1154	6	10	**213**
	ScpBnd-Rel	0	oom	-	1	497	11041	2	36	1175	6	11	213

Table 3.3: Exact evaluation for **pedigree** instances. The best times and node counts are **bolded** per i-bound and the best times and node count overall for a given instance are also **underlined**. If the optimal solution was not found, then we report 'oom' to denote that the experiment ran out of memory. We also report the pre-processing time to the left of each total time. n - number of variables, k - maximum domain size, w^* - induced width, h - pseudo-tree height

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

Table 3.4: Summary for exact solutions on **pedigree** instances: win counts of instances that the ordering heuristic had a lower time or lower number of nodes than the baseline. The number of instances solved by any heuristic (including the baseline) is shown under each i-bound label. Each number in parentheses is the percentage of solved instances that performed better for that particular i-bound.

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

Promedas. **Table 3.5** reports on a selection of **promedas** instances. We used higher i-bounds for this benchmark because the instances are harder compared to the **pedigrees**. As in the previous benchmark, we observe that the number of nodes expanded by an error-guided ordering is better than the baseline ordering in many cases. Notably, we see here for the high i-bound of 18, where the savings in nodes on a few harder instances translated well to savings in the runtime. For example, on *or_chain_25.fg*, both the runtime and number of nodes expanded using any of the error-guided orderings were about half of those of the baseline ordering. Still, the error-guided orderings may still be worse than the baseline (e.g. *or_chain_140.fg*, but this is usually the exception.

Table 3.6 provides the win counts for each error-guided ordering heuristic. The **ScpBnd-Abs** heuristic is the worst performer, as we saw before. Both relative error base orderings (**Const-Rel** and **ScpBnd-Rel**) have similar positive performance, demonstrating positive impact of the orderings on at least 50% of the instances across all i-bounds. Overall, **Const-Abs** was best, yielding improvements on at least 70% of the instances. The time savings in nodes carries over to most cases using lower i-bounds and slightly fewer at higher i-bounds. For example, at an i-bound of 12 using **Const-Abs**, 9 out of the 10 instances (90%) had

instance	heur	$i=12$			$i=14$			$i=16$			$i=18$		
(n,k,w^*,h)		pre	time	nodes	pre	time	nodes	pre	time	nodes	pre	time	nodes
	Heur	0	oom	-	0	239	10020	0	164	7009	1	319	13152
	Const-Abs	2	oom	-	3	172	6686	4	**115**	**4629**	4	161	6225
or_chain_25.fg	Const-Rel	2	oom	-	3	175	**6668**	4	117	4671	4	**159**	6226
(1075,2,43,80)	ScpBnd-Abs	2	oom	-	2	oom	-	3	117	4678	4	163	6225
	ScpBnd-Rel	2	oom	-	2	**170**	6670	3	115	4657	4	161	**6224**
	Heur	0	oom	-	0	oom	-	1	195	8642	2	109	5084
	Const-Abs	2	oom	-	3	oom	-	5	168	7039	6	60	**2550**
or_chain_40.fg	Const-Rel	2	oom	-	3	oom	-	5	**144**	**5721**	7	**60**	2550
(988,2,43,87)	ScpBnd-Abs	2	oom	-	3	oom	-	4	302	12748	5	231	9579
	ScpBnd-Rel	2	oom	-	3	oom	-	4	152	6151	6	61	2550
	Heur	0	42	2036	0	11	563	0	**7**	349	1	9	411
	Const-Abs	1	**31**	**1391**	2	10	350	3	8	**226**	4	**8**	251
or_chain_63.fg	Const-Rel	1	32	1391	2	10	**350**	3	8	244	4	9	**251**
(731,2,38,81)	ScpBnd-Abs	1	52	2266	2	15	608	3	10	382	4	12	467
	ScpBnd-Rel	1	44	2038	2	**10**	361	3	7	230	4	9	273
	Heur	0	128	5374	0	80	3398	1	47	1977	2	46	1904
	Const-Abs	2	112	4550	4	**75**	3019	5	47	1724	7	42	**1443**
or_chain_80.fg	Const-Rel	2	**99**	**4550**	4	76	3029	5	47	1738	7	41	1443
(840,2,50,108)	ScpBnd-Abs	2	248	9001	3	174	6222	5	83	3108	6	84	3127
	ScpBnd-Rel	3	114	4551	3	76	**3018**	4	**46**	**1724**	6	**40**	1444
	Heur	0	74	3659	0	58	2945	1	45	2498	1	28	1582
	Const-Abs	1	39	1690	3	32	1416	4	28	1318	4	21	937
or_chain_94.fg	Const-Rel	1	**35**	1690	3	32	1416	4	28	**1298**	4	21	937
(762,2,32,97)	ScpBnd-Abs	1	370	12931	2	206	8068	3	114	5048	4	59	2787
	ScpBnd-Rel	1	37	**1624**	2	**30**	**1357**	3	**27**	1307	4	**21**	**931**
	Heur	0	135	5103	0	79	3946	1	**41**	2224	1	**32**	**1714**
	Const-Abs	2	**128**	**4950**	3	66	**2913**	4	51	2290	4	55	2448
or_chain_140.fg	Const-Rel	2	139	5633	3	66	2934	4	49	2288	4	69	3328
(1260,2,32,79)	ScpBnd-Abs	2	139	4979	2	70	2962	3	43	**2008**	4	51	2240
	ScpBnd-Rel	2	138	5305	2	**65**	2944	3	47	2119	4	67	3220
	Heur	0	oom	-	0	284	13893	1	253	14811	1	231	11752
	Const-Abs	2	265	11758	3	134	**6389**	5	**145**	**6958**	5	114	5339
or_chain_178.fg	Const-Rel	2	**264**	**11724**	3	**133**	6390	5	254	12591	6	**111**	**5339**
(1012,2,35,97)	ScpBnd-Abs	2	oom	-	3	oom	-	4	oom	-	5	oom	-
	ScpBnd-Rel	2	300	12810	3	157	7303	4	278	13411	5	113	5461
	Heur	0	33	1677	0	**42**	2259	0	33	1850	1	**20**	1098
	Const-Abs	1	**29**	**1332**	2	43	**2094**	3	**30**	**1496**	4	20	971
or_chain_199.fg	Const-Rel	1	39	1708	2	71	2921	3	30	1498	4	20	**971**
(917,2,33,79)	ScpBnd-Abs	1	93	3330	2	79	3048	2	70	3050	3	34	1527
	ScpBnd-Rel	1	39	1693	2	58	2596	2	38	1789	3	22	1065
	Heur	0	103	5031	0	60	3110	0	43	2382	1	17	911
	Const-Abs	1	55	**2476**	2	**33**	**1569**	3	**26**	1165	4	14	**549**
or_chain_212.fg	Const-Rel	1	**55**	2476	2	33	1570	3	26	**1164**	4	14	549
(773,2,33,79)	ScpBnd-Abs	1	oom	-	2	282	12138	3	215	9986	3	86	4305
	ScpBnd-Rel	1	66	2830	2	35	1670	3	31	1399	3	**14**	566
	Heur	0	197	8434	0	212	9580	1	75	3443	2	34	1528
	Const-Abs	2	156	5322	4	**90**	**3330**	5	**45**	1611	6	28	**921**
or_chain_226.fg	Const-Rel	2	**154**	**5320**	4	104	3905	5	45	**1611**	5	26	976
(735,2,42,87)	ScpBnd-Abs	2	291	11595	3	346	15606	4	117	4854	5	62	2525
	ScpBnd-Rel	2	159	5482	3	98	3663	4	46	1697	4	**25**	976

Table 3.5: Exact evaluation for **promedas** instances. The best times and node counts are **bolded** per i-bound and the best times and node count overall for a given instance are also **underlined**. If the optimal solution was not found, then we report 'oom' to denote that the experiment ran out of memory. We also report the pre-processing time to the left of each total time. n - number of variables, k - maximum domain size, w^* - induced width, h - pseudo-tree height

a positive impact of ordering also had better runtime than the baseline. In contrast, at an i-bound of 18, only 12 out of the 16 instances (75%) had improved times.

Type4. **Table 3.7** shows all the instances of the **type4** benchmark that could be solved by AOBF with the given i-bounds. This benchmark is the hardest of the 3 benchmarks and thus we used even higher i-bounds compared to the **promedas** benchmark. Even then, only with an i-bound of 20 are we able to solve all 6 of the instances. Like with the previous

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

Table 3.6: Summary for exact solutions on **promedas** instances: win counts of instances that the ordering heuristic had a lower time or lower number of nodes than the baseline. The number of instances solved by any heuristic (including the baseline) is shown under each i-bound label. Each number in parentheses is the percentage of solved instances that performed better for that particular i-bound.

instance (n, k, w^*, h)	heur	$i = 14$			$i = 16$			$i = 18$			$i = 20$		
		pre	time	nodes	pre	time	nodes	pre	time	nodes	pre	time	nodes
	Heur	4	oom	-	9	oom	-	23	2436	11898	61	208	1415
	Const-Abs	16	oom	-	24	oom	-	36	**1965**	**10036**	70	176	**1231**
t4-haplo_100_19	Const-Rel	16	oom	-	25	oom	-	35	2024	10036	70	177	1231
(3927,5,28,362)	ScpBnd-Abs	14	oom	-	21	oom	-	33	2953	13625	68	222	1654
	ScpBnd-Rel	14	oom	-	21	oom	-	33	2039	10343	68	**175**	1231
	Heur	3	oom	-	7	568	1846	17	32	162	32	34	10
	Const-Abs	12	oom	-	16	325	**1823**	21	29	155	34	34	8
t4-haplo_120_17	Const-Rel	12	oom	-	14	311	1825	21	29	**155**	34	34	8
(4302,5,23,300)	ScpBnd-Abs	10	oom	-	14	**211**	1880	20	**25**	155	33	**34**	**8**
	ScpBnd-Rel	10	oom	-	14	338	1828	20	28	155	33	34	8
	Heur	2	**1390**	**5299**	3	41	**300**	8	12	21	13	16	6
	Const-Abs	9	3239	5820	11	31	359	11	12	16	14	14	6
t4-haplo_170_23	Const-Rel	9	3323	5821	11	32	385	11	12	16	14	**14**	6
(6933,5,21,396)	ScpBnd-Abs	7	2046	6675	9	**22**	398	10	11	22	14	14	6
	ScpBnd-Rel	8	2067	6484	10	31	383	10	**11**	**16**	14	14	**6**
	Heur	5	oom	-	11	oom	-	30	oom	-	101	880	7757
	Const-Abs	18	oom	-	26	oom	-	43	oom	-	111	832	7664
t4b_100_19	Const-Rel	18	oom	-	26	oom	-	43	oom	-	111	678	**7588**
(3938,5,29,354)	ScpBnd-Abs	16	oom	-	22	oom	-	35	oom	-	109	973	7891
	ScpBnd-Rel	16	oom	-	22	oom	-	40	oom	-	109	**634**	7598
	Heur	3	oom	-	8	508	1956	22	**24**	119	38	**39**	48
	Const-Abs	11	oom	-	16	336	1922	27	31	104	41	42	35
t4b_120_17	Const-Rel	11	oom	-	14	**307**	1919	27	31	104	41	42	**35**
(4072,5,24,319)	ScpBnd-Abs	10	oom	-	14	364	1937	26	29	**103**	40	41	38
	ScpBnd-Rel	10	oom	-	14	329	**1919**	26	30	105	40	41	37
	Heur	3	382	2968	4	**5**	37	7	**7**	20	9	**9**	5
	Const-Abs	8	287	2879	9	10	36	8	9	6	9	10	5
t4b_170_23	Const-Rel	8	333	2809	9	10	35	8	8	6	9	9	5
(5590,5,21,427)	ScpBnd-Abs	7	**176**	**2630**	8	9	53	8	8	**6**	9	10	5
	ScpBnd-Rel	7	281	2736	8	9	**34**	8	8	6	9	10	**5**

Table 3.7: Exact evaluation for **type4** instances. The best times and node counts are **bolded** per i-bound and the best times and node count overall for a given instance are also **underlined**. If the optimal solution was not found, then we report 'oom' to denote that the experiment ran out of memory. We also report the pre-processing time to the left of each total time. n - number of variables, k - maximum domain size, w^* - induced width, h - pseudo-tree height

Win counts for **type4** instances				
	$i = 14$ (solved=2)	$i = 16$ (solved=4)	$i = 18$ (solved=5)	$i = 20$ (solved=6)
heuristic	wins by time(%)/nodes(%)	wins by time(%)/nodes(%)	wins by time(%)/nodes(%)	wins by time(%)/nodes(%)
Const-Abs	1 (50.0)/1 (50.0)	3 (75.0)/3 (75.0)	3 (60.0)/5 (100.0)	3 (50.0)/4 (66.7)
Const-Rel	1 (50.0)/1 (50.0)	3 (75.0)/3 (75.0)	3 (60.0)/5 (100.0)	3 (50.0)/4 (66.7)
ScpBnd-Abs	1 (50.0)/1 (50.0)	3 (75.0)/1 (25.0)	2 (40.0)/3 (60.0)	2 (33.3)/2 (33.3)
ScpBnd-Rel	1 (50.0)/1 (50.0)	3 (75.0)/3 (75.0)	3 (60.0)/5 (100.0)	3 (50.0)/4 (66.7)

Table 3.8: Summary for exact solutions on **type4** instances: win counts of instances that the ordering heuristic had a lower time or lower number of nodes than the baseline. The number of instances solved by any heuristic (including the baseline) is shown under each i-bound label. Each number in parentheses is the percentage of solved instances that performed better for that particular i-bound.

two benchmarks, we see that the error-guided orderings yield better performance on many instances. However, the variation is smaller. For example, on *type4-haplo_100_19* using an i-bound of 20, the number of nodes expanded using the **Const-Abs** ordering is still about 87% of what the baseline ordering yields, compared with the lower 50% rates seen on some instances in the other two benchmarks. On two of the instances here (*type4-haplo_170_23* and *type4b_170_23*), there is no difference in the number of nodes expanded, indicating that most of the errors likely evaluated to zero, thus falling back on the baseline ordering. Still, we see improved runtime on the hardest of the instances here (*type4-haplo_100_19* and *type4b_100_19*), where any savings in the number of nodes expanded did carry over to overall savings in runtime.

Table 3.8 aggregates the results by win counts as before. Across the i-bounds, every error-guided ordering was better on at 50-100% of the instances except for **ScpBnd-Abs**. Notably, improvement was achieved on all instances by these 3 orderings at an i-bound of 18 and a majority at the highest i-bound of 20. In terms of overall runtime improvement, most instances also had better runtimes when their orderings were better in terms of node expansions.

3.5.1.1 Discussion on Finding Exact Solutions

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

3.5.2 Anytime Lower Bounding

Next, we will evaluate AOBF for generating lower bounds in an anytime fashion and the impact of the f_2 ordering heuristic on this anytime performance. **Figures 3.2, 3.4,** and **3.6** report the lower bound obtained as a function of time for each subproblem ordering f_2. A profile that is higher earlier in time is superior. The first point of each line is always the bound returned by the MBE-MM heuristic itself, recorded whenever search starts following all pre-processing. If known, the exact solution is also plotted as a dashed gray line. For each benchmark, we select 2 representative instances, one of which was exactly solved and another which was not. For each instance, we show 4 different i-bounds. For pedigrees both

instances are exactly solved (**Figure 3.2**) Each ordering is labeled with abbreviated names for clarity.

We also aggregated the results per benchmark in **Figures 3.3, 3.5,** and **3.7**. We normalized the time scale for each instance to that of the baseline, ranked the bounds yielded by each variant across time, and aggregated across the instances by averaging. The number of instances varies with the different i-bounds since for some large instances, compiling the MBE heuristics at the highest i-bounds exceeds our memory limit of 24GB.

Pedigrees. **Figure 3.2** shows the lower bounds as a function of time on 2 selected instances from the **pedigree** benchmark. First, on *pedigree9* (which is solved exactly). For the lower i-bounds of 6 and 10, all of the methods except **ScpBnd-Abs** perform better early on. Since the problem is easy at higher i-bounds, AOBF quickly finds the optimal solution after the initial bound generated by MBE-MM. Still, at an i-bound of 14, everything except **ScpBnd-Abs** improves over the baseline. At the highest i-bound of 18, the preprocessing overhead makes the error-guided orderings not cost-effective. In *pedigree51*, where AOBF ran out of memory before finding the exact solution, the lowest i-bounds of 6 and 10 also benefited from error-guided orderings. Increasing to the i-bound 14 yields marginally better performance compared with the baseline (except for **ScpBnd-Abs**). Finally, at the highest i-bound the baseline performs best.

Figure 3.3 presents the average ranks for each ordering heuristic based on normalizing the time across the instances and averaging as explained earlier. As seen in the instance-by-instance results, all error-guided orderings except **ScpBnd-Abs** outperform the baseline. As the i-bound increases the average rank of the baseline improves. For an i-bound of 18, only **ScpBnd-Rel** ranks similarly to the baseline, but the baseline is better overall.

Promedas. **Figure 3.4** shows results on 2 selected instances from the **promedas** bench-

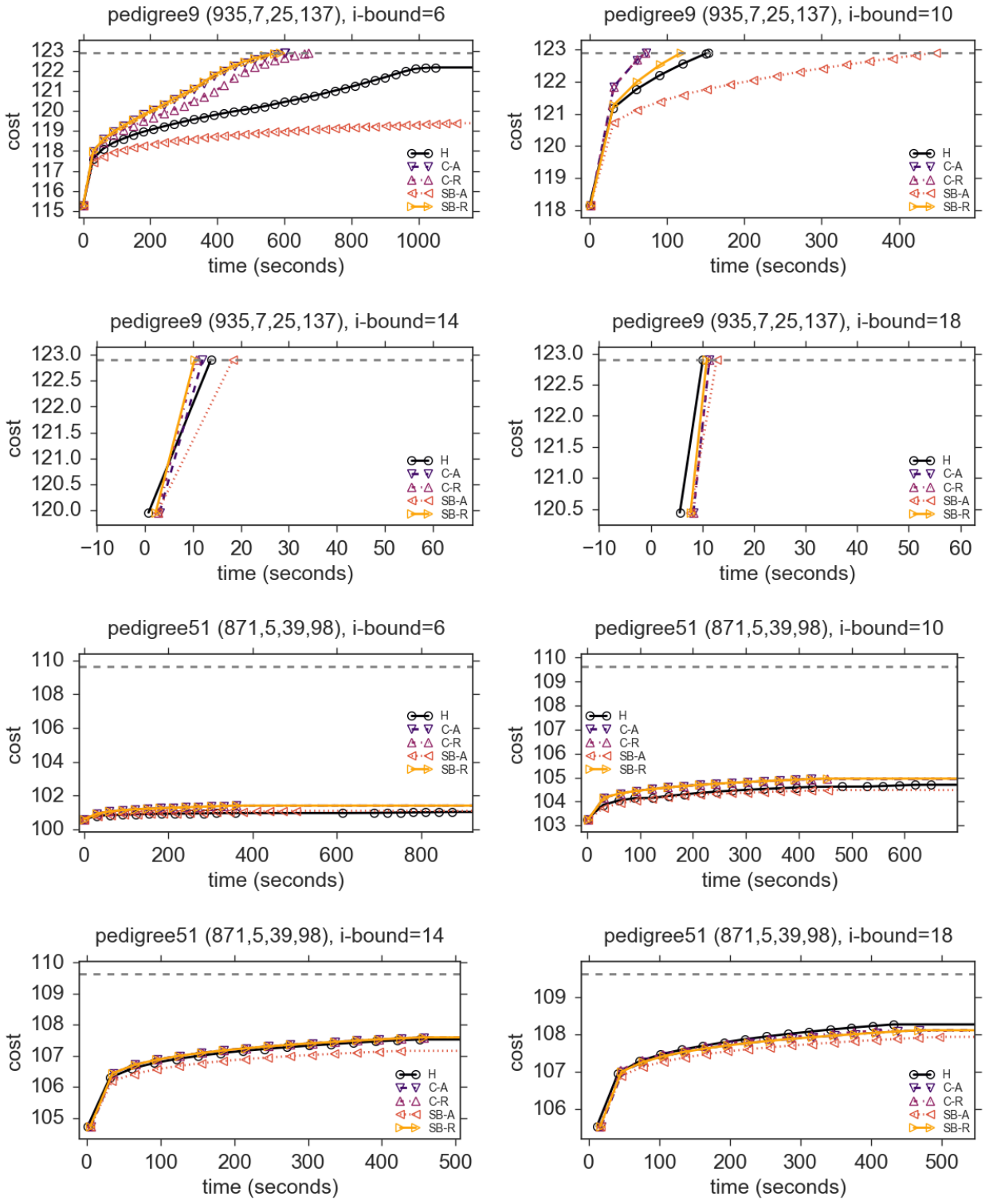


Figure 3.2: Lower bounds as a function of time for two instances from the **pedigree** benchmark. Higher is better.

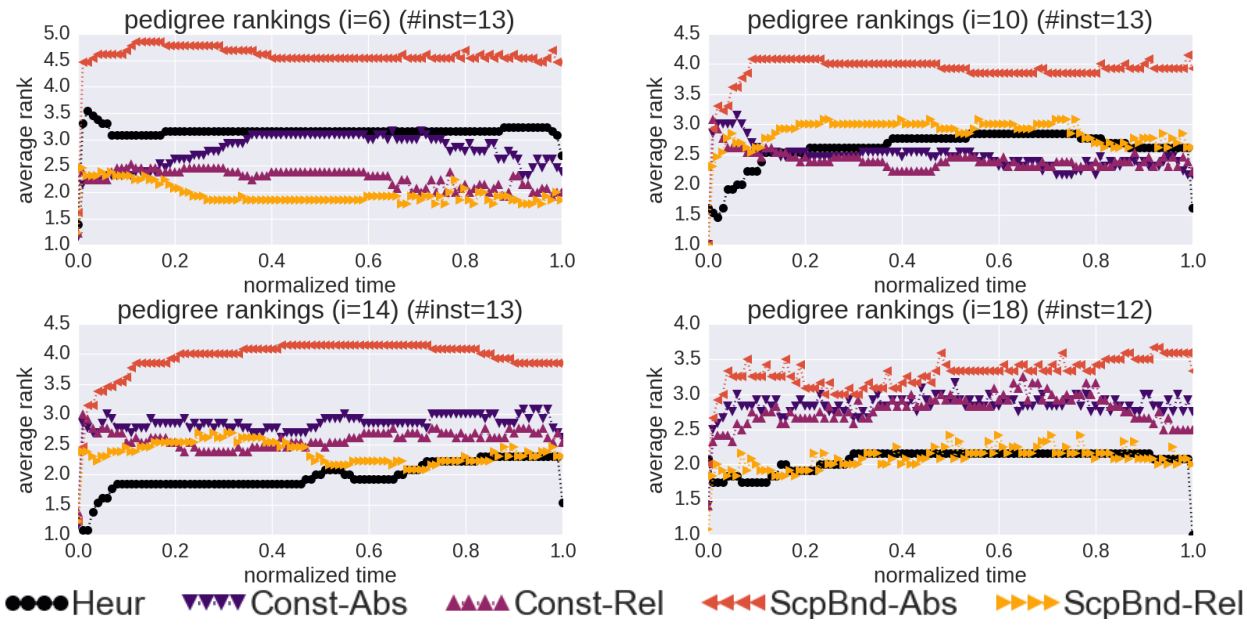


Figure 3.3: Average rank of each ordering as a function of normalized time across all of the instances in the **pedigree** benchmark. Lower is better.

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

Figure 3.5 presents the ranking summary over this benchmark. For all i-bounds the baseline seems superior early on due to the preprocessing overhead of the error-guided orderings. However, it is overtaken by the other methods eventually at different points on the normalized time scale. At an i-bound of 12, the **Const** methods outrank the baseline early on. Moving to i-bounds of 14 and 16, everything but **ScpBnd-Abs** approaches the baseline eventually and outrank it with time. Finally, at the highest i-bound of 18, the **ScpBnd-Rel** method performs better early around 0.1 on the time scale, with the **Const** methods overtaking the

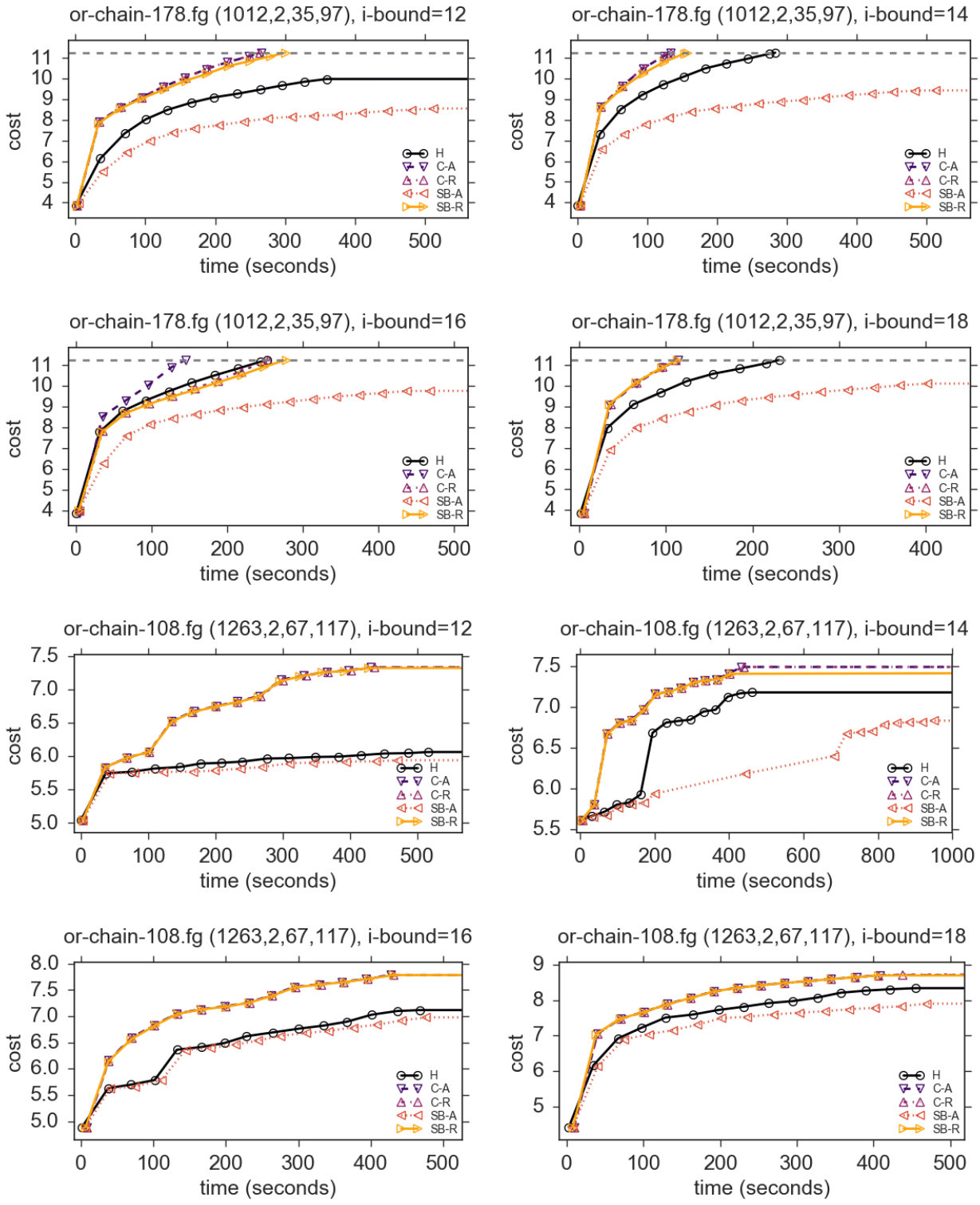


Figure 3.4: Lower bounds as a function of time for two instances from the **promedas** benchmark. Higher is better.

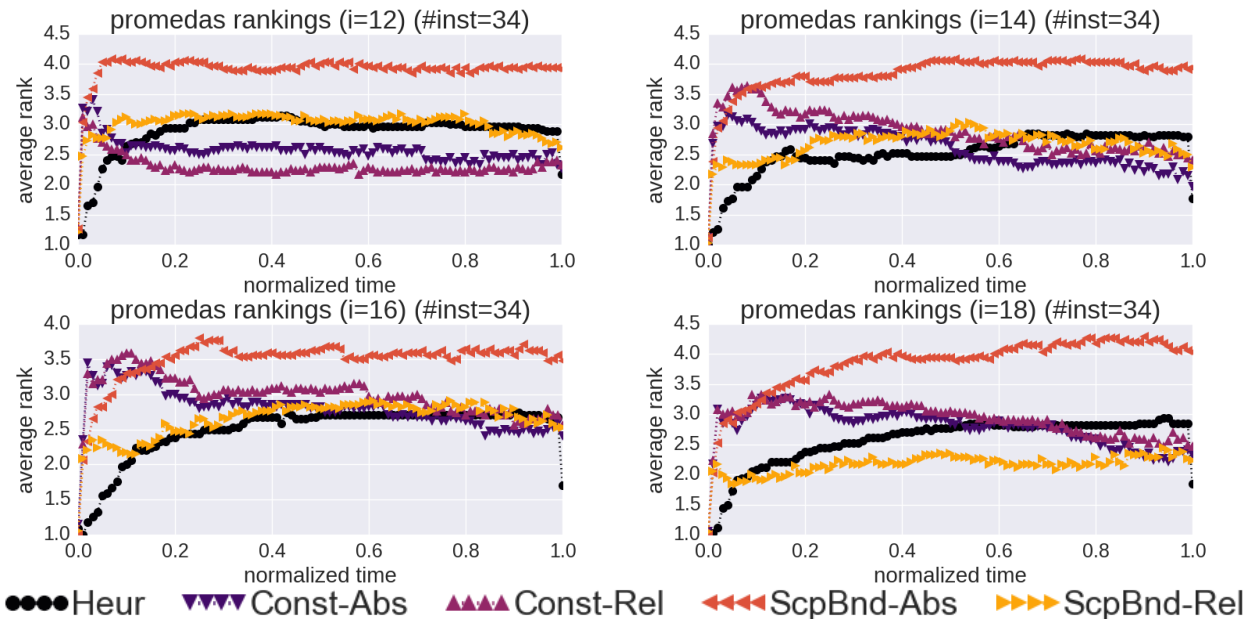


Figure 3.5: Average rank of each ordering as a function of normalized time across all of the instances in the **promedas** benchmark. Lower is better.

baseline at around 0.6 on the time scale.

Type4. **Figure 3.6** shows lower bounds over time for 2 instances of our last benchmark. The first instance (*t4-haplo-100-19*) could not be solved at i-bounds of 14 and 16, but were solved with higher i-bounds. Once again, all of the error-guided heuristics except **ScpBnd-Abs** improve performance over all i-bounds of 16 and up. At the lowest i-bound of 14, the performance is close to the baseline. Moving to *t4-haplo-190-20* which was not solved, we have similar behavior. Specifically, **ScpBnd-Abs** is slightly better than the rest at i-bounds up to 18, but performs significantly worse than both **Rel** methods at the highest i-bound of 20.

Figure 3.7 provides summary rankings of the orderings over normalized time for all of the instances that did not run out of memory for each i-bound. Notably, the number of instances reported decreases significantly as we increase the i-bound because we were unable to compile the MBE heuristics given our memory bound for many instances. At an i-bound of

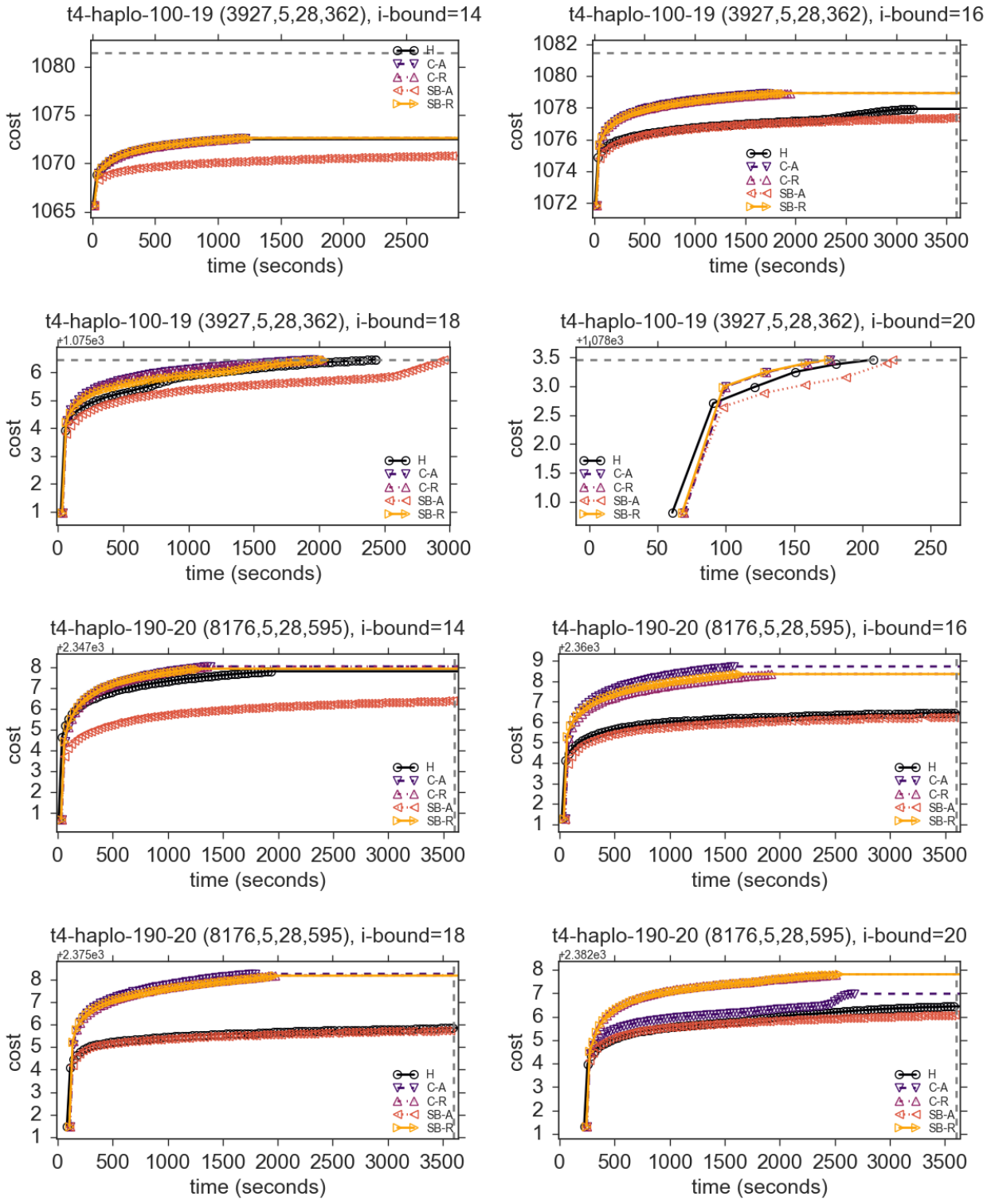


Figure 3.6: Lower bounds as a function of time for two instances from the **type4** benchmark. Higher is better.

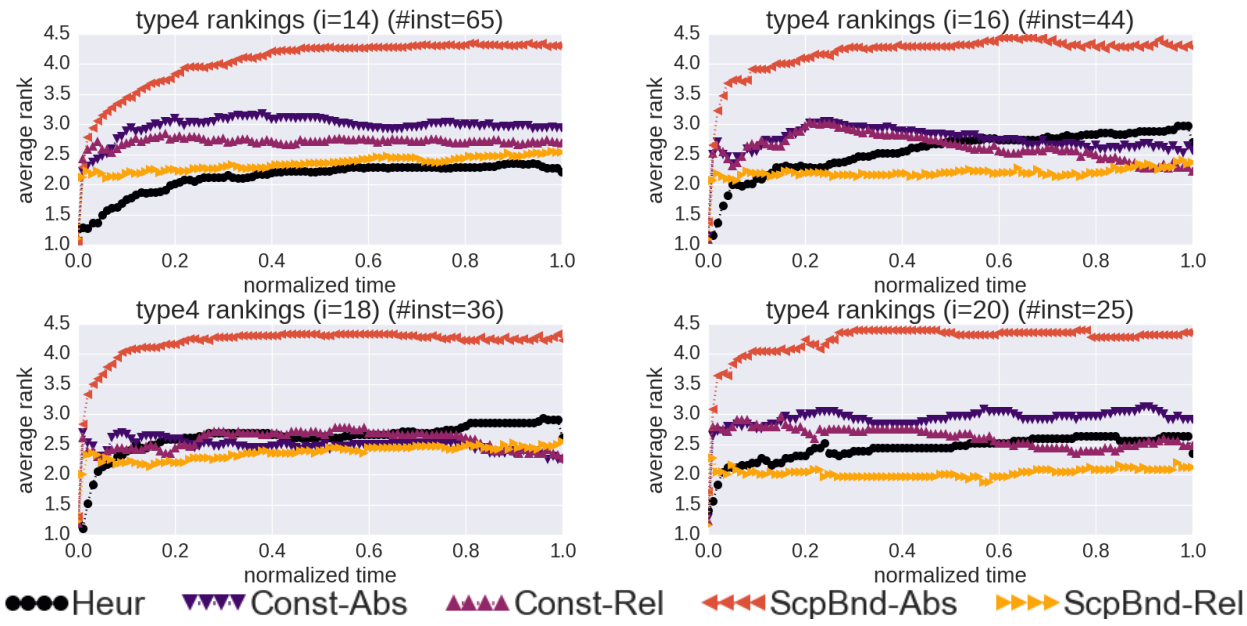


Figure 3.7: Average rank of each ordering as a function of normalized time across all of the instances in the **type4** benchmark. Lower is better.

14, the baseline performs best. Many of the instances had a large number of variables, which substantially increased the time and memory needed to compile the MBE heuristics. In turn, this extra cost carried over to increasing the preprocessing time for the error-guided ordering heuristics, leading to a negative impact. For higher i-bounds, we observe the same behavior as in the previous benchmark, where the baseline ordering is best initially. However, the error-guided orderings other than **ScpBnd-Abs** dominate with more time. Notably, with the highest i-bound, the **ScpBnd-Rel** ordering outperforms the baseline early and maintains its continuously.

3.5.2.1 Discussion on Anytime Performance

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

Overall, from the instance-by-instance lower bound plots **ScpBnd-Rel** was most consistent in producing superior lower bounds.

3.6 Conclusion

This chapter focuses on the potential of using AND/OR best-first search for generating lower bounds in an anytime fashion. Within this context, it explores the impact of subproblem ordering heuristics (the so-called secondary evaluation function [41]) for both exact and anytime performance. We present new heuristics for subproblem ordering which are based on pre-compiled information regarding the error associated with the primary heuristic, guided by the notion of bucket error. In an extensive empirical evaluation, we showed that in the context of evaluating the exact solution, all of our proposed error-based variants, with the exception **ScpBnd-Abs**, were equally good, improving the runtime on 50-60% of the instances. This accounted for most of the hard instances which had enough error to impact subproblem ordering. In the anytime evaluation, we found that **ScpBnd-Rel** was the best scheme overall, illustrating the advantage of having more informed functional error information over the constant-based ones.

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

As far as we know, there has been little focus on subproblem ordering in AND/OR Best-First search in recent literature and the work presented here illustrates that there is potential to be realized by ordering the subproblems in a better informed manner. Our ideas presented here generalize to any type of AND/OR search. In particular, various memory-efficient A* variants (e.g. IDA*, RBFS) [29, 28] use the idea of repeatedly *deleting and re-expanding* nodes, which would potentially also benefit from the savings yielded by better subproblem orderings.

Chapter 4

Dynamic FGLP Heuristics

In the work presented in this thesis so far, we use static mini-bucket heuristics for all the search algorithms. To reiterate, the typical setup is to compile the heuristics before search begins as a preprocessing step, then use table lookups over the generated mini-bucket messages to obtain heuristic values for each node that is generated. While this has the advantage of low overhead in node expansion, the quality of the heuristics are bounded by the amount of memory available to store the pre compiled heuristics. More importantly, static heuristics do not capture any special structure that may be induced into a problem when it is partially instantiated.

This chapter explores the alternative of computing heuristics *dynamically* under the OR search framework. In contrast to the table lookups of static heuristics, we compute the heuristic on the subproblem induced by the current partial conditioning during search, thus allowing additional structure to be exploited. However, this means that we must also keep the computational cost of generating heuristics low. For example, running MBE with a high i-bound at every node generation is unlikely to be cost-effective. In the constraint optimization literature, one efficient approach is maintaining *soft arc consistency* (SAC) [30] during search. Maintaining SAC is a method of *re-parameterizing* the problem by shifting costs from higher arity functions toward lower arity functions, which bounds the problem with a single nullary function that has cost shifted into it. One of these algorithms is optimal

soft arc consistency (OSAC), which formulates finding the set of re-parameterizations as a linear programming (LP) task [10]. However, maintaining OSAC entails solving a large LP for each search node and thus not cost-effective. Another algorithm is virtual arc-consistency, which finds a sequence of cost shifts to tighten the bound based on a connection with classical arc consistency, yielding an iterative cost-shifting method [9].

In other literature, there are several iterative approximation techniques based on solving the LP relaxation of a graphical model [48]. This initial work established connections between these LP relaxations and message-passing, which led to coordinate descent update algorithms such as max-product linear programming (MPLP) [23].

As we first introduced in the background of Chapter 1, the ideas of cost-shifting were used to tighten the bounds generated by MBE heuristics [25]. One of these methods employs a similar algorithm to MPLP, known as *factor graph linear programming/join graph linear programming* (FGLP/JGLP) as a preprocessing step on the original problem. Another scheme is *MBE with moment-matching* (MBE-MM), an enhanced version of MBE that includes cost-shifting to enforce consistency between the duplicated variables. However, all these schemes were used so far in the context of static heuristics.

In this work, we aim to 1) explore the use of FGLP as a heuristic generator *dynamically* for every node during the search, and compare with the most advanced statically generated heuristics as in [25], and 2) to combine both static and dynamic schemes into a single, potentially more powerful heuristic for branch-and-bound. While generating dynamic heuristics based on FGLP is closely related to the soft-arc consistency algorithms such as those in the *toulbar2*[1] solver, our work provides an alternative based on techniques that come from the LP literature. In particular, FGLP solves a problem that is identical to that of the LP for OSAC, which we show later. Since FGLP is an LP coordinate descent algorithm,

[1] `http://mulcyber.toulouse.inra.fr/projects/toulbar2/`

it allows us to aim towards an optimal re-parameterization, yet terminating short of solving it to optimality, based on a given computation budget. We will compare our presented dynamic re-parameterization schemes against the state-of-the art static mini-bucket based schemes. We also present and experiment with a combination of the two approaches (the static, mini-bucket-based and the dynamic, re-parameterization-based).

We present empirical results showing that on some problem instances for which static mini-bucket evaluation is quite weak due to memory restrictions, the dynamic FGLP scheme can prune the search space far more effectively, and in some cases this is carried out in a cost-effective manner despite the significant overhead inherent in the dynamically generated heuristics. We acknowledge however that the overhead of our dynamic re-parameterization is often quite prohibited limiting its effectiveness both when applied in a pure form and within a hybrid scheme.

We present the background in section 4.1, reviewing the framework of bounding via re-parameterization and presenting the FGLP algorithm. In section 4.2, we present a new version of the FGLP algorithm that employs scheduled updates, taking into account the fact that it is used as a dynamic heuristic. In section 4.3, we present experimental results comparing various settings of the dynamic and static heuristics and their performance in OR search spaces. We conclude in the last section.

4.1 Background: Factor Graph Linear Programming

We discussed cost-shifting methods in the background in section 1.2.3.4. We briefly re-state some of the concepts here once again to put FGLP, the core building block of the work in this chapter, into context.

Recall that cost-shifting methods start from considering a bound the min-sum objective by

exchanging the min and sum operators yielding

$$\min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot) \geq \sum_{f_j \in \mathbf{F}} \min_{\mathbf{x}} f_j(\cdot), \tag{4.1}$$

which can be viewed as optimizing each function independently. One view of this is that each function has its own copy of each variable so each is optimized independently.

By introducing a collection of functions $\{\lambda_{f_j}(X_i) | f_j \in \mathbf{F}_i\}$ for each variable X_i and enforcing that $\forall X_i, \quad \sum_{f_j \in \mathbf{F}_i} \lambda_{f_j}(X_i) = 0$ (so the global cost function does not change), we can define a re-parameterization of the graphical model as follows

$$\begin{aligned} C^* &= \min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} f_j(\cdot) \\ &= \min_{\mathbf{x}} \sum_{f_j \in \mathbf{F}} \left(f_j(\cdot) + \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i) \right) \\ &\geq \sum_{f_j \in \mathbf{F}} \min_{\mathbf{x}} \left(f_j(\cdot) + \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i) \right) \end{aligned} \tag{4.2}$$

The goal to optimize over $\mathbf{\Lambda} = \{\lambda_{f_j}(X_i) | f_j \in \mathbf{F}_i, X_i \in \mathbf{X}\}$ in order to maximize (4.2). One common class of optimization methods is based on coordinate descent on each variable X_i. Known as *LP-tightening* [25], a choice of $\lambda_{f_j}(X_i)$ that maximizes with respect to X_i is making all min-marginals of the updated functions equal, where the min-marginal is defined as $\gamma_{f_j}(X_i) = \min_{S_{f_j} \setminus X_i} f_j(\cdot)$. We also define the *average min-marginal* over $f_j \in \mathbf{F}_i$ by

$$\bar{\gamma}_{\mathbf{F}_i}(X_i) = \frac{1}{|\mathbf{F}_i|} \sum_{f_j \in \mathbf{F}_i} \gamma_{f_j}(X_i)$$

Formally, we choose

$$\lambda_{f_j}(X_i) = \bar{\gamma}_{\mathbf{F}_i}(X_i) - \gamma_{f_j}(X_i) \tag{4.3}$$

which can be shown to solve equality between the min-marginals.

The coordinate descent proceeds by repeatedly applying LP-tightening over different variables, thus updating a subset of the functions in the graphical model in each iteration such that their min-marginals become equal. This process is carried out until convergence.

We now present the FGLP algorithm, which works by using the LP-tightening update.

Algorithm 15: FGLP($\mathcal{M}, m$) [25]

Input: Graphical Model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, where f_j is a function defined on variables S_{f_j}, number of iterations m

Output: Re-parameterized factors $\mathbf{F}'$, bound on optimum value

for *m iterations* **do**
  foreach *variable* X_i **do**
    Let $\mathbf{F}_i = \{f_j | X_i \in S_{f_j}\}$
    $\forall f_j \in \mathbf{F}_i$, *compute min-marginals:*
      $\gamma_{f_j}(X_i) = \min_{S_{f_j} \setminus X_i} f_j(\cdot)$
    $\bar{\gamma}_{\mathbf{F}_i}(X_i) = \frac{1}{|\mathbf{F}_i|} \sum_{f_j \in \mathbf{F}_i} \gamma_{f_j}(X_i)$
    $\forall f_j \in \mathbf{F}_i$, *update parameterization:*
      $f_j(\cdot) \leftarrow f_j(\cdot) + \bar{\gamma}_{\mathbf{F}_i}(X_i) - \gamma_{f_j}(X_i)$
return Re-parameterized factors $\mathbf{F}'$ and bound $\sum_{f_j \in \mathbf{F}} \min_{\mathbf{x}} f_j(\cdot)$

We show the pseudocode of FGLP in Algorithm 15. In contrast to the original presentation in [25], we include a parameter m to explicitly control the number of iterations, where an iteration is defined as a single loop over all the variables.

The time complexity of FGLP for a single iteration is $O(n \cdot |\mathbf{F}_s| \cdot l)$, where n is the number of variables, k is the maximum domain size, $|\mathbf{F}_s| = \max_i |\mathbf{F}_i|$ is the largest number of functions having the same variable X_i in their scopes, and a is the maximum arity of the functions, and $l \leq k^a$ bounds the number of entries in a single function. The space complexity is $O(|\mathbf{F}| \cdot l)$, the size of the input graphical model [21].

4.2 Search with Dynamic FGLP Heuristics

In this work, we re-purpose the FGLP algorithm as a heuristic generator for branch-and bound. The main idea is to not only apply it to the original problem, but also to each conditioned subproblem during search. While we can generate valid heuristics by treating FGLP as a black-box heuristic applied independently to each node in the search space, we can make the process more efficient by exploiting the relationship along a path in the search tree. In particular, any bound tightening performed earlier at a particular node can be inherited by its children. This introduces a trade-off between time and accuracy for each node expansion. In general, this process is similar to the method of heuristic generation in the WCSP literature known as *maintaining soft-arc consistency* [30]. However, unlike our setup, trade-offs explored in the WCSP literature focus on varying the strengths of soft-arc consistency.

Algorithm 16: BB-FGLP($\mathcal{M}$,$\mathcal{O}$,m,UB)

Input: Graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, where $f_j \in \mathbf{F}$ is a function defined on variables S_{f_j}, variable ordering $\mathcal{O}$, number of iterations for FGLP m, current upper bound UB

Output: Optimal cost to $\mathcal{M}$

1 **if** $\mathbf{X} = \emptyset$ **then return** 0;
2 **else**
3 Apply FGLP on $\mathcal{M}$ for m iterations to generate re-parameterized $\mathcal{M}' = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}' \rangle$
4 $X_k \leftarrow SelectVar(\mathbf{X})$ according to $\mathcal{O}$
5 $h(X_k) \leftarrow \sum_{f'_j \in \mathbf{F}'} \min_{S_{f_j} \setminus X_k} f'_j(\cdot)$; `// Compute heuristic for each` x_k
6 **foreach** $x_k \in D_k$ **do**
7 **if** $c'(X_k, x_k) + h(X_k = x_k) < UB$ **then**
8 $UB \leftarrow \min(UB, c'(X_k, x_k) + \text{BB-FGLP}(\mathcal{M}'(x_k), \mathcal{O}, m, UB - c'(X_k, x_k)))$
9 **return** UB

Our algorithm incorporating Branch-and-Bound (BB) with FGLP is presented in Algorithm 16. It finds the optimal solution to $\mathcal{M}$ on an OR search tree. It takes in an upper bound UB, which is set to ∞ at the root, and is used as a pruning threshold. Line 1 is a self-explanatory

base case. Otherwise we perform the steps in lines 2-9. First, line 3 re-parameterizes the problem by running FGLP for the specified number of iterations, creating problem $\mathcal{M}'$. In line 4, we select the variable based on the variable ordering. Line 5 computes the heuristic for each value x_k of variable X_k in a similar way to to the bound on C^* shown in Equation 4.1, except we do not minimize over X_k. This generates a function of a single variable, X_k, which provides a heuristic value for each x_k. Lines 6-8 solves each subproblem formed by conditioning X_k in $\mathcal{M}'$ with x_k (denoted $\mathcal{M}'(x_k)$). Within this loop, line 7 is a pruning check which skips the subproblem based on its heuristic value and the arc cost. Line 8, updates the UB value by recursively calling BB on the subproblem using the current UB value. Note that the arc cost, denoted $c'(X_k, x_k)$, is defined by the *re-parameterized* function rather than the original problem. Finally, line 9 returns the value UB, which is the optimal solution to $\mathcal{M}$.

The key difference here compared with regular BB for graphical model problems is that we use $\mathcal{M}'$ in the recursion, which is the *re-parameterized* version of problem $\mathcal{M}$, thus any updates performed by FGLP can by inherited by all children. This step allows FGLP to start at a state potentially near convergence, thus maximizing the effectiveness of the m iterations performed each time (or running fewer if a given convergence criterion is met).

Theorem 4.1 (**complexity of BB-FGLP**). *Given a problem with n variables having a maximum domain size of k with functions $\mathbf{F}$ having maximum arity a, the time complexity of BB-FGLP is $O(N \cdot m \cdot n \cdot |\mathbf{F}_s| \cdot l)$ and the space complexity is $O(n \cdot |\mathbf{F}| \cdot l)$, where $N \leq k^n$ bounds the size of the search space, $|\mathbf{F}_s| = \max_i |\mathbf{F}_i|$ is the largest number of functions having the same variable X_i in their scopes, and $l \leq k^a$ bounds the number of entries in a single function.*

Proof. For each node expansion, since FGLP is run once with m iterations, the time complexity is $O(n \cdot m \cdot |\mathbf{F}_s| \cdot l)$. Thus the total time complexity over the search space is $O(N \cdot n \cdot m \cdot |\mathbf{F}_s| \cdot l)$

Each recursive call needs to make a copy of problem $\mathcal{M}'$ before conditioning. The space required to store a copy of $\mathcal{M}'$ is bounded by $O(|\mathbf{F}| \cdot l)$. Since we store a copy for each node along a depth-first path which is at most the maximum depth n, the space complexity is $O(n \cdot |\mathbf{F}| \cdot l)$. □

4.2.1 Improving FGLP for Branch-and-Bound

In the context of branch-and-bound, we have additional assumptions that can be made to improve the performance of FGLP in the context of heuristic generation. In particular, since we inherit the re-parameterization performed by FGLP prior to conditioning, it is likely that the best candidates for re-parameterization in the conditioned problem are the functions that were affected by the conditioning. At the same time, a standard round-robin schedule may be wasting computation on updating parts of the problem that are already near convergence. We address these issues in two steps as follows.

4.2.1.1 Normalization

The LP-tightening update used in FGLP (Equation 4.3) averages the min-marginals of all functions. This has the effect of spreading the cost of the minimum of each function across all functions. Consequently, all functions may be updated if changes via conditioning are made to a small subset of the functions. This is a key step in the context of search with dynamic heuristics. In order to reduce the number of updates needed in this situation, we normalize the re-parameterized functions such that their minimum is zero. The LP is redefined to enforce this condition below.

For each variable X_i, we introduce an additional constant λ_{X_i}, therefore $\Lambda_i = \{\lambda_{f_j}(X_i) | f_j \in \mathbf{F}_i\} \cup \{\lambda_{X_i}\}$.

The new constraint set is as follows:

$$\forall X_i \in \mathbf{X}, \quad \lambda_{X_i} + \sum_{f_j \in \mathbf{F}_i} \lambda_{f_j}(X_i) = 0 \tag{4.4}$$

which enforces no change to the global function of the graphical model, as before. Next,

$$\forall f_j \in \mathbf{F}, \quad \min_{\mathbf{x}} \left(f_j(\cdot) + \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i) \right) = 0 \tag{4.5}$$

is the *normalization constraint* enforcing that the resulting re-parameterization to the functions have 0 as their minimum.

The λ_{X_i} constants represent the cost that can be shifted out of all $\lambda_{f_j}(X_i)$ into a *nullary function* $f_\emptyset$, which represents the global lower bound.

We now rewrite the objective (4.2) to include the λ_{X_i} terms:

$$C^* \geq \sum_{X_i} \lambda_{X_i} + \sum_{f_j \in \mathbf{F}} \min_{\mathbf{x}} \left(f_j(\cdot) + \sum_{X_i \in S_{f_j}} \lambda_{f_j}(X_i) \right) \tag{4.6}$$

From the normalization constraint (4.5), we can drop the second term, yielding

$$= \sum_{X_i} \lambda_{X_i} \tag{4.7}$$

As before, $\mathbf{\Lambda} = \{\Lambda_i | X_i \in \mathbf{X}\}$ and the objective is then to find an optimal set of functions $\mathbf{\Lambda}$ to maximize (4.2), which can be interpreted as finding a re-parameterization that shifts as much cost as possible to $f_\emptyset$.

The previous LP-tightening update (Equation 4.3) no longer satisfies the constraints, so we derive a new, but similar update as follows. Here, we iteratively optimize over $\lambda_{X_i}, \lambda_{f_j}(X_i)$,

fixing all other $\lambda_{X_k}, \lambda_{f_j}(X_k)$ $s.t.$ $X_k \neq X_i$. So we want to solve the following local optimization

$$\begin{aligned}
&\max_{\lambda_{X_i}, \lambda_{f_j}(X_i)} \left[\lambda_{X_i} + \left(\sum_{f_j \in \mathbf{F}_i} \min_{S_{f_j}} f_j(\cdot) + \lambda_{f_j}(X_i) \right) \right] \\
= &\max_{\lambda_{X_i}, \lambda_{f_j}(X_i)} \left[\lambda_{X_i} + \left(\sum_{f_j \in \mathbf{F}_i} \min_{X_i} \left[\min_{S_{f_j} \setminus X_i} f_j(\cdot) + \lambda_{f_j}(X_i) \right] \right) \right]
\end{aligned}$$

Replacing terms with the min-marginal $\gamma_{f_j}(X_i)$,

$$\begin{aligned}
&= \max_{\lambda_{X_i}, \lambda_{f_j}(X_i)} \left[\lambda_{X_i} + \left(\sum_{f_j \in \mathbf{F}_i} \min_{X_i} \left[\gamma_{f_j}(X_i) + \lambda_{f_j}(X_i) \right] \right) \right] \\
&\leq \max_{\lambda_{X_i}, \lambda_{f_j}(X_i)} \left[\lambda_{X_i} + \left(\min_{X_i} \sum_{f_j \in \mathbf{F}_i} \left[\gamma_{f_j}(X_i) + \lambda_{f_j}(X_i) \right] \right) \right]
\end{aligned} \tag{4.8}$$

Here, if we choose $\lambda_{f_j}(X_i) = \bar{\gamma}_{\mathbf{F}_i}(X_i) - \gamma_{f_j}(X_i) = \frac{1}{|\mathbf{F}_i|} \sum_{f'_j \in \mathbf{F}_i} \gamma_{f'_j}(X_i) - \gamma_{f_j}(X_i)$ as before, the normalization constraint (4.5) is violated since this choice would yield

$$\begin{aligned}
&\min_{X_i} \left[\gamma_{f_j}(X_i) + \bar{\gamma}_{\mathbf{F}_i}(X_i) - \gamma_{f_j}(X_i) \right] \\
= &\min_{X_i} \left[\bar{\gamma}_{\mathbf{F}_i}(X_i) \right]
\end{aligned}$$

To enforce this constraint, we subtract $\bar{\gamma}_{\mathbf{F}_i}(X_i)$ with its minimum $\min_{X'_i} \bar{\gamma}_{\mathbf{F}_i}(X'_i)$ yielding

$$\min_{X_i} \left[\bar{\gamma}_{\mathbf{F}_i}(X_i) - \min_{X'_i} \bar{\gamma}_{\mathbf{F}_i}(X'_i) \right] = 0$$

Thus, we choose to assign $\lambda_{f_j}(X_i)$ as

$$\lambda_{f_j}(X_i) = \bar{\gamma}_{\mathbf{F}_i}(X_i) - \min_{X_i'} \bar{\gamma}_{\mathbf{F}_i}(X_i') - \gamma_{f_j}(X_i). \tag{4.9}$$

Next, we also need to choose λ_{X_i}. From constraint (4.4), we can directly solve for λ_{X_i}.

$$\lambda_{X_i} + \sum_{f_j \in \mathbf{F}_i} \lambda_{f_j}(X_i) = 0$$

$$\lambda_{X_i} = - \sum_{f_j \in \mathbf{F}_i} \lambda_{f_j}(X_i)$$

Substituting in the $\lambda_{f_j}(X_i)$ terms we chose (4.9) and rearranging, we obtain

$$\begin{aligned} \lambda_{X_i} &= - \sum_{f_j \in \mathbf{F}_i} \left[\bar{\gamma}_{\mathbf{F}_i}(X_i) - \min_{X_i'} \bar{\gamma}_{\mathbf{F}_i}(X_i') - \gamma_{f_j}(X_i) \right] \\ &= \sum_{f_j \in \mathbf{F}_i} \min_{X_i'} \bar{\gamma}_{\mathbf{F}_i}(X_i') = |\mathbf{F}_i| \cdot \min_{X_i'} \bar{\gamma}_{\mathbf{F}_i}(X_i') \end{aligned}$$

which serves to shift an equal amount of cost from each the average min-marginals into $f_\emptyset$.

Algorithm 17: Normalized FGLP-Variable-Update(X_i)

Input: Graphical Model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, where f_j is a function defined on variables S_{f_j}, variable X_i to update, current lower bound $f_\emptyset$

Output: Re-parameterized factors $\mathbf{F}'$

Let $\mathbf{F}_i = \{f_j : X_i \in S_{f_j}\}$

$\forall f_j \in \mathbf{F}_i$, *compute min-marginals:*

  $\gamma_{f_j}(X_i) = \min_{S_{f_j} \setminus X_i} f_j(\cdot)$

$\bar{\gamma}_{\mathbf{F}_i}(X_i) = \frac{1}{|\mathbf{F}_i|} \sum_{f_j \in \mathbf{F}_i} \gamma_{f_j}(X_i)$

$\forall f_j \in \mathbf{F}_i$, *update parameterization:*

  $f_j(\cdot) \leftarrow f_j(\cdot) + \bar{\gamma}_{\mathbf{F}_i}(X_i) - \min_{X_i} \bar{\gamma}_{\mathbf{F}_i}(X_i) - \gamma_{f_j}(X_i)$

  $f_\emptyset \leftarrow f_\emptyset + \min_{X_i} \bar{\gamma}_{\mathbf{F}_i}(X_i)$

return Re-parameterized $\mathbf{F}'$ containing updates to f_j with X_i in their scope

Algorithm 17 presents changes to the FGLP algorithm in terms of a single variable update, using the above choices for λ_{X_i} and $\lambda_{f_j}(X_i)$. Everything is identical the original FGLP

algorithm (Algorithm 15) until lines 6-7. Here, we also subtract off the minimum of the average min-marginal in order to enforce that the minimum of the resulting re-parameterized $f_j(\cdot)$ is 0. These quantities are then shifted into the nullary function $f_\emptyset$.

Theorem 4.2 (**complexity of Normalized FGLP-Variable-Update).** *Given functions* $\mathbf{F_i} \subseteq \mathbf{F}$ *containing variable* X_i *in a graphical model, let* k *be the maximum domain size,* a *be the maximum arity, and* $l \leq k^a$ *bound the number of entries in a single function, then the time complexity of Normalized FGLP-Variable-Update is* $O(|\mathbf{F}_i| \cdot l)$ *and the space complexity is also* $O(|\mathbf{F}_i| \cdot l)$.

Proof. Computing each min-marginal takes $O(l)$ time. Since we need to do this $|\mathbf{F}_i|$ times, the total time complexity is $O(|\mathbf{F}_i| \cdot l)$. The space complexity is bounded by the size of the functions to be updated, thus we also have $O(|\mathbf{F}_i| \cdot l)$. □

Note that the objective we derived (Equation 4.7), is equivalent to the OSAC objective in [10]. In that work, enforcing OSAC was accomplished by using standard LP solvers such as CPLEX. It was only applied to the original problem (i.e. at the root node of the search space), being deemed to costly to maintain during search. However, this equivalence implies that FGLP, which can be fine tuned to stop short of reaching the optimal solution to the LP, is an alternative method that allows us to aim at maintaining OSAC, but stop short of it if it is not cost-effective. Our update is also closely related to the one used in the MPLP algorithm [23].

4.2.1.2 Scheduling Updates

FGLP updates the variables in a fixed order during each iteration. To further reduce the number of updates required to tighten the bound, we propose a scheduling scheme that identifies the most important variables to update. There are a number of works which

aim to solve the general problem of update scheduling in related coordinate update based algorithms. This includes scoring methods for either choosing entire regions for updates [46] or adding entire clusters for higher order updates [3] for LP-based methods, but those methods are less suitable in the context of being embedded in a complete search algorithm which rapidly changes the functions by conditioning.

Thus, we consider Residual BP (RBP) [18], a variant of BP that defines a message update schedule which enables better convergence properties. In BP, different parts of the graphical model may converge at different rates when performing message updates. First, a *message norm* $\|\cdot\|$ is used to measure the distance between two messages, which is computed by treating the messages as vectors and taking vector norms (e.g. $L_1, L_2, L\infty,$). The schedule is based on maintaining a priority queue of messages by computing the *residual* of each message, defined as the message norm between the current message and the message after the update if it were applied. Thus, RBP computes the next message update for each message first, in order to obtain the residuals, and then applies the message with the largest residual.

We can apply a similar method to variable updates in FGLP. In order to avoid computing the updates beforehand, our priority for a given variable X_i are instead based on the previously applied re-parameterizations to its *neighboring* variables. We redefine an iteration of FGLP as a *single* variable update rather than a set of updates over all of the variables.

Definition 4.1 (**FGLP variable update priority).** *At iteration t, the* priority *for X_i, denoted p_{X_i}, is $\max_{f_j \in \mathbf{F}_i} \max_{k \in S_{f_j}} \|\lambda^{t^-}_{f_j}(X_k)\|$, where $\lambda^{t^-}_{f_j}(X_k)$ denotes the most recent re-parameterization over X_k performed on function f_j.*

We present the full version of the FGLP algorithm with normalization and scheduling called *prioritized FGLP* (pFGLP) in Algorithm 18. To stop the computation early we have two parameters: a tolerance ϵ that stops updates with the highest priority p_{X_i} if it is less than ϵ, and a maximum number of iterations m'. The algorithm loops to perform updates while the

Algorithm 18: pFGLP($\mathcal{M}, P, \epsilon, m'$)

Input: Graphical Model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, where f_j is a function defined on variables S_{f_j}, initial priorities $P = \{p_{X_i} | X_i \in \mathbf{X}\}$, convergence tolerance ϵ, maximum number of iterations m'

Output: Re-parameterized factors $\mathbf{F}'$, bound on optimal value $f'_\emptyset$, updated priorities $p'_{X_1}, ...p'_{X_n}$

1 **while** $\max_{X_i} p_{X_i} > \epsilon$ *and* # *iterations* $\leq m$ **do**
2 $\quad X_i \leftarrow \arg\max_{X_i} p_{X_i}$
3 $\quad p_{X_i} \leftarrow 0$
4 $\quad$ Let $\mathbf{F}_i = \{f_j : X_i \in S_{f_j}\}$
5 $\quad \forall f_j \in \mathbf{F}_i$, *compute min-marginals and shift their minimum into* $f_\emptyset$*:*
6 $\quad\quad \lambda_{f_j}(X_i) = \min_{S_{f_j} \setminus X_i} f_j(\cdot)$
7 $\quad\quad \lambda'_{f_j}(X_i) = \lambda_{f_j}(X_i) - \min_{x_i} \lambda_{f_j}(X_i)$
8 $\quad\quad f_\emptyset \leftarrow f_\emptyset + \min_{x_i} \lambda_{f_j}(X_i)$
9 $\quad \forall f_j \in \mathbf{F}_i$, *update parameterization::*
10 $\quad\quad f_j(\cdot) \leftarrow f_j(\cdot) - \lambda_{f_j}(X_i) + \frac{1}{|\mathbf{F}_i|} \sum_{f'_j \in \mathbf{F}_i} \lambda'_{f'_j}(X_i)$
11 $\quad\quad \forall X_k \in S_{f_j} \setminus X_i$, *update priorities:*
12 $\quad\quad\quad p_{X_k} \leftarrow \max(p_{X_k}, \| \frac{1}{|\mathbf{F}_i|} \sum_{f'_j \in \mathbf{F}_i} \lambda'_{f_j}(X_i) - \lambda_{f_j}(X_i) \|)$
13 **return** Re-parameterized factors $\mathbf{F}'$ and priorities $p'_{X_1}, ...p'_{X_n}$

maximum priority is greater than ϵ or the number of iterations m' has not been exceeded (line 1). In each iteration, the variable with the maximum priority is extracted and its priority value is set to 0 (lines 2-3). The following lines are those of FGLP-Variable-Update (Algorithm 17) (lines 4-10). Finally, we update all of the neighboring variable priorities with a priority based on the magnitude of the update just performed on variable X_i in lines 11-12.

When applied to the original problem (or the root of the search space), the initial priorities are initialized to ∞ to ensure each variable is updated at least once.

Theorem 4.3 (complexity of pFGLP). *The time complexity is* $O(m'|\mathbf{F}_s| \cdot l)$ *and the space complexity is* $O(\cdot|\mathbf{F}| \cdot l)$*, where* m' *is the number of iterations,* k *is the maximum domain size,* a *is the maximum function arity,* $|\mathbf{F}_s| = \max_i |\mathbf{F}_i|$ *is the largest number of functions having the same variable* X_i *in their scopes, and* $l \leq k^a$ *bounds the number of entries in a single function. The space complexity is* $O(|\mathbf{F}| \cdot l)$*.*

Proof. Each of the m' iterations is a single execution of Normalized-FGLP-Variable Update, thus yielding a time complexity is $O(m' \cdot |\mathbf{F}_s| \cdot l)$. The space complexity, like FGLP, is $O(|\mathbf{F}_s| \cdot l)$ the size of the input graphical model. □

As expected, the complexity is similar to that of FGLP, but the time complexity depends on updating m' variables rather then the coarser schedule which is mn variables, where n is a fixed quantity. Thus, pFGLP gives us a finer-grained control of the time and accuracy trade-off while performing more effective updates.

Algorithm 19: BB-pFGLP($\mathcal{M}, \mathcal{O}, P, \epsilon, m, UB$)

Input: Graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, where $f_j \in \mathbf{F}$ is a function defined on variables S_{f_j}, variable ordering $\mathcal{O}$, priorities $P = \{p_{X_i} | X_i \in \mathbf{X}\}$, convergence tolerance ϵ, number of iterations for each pFGLP m, current upper bound UB

Output: Optimal cost to $\mathcal{M}$

1 **if** $\mathbf{X} = \emptyset$ **then return** 0;
2 **else**
3 Apply pFGLP on $\mathcal{M}$ for m iterations using priorities P and tolerance ϵ to generate re-parameterized $\mathcal{M}'$ and updated priorities P'
4 $X_k \leftarrow SelectVar(\mathbf{X})$ according to $\mathcal{O}$
5 $h(X_k) \leftarrow \sum_{f'_j \in \mathbf{F}'} \min_{S_{f_j} \setminus X_k} f'_j(\cdot)$; `// Compute heuristic for each` x_k
6 **foreach** $x_k \in D_k$ **do**
7 **if** $c'(X_k, x_k) + h(X_k = x_k) < UB$ **then**
8 **foreach** $X_i \in \bigcup_{f'_j \in \mathbf{F}'_k} S_{f'_j}$ **do**
9 $p'_{X_i} \leftarrow \infty$
10 $UB \leftarrow$ $\min(UB, c'(X_k, x_k) + \text{BB-pFGLP}(\mathcal{M}'(x_k), \mathcal{O}, P', \epsilon, m, UB - c'(X_k, x_k)))$
11 **return** UB

For the sake of completeness, we present branch-and-bound once again, but using pFGLP instead of FGLP in Algorithm 19. We discuss only the changes compared to BB-FGLP (Algorithm 16) in the following. The algorithm's parameters are changed (from BB-FGLP) to include the priorities P, convergence tolerance ϵ, and iterations m (defined as single variable updates). Line 3 also generates an updated set of priorities P'. Before the recursive call to BB-FGLP, we reset all the priorities in P' to ∞ for variables in functions that would

be affected by the conditioning in lines 8-9. Indeed, the priorities play a larger role in this context by allowing pFGLP to continue where it left off as nodes are expanded.

4.2.2 Search on Graphs

All of the above algorithms assume search on an OR tree. Naturally, the next step is to consider search on an OR graph, which can be more compact. Graph search is often implemented by using a cache indexed by the instantiated context that stores the cost of optimally solved subproblems. However, due to the nature of re-parameterization which shifts costs between functions, the information stored in the cache may be invalidated. The main problem with the BB algorithms as presented is the use of the re-parameterized path costs $c'(X_k, x_k)$ rather than ones based on the original problem. Thus, any optimal cost of a subproblem is given in terms of the re-parameterized version instead of the original one.

In the following, we provide a simple analysis of the costs involved. Let $f^*(n) = g(n) + h^*(n)$ denote the optimal cost through n in terms of the original parameterization and $f^{*'}(n) = g'(n) + h^{*'}(n)$ denote the same quantity for some re-parameterization. Clearly, since $f^*(n) = f*'(n)$, we have $g(n) + h^*(n) = g'(n) + h^{*'}(n)$. The goal in caching is to record the cost of a subproblem rooted by n in terms of $h^*(n) = g'(n) - g(n) + h^{*'}(n)$. Since BB-FGLP returns $h^{*'}(n)$ if the problem rooted by n is solved (and thus cached), we only need to keep track of the complete path costs $g'(n)$ and $g(n)$ and apply the equation above. Thus, when we encounter node n again, we do not need call BB-FGLP on it again. This adjustment allows search to be performed on the same search graph defined by the original problem, while using FGLP heuristics. This can also be viewed as a way for FGLP (or any type of re-parameterizing method) to generate heuristics on subproblems in terms of the original local functions without losing the benefits of maintaining the various re-parameterized versions along a path of the search space.

Algorithm 20: BB-pFGLP-C($\mathcal{M}, \mathcal{O}, P, \epsilon, m, UB$)

Input: Graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, where $f_j \in \mathbf{F}$ is a function defined on variables S_{f_j}, variable ordering $\mathcal{O}$, priorities $P = \{p_{X_i} | X_i \in \mathbf{X}\}$, convergence tolerance ϵ, number of iterations for each pFGLP m, current upper bound UB, original path cost g, re-parameterized path cost g_r, current path π,

Output: Cost to $\mathcal{M} \leq UB$, flag *opt* indicating whether the cost returned is the optimal cost

Initialize: $Cache \leftarrow \emptyset$
if $\mathbf{X} = \emptyset$ **then return** $(0, true)$;
else
  Apply pFGLP on $\mathcal{M}$ for m iterations using priorities P and tolerance ϵ to generate re-parameterized $\mathcal{M}'$ and updated priorities P'
  $X_k \leftarrow SelectVar(\mathbf{X})$ according to $\mathcal{O}$
  $h(X_k) \leftarrow \sum_{f'_j \in \mathbf{F}'} \min_{S_{f_j} \setminus X_k} f'_j(\cdot) + g_r - g$; `// Compute heuristic for each` x_k
  $opt \leftarrow false$
  foreach $x_k \in D_k$ **do**
    if $c(X_k, x_k) + h(X_k = x_k) < UB$ **then**
      if $Cache(ctxt(X_k, \pi))$ *exists* **then**
        $childcost \leftarrow Cache(ctxt(X_k, \pi))$
        $childopt \leftarrow true$
      else
        foreach $X_i \in \bigcup_{f'_j \in \mathbf{F}'_k} S_{f'_j}$ **do**
          $p'_{X_i} \leftarrow \infty$
        $\pi' \leftarrow \pi \cup \{(X_k, x_k)\}$
        $g' \leftarrow g + c(X_k, x_k)$
        $g'_r \leftarrow g'_r + c'(X_k, x_k)$
        $(childcost, childopt) \leftarrow$ BB-pFGLP-C($\mathcal{M}'(x_k), \mathcal{O}, P', \epsilon, m, UB - c(X_k, x_k), g', g'_r, \pi'$))
      if $c(X_k, x_k) + childcost \leq UB$ **then**
        $UB \leftarrow c(X_k, x_k) + childcost$
        if *childopt* **then**
          $opt \leftarrow childopt$
  if *opt* **then**
    $Cache(ctxt(X_k, \pi)) \leftarrow UB$
  return (UB, opt)

Algorithm 20 presents a version of BB-pFGLP that searches the OR graph instead, which we call BB-pFGLP with caching (BB-pFGLP-C). To accomplish graph search, there are two main changes: 1) the adjustment to the heuristic value as described above, and 2) a cache which is indexed by the instantiations to the contexts. We discuss these changes one at a time.

To implement the first change, we include two additional parameters g and g_r to the input which are used to keep track of the path costs with respect the original functions and the re-parameterized functions, respectively. These are primarily used in line 6, which adds the quantity $g_r - g$ to the heuristic value for each x_k, thus $h(X_k)$ becomes a heuristic with respect to the original functions. Lines 17-18 are update g and g_r with the arc costs according to the original problem and re-parameterized problem, which are passed into the recursive call to BB-pFGLP-C on line 19. Furthermore, UB is adjusted based on the original parameterization as well.

With this change, it makes it so the arc costs of the search space are fixed, thus allowing us to implement graph search using a cache. The cache is indexed by instantiations to the contexts. To use the cache, we include a parameter π, which stores the partial assignment. We use $ctxt(X_k, \pi)$ to denote the instantiation to the context of X_k w.r.t. to π. The algorithm also outputs whether the cost returned is the optimal cost, since a cost should only be cached if it is. The main changes occur when we do not perform pruning (lines 10-23). Lines 10-12 retrieve the cost of the subproblem from the cache if it exists based on the context. Otherwise, in lines 13-19, we proceed as before, recursively calling BB-pFGLP on the child. Line 16 keeps track of the path π by appending the current child and is passed into the recursive call. The call to BB-pFGLP-C on line 19 then returns the tuple containing the cost of the subproblem and whether it is the exact optimal value. Lines 20-23 updates the UB value as before, but also sets a flag to note whether this new value is the optimal solution, which is only the case if the child's cost was also optimal. Lines 24-25 caches the

result UB's most recent update in line 21 was from an optimal child.

Theorem 4.4 (complexity of BB-pFGLP-C). *Given a problem with n variables having a maximum domain size of k with functions $\mathbf{F}$ having maximum arity a and pathwidth pw, the time complexity of BB-pFGLP is $O(N{\cdot}m{\cdot}n{\cdot}|\mathbf{F}_s|{\cdot}l)$ and the space complexity is $O(N{+}n{\cdot}|\mathbf{F}|{\cdot}l)$, where $N \leq nk^{pw}$ bounds the size of the search space, $|\mathbf{F}_s| = \max_i |\mathbf{F}_i|$ is the largest number of functions having the same variable X_i in their scopes, and $l \leq k^a$ bounds the number of entries in a single function.*

Proof. The time complexity is identical to that of BB-FGLP (**Algorithm 16**), except that the search space is smaller. The size of the OR graph is bounded by $O(nk^{pw})$ [33], thus $N \leq nk^{pw}$ in this case. The space complexity is increased by $O(N)$, since we may need to cache each node [33]. □

4.3 Experiments

Benchmark	# inst	n	k	w	pw	$\|F\|$	a
Block world	15	192	3	17	101	196	5
		2695	3	60	1699	2703	5
WCSPs	61	16	2	5	10	58	2
		1057	100	287	387	21787	3
Pedigree	22	298	3	15	86	335	4
		1015	7	39	357	1290	5
CPD	46	11	48	10	10	67	2
		115	198	114	114	6671	2

Table 4.1: Benchmark statistics for. # inst - number of instances, n - number of variables, w - induced width, pw - pathwidth, k - maximum domain size, $|F|$ - number of functions, a - maximum arity. The top value is the minimum and the bottom value is the maximum for that statistic.

In the following empirical evaluation, we experiment with FGLP and pFGLP as dynamic heuristic generators for Branch-and-Bound on the OR search graph, (**Algorithm 20** and a

version using FGLP instead). We used several benchmarks including **block world planning**, **WCSPs**, **pedigrees**, and **CPD** (computational protein design). **Table 4.1** presents the ranges of problem parameters for each benchmark. We report the pathwidth *pw* here, relevant for the BB-FGLP-C algorithm. We can see that the pathwidth is relatively high compared with the number of deadends, implying that most caches are dead caches [15].

For every instance, we ran FGLP for 30 seconds as preprocessing and stochastic local search (SLS) for a maximum of 10 seconds to generate an initial solution. The baseline algorithm uses the static MBE-MM heuristic [25]. Our dynamic heuristic schemes consisted of both pFGLP and FGLP as the heuristic generator, denoted d-pFGLP and d-FGLP respectively. For d-pFGLP, we ran between 100 and 150 iterations per node and for dFGLP we ran between 1 and 2 iterations (suffixed with N to indicate the number of updates in terms of single variable updates). We will refer to these as *pure* dynamic schemes. For each of d-pFGLP and d-FGLP, we also combined them together with MBE-MM by taking the maximum heuristic value of the two. We will refer to these as *hybrid* dynamic schemes. We used a time limit of 1 hour and memory limit of 1GB. All algorithms were implemented in C++ and ran on a 2.66Ghz processor.

4.3.1 Evaluating for Exact Solutions

In this section, we compared MBE-MM against our dynamic heuristic schemes for finding an exact solution. **Tables 4.2, 4.3, 4.4,** and **4.5** report the preprocessing time, total time, and number of OR nodes expanded for each scheme. **Figures 4.1, 4.2, 4.3,** and **4.4** summarize the performance of each scheme on each benchmark by plotting the number of instances solved as a function of time.

Block world planning									
problem (n,w,pw,k,fn,ar,ib)	MBEMM			d-pFGLP(100) d-pFGLP(150) d-FGLP(1N) d-FGLP(2N)			MBEMM+d-pFGLP(100) MBEMM+d-pFGLP(150) MBEMM+d-FGLP(1N) MBEMM+d-FGLP(2N)		
	ptime	time	nodes	ptime	time	nodes	ptime	time	nodes
bw2_2_4_5 (314,17,175,3,318,5,17)	36	36	0	32	36	6685	36	36	358
				32	36	6625	36	37	331
				32	39	7234	36	37	318
				32	38	6761	36	37	335
bw2_2_4_7 (436,17,249,3,440,5,17)	38	38	0	33	2608	1648250	38	39	437
				33	2549	1648182	38	39	437
				33	2862	1648452	39	93	41942
				33	2954	1648227	39	39	437
bw6_3_4_4 (600,34,354,3,607,5,16)	68	68	601	36	36	601	68	69	601
				36	36	601	68	69	601
				35	37	601	65	67	601
				36	37	601	68	70	601
bw6_3_4_5 (746,34,434,3,753,5,15)	69	72	464803	44	76	18675	69	76	2791
				43	76	18458	69	76	2534
				43	89	18631	70	89	6172
				43	88	18418	69	79	2440
bw6_3_4_6 (892,34,527,3,899,5,15)	76	610	66384363	47	392	96408	77	188	23648
				48	202	48058	76	165	17514
				47	398	66517	76	334	36176
				47	323	45198	79	285	21745
bw6_3_4_8 (1184,34,715,3,1191,5,15)	103	-	-	72	-	-	103	241	21994
				69	-	-	104	308	38259
				69	-	-	128	579	34338
				69	-	-	104	590	61578
bw7_4_4_6 (1627,58,1001,3,1635,5,13)	136	-	-	123	270	12992	135	-	-
				125	226	9361	132	-	-
				119	229	7899	136	-	-
				118	230	7043	132	-	-

Table 4.2: Exact evaluation for **block world** instances. The best times and node counts are boxed. If an optimal solution was not found, the time and nodes are marked as '-'. We also report the preprocessing time to the left of each total time. n - number of variables, w - induced width, pw - pathwidth, k - maximum domain size, fn - number of functions, ar - maximum function arity, ib - i-bound used for MBE-MM.

4.3.1.1 Block World Planning

Table 4.2 shows results for selected instances from the **block world planning** benchmark. This benchmark contains a fair amount of determinism that can be exploited with conditioning and methods such as FGLP. Some instances are solved by MBE-MM, since we could use an i-bound which is equal to the induced width. However, for harder instances, we see that the FGLP heuristics could be useful. For example, on *bw6_3_4_6*, all of the pure dynamic schemes have about half the runtime of MBE-MM, and search about an order of 10^3 fewer nodes. Yet, the hybrid scheme yields even better performance here, with even lower runtime and fewer nodes. Finally, the last two instances timed out when using static heuristics, but were solvable using either the pure or hybrid dynamic schemes.

Figure 4.1 shows the number of instances solved in this benchmark over time. Many of the

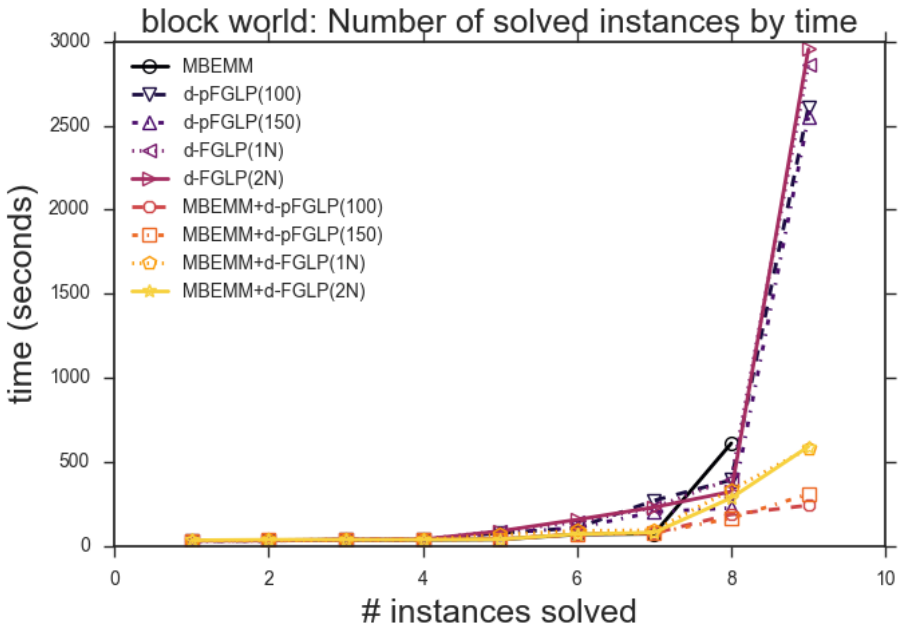


Figure 4.1: Number of instances solved by time over all **block world** instances. A line that is farther to the right and lower is better.

instances are quite easy for all of the methods. However, with the static MBE-MM heuristic, we solve just 8 instances. Overall, the hybrid scheme using d-pFGLP at 100 iterations is the best performer here.

4.3.1.2 WCSPs

Table 4.3 shows results for selected instances from the **WCSP** benchmark. Some instances in this benchmark have high domain sizes, thus drastically decreasing the highest feasible i-bound for MBE-MM. For example, on *bwt5ac.wcsp*, the domain size is 27, allowing the i-bound to be just 5. This problem is quite difficult, having an induced width of 61. Indeed, the static MBE-MM heuristic times out, but all of the dynamic heuristics were able to solve the problem relatively quickly in under 3 minutes. On most instances here, the number of nodes expanded using a dynamic heuristic is usually smaller. However, this is not always the case. For instance, on *queen5_5_4.wcsp*, the i-bound is sufficiently high, thus the MBE-MM

Weighted CSPs									
problem (n,w,pw,k,fn,ar,ib)	MBEMM			d-pFGLP(100) d-pFGLP(150) d-FGLP(1N) d-FGLP(2N)			MBEMM+d-pFGLP(100) MBEMM+d-pFGLP(150) MBEMM+d-FGLP(1N) MBEMM+d-FGLP(2N)		
	ptime	time	nodes	ptime	time	nodes	ptime	time	nodes
GEOM30a_3.wcsp (30,6,15,3,82,2,6)	30	30	0	30	3229	4239791	30	30	0
				30	-	-	30	30	0
				30	1735	8466176	30	30	0
				30	1506	4827347	30	30	0
myciel5g_4.wcsp (47,19,26,4,237,2,11)	76	243	33188442	30	-	-	76	3577	1645072
				30	-	-	76	-	-
				30	-	-	76	-	-
				30	-	-	76	3365	2743578
queen5_5_4.wcsp (25,18,21,4,161,2,11)	80	92	2773158	30	1026	353526	79	732	282346
				30	1335	318584	79	935	253766
				30	719	1075466	79	457	745422
				30	629	599570	74	414	462678
bwt5ac.wcsp (431,61,165,27,9246,2,5)	67	-	-	42	135	3686	67	158	3791
				41	112	2650	67	146	2614
				42	95	1527	67	120	1516
				42	87	934	68	113	940
driverlog05ac.wcsp (351,66,177,11,6493,2,6)	67	-	-	41	-	-	67	-	-
				41	-	-	67	-	-
				42	-	-	68	-	-
				41	3260	182543	70	3245	182349
satellite01ac.wcsp (79,19,34,8,744,2,7)	78	93	1986417	30	31	330	78	79	332
				30	32	295	78	79	294
				30	32	399	77	79	399
				30	31	199	77	78	199
zenotravel02ac.wcsp (116,18,41,19,1203,2,6)	61	61	1259	31	31	105	62	62	201
				31	31	99	61	62	190
				31	31	84	61	62	84
				31	31	62	67	67	62
CELAR6-SUB0.wcsp (16,7,10,44,58,2,4)	131	-	-	30	40	1176	139	151	13492
				30	41	1825	138	149	10808
				30	49	8776	137	155	9881
				30	42	5324	137	151	18690
graph05.wcsp (100,24,60,44,417,2,3)	54	54	2188	30	32	755	54	56	101
				30	32	755	54	56	101
				30	43	3062	54	55	101
				30	43	3062	54	55	101
54.wcsp (67,11,22,4,272,3,11)	30	30	0	30	882	342791	30	30	0
				30	1184	326976	30	30	0
				30	497	529156	30	30	0
				30	596	418783	30	30	0

Table 4.3: Exact evaluation for **WCSP** instances. The best times and node counts are boxed. If an optimal solution was not found, the time and nodes are marked as '-'. We also report the preprocessing time to the left of each total time. n - number of variables, w - induced width, pw - pathwidth, k - maximum domain size, fn - number of functions, ar - maximum function arity, ib - i-bound used for MBE-MM.

heuristic maintains its dominance. In the extreme when the i-bound is equal to the induced width (e.g. *GEOM30a_3.wcsp*), no search is needed, while it can be difficult for the dynamic heuristics to perform well.

Figure 4.2 summarizes by plotting the number of instances solved over time. The pure dynamic schemes are not cost-effective on half of the instances here, which correspond to problems where the i-bound used for MBE-MM was close to the induced width. The hybrid schemes are overall superior here, showing the ability to perform well on instances where MBE-MM is strong and where dynamic FGLP is strong. This is seen with how the hybrid

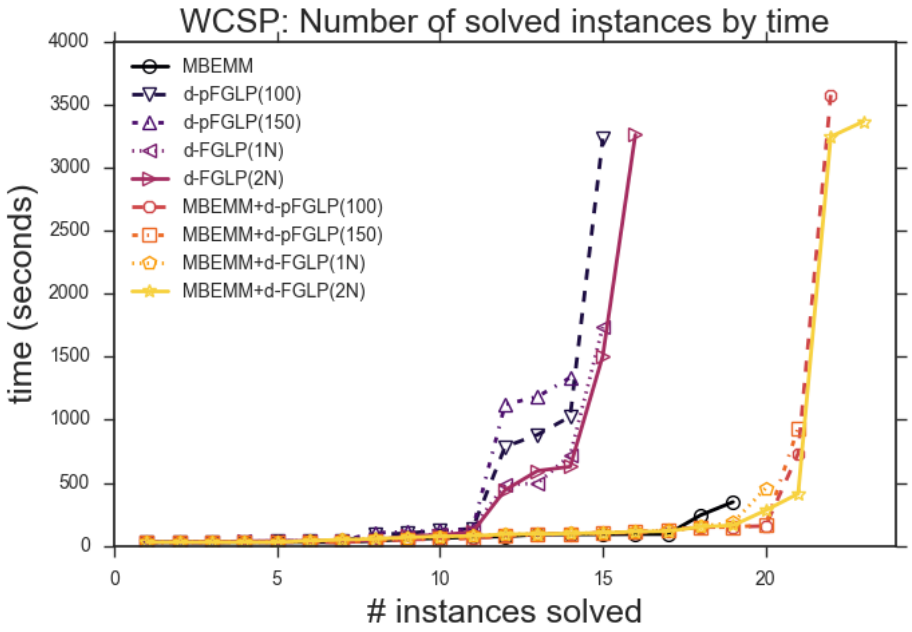


Figure 4.2: Number of instances solved by time over all **WCSP** instances. A line that is farther to the right and lower is better.

scheme maintains a low runtime for most instances comparable to MBE-MM while also solving more instances than MBE-MM.

4.3.1.3 Pedigree

Table 4.4 shows the results on the **pedigree** instances. We show a subset containing the hardest instances that could still be solved within the time limit in the table. The dynamic schemes perform poorly on this benchmark. Although, the hybrid scheme can produce stronger heuristics for some configuration of pFGLP/FGLP with a certain number of iterations on each of the instances that were solved, as demonstrated by the lower numbers of nodes expanded, it was never cost effective. None of the pure dynamic schemes solved any of the instances shown within the time limit.

Figure 4.3 plots the number of instances solved over time for this benchmark. Indeed, we see that the pure dynamic schemes only managed to solved 3 of the instances with worse

Pedigree networks									
problem (n,w,pw,k,fn,ar,ib)	MBEMM			d-pFGLP(100) d-pFGLP(150) d-FGLP(1N) d-FGLP(2N)			MBEMM+d-pFGLP(100) MBEMM+d-pFGLP(150) MBEMM+d-FGLP(1N) MBEMM+d-FGLP(2N)		
	ptime	time	nodes	ptime	time	nodes	ptime	time	nodes
pedigree7 (867,28,259,4,1069,4,18)	61	117	8020188	39	-	-	61	-	-
				39	-	-	61	-	-
				39	-	-	62	-	-
				39	-	-	62	-	-
pedigree13 (888,32,312,3,1078,4,20)	56	688	122555546	44	-	-	60	-	-
				44	-	-	60	-	-
				44	-	-	60	-	-
				45	-	-	60	-	-
pedigree25 (993,23,259,5,1290,5,21)	59	74	515627	49	-	-	62	75	3742
				49	-	-	62	113	17031
				49	-	-	62	3158	667854
				49	-	-	62	3561	653074
pedigree31 (1006,30,343,5,1184,5,17)	65	713	94359897	51	-	-	70	-	-
				51	-	-	70	-	-
				52	-	-	70	-	-
				52	-	-	70	-	-
pedigree34 (922,28,274,5,1161,4,15)	50	1134	144332505	39	-	-	51	-	-
				39	-	-	51	-	-
				39	-	-	51	-	-
				40	-	-	51	-	-
pedigree44 (644,24,257,4,812,5,20)	50	58	493931	37	-	-	50	332	78139
				37	-	-	50	344	78139
				37	-	-	50	1803	500221
				37	-	-	50	2283	502428
pedigree50 (478,16,168,6,515,4,10)	38	53	2346914	31	-	-	38	1828	756009
				31	-	-	38	1763	736027
				31	-	-	38	2140	760121
				32	-	-	38	3074	749242

Table 4.4: Exact evaluation for **pedigree** instances. The best times and node counts are boxed. If an optimal solution was not found, the time and nodes are marked as '-'. We also report the preprocessing time to the left of each total time. n - number of variables, w - induced width, pw - pathwidth, k - maximum domain size, fn - number of functions, ar - maximum function arity, ib - i-bound used for MBE-MM.

runtimes. While the hybrid scheme was able to extend the reach of the number of problems solved with respect to the dynamic heuristics, there were also cases where the overhead of the dynamic portion of the heuristic resulted in timeouts. The static MBE-MM scheme alone was overall dominant in terms of the both the number of problems solved and runtime.

4.3.1.4 Computational Protein Design

Table 4.5 shows results on selected instances of the **CPD** benchmark. This benchmark has the notable property of all the structures being complete graphs. Thus, the induced width of each problem is always $n - 1$. It also contains variables with very high domain size (up to 194). These factors make it challenging for static MBE-MM. Indeed, the pure dynamic schemes tend to perform better here. On many of the instances shown here, the number of variables is relatively low compared to other benchmarks, so static MBE-MM is still able to

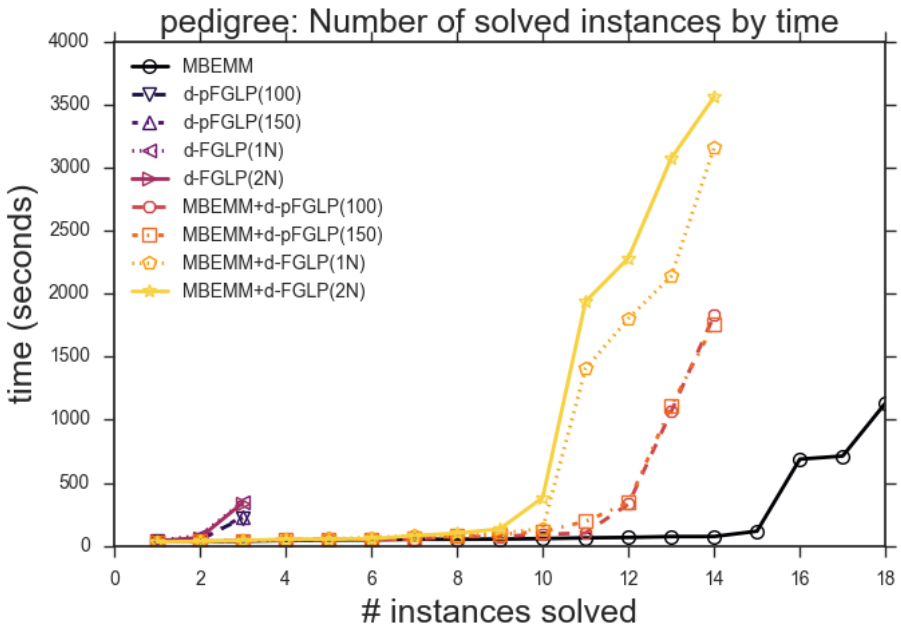


Figure 4.3: Number of instances solved by time over all **pedigree** instances. A line that is farther to the right and lower is better.

solve some of these instances. However, the instances tend to be trivially easy for dynamic FGLP heuristics in a number of cases, expanding only under 1000 nodes. For example, on *1BK2*, a significant amount of time was spend constructing the MBE-MM heuristic (about 40 seconds), followed by 12 more seconds for search to find the solution. However, the dynamic schemes provided nearly exact heuristics, solving the problem with just 4 seconds of search without the need for large preprocessing (other than the initial 30 seconds of FGLP we ran for every setting). On the much harder *2TRX*, the static MBE-MM heuristic times out, while all of the dynamic schemes manage to solve the problem.

Figure 4.4 plots the number of instances solved by time for these instances. While the static MBE-MM scheme performs decently overall, the pure dynamic schemes dominate it. The hybrid schemes here perform worse than the pure dynamic schemes, but still outperform MBE-MM alone. Overall, the pure dynamic schemes using pFGLP were best here, with similar performance between the two iteration settings.

Protein CPD									
problem (n,w,pw,k,fn,ar,ib)	MBEMM			d-pFGLP(100) d-pFGLP(150) d-FGLP(1N) d-FGLP(2N)			MBEMM+d-pFGLP(100) MBEMM+d-pFGLP(150) MBEMM+d-FGLP(1N) MBEMM+d-FGLP(2N)		
	ptime	time	nodes	ptime	time	nodes	ptime	time	nodes
1BK2.mat... (24,23,23,182,301,2,3)	70	82	329336	30	35	28	70	75	32
				30	34	25	70	74	27
				30	35	133	70	75	165
				30	34	85	71	77	96
1CSK.mat... (30,29,29,49,466,2,4)	62	62	1850	30	31	59	62	63	58
				30	31	36	64	66	36
				30	31	83	64	64	83
				30	30	66	66	66	66
1ENH.mat... (36,35,35,182,667,2,2)	53	78	139403	33	34	1	54	54	1
				33	34	1	53	53	1
				33	-	-	54	-	-
				35	-	-	54	-	-
1NXB.mat... (34,33,33,56,596,2,4)	98	98	35	30	31	35	99	99	35
				30	31	35	99	100	35
				30	30	35	102	102	35
				30	30	35	106	107	35
1PIN.mat... (28,27,27,194,407,2,3)	631	1150	14992222	31	960	582	712	1534	585
				31	1350	325	636	1612	322
				31	1589	10086	646	2291	10089
				31	1071	3049	645	1650	3054
1TEN.mat... (39,38,38,66,781,2,4)	119	123	218381	31	34	133	241	246	195
				30	37	176	131	136	139
				31	37	539	123	128	475
				30	37	390	133	139	348
2PCY.18p... (18,17,17,48,172,2,3)	52	52	29	30	50	33	52	53	19
				30	31	19	52	53	19
				30	30	34	52	52	20
				30	30	32	52	52	19
2TRX.mat... (61,60,60,186,1892,2,4)	176	-	-	32	604	3607	187	677	3584
				32	893	3250	225	994	3121
				32	300	4272	270	509	4042
				32	316	3343	250	575	3630

Table 4.5: Exact evaluation for **CPD** instances. The best times and node counts are boxed. If an optimal solution was not found, the time and nodes are marked as '-'. We also report the preprocessing time to the left of each total time. n - number of variables, w - induced width, pw - pathwidth, k - maximum domain size, fn - number of functions, ar - maximum function arity, ib - i-bound used for MBE-MM.

4.3.1.5 Discussion

Many of the results we saw performed as expected. Namely, on instances such as the **pedigrees** where the static MBE-MM scheme can execute with a sufficiently high i-bound, the static heuristics much more cost-effective. On the other hand, when the feasible i-bound for MBE-MM was drastically decreased from due to large variable domain sizes, the dynamic schemes perform are stronger and cost-effective, often solving instances that the static heuristic timed out on. In some cases, pFGLP performed worse than FGLP likely due to overhead in maintaining the priority queue for scheduling updates. However, it still maintained an advantage over MBE-MM whenever FGLP did. Overall, we demonstrate here that dynamic FGLP schemes should be used instead of static MBE-MM on hard problems with high variable domain sizes and high induced width.

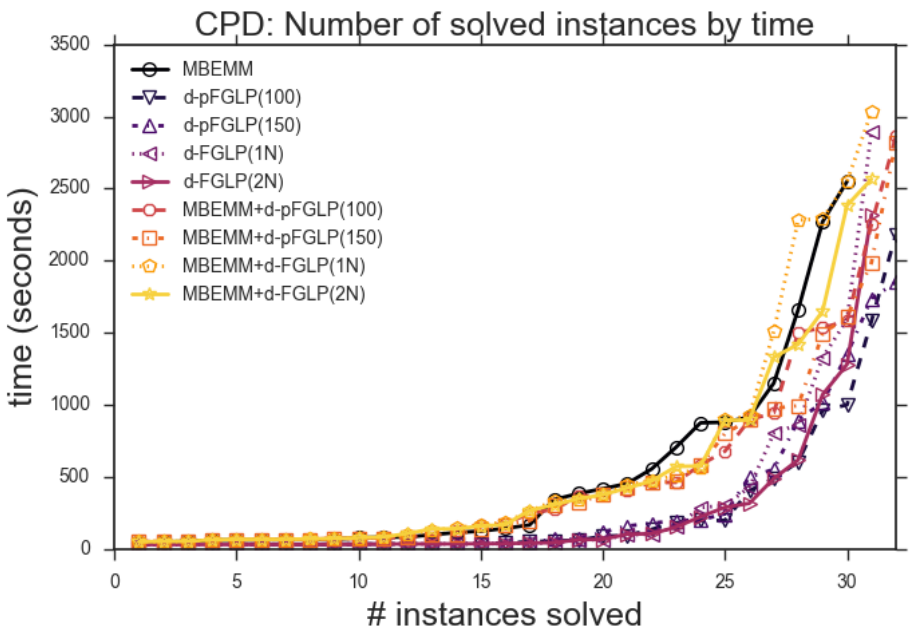


Figure 4.4: Number of instances solved by time over all **CPD** instances. A line that is farther to the right and lower is better.

4.3.2 Anytime Behavior

We next report on the anytime performance of the different heuristics. **Figures 4.5, 4.6, 4.7,** and **4.8** plot upper bounds on the optimal solution as a function of time on 2 selected instances from each benchmark. To reduce clutter, the legend on these plots abbreviates the algorithms: *MM* for *MBE-MM* and *F* for *FGLP*.

Tables 4.6, 4.7, 4.8, and **4.9** give an account of the anytime performance of all the hardest instances in each benchmark by reporting the percentage of instances for which a heuristic obtains the best solution at time bounds of 45, 60, 600, 1200, 2400, and 3600 seconds.

4.3.2.1 Block World Planning

Figure 4.5 shows the anytime performance on 2 selected instances from the **block world** benchmark. There is generally not a lot of change in the solution quality in these instances.

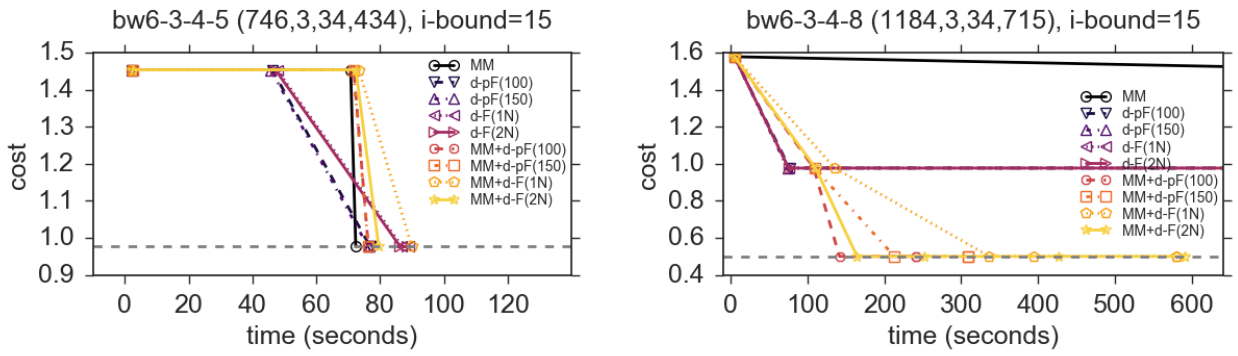


Figure 4.5: Upper bounds as a function of time for two instances from the **block world** benchmark. Lower is better. The dotted gray line indicates the optimal solution, if known.

	Time bound (sec)					
Block world,#inst = 10, n = 600-2695, k = 3-3, w = 34-60, pw = 354-1699						
Heuristic	45	60	600	1200	2400	3600
MBEMM	**1.00**	**1.00**	0.80	0.80	0.80	0.70
d-pFGLP(100)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	0.80
d-pFGLP(150)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	0.80
d-FGLP(1N)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	0.80
d-FGLP(2N)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	0.80
MBEMM+d-pFGLP(100)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	0.80
MBEMM+d-pFGLP(150)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	**0.90**
MBEMM+d-FGLP(1N)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	0.80
MBEMM+d-FGLP(2N)	**1.00**	**1.00**	**0.90**	**0.90**	**0.90**	0.80

Table 4.6: Percentage of instances for which a heuristic finds the best solution amongst all of the heuristics. #inst - the number of instances, n - number of variables, k - maximum domain size, w - induced width, pw - pathwidth.

On *bw6-3-4-5*, all of the algorithms begin with the same initial solution obtained by stochastic local search (SLS) at an early stage and reach the optimal solution at about the same time. Here, the static MBE-MM heuristic is slightly superior. On the more difficult *bw6-3-4-8*, we see a grouping in the behavior across the algorithms. The static MBE-MM heuristic does not manage to improve over the initial SLS solution, while the pure FGLP heuristics improve once. The hybrid heuristics improve once to the solution the pure FGLP heuristics obtain, then improve again to the optimal solution. On this particular instance, the hybrid scheme using pFGLP with 100 iterations performed best.

Table 4.6 provides winning percentages for each algorithm at various time points. All of the dynamic schemes have about the same performance at all times, except for the hybrid using pFGLP with 150 iterations at the final time point of 3600 where it performs better than the rest of the algorithms. The static MBE-MM heuristic starts to fall behind at 600 seconds. Like in the exact evaluation, the hybrid scheme gets the best of both the static and dynamic heuristics on this benchmark.

4.3.2.2 Pedigree

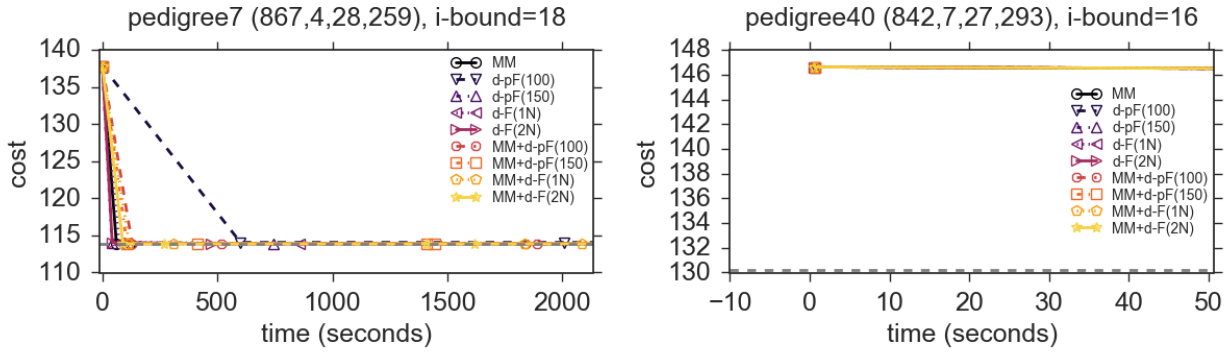


Figure 4.6: Upper bounds as a function of time for two instances from the **pedigree** benchmark. Lower is better. The dotted gray line indicates the optimal solution, if known.

Figure 4.6 shows the results for 2 representative instances from the **pedigree** benchmark. Like the previous benchmark, there is relatively little change in the upper bounds over time. On *pedigree7*, nearly all of the schemes obtain the optimal solution quickly and the rest of the time is spent exhausting the search space to prove optimality. For harder instances such as *pedigree40*, all algorithms began with the initial solution from SLS, but could not improve it over time.

Table 4.7 summarizes the anytime performance over the entire benchmark via winning percentages for each algorithm. As expected and also observed in the previous section on exact evaluation, the static MBE-MM heuristic is dominant, winning on almost all of the

	Time bound (sec)					
Pedigree,#inst = 15, n = 390-1006, k = 3-7, w = 16-39, pw = 125-357						
Heuristic	45	60	600	1200	2400	3600
MBEMM	0.93	**0.93**	**1.00**	**1.00**	**1.00**	0.93
d-pFGLP(100)	0.87	0.80	0.60	0.60	0.67	0.67
d-pFGLP(150)	0.93	0.87	0.60	0.67	0.73	0.67
d-FGLP(1N)	0.87	0.80	0.60	0.67	0.73	0.67
d-FGLP(2N)	**1.00**	0.87	0.73	0.73	0.80	0.73
MBEMM+d-pFGLP(100)	0.93	0.80	0.87	0.87	0.93	0.87
MBEMM+d-pFGLP(150)	0.93	0.80	0.87	0.87	0.93	0.87
MBEMM+d-FGLP(1N)	0.93	0.80	0.80	0.80	0.93	0.87
MBEMM+d-FGLP(2N)	0.93	0.80	0.80	0.80	0.87	**1.00**

Table 4.7: Percentage of instances for which a heuristic finds the best solution amongst all of the heuristics. #inst - the number of instances, n - number of variables, k - maximum domain size, w - induced width, pw - pathwidth.

instances for nearly all time points. Though there are time points at the beginning and end where one of the dynamic schemes is better than the static heuristic, it is clear that the static MBE-MM heuristic is preferred for the anytime performance on this benchmark.

4.3.2.3 WCSPs

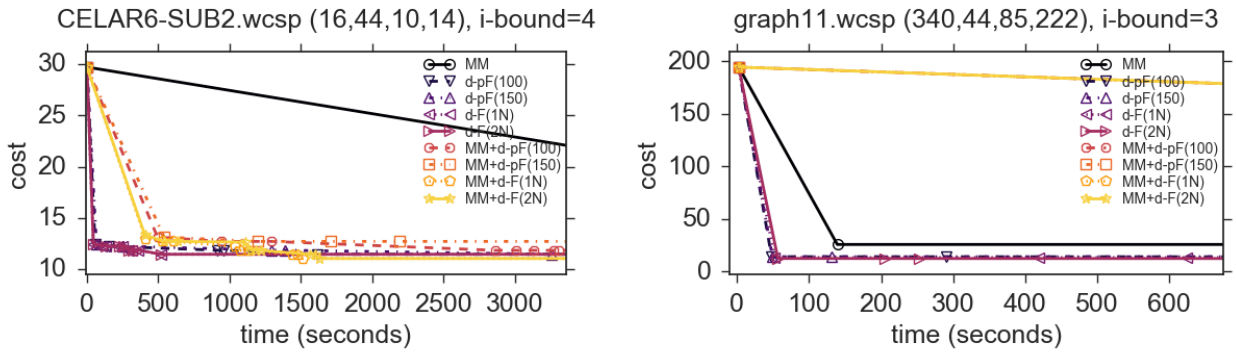


Figure 4.7: Upper bounds as a function of time for two instances from the **WCSP** benchmark. Lower is better. The dotted gray line indicates the optimal solution, if known.

Figure 4.7 shows the performance for 2 instances from the **WCSPs**. Here we, see that the pure dynamic schemes dominate on these two difficult instances. On the first instance,

	Time bound (sec)					
WCSP,#inst = 52, n = 14-1057, k = 2-100, w = 7-287, pw = 10-387						
Heuristic	45	60	600	1200	2400	3600
MBEMM	0.87	0.83	0.83	0.83	0.83	0.85
d-pFGLP(100)	0.92	0.88	0.83	0.83	0.83	0.83
d-pFGLP(150)	0.90	0.85	0.81	0.81	0.81	0.81
d-FGLP(1N)	0.90	0.87	0.83	0.83	0.81	0.81
d-FGLP(2N)	**0.96**	**0.94**	**0.90**	**0.90**	**0.88**	**0.88**
MBEMM+d-pFGLP(100)	0.83	0.79	0.77	0.79	0.79	0.79
MBEMM+d-pFGLP(150)	0.83	0.79	0.77	0.79	0.79	0.79
MBEMM+d-FGLP(1N)	0.85	0.81	0.79	0.81	0.83	0.83
MBEMM+d-FGLP(2N)	0.87	0.83	0.81	0.81	0.83	0.83

Table 4.8: Percentage of instances for which a heuristic finds the best solution amongst all of the heuristics. #inst - the number of instances, n - number of variables, k - maximum domain size, w - induced width, pw - pathwidth.

CELAR6-SUB2.wcsp, the pure dynamic heuristics obtain much better solution than the static heuristic early on. It is also better than the hybrid schemes here. However, with more time, the hybrids MBEMM+dFGLP obtain a slightly better solution. On the other hand, the hybrid scheme performs worst on *graph11.wcsp*, having a solution that is much worse than even the static heuristic alone. Still, the pure dynamic schemes are all better than the static heuristic. All variants of the pure dynamic scheme performed similarly here on both instances.

Table 4.8 gives an account of the anytime performance on all the instances. Performance is similar across the algorithms, but the pure d-FGLP(2N) scheme dominates the rest here for all time bounds. A large majority of the instances in this benchmark tend to be unfavorable for MBE-MM due to the large domain sizes present. In contrast with the results seen for evaluating exact solutions, the pure dynamic schemes work well in the anytime context on most of this benchmark.

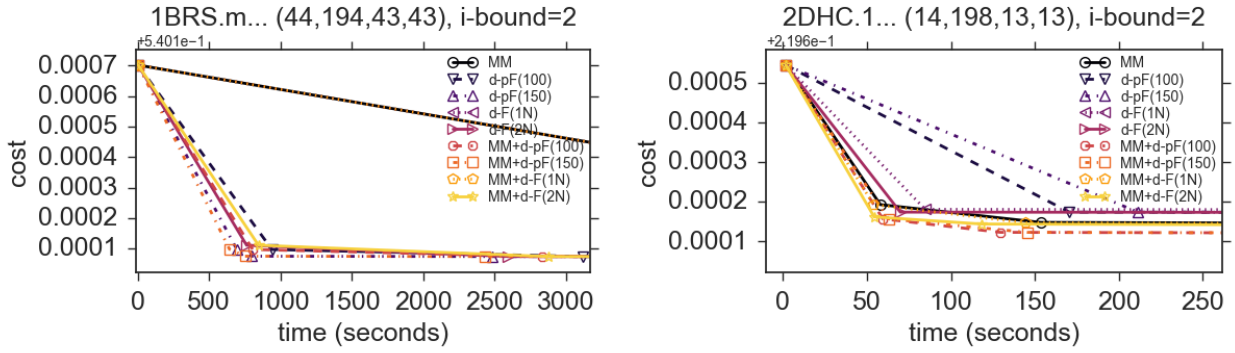


Figure 4.8: Upper bounds as a function of time for two instances from the **CPD** benchmark. Lower is better. The dotted gray line indicates the optimal solution, if known.

	Time bound (sec)					
CPD,#inst = 46, n = 11-115, k = 48-198, w = 10-114, pw = 10-114						
Heuristic	45	60	600	1200	2400	3600
MBEMM	0.26	0.28	0.65	0.76	0.78	0.78
d-pFGLP(100)	0.85	0.85	**0.93**	0.89	0.93	0.96
d-pFGLP(150)	0.78	**0.91**	**0.93**	**0.93**	**0.98**	**0.98**
d-FGLP(1N)	0.85	0.85	0.89	0.85	0.87	0.87
d-FGLP(2N)	**0.96**	**0.91**	0.91	0.91	0.93	0.93
MBEMM+d-pFGLP(100)	0.26	0.30	0.83	0.85	0.93	**0.98**
MBEMM+d-pFGLP(150)	0.26	0.28	0.85	0.91	0.96	**0.98**
MBEMM+d-FGLP(1N)	0.26	0.30	0.80	0.83	0.87	0.89
MBEMM+d-FGLP(2N)	0.26	0.30	0.83	0.85	0.93	0.96

Table 4.9: Percentage of instances for which a heuristic finds the best solution amongst all of the heuristics. #inst - the number of instances, n - number of variables, k - maximum domain size, w - induced width, pw - pathwidth.

4.3.2.4 Computational Protein Design

Figure 4.8 presents the anytime performance for 2 **CPD** instances. Almost all of the dynamic schemes behave similarly on these instances. On the *1BRS* instance, all but MBE-MM and MBE-MM+dFGLP(2N) have similar anytime profiles. There is, however, some variance, as MBE-MM+dFGLP(150) performs slightly better out of the rest. Still most of the schemes converge to the same solution given enough time. The other instance, *2DHC*, is easier with a lower induced width. Here, MBE-MM remains competitive with the dynamic

schemes. We also see that both d-pFGLP schemes perform worse than the rest here, while the MBE-MM+d-pFGLP schemes are best.

Table 4.9 summarizes the anytime performance. The pure d-FGLP(2N) scheme performs the best at low time bounds below 60 seconds. Notably, both MBE-MM alone and the hybrid schemes are very poor at low time bounds, due to the long preprocessing time for MBE-MM on these instances. At above 60 seconds, the pure d-pFGLP(150) scheme leads the way, providing the best solutions at all other time bounds. The hybrid schemes using d-pFGLP also catch up over time, eventually converging to the same performance.

4.3.2.5 Discussion

Many of the positive or negative impacts we observed carried over from the exact evaluation to this anytime evaluation. The same conclusions can be made, that hard problems having large domain sizes are prime candidates for dynamic FGLP heuristics. We also saw once again that performance was unfavorable on benchmarks where the static MBE-MM works well on like the **pedigrees**. Still, we demonstrated the strength of both FGLP as a pure dynamic scheme and as a component of a hybrid scheme with MBE-MM on the other benchmarks.

4.4 Conclusion

We demonstrated how to use FGLP as a dynamic heuristic generator. In particular, we derived a new version of FGLP with a schedule for updates and in the process, established its direct connection with optimal soft arc-consistency. We then conducted an empirical evaluation on using both the new and old version of FGLP as heuristics, as well as combining them with MBE-MM to generate a superior heuristic. This demonstrated how dynamic heuristics are preferable on many instances known to be difficult when using static heuristics.

Our pFGLP algorithm was not always superior to the FGLP algorithm alone when used as a dynamic heuristic generator. Though it converges in fewer iterations, it needs to maintain a priority queue, which requires extra work to maintain. We also did not experiment with running more than 150 iterations per node, which can be fewer than what FGLP would run in a single round-robin pass over the variables on a number of problems. Further empirical work tuning the number of iterations to each problem for pFGLP would shed more light on the relationship between pFGLP and FGLP.

The most important method we did not incorporate our heuristics into is search with dynamic variable orderings, which are proven to be effective for dynamic heuristics, as seen in solvers such as *toulbar*. Although, moving to dynamic variable orderings would not allow for a hybrid scheme, we expect the dynamic FGLP heuristics to perform much better when it is able to exploit structure with minimal amounts of conditioning, allowing for larger portions of the search space to be pruned off sooner. This would also allow a comparison of FGLP with the existing soft arc-consistency methods to know where FGLP stands in the spectrum of cost-shifting based heuristics.

Chapter 5

Extensions to AND/OR Multi-valued Decision Diagrams

In this chapter, we explore AND/OR Multi-valued Decision Diagrams (AOMDDs), which combines the two frameworks of AND/OR search spaces and Multi-valued Decision Diagrams (MDDs). Decision diagrams are generally used to represent functions compactly, widely used in formal verification [6].

The main motivation of AOMDDs [37] is to compile the global function into a more compact and explicit representation on which queries can be answered in linear time.

Our main contribution over [37] is the introduction of an elimination operator for AOMDDs, thus allowing the use of AOMDDs as a alternative representation to the tabular representation of functions in graphical models. This facilitates query processing on otherwise difficult problems having high treewidth. Similar work is presented in [7], where an algebraic decision diagram (ADD) structure is considered. In [45], ADDs are extended with affine transformations to capture additive and multiplicative structures in graphical models. However, AND problem decomposition is still not exploited in these alternative decision diagram variants.

We perform an empirical evaluation of the BE-AOMDD algorithm first presented in [36] to

generate an AOMDD. Next, we present BE-AOMDD-I, which is the standard bucket elimination algorithm using AOMDDs to represent all functions and messages and demonstrate its performance for exact inference.

The rest of this chapter is organized as follows: section 5.1 provides background on decision diagrams in general and previous work on AOMDDs. Section 5.2 presents our main contribution of the elimination operator for AOMDDs. Section 5.3 presents the empirical results and section 5.3.3 concludes.

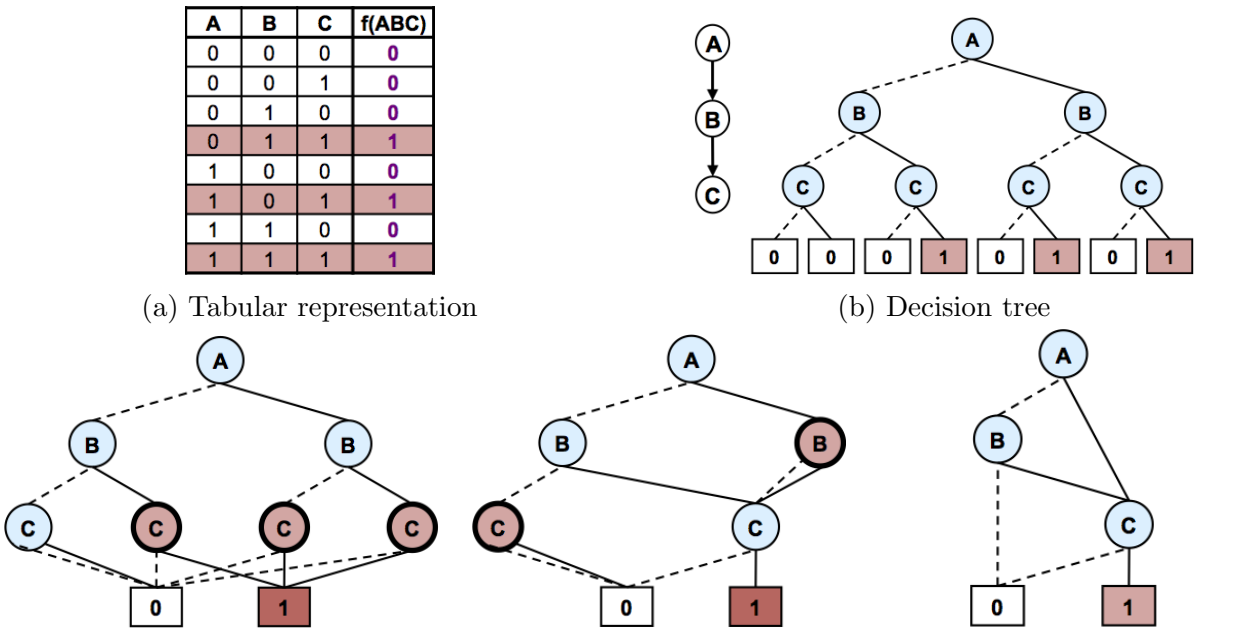
5.1 Background

A *decision diagram* is a compilation of a function that exploits problem function structure to create a compact representation. Let us consider the most basic version of a *binary decision diagram (BDD).*

Definition 5.1. *(**binary decision diagram (BDD)**) Given* $\mathbf{B} = \{0, 1\}$*, a BDD is a directed acyclic graph representing a boolean function* $f : \mathbf{B}^N \Rightarrow \mathbf{B}$*. The graph has two terminal nodes* 0 *and* 1 *that represent the right-hand side of the mapping. Each internal node corresponds to one of the* N *boolean variables each with two pointers to either other internal nodes or terminal nodes.*

A BDD is defined relative to a variable ordering. **Figure 5.1b** shows a decision tree that represents the function in **Figure 5.1a** along ordering (A,B,C). Each node corresponds to a variable, dotted edges correspond to a value of 0, and solid edges correspond to a value of 1. The leaf nodes correspond to the value of the function along a path. There are now two reduction rules that can be applied [5]:

- *isomorphism*: merge nodes of the same variable that have the same children.



(a) Tabular representation

(b) Decision tree

(c) Isomorphic nodes

(d) Redundant nodes

(e) Reduced Ordered Binary Decision Diagram (OBDD)

Figure 5.1: BDD Example: from table to OBDD (from [37])

- *redundancy*: delete nodes whose children are identical and connect its parent to that child.

Applying both of these rules yields the canonical reduced ordered binary decision diagram (OBDD) [5].

Continuing the example, we can first merge the leaves with the same value, yielding the graph in **Figure 5.1c**. However, we can merge more nodes here, since the marked C nodes are isomorphic. Carrying out this merge yields the graph in **Figure 5.1d**. Here, we can identify redundant nodes (the marked B and C nodes). Applying the rule removes them and redirects their parents' arcs toward the terminal 0 and remaining C node, respectively. This results in the OBDD in **Figure 5.1e**.

An important property of the OBDD is that it is a canonical compact representation of the function with respect to the chosen variable ordering. The extension of BDDs to non-binary

variables is straightforward, yielding multi-valued decision diagrams (MDDs).

5.1.1 AND/OR Multi-Valued Decision Diagrams

The AND/OR Multi-valued Decision Diagram (AOMDD), which is a compact representation of an AND/OR search graph was introduced in [36]. It is a data structure based on applying the reduction rules of decision diagrams to (weighted) context-minimal AND/OR search graphs [37]. The basic unit of an AOMDD is the meta-node, which groups OR and their AND child nodes together. We provide the following definitions and notation from [37].

Definition 5.1 (**meta-node**). *A meta-node u in an AOMDD is either: (1) a terminal node labeled with 0 or 1, or (2) a nonterminal node grouping an OR node labeled with a variable X and its k AND children labeled x_i representing each assignment to X. Each AND node stores a set of pointers to child meta-nodes, denoted $u.children_i$. Additionally, the AND node stores a weight $u.c(X, x_i)$ for weighted graphical models.*

An example of an AOMDD and its meta-nodes can be seen in **Figure 5.2c**.

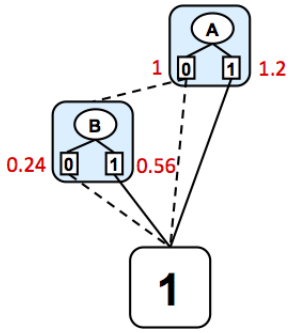
Definition 5.2 (**isomorphic meta-node [37]**). *Given a weighted AND/OR search graph represented with meta-nodes, two meta-nodes u and v having $var(u) = var(v) = X$ with domain size k are* isomorphic *iff*

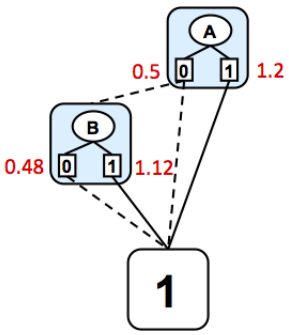
1. $\forall i \in \{1, \ldots, k\}$ $u.children_i = v.children.i$, *and*
2. $\forall i \in \{1, \ldots, k\}$ $u.c(X, x_i) = v.c(X, x_i)$

Definition 5.3 (**redundant meta-node [37]**). *Given a weighted AND/OR search graph represented with meta-nodes, a meta-nodes u having $var(u) = X$ with domain size k is* redundant *iff the child nodes and the arc-costs are all the same. Namely if,*

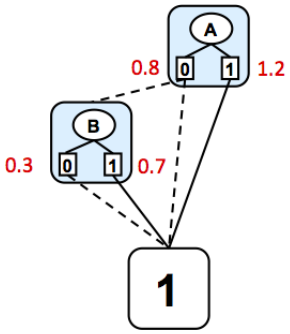
1. $u.children_1 = \ldots = u.children_k$ *and*

A	B	f
0	0	0.24
0	1	0.56
1	0	1.2
1	1	1.2

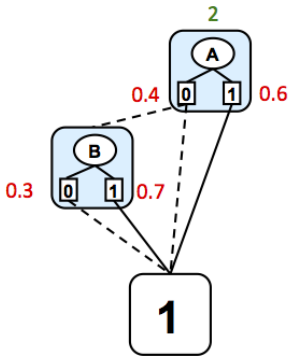




(a) Two possible distributions of weights for the function on the left



(b) AOMDD with meta-node B normalized



(c) Normalized AOMDD with a canonical distribution of weights

Figure 5.2: Weight normalization example for AOMDDs.

2. $u.c(X, x_1) = \ldots = u.c(X, x_k)$

Even with the reduction rules applied, AOMDDs are not yet canonical representations a graphical models such as Bayesian networks, Markov networks, or weighted CSPs. This is due to how weights along the paths can be factorized in many ways to yield the same cost.

We illustrate this with an example in **Figure 5.2**. The top part shows a function in a tabular representation along with two AOMDDs with the same structure, but different weights. Recall that like in weighted AND/OR search graphs, the cost of a solution is the product of the arc costs in a solution graph. In this case, when taking identical solutions trees between each of the AOMDDs, the resulting product is equal. However, based on the definition of isomorphism given above, these two AOMDDs would not be considered isomorphic since their weights differ. In order to restore this property, we can normalize the weights. Namely, we

require that the weights of AND nodes within a metanode sum to 1. The weights of AOMDD can be normalized by processing nodes bottom-up. In our example, taking the first AOMDD in **Figure 5.2a**, we take the weights of B and sum them up, yielding $0.24 + 0.56 = 0.8$. We normalize each of these weights by dividing each by 0.8, yielding 0.3 and 0.7, respectively, then propagating up the normalizer 0.8 to its parent, yielding the graph in **Figure 5.2b**. Following this, we normalize the weights of A, yielding a normalizer of $0.8+1.2 = 2$, therefore giving weights of 0.4 and 0.6. Since there is no parent of A here, the normalizer 2 is placed at the root meta-node. **Figure 5.2c** shows the final normalized AOMDD.

Thus, given a graphical model and a pseudo-tree $\mathcal{T}$, an AOMDD of this graphical model is and AND/OR search graph along $\mathcal{T}$ is defined as follows.

Definition 5.4 (**AOMDD [37]**). *An AND/OR Multi-valued Decision Diagram (AOMDD) is a weighted AND/OR search graph such that it is completely reduced and non-redundant. Namely, its meta nodes obey that 1) each meta-node is normalized (its weights sum to 1), 2) the root meta-node has a constant associated with it, and 3) it is completely reduced, namely, it has no isomorphic meta-nodes and no redundant meta-nodes.*

5.1.1.1 Apply Operator

In the decision diagram literature such as [5], the idea is to have BDDs represent smaller functions, which can then be combined into larger ones through the use of a so-called *apply* operator. This allows for standard binary operators (union, intersection, join, etc.) to be performed on two boolean functions represented by BDDs.

We next present the *apply* operator for AOMDDs, first introduced in [36], and extended for weighted models in [37]. In OBDDs, in order to combine two BDDs using *apply*, they must have compatible variable orderings. In AOMDDs, we define the notion of *strictly compatible pseudo trees*.

Definition 5.5 (**strictly compatible pseudo trees [37]**). *Given a pseudo-tree $\mathcal{T}$ with nodes $\mathbf{X}$ and pseudo-tree $\mathcal{T}_\infty$ with nodes $\mathbf{X_1} \subseteq \mathbf{X}$, $\mathcal{T}_1$ is embeddable in $\mathcal{T}$ if $\mathcal{T}_1$ can be obtained from $\mathcal{T}$ by removing each node in $\mathbf{X} \setminus \mathbf{X_1}$ and connecting its parent to each of its descendants. Two pseudo-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are strictly compatible if there exists a pseudo-tree $\mathcal{T}$ which they can both be embeddable in.*

The APPLY operator for AOMDDs is given in **Algorithm 21** [37]. It requires that the pseudo-trees of the input AOMDDs are strictly compatible with respect to a pseudo-tree $\mathcal{T}$. The inputs specifically take the root meta-node of AOMDD f with the list of meta-nodes from AOMDD g. Initially, the list of nodes from g consists of only the root node of g. Furthermore, this list of nodes from g must satisfy that the single node from f is a common ancestor and that none of them are ancestors of each other. Thus, the list of nodes in g represent meta-nodes rooting decomposed parts with respect to $\mathcal{T}$. Note that we consider a node to be an ancestor of itself here, thus having v_1 and w_1 with the same variable is valid. APPLY makes use of two caches: H_1 for detecting isomorphism and H_2 to prevent the same operation from being computed more than once.

The first line of the algorithm makes use of H_2 in this way. Otherwise, several other simple cases occur in lines 2-4. Line 5 initializes a new meta node u with the same variable as v_1, then proceeds to generate the children for each of its AND nodes in lines 6-20.

Line 14 is a step which serves to dictate how to recursively work down the decision diagram structure to combine the children. Intuitively, this moves the down each side of f and g as needed and generates APPLY calls that represent each of the decompositions formed by multiple children if they exist. Once, these parameters are determined, lines 15-20 generate a meta-node for each parameter set, which become the children of u's j^{th} AND node. We can also terminate early if any of the metanodes generated is $\mathbf{0}$. After the meta-node u has been completely generated, lines 21-22 checks if it is redundant, promoting its weight to its

Algorithm 21: APPLY($v_1; w_1, ..., w_m$)[37]

Input: AOMDDs f with nodes v_i and g with nodes w_j, based on *strictly compatible* pseudo trees $\mathcal{T}_1, \mathcal{T}_2$ that can be embedded in $\mathcal{T}$.
$var(v_1)$ is an ancestor of all $var(w_1), ..., var(w_m)$ in $\mathcal{T}$.
$var(w_i)$ and $var(w_j)$ are not in an ancestor-descendant relation in $\mathcal{T} \forall i \neq j$
Output: $v_1 \otimes (w_1 \otimes ... \otimes w_m)$, based on $\mathcal{T}$

1 **if** $H_2(v_1, w_1, ..., w_m)$ *exists* **then return** $H_2(v_1, w_1, ..., w_m)$;
2 **if** *any of* $v_1, w_1, ..., w_m = \mathbf{0}$ **then return 0** ;
3 **if** $v_1 = \mathbf{1}$ **then return 1** ;
4 **if** *there are no* w_i **then return** v_1 ;
5 Create new nonterminal metanode u with variable $var(v_1) = X_i$ (which has a domain from 1 to k_i.
6 **for** $j \leftarrow 1$ *to* k_i **do**
7 $u.children_j \leftarrow \varnothing$; `// children of the j-th AND node of u`
8 $u.c(X_i, x_j) \leftarrow v_1.c(X_i, x_j)$
9 **if** $m = 1$ *and* $var(v_1) = var(w_1) = X_i$ **then**
10 $u.c(X_i, x_j) \leftarrow v_1.c(X_i, x_j) \otimes w_1.c(X_i, x_j)$; `// combine weights`
11 $tempChildren \leftarrow w_1.children_j$
12 **else**
13 $tempChildren \leftarrow \{w_1, ..., w_m\}$
14 Group nodes from $v_1.children_j \cup tempChildren$ into sets $\{v^1; w^1, ..., w^r\}$
15 **foreach** $\{v^1; w^1, ..., w^r\}$ **do**
16 $y \leftarrow$ APPLY($v^1; w^1, ..., w^r$)
17 **if** $y = \mathbf{0}$ **then**
18 $u.children_j \leftarrow \mathbf{0}$; **break**
19 **else**
20 $u.children_j \leftarrow u.children_j \cup \{y\}$
21 **if** $u.children_1 = ... = u.children_{k_i}$ *and* $u.c(X, x_1) = \ldots = u.c(X, x_k)$ **then**
22 Promote $u.c(X_i, x_1)$ to parent
23 **return** $u.children_1$; `// redundancy`
24 **if** $\exists u' \in H_1$ *s.t.* u *is isomorphic to* u' **then**
25 **return** u'
26 Insert u into H_1
27 Let $H_2(v_1; w_1, ..., w_m) = u$
28 **return** u

parent and returning a set of children from any of its AND nodes, thus removing it. Lines 23-24 checks if u has already been generated (and thus isomorphic) by consulting H_1 and simply returns the previous generated version instead if so. Finally, lines 25-27 update H_1 and H_2 and return the newly created meta-node u.

The runtime of *apply* is quadratic in the size of the input AOMDDs [37].

5.1.1.2 Compiling AOMDDs

Algorithm 22: BE-AOMDD-C [37]

Input: Graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, pseudo tree $\mathcal{T}$
Output: AOMDD representing the global function of $\mathcal{M}$

```
1 foreach X_p ∈ X in bottom up order according to T do
2     B_p ← {f_j^AOMDD ∈ F | X_p ∈ S_j}
3     F ← F − F_p
4     Put child message λ_ch(p)^AOMDD in B_p
5     λ_p→pa(p)^AOMDD ← 1
6     foreach f_j^AOMDD ∈ B_p do
7         λ_p^AOMDD ← APPLY(λ_p^AOMDD, f_j^AOMDD)
8 return λ_root(T)^AOMDD
```

Algorithm 22 presents an algorithm for compiling the AOMDD [37]. Introduced as BE-AOMDD, we call it BE-AOMDD-C here to emphasize its purpose for *compilation*. Using APPLY, it creates the AOMDD of the graphical model using a bucket elimination (BE) based schedule. It works exactly like BE, except it combines functions using the APPLY algorithm and does not eliminate variables.

5.2 AOMDDs for Exact Inference

It is easy to see that in BE-AOMDD-C, if we include a step to eliminate the bucket variable, then the result is the same as BE, but carried out via functions represented as AOMDDs.

Thus, we need an elimination operator to eliminate variables from and AOMDD, namely summation.

We next define a simple form of the operator that is restricted to eliminating variables that sit at the leaves of the embedded pseudo-tree of a given AOMDD.

The pseudo code is given in **Algorithm 23**. Lines 1-4 cover the simple cases when the AOMDD is a terminal 0 or 1. The only special case is when the operator is summation (which is what we plan to use), where we multiply the input by the domain size of the variable to eliminate. Otherwise, given the embedded pseudo tree of the AOMDD, we identify the metanodes associated with variables which lie on the path from the variable to be eliminated to the root (line 5). We will refer to these variables as *elimination relevant*. The loop beginning at line 7 then processes each metanode in the AOMDD in a bottom-up fashion according to the pseudo tree if its variable is one of the relevant variables. If the node is an elimination variable, we eliminate the node by performing the necessary operator, promote the weight to the parents (lines 10-15). Otherwise, we first need to perform a redundancy reduction on the node if necessary due to the weights promoted from the children through elimination, promoting its weight further up to its parents (lines 21-28). If the node is not redundant, then we normalize the node and pass on the normalization constant to the parent (line 29-34).

There is also a special case for processing a metanode that is not the variable to eliminate when the operator is summation. In the event that the metanode for a variable to eliminate is not present in the decision diagram due to the redundancy reduction, we need to detect this and adjust the weights properly, namely multiplying them by the domain size of the elimination variable to carry out the summation operation. To identify which AND nodes do not require this step, we keep track of the AND nodes that have had their weights updated via either elimination (lines 8-15), redundancy reduction (lines 21-28), or normalization (lines

Algorithm 23: ELIMINATE($f, eVar, \Downarrow$: elimination operator : $(\sum, \max, \min)$)

Input: AOMDD f (with embedded pseudo tree $\mathcal{T}$ containing $eVar$ as a leaf)
Output: AOMDD f' representing $\Downarrow_{eVar} f$ (with embedded pseudo tree $\mathcal{T}'$)

```
1  if f = 0 or f = 1 then
2      Remove eVar from T
3      if ⇓ = Σ then return |D_eVar| · f ;
4      else return f ;
5  R := set of metanodes of variables on the path from eVar to the root of T
6  weightUpdated := ∅
7  Process each m ∈ R bottom-up, according to T:
8      if var(m) = eVar then
9          weight :=⇓_{c∈ch(m)} (c.weight)
10         foreach p ∈ pa(m) do
11             p.weight := p.weight · weight
12             Insert p into weightUpdated
13             Remove m from ch(p)
14             if weight = 0 then ch(p) := {0} ;
15             else if ch(p) = ∅ then ch(p) := {1} ;
16     else
17         if ⇓= Σ then
18             foreach c ∈ ch(m) do
19                 if c ∉ weightUpdated then
20                     c.weight := c.weight · |D_eVar|
21         if m is redundant then
22             weight := c.weight for any c ∈ ch(m)
23             foreach p ∈ pa(m) do
24                 p.weight := p.weight · weight
25                 Insert p into weightUpdated
26                 Remove m from ch(p)
27                 if weight = 0 then ch(p) := {0} ;
28                 else if ch(p) = ∅ then ch(p) := {1} ;
29         else
30             z := Σ_{c∈ch(m)} c.weight
31             foreach c ∈ ch(m) do
32                 c.weight := c.weight / z
33             foreach p ∈ pa(m) do
34                 p.weight := p.weight · z
35                 Insert p into weightUpdated
```

29-34). If an AND node has never been updated, then it means that we must apply the weight adjustment. Lines 17-20 carries out the weight update based on these conditions.

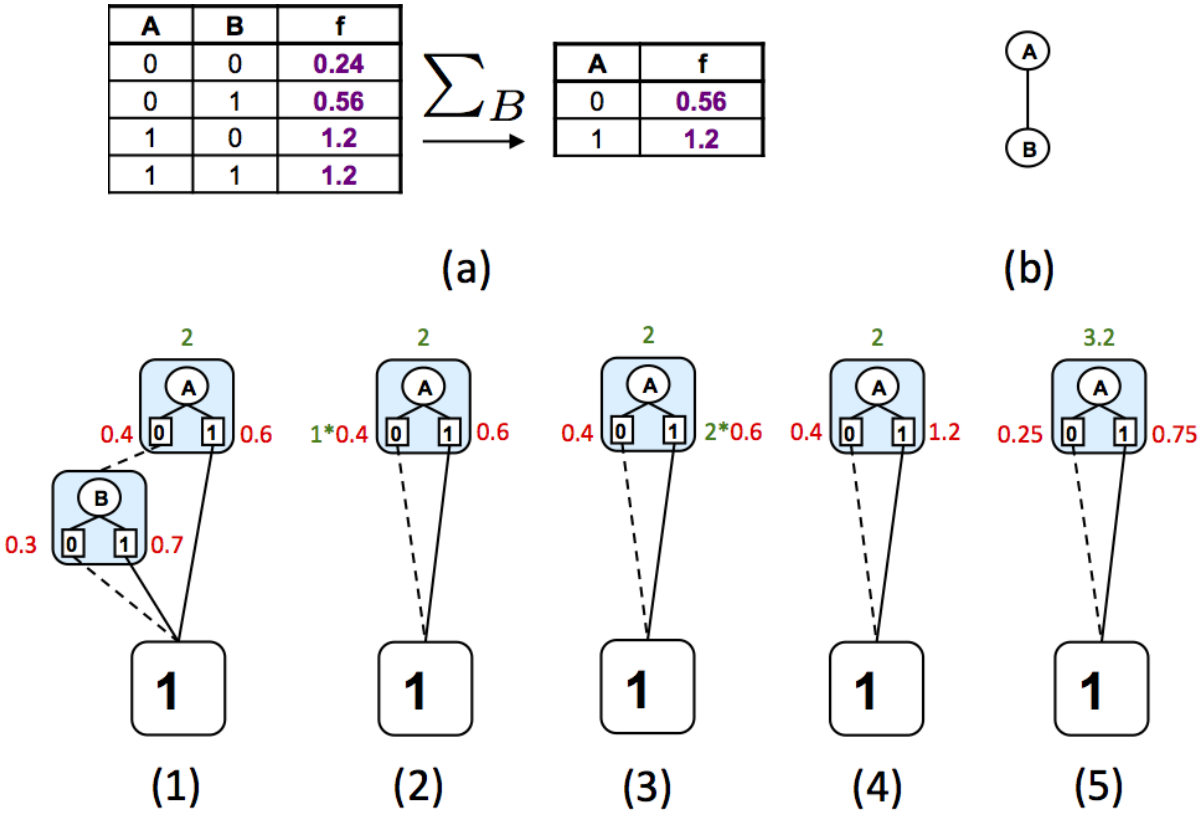


Figure 5.3: Example of elimination on AOMDDs. The state of the AOMDD is shown through the process.

We demonstrate the algorithm on a small example, shown in **Figure 5.3**. The function tables above (a) the AOMDDs demonstrate the operation performed in a standard representation. We are interested in summing out variable B (namely eliminating B). The embedded pseudo tree (b) is used to determine the set of relevant variables, which in this case is $\{A, B\}$.

We begin with the AOMDD shown at **Figure 5.3.1**, which represents the same function as the input table. Visiting the relevant nodes in a reverse BFS order, we visit the metanode B first, eliminate it, and propagate its result up to the parent AND node in metanode A, shown in **5.3.2**. At **5.3.3**, we are left with only metanode A. Checking the 0 AND node, since it received a weight from something, we are done with it. Checking the 1 AND node,

since it has not received a weight, we multiply it by the domain size of B, which is 2 in this case. The result is now shown at (4). Finally, we normalize the AND node weights of metanode A and propagate its normalization term up to the root, yielding the resulting AOMDD in (5), which represents the same function as the output table.

In the cases of maximization and minimization, we do not encounter the same problem since these operators choose one from the set of values, which has no effect on functions where all the output values are identical.

PROPOSITION **7** (**complexity of ELIMINATE).** *The time complexity of ELIMINATE is linear in the size of the AOMDD. The output size of the AOMDD is also linear in the size of the input AOMDD.*

Proof. In the worst case, when the AOMDD has an embedded pseudo tree that is a chain, then every meta-node in the AOMDD will be processed bottom-up, starting with the variable to eliminate. Thus, the time complexity of ELIMINATE is linear in the size of the AOMDD. In the worst case, there is only a single meta-node corresponding to the variable we eliminate, thus the output is still be linear in the size of the AOMDD. □

5.2.1 Bucket Elimination

Now that we have the APPLY and ELIMINATE operators, we have everything necessary to perform bucket elimination using AOMDDs as the function representation. We call the resulting algorithm BE-AOMDD-I. The algorithm is identical to BE-AOMDD-C (see **Algorithm 22**), except that we also eliminate the bucket variable when generating the message. BE-AOMDD-I is therefore a version of BE where the functions of the graphical model use AOMDDs as the representation instead of tables. Since the elimination operator is restricted to eliminating only leaf variables, each function must also be constructed based on a pseudo-tree that can be embedded in the pseudo-tree guiding the elimination ordering of BE.

5.3 Experiments

In this section we provide results in two parts. The first uses the previously introduced BE-AOMDD-C algorithm which compiles the AOMDD in the same order as bucket elimination (i.e. in the variable elimination ordering) using the APPLY operator to combine functions. The second reports on experiments using AOMDDs as a general framework for the representation of functions inside the BE algorithm for performing the weighted counting problem (summation query).

The AOMDDs and all operators were implemented in C++ and run on a 2.66GHz processor with 24GB of RAM. All experiments were run with no time limit and a maximum of 24GB memory.

5.3.1 Compilation

In previous work [37], results on the compilation of AOMDDs were carried out via a top-down compilation algorithm that worked by tracing of an execution of AND/OR search. We compare with a variant this algorithm known as AOMDD-BCP, which applies boolean constraint propagation to the original problem as a preprocessing step to better capture determinism during search. In our experiments, the implementation of AOMDD-BCP used was provided by the authors of [37]. The redundancy rule was not applied in this implementation, so our results on BE-AOMDD-C are the first for compiling canonical AOMDDs.

We ran experiments on the UAI 2006 ISCAS circuits and protein side chain prediction networks. In each table, we report various problem instances parameters, runtime for each of the algorithms, the size of the context-minimal AND/OR graph (in OR nodes), and the sizes of the AOMDDs for each algorithm.

5.3.1.1 UAI 2006 ISCAS Circuits

name	n	w	h	k	f	time (s) [BE-AOMDD-C] [AOMDD-BCP]	CM OR	Metanodes [BE-AOMDD-C] [AOMDD-BCP]
BN_42	850	20	50	2	879	10 36	5623680	**25841** 95963
BN_43	850	21	50	2	881	73 647	22731586	**148184** 629027
BN_44	850	21	60	2	880	OOM 110	11681649	OOM 396583
BN_45	850	21	56	2	875	17 142	15778481	**122763** 260917
BN_46	850	19	47	2	499	OOM 1	4277086	OOM 10237
BN_47	632	36	54	2	662	OOM 0.42	4.49E+11	OOM 1974
BN_49	632	40	60	2	663	OOM 0.33	9.56E+12	OOM 1277
BN_51	632	40	59	2	665	OOM 0.43	6.39E+12	OOM 1884
BN_53	532	42	82	2	562	OOM 1.3	3.02E+13	OOM 5317
BN_55	532	42	85	2	563	OOM 0.44	2.66E+13	OOM 1704
BN_57	532	41	82	2	562	OOM 1.47	1.54E+13	OOM 6425
BN_59	511	46	86	2	542	OOM 3.47	4.54E+14	OOM 15679
BN_61	638	40	60	2	668	OOM 0.34	8.11E+12	OOM 1278
BN_63	511	47	81	2	539	OOM 6.52	1.03E+15	OOM 17888
BN_65	411	51	98	2	460	OOM 230.8	1.61E+16	OOM 9631
BN_67	411	50	79	2	460	OOM 449.13	9.44E+15	OOM 1135

Table 5.1: Compilation results on UAI 2006 benchmarks (ISCAS circuits). For each instance, report the number of variables (n), induced width (w), pseudo-tree height (h), maximum domain size (k), and number of functions (f). In the columns labeled "time(s)" and "Metanode", the top and bottom are the runtimes/metanode counts for BE-AOMDD-C and AOMDD-BCP respectively. We also report the number of OR nodes in the context-minimal AND/OR graph (CM OR).

Table 5.1 presents the compilation results on ISCAS circuits from the UAI 2006 benchmark. Notice that the target here is to generated the full weighted AOMDD. There are a total of 16 ISCAS instances, but only 3 could be successfully compiled by BE-AOMDD-C. BE-AOMDD-C ran out of memory on the rest. On these 3 instances, we observe that the AOMDDs generated by BE-AOMDD-C are smaller and also have a faster runtime. For example, on *BN_42*, BE-AOMDD-C took 10 seconds to construct an AOMDD with 25841 meta-nodes

while AOMDD-BCP took 36 seconds to build an AOMDD with 95963 meta-nodes. For the instances where BE-AOMDD-C ran out of memory, AOMDD-BCP has an advantage through its preprocessing via boolean constraint propagation, which detects determinism in the network and prunes out branches of the search graph before they are generated. Our implementation of BE-AOMDD-C performs no preprocessing on the input problem which is likely to be a major contribution for the poor performance. This performance difference can be additionally attributed to the difference in a bottom-up vs. a top-down compilation approach. We observe this by noticing that during the execution of BE-AOMDD-C, the intermediate messages are generally larger than the final AOMDD. Given enough time and memory resources, we expect BE-AOMDD-C to always yield a smaller compiled AOMDD than AOMDD-BCP, since it generates the canonical, fully reduced AOMDD for any given function.

5.3.1.2 Protein Networks

Table 5.2 shows compilation results on protein networks. For these instances, we were unable to run the supplied implementation of AOMDD-BCP due to unknown technical issues. Thus, we only provide a comparison to the context-minimal AND/OR graph size here. In general, this benchmark has low induced width, but high domain sizes, thus still yielding large context-minimal AND/OR graphs. In all cases, the AOMDDs generated are smaller as expected. In particular, we can see reductions as high as 87.9 times (see *pdb1rzl*). Indeed, despite the high domain size, when inspecting the functions of these instances, there are many variable assignments that have the same value, a feature that AOMDDs can exploit by mapping all of them to the same meta-node. These types of compilations are computed here for the first time.

name	n	w	h	k	time (s)	CM OR	Metanodes [BE-AOMDD-C]
pdb1ajj	32	5	12	81	35	2076152	188974
pdb1akg	14	2	4	18	0	271	49
pdb1etl	9	1	3	27	0	70	19
pdb1etm	10	1	3	27	0	82	25
pdb1etn	9	1	2	27	0	85	22
pdb1fna	75	6	18	81	136	1983522	56377
pdb1fxd	51	5	17	81	121	1361708	109076
pdb1hh5	54	5	14	81	704	11069620	302913
pdb1hoe	60	5	16	81	108	584840	51972
pdb1j8e	39	6	12	81	294	2714323	258198
pdb1k51	57	5	14	81	47	910288	15388
pdb1mof	46	4	16	81	362	2970412	54002
pdb1noa	80	4	21	81	390	3193958	312262
pdb1not	11	2	5	81	0	179	72
pdb1pef	17	6	11	81	430	4123288	342367
pdb1pen	13	2	4	18	0	432	104
pdb1piq	29	4	20	81	359	2568277	223488
pdb1rb9	42	7	14	81	1127	13370233	1163424
pdb1rh4	21	4	14	81	35	386047	55054
pdb1rzl	65	5	14	81	2232	43949983	499710
pdb1xy2	7	2	2	36	0	480	62
pdb2erl	34	4	12	81	22	359718	13914
pdb2fdn	42	4	11	81	0	9704	1761
pdb2igd	50	6	19	81	1295	33711674	451081
pdb2mcm	80	4	18	81	52	744266	10555
pdb3cao	84	5	21	81	1022	12535345	913621
pdb3ezm	88	3	13	81	7	514561	6959

Table 5.2: Compilation results on protein networks using BE-AOMDD-C. The information is the same as the previous table, except without AOMDD-BCP.

5.3.2 Exact Inference

In this section, we do not compile a graphical model to an AOMDD, but rather perform inference using the bucket elimination framework. The following evaluates the BE-AOMDD-I algorithm, which is the same as bucket elimination, but uses AOMDDs to represent all functions. We ran experiments on the UAI 2006 evaluation problems, mastermind instances, and genetic linkage analysis networks, available at `http://graphmod.ics.uci.edu`. In each table, we compare the time and memory usages of standard BE vs. BE-AOMDD-I. Times reported as "OOM" indicate that the algorithm exceeded our memory bound of 8GB. Results on memory usage are based on the usage of the cache storing nodes of the AOMDDs. For instance where BE runs out of memory, we simulated its execution by only passing information about scope sizes to compute the memory usage.

problem	n	w	h	k	time (s) [BE]	time (s) [BE-AOMDD-I]	Mem (MB) [BE]	Mem (MB) [BE-AOMDD-I]
BN_22	2425	5	575	91	1	13	**26.93**	581.27
BN_24	1819	5	381	91	1	23	**24.07**	977.52
BN_28	24	5	9	10	1	13	**1.79**	568.36
BN_30	1156	48	179	2	OOM	38	1.50E+10	**245.93**
BN_32	1444	56	219	2	OOM	4384	4.45E+12	**3006.08**
BN_34	1444	55	220	2	OOM	145	2.30E+12	**515.45**
BN_36	1444	56	210	2	OOM	7792	3.51E+12	**2629.44**
BN_40	1444	55	235	2	OOM	91	1.82E+12	**322.76**
BN_42	880	23	54	2	21	2	314.04	**21.62**
BN_43	880	22	53	2	10	2	153.66	**11.83**
BN_44	880	22	55	2	10	2	159.78	**18.47**
BN_45	880	22	55	2	10	2	162.39	**16.56**
BN_46	499	22	49	2	18	<1	248.97	**1.99**
BN_49	661	44	59	2	OOM	1188	7.83E+08	**2991.78**
BN_51	661	44	61	2	OOM	3433	1.17E+09	**2274.11**
BN_53	561	48	95	2	OOM	4063	8.43E+09	**3303.48**
BN_61	667	44	61	2	OOM	17	9.46E+08	**235.72**
BN_65	440	61	95	2	OOM	1062	Overflow*	**2843.65**
BN_67	440	61	99	2	OOM	9893	Overflow*	**1270.54**
BN_78	54	13	24	2	<1	<1	**0.51**	29.82
BN_84	360	20	24	2	4	22	**24.76**	546.21
BN_86	422	22	40	2	26	73	**179.44**	1084.59
BN_92	422	22	33	2	26	23	**187.43**	433.65

Table 5.3: BE-AOMDD-I on UAI 2006 benchmarks (22-28: unrolled dynamic Bayesian networks, 30-40: grid structured networks, 42-67: ISCAS circuits, 78-92: medical diagnosis). (* The size in MB could not be stored within a double precision number representation.)

5.3.2.1 Results of Inference for UAI 2006 Instances

Table 5.3 presents results on running bucket elimination using the traditional representation of functions by tables against our implementation using AOMDDs. We see that our scheme is able to solve some problems which do not fit in standard main memory. For example, on *BN_65* having an induced width of 61, BE would require an amount of memory (in MB) larger than the maximum double-precision number (10^{308}), yet BE-AOMDD-I requires only 3303 MB of memory to solve the problem. A large number of these instances, on which AOMDD performs well, have a significant amount of determinism and mostly constant functions (42-67 are ISCAS circuits), which AOMDDs can exploit. On the other hand, there must be a significant amount of compression before we get any memory savings, as seen on instances such as *BN_86* where far more memory was needed by BE-AOMDD-I. This is due to the overhead of the AOMDDs graph representation compared with a simple tabular

representation. Whenever there is not enough structure to exploit, AOMDD-based schemes end up using more resources in both time and memory. Overall, we observe that as expected for easier problems, standard BE is sufficient and yields better runtime.

5.3.2.2 Mastermind

name	n	w	h	k	time (s) [BE]	time (s) [BE-AOMDD-I]	Mem (MB [BE]	Mem (MB) [BE-AOMDD-I]
03_08_03-0000	1220	18	53	2	4	5	**48.23**	154.45
03_08_03-0001	1220	18	54	2	4	4	**53.63**	105.21
03_08_03-0006	1220	18	41	2	2	2	44.38	**41.70**
03_08_03-0007	1220	18	52	2	2	1	46.40	**21.64**
03_08_04-0000	2288	29	79	2	OOM	643	49865.56	**4187.84**
03_08_04-0001	2288	28	76	2	OOM	293	39769.34	**2610.66**
03_08_05-0006	3692	37	101	2	OOM	6	24847465.64	**39.04**
03_08_05-0007	3692	37	80	2	OOM	9	25456599.73	**120.19**
04_08_03-0001	1418	22	58	2	46	54	**638.49**	907.49
04_08_03-0002	1418	22	55	2	41	33	**621.16**	786.86
04_08_03-0006	1418	22	59	2	46	11	621.98	**270.47**
04_08_03-0007	1418	22	52	2	36	2	617.11	**47.70**
04_08_04-0006	2616	35	88	2	OOM	17	4675522.59	**371.95**
04_08_04-0007	2616	35	93	2	OOM	17	3467438.50	**349.37**
05_08_03-0000	1616	26	57	2	690	1064	9092.90	**5742.06**
05_08_03-0001	1616	26	67	2	758	759	8853.24	**3919.56**
06_08_03-0006	1814	29	73	2	OOM	135	92849.60	**1150.94**
06_08_03-0007	1814	29	66	2	OOM	27	76976.00	**588.10**
06_08_03-0008	1814	29	64	2	OOM	24	85068.34	**523.36**
06_08_03-0010	1814	29	66	2	OOM	13	93097.25	**253.99**
10_08_03-0006	2606	43	92	2	OOM	654	2085939395.05	**673.54**

Table 5.4: BE-AOMDD-I on Mastermind instances.

Table 5.4 shows results on instances derived from mastermind games. Much like the ISCAS circuits, these networks also exhibit high amounts of determinism and have near constant functions, which can be exploited by AOMDDs. Here, half of the instances cannot be solved by BE, but BE-AOMDD-I manages to solve them despite the extremely high memory required by BE. See, for example, instance *10_08_03-0006*.

5.3.2.3 Pedigree Networks

Table 5.5 shows results on the pedigree networks. On these instances, many of the partition function values were not known before the work of [27], which uses hard disk to push the

name	n	w	h	k	time (s) [BE]	time (s) [BE-AOMDD-I]	Mem (MB) [BE]	Mem (MB) [BE-AOMDD-I]
pedigree1	334	15	61	4	2	14	**23.61**	210.09
pedigree7	1068	28	123	4	OOM	OOM	404625.25	OOM
pedigree9	1118	25	137	7	550	5301	7499.77	**4030.34**
pedigree13	1077	29	161	3	OOM	OOM	150793.98	OOM
pedigree18	1184	19	102	5	7	200	**136.13**	959.28
pedigree19	793	21	118	5	OOM	OOM	23478.49	OOM
pedigree20	437	21	58	5	131	291	1393.90	**1030.66**
pedigree23	402	20	58	5	19	52	**241.57**	532.46
pedigree25	1289	23	86	5	146	1284	**2037.69**	2999.84
pedigree30	1289	20	102	5	13	307	**220.63**	1044.76
pedigree31	1183	28	106	5	OOM	OOM	919612.70	OOM
pedigree33	798	24	116	4	347	883	4277.26	**1368.42**
pedigree34	1160	28	143	5	OOM	OOM	762082.00	OOM
pedigree37	1032	20	62	5	OOM	3535	251109.68	**7992.43**
pedigree38	724	16	67	5	OOM	2201	172249.65	**6253.16**
pedigree39	1272	20	83	5	46	400	**772.20**	1555.68
pedigree40	1030	27	111	7	OOM	OOM	3884488.59	OOM
pedigree41	1062	28	142	5	OOM	OOM	261551.31	OOM
pedigree42	448	21	67	5	OOM	OOM	39007.50	OOM
pedigree44	811	24	79	4	516	3795	6153.63	**4782.29**
pedigree50	514	16	53	6	OOM	OOM	682521.86	OOM
pedigree51	1152	34	121	5	OOM	OOM	2900534.96	OOM

Table 5.5: BE-AOMDD-I on Pedigree networks.

memory restrictions of BE.

Our results are less promising on these networks. There are only two instances on which BE-AOMDD-I manages to perform very well, where standard BE would require about 30 times the amount of memory (*pedigree37* and *pedigree38*). For the rest of the problems that BE-AOMDD-I managed to solve, on half of the instances it was less efficient than standard BE. For the other half where BE-AOMDD-I uses less memory, the runtime is often much worse, due to overhead in maintaining the properties of a canonical AOMDD. We can attribute these results this set of problems having much less structure than networks derived from digital applications such as circuits and board games (ISCAS and mastermind). However, these results also demonstrate the use of decision diagrams on non-binary networks for inference, when compared to related work using ADDs [7, 24].

5.3.3 Conclusion

To recap, we explored the potential of AOMDDs in two ways: first for compilation and then for exact inference. On compilation, our results are preliminary, but demonstrate a reduction in AOMDD size on instances on which the new compilation algorithm was feasible. The feature of canoncity is useful, as compiling a graphical model into an AOMDD can reveal whether it is truly as hard as implied by the induced width. The notion of semantic induced width can be calculated based on the size of an AOMDD [37]. For exact inference, we demonstrated the potential of using AOMDD as a function representation by solving many difficult instances exactly that would have required the traditional tabular representations an infeasible amount of memory. This was demonstrated on instances having high levels of determinism and near constant functions. This suggests that the AOMDD representation should be considered for such types of problems.

For future work, performing some preprocessing may allow us to compile a wider variety of problems with high treewidth. More importantly, the bucket elimination based schedule for compiling AOMDDs is needlessly restrictive. With the flexibility of the apply operator, we can consider other schedules. A particularly useful one might be to combine the functions with the smallest AOMDDs first, which are likely to have determinism or are near constant. Incorporating this information into a compilation process as soon as possible may mitigate the increase of size of the intermediate AOMDDs.

For problems which are still too large to compile, we still have the option of using AOMDDs as an alternative function representation. For certain problems such as the pedigree networks, the overhead results in worse performance due to overhead which is not mitigated by exploitable structure.

Chapter 6

Summary and Conclusion

In this dissertation, we focused primarily on methods for improving state-of-the-art heuristics for combinatorial optimization tasks. We also extend previous work using a compact representation of graphical models for the weighted counting task.

We first (Chapter 2) considered the well-known technique of look-ahead which we adapted to heuristic search for the MPE task in graphical models. Through an analysis of the most commonly used mini-bucket (MBE) heuristic for AND/OR Branch-and-Bound for this task, we established a connection between the so-called bucket error of MBE with the concept of known as the residual in the Bellman backup value [22]. This enabled developing a cost-effective scheme which, by using pre-compiled information about the bucket error, allowed a selective look-ahead which focuses on the most promising nodes during search. In an extensive empirical evaluation, we demonstrated improved runtime when using our error-guided look-ahead to improve the state-of-the-art mini-bucket elimination with moment matching (MBE-MM) heuristic.

Next (Chapter 3), focusing on finding algorithms generating lower bounds on the optimal cost, we explored the potential of AND/OR Best-First (AOBF) search to generate such bounds in an anytime fashion. In this context, we tackled the issue of subproblem ordering. Namely, we analyzed the impact of subproblem ordering on the performance of AOBF and

showed that the impact could be significant. Indeed this was a problem posed by Pearl [41], but was hardly explored. We identified that look-ahead would be a useful heuristic for subproblem ordering and thus leveraged our findings on bucket error. Our experiments demonstrated that our better informed subproblem ordering heuristics can provide a positive impact for both finding exact solutions and for generating lower bounds in an anytime manner.

Another focus of this dissertation (Chapter 4) is on dynamic heuristics that are orthogonal to static MBE. On some problems, especially when the domain sizes are large, MBE-MM can yield a poor heuristic, thus calling for alternative methods of heuristic generation that can exploit the new structure induced during search. We presented a dynamic heuristic generator based on factor-graph linear programming (FGLP), a coordinate-descent algorithm for solving an LP based on re-parameterizing the functions of a graphical model [25]. The algorithm was originally designed to be used once during preprocessing rather than run repeatedly during search. We address this in two ways. First, we redefine the LP that FGLP solves in order reduce the number of iterations required to for convergence conditioning a variable during search. Second, we define a finer-grained update schedule that leads to the most bound tightening. Our resulting method is similar to soft arc-consistency methods in the WCSP literature [10], but aims toward the optimal re-parameterization while balancing time and accuracy through the use of a controlling parameter. On challenging benchmarks for MBE-MM, we demonstrated the potential of FGLP as a dynamic heuristic generator.

Lastly (Chapter 5), we advanced the state-of-the-art on AND/OR Multi-valued Decision Diagram (AOMDD) framework. We provided the first empirical evaluation on the class of bottom-up compilation algorithms that were previously introduced. We also extended the framework to allow for the elimination of variables from a function represented by an AOMDD, thus allowing for the exact bucket elimination algorithm to be carried out with AOMDD-based functions. For compilation, we illustrated that the bottom-up methods ap-

plying full reduction rules are able to yield smaller complied AOMDD than previously reported. For exact inference via bucket elimination using AOMDDs, we demonstrated that we could solve problems that would otherwise take an infeasible amount of memory, if we used bucket elimination with a table representation of functions.

Bibliography

[1] D. Allouche, S. De Givry, G. Katsirelos, T. Schiex, and M. Zytnicki. Anytime hybrid best-first search with tree decomposition for weighted csp. In *International Conference on Principles and Practice of Constraint Programming*, pages 12–29. Springer, 2015.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[3] D. Batra, S. Nowozin, and P. Kohli. Tighter relaxations for map-mrf inference: A local primal-dual gap based separation algorithm. In *AISTATS*, volume 11, pages 146–154, 2011.

[4] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, pages 5–33, 2001.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, pages 677–691, 1986.

[6] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. *Proceedings of International Conference on Computer-Aided Design*, pages 236–243, 1995.

[7] M. Chavira and A. Darwiche. Compiling bayesian networks using variable elimination. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2443–2449, 2007.

[8] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1):199–227, 2004.

[9] M. C. Cooper, S. de Givry, M. Sánchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted csp. In *AAAI*, volume 8, pages 253–258, 2008.

[10] M. C. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *IJCAI*, volume 7, pages 68–73, 2007.

[11] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

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

[13] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

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

[15] R. Dechter, L. H. Lelis, and L. Otten. Caching in context-minimal or spaces. In *Eighth Annual Symposium on Combinatorial Search*, 2015.

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

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

[18] G. Elidan, I. McGraw, and D. Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Uncertainty in Artificial Intelligence (UAI)*, 2006.

[19] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 2002.

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

[21] N. Flerova. Methods for advancing combinatorial optimization over graphical models. Technical report, Ph.D. thesis, University of California, Irvine, California, 2015.

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

[23] A. Globerson and T. S. Jaakkola. Fixing max-product: Convergent message passing algorithms for map lp-relaxations. In *Advances in neural information processing systems (NIPS 2007)*, pages 553–560, 2007.

[24] V. Gogate and P. Domingos. Approximation by quantization. In *Uncertainty in Artificial Intelligence*, 2011.

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

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

[27] K. Kask, R. Dechter, and A. E. Gelfand. Beem: bucket elimination with external memory. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 268–276. AUAI Press, 2010.

[28] R. Korf. Linear-space best-first search. In *Artificial Intelligence*, pages 41–78, 1993.

[29] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[30] J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1):1–26, 2004.

[31] Q. Lou, R. Dechter, and A. Ihler. Anytime anyspace and/or search for bounding the partition function. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, 2017.

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

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

[34] R. Marinescu, J. Lee, A. Ihler, and R. Dechter. Anytime best+depth-first search for bounding marginal map. In *AAAI*, 2017.

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

[36] R. Mateescu and R. Dechter. Compiling constraint networks into and/or multi-valued decision diagrams. In *Constraint Porgramming (CP2006)*, 2006.

[37] R. Mateescu, R. Dechter, and R. Marinescu. And/or multi-valued decision diagrams (aomdds) for graphical models. *J. Artif. Intell. Res. (JAIR)*, 33:465–519, 2008.

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

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

[40] L. Otten, A. Ihler, K. Kask, and R. Dechter. Winning the pascal 2011 map challenge with enhanced and/or branch-and-bound. In *In NIPS Workshop DISCML*. Citeseer, 2012.

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

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

[43] E. Rollon, J. Larrosa, and R. Dechter. Semiring-based mini-bucket partitioning schemes. In *IJCAI*, pages 3–9. Citeseer, 2013.

[44] S. J. Russell and P. Norvig. Artificial intelligence: a modern approach (3rd edition), 2009.

[45] S. Sanner and D. A. McAllester. Affine algebraic decision diagrams (aadds) and their application to structured probabilistic inference. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1384–1390, 2005.

[46] D. Tarlow, D. Batra, P. Kohli, and V. Kolmogorov. Dynamic tree block coordinate ascent. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 113–120, 2011.

[47] V. Vidal. 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*, pages 150–160, 2004.

[48] M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky. Map estimation via agreement on trees: message-passing and linear programming. *Information Theory, IEEE Transactions on*, 51(11):3697–3717, 2005.

[49] B. Wemmenhove, J. M. Mooij, W. Wiegerinck, M. Leisink, H. J. Kappen, and J. P. Neijt. Inference in the promedas medical expert system. In *Artificial Intelligence in Medicine*, volume 4594 of *Lecture Notes in Computer Science*, pages 456–460. Springer Berlin Heidelberg, 2007.

[50] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Constructing free-energy approximations and generalized belief propagation algorithms. *Information Theory, IEEE Transactions on*, 51(7):2282–2312, 2005.

Appendices

A Cost-Directed Look-ahead in AND/OR Search: Additional Proofs

A.1 Proposition 1

If $\bar{x}_p$ is a partial assignment, then the look-ahead heuristic for MBE can be expressed as

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

Proof. We rewrite the $h^d(\bar{x}_p)$ term by using **Definition 2.1** and unroll the recursive $h^{d-1}(\bar{x}_q)$ term. Doing so replaces the summation over children and minimization of each child with a minimization over all of the variables within the look-ahead subtree $\mathcal{T}_{p,d}$, plus the heuristic below the look-ahead subtree. Let $\bar{x}^{\downarrow}_{p,d}$ denote an extension to the assignment of all the variables in $\mathcal{T}_{p,d}$. Then we have

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

Using **Equation 1.2**, we replace $h(\bar{x}_p, \bar{x}^{\downarrow}_{p,d})$ with the appropriate term, yielding

$$h^d(\bar{x}_p) = \min_{\bar{x}^{\downarrow}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} f_k(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_j(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) \right\} \tag{1}$$

We then further expand Λ_k term using **Equation 1.1**,

$$h^d(\bar{x}_p) = \min_{\bar{x}^{\downarrow}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} f_k(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{X_q \in \mathcal{T}_{p,d} \cup \bar{X}_p} \lambda_{j \to q}(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) \right\}$$

Next, the second term's inner sum $\sum_{X_q \in \mathcal{T}_{p,d} \cup \bar{X}_p} \lambda_{j \to q}(\bar{x}_p, \bar{x}^{\downarrow}_{p,d})$ can be broken down into $\sum_{X_q \in \mathcal{T}_{p,d}} \lambda_{j \to q}(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_q \in \bar{X}_p} \lambda_{j \to q}(\bar{x}_p)$ where we drop the $\bar{x}^{\downarrow}_{p,d}$ argument in the second term since those messages do not contain any variables in $\mathcal{T}_{p,d}$. Substituting back into **Equation 1**, we obtain

$$h^d(\bar{x}_p) = \min_{\bar{x}^{\downarrow}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} f_k(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \left[\sum_{X_q \in \mathcal{T}_{p,d}} \lambda_{j \to q}(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_q \in \bar{X}_p} \lambda_{j \to q}(\bar{x}_p) \right] \right\}$$

Factoring out the terms that do not depend on the minimization over $\bar{x}^{\downarrow}_{p,d}$ and applying **Equation 1.1** on the factored out terms,

$$\begin{aligned} h^d(\bar{x}_p) &= \min_{\bar{x}^{\downarrow}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} f_k(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \left[\sum_{X_q \in \mathcal{T}_{p,d}} \lambda_{j \to q}(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) \right] \right\} \\ &\quad + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \sum_{X_q \in \bar{X}_p} \lambda_{j \to q}(\bar{x}_p) \\ &= \min_{\bar{x}^{\downarrow}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} f_k(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \left[\sum_{X_q \in \mathcal{T}_{p,d}} \lambda_{j \to q}(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) \right] \right\} \\ &\quad + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_j(\bar{x}_p) \end{aligned}$$

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

$$h^d(\bar{x}_p) = \min_{\bar{x}^{\downarrow}_{p,d}} \left\{ \sum_{X_k \in \mathcal{T}_{p,d}} \left[f_k(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \lambda_{j \to k}(\bar{x}_p, \bar{x}^{\downarrow}_{p,d}) \right] \right\} + \sum_{X_j \in \mathcal{T}_p - \mathcal{T}_{p,d}} \Lambda_j(\bar{x}_p)$$

By **Definition 2.4**, we replace the first term with $L^d(\bar{x}_p)$, therefore showing that

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

□

A.2 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 d-level look-ahead path $\{n_k^{opt(d)} \in N_k|_{0 \leq k \leq d}\}$. We derive the following to relate the 1-level 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 2.1**) 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 d-level 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 1-level residual (**Definition 2.2**), we can derive the following:

$$\begin{aligned} res^1(n_k^{opt(d)}) &= h^1(n_k^{opt(d)}) - h(n_k^{opt(d)}) \\ &\leq c(n_k^{opt(d)}, n_{k+1}^{opt(d)}) + h(n_{k+1}^{opt(d)}) - h(n_k^{opt(d)}) \end{aligned} \tag{2}$$

We will refer to Equation 2, which is an upper-bound on the 1-level 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 d-level residual $res^d(n)$.

Lemma 3. *If $res^d(n)$ is the d-level 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 d-level 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 d-level residual, we have

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

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

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

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 2, 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 d-level residual.

$$res^d(n) = \sum_{k=0}^{d-1} res^1_{\leq}(n_k^{opt(d)}) \geq \sum_{k=0}^{d-1} res^1(n_k^{opt(d)}) \tag{3}$$

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

$$res^1(n_k^{opt(d)}) \geq \min_{n_k \in N_k} res^1(n_k) \tag{4}$$

Applying Equations 3 and 4 together, we obtain our proposed statement.

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

□