Anytime Anyspace AND/OR Search for Bounding the Partition Function

Qi Lou
University of California, Irvine
Irvine, CA 92697, USA
qlou@ics.uci.edu

Rina Dechter
University of California, Irvine
Irvine, CA 92697, USA
dechter@ics.uci.edu

Alexander Ihler
University of California, Irvine
Irvine, CA 92697, USA
ihler@ics.uci.edu

Abstract

Bounding the partition function is a key inference task in many graphical models. In this paper, we develop an anytime anyspace search algorithm taking advantage of AND/OR tree structure and optimized variational heuristics to tighten deterministic bounds on the partition function. We study how our priority-driven best-first search scheme can improve on state-of-the-art variational bounds in an anytime way within limited memory resources, as well as the effect of the AND/OR framework to exploit conditional independence structure within the search process within the context of summation. We compare our resulting bounds to a number of existing methods, and show that our approach offers a number of advantages on real-world problem instances taken from recent UAI competitions.

Introduction

Probabilistic graphical models, including Bayesian networks and Markov random fields, provide a framework for representing and reasoning with probabilistic and deterministic information (Dechter, Geffner, and Halpern 2010; Darwiche 2009; Dechter 2013). Reasoning in a probabilistic graphical model often requires computing the partition function, i.e., the normalizing constant of the underlying distribution. In general, exact computation of the partition function is known to be #P-hard (Valiant 1979), leading to the development of a broad array of approximate schemes. Particularly useful are schemes that provide guarantees, such as a confidence interval (upper and lower bounds), that can also improve them in an anytime and anyspace manner.

Approximate elimination methods (Dechter and Rish 2003; Liu and Ihler 2011) and closely related variational bounds (Wainwright and Jordan 2008) provide deterministic guarantees on the partition function. However, these bounds are not anytime; their quality often depends on the amount of memory available, and do not improve without additional memory. On the other hand, Monte Carlo methods, such as those based on importance sampling (Liu, Fisher, and Ihler 2015), or approximate hash-based counting for weighted SAT (e.g., Chakraborty, Meel, and Vardi, 2016) can smoothly trade time for quality, but provide only probabilistic bounds (e.g., they hold with probability $1 - \delta$ for some confidence parameter δ), and can be slow to provide tight intervals.

In this work, we explore AND/OR search algorithms for providing anytime, deterministic bounds on the partition function. Historically, search techniques are well studied in graphical models for optimization (e.g., MAP, or maximum *a posteriori* and weighted CSP tasks), (Shimony and Charniak 1991; Santos 1991; Kask and Dechter 2001; F. Bacchus and Piassi 2003b) and exact summation (Darwiche 2001; Chavira and Darwiche 2008; F. Bacchus and Piassi 2003a; Sang, Beame, and Kautz 2005; Darwiche 2009). However, there have been relatively little recent study for approximate summation problems such as the partition function via search. One exception is Viricel et al. (2016), which adapts a depth-first branch-and-bound search scheme to provide deterministic upper and lower bounds on the partition function.

AND/OR search spaces (Dechter and Mateescu 2007) provide an elegant framework that exploits conditional, possibly context-specific independence structure during search. In contrast to methods such as recursive conditioning or "clamping" (Darwiche 2001; Weller and Domke 2015; Dechter and Mateescu 2007) and recent work on knowledge compilation (Kisa et al. 2014), that can also explore the AND/OR search space, most explicit AND/OR search algorithms were used for optimization, employing a fixed search order that restricts the search but enables the use of strong, pre-compiled heuristic functions and can lead to faster exploration and better early pruning (Marinescu and Dechter 2009a; 2009b; Otten et al. 2011; Otten and Dechter 2012; Marinescu, Dechter, and Ihler 2014; 2015).

Other, related approaches for summation queries include cutset-conditioning for exact solutions (Pearl 1988; Dechter 2013) or approximation with sampling (Bidyuk and Dechter 2007). Bidyuk, Dechter, and Rollon (2010) used conditioning to combine bound intervals on marginal probabilities.

Our contributions In this paper, we develop an anytime anyspace AND/OR best-first search algorithm to improve deterministic bounds for the partition function. Our priority-driven best-first search scheme takes advantage of both AND/OR tree search and optimized variational heuristics, to efficiently reduce the bound gap on the partition function. Empirical results demonstrate that our approach with heuristics extracted from weighted mini-bucket (Liu and Ihler 2011) is almost always superior to the baselines on various benchmark-memory settings.

Background

Let $\mathbf{X} = (X_1, \ldots, X_N)$ be a vector of random variables, where each X_i takes values in a discrete domain $\mathcal{X}_i$; we use lower case letters, e.g. $x_i \in \mathcal{X}_i$, to indicate a value of X_i. A graphical model over $\mathbf{X}$ consists of a set of factors $\mathbf{F} = \{f_\alpha(X_\alpha) \mid \alpha \in \mathcal{I}\}$, where each factor f_α is defined on a subset $X_\alpha = \{X_i \mid i \in \alpha\}$ of $\mathbf{X}$, called its scope.

We associate an undirected graph $\mathcal{G} = (V, E)$ with $\mathbf{F}$, where each node $i \in V$ corresponds to a variable X_i and we connect two nodes, $(i, j) \in E$, iff $\{i, j\} \subseteq \alpha$ for some α. The set $\mathcal{I}$ then corresponds to cliques of $\mathcal{G}$. We can interpret $\mathbf{F}$ as an unnormalized probability measure, so that

$$f(x) = \prod_{\alpha \in \mathcal{I}} f_\alpha(x_\alpha) \qquad Z = \sum_x \prod_{\alpha \in \mathcal{I}} f_\alpha(x_\alpha)$$

Z is a normalizing constant called the *partition function* which ensures that $p(x)$ sums to one. Computing Z is often a key task in evaluating the probability of observed data, model selection, or computing predictive probabilities.

AND/OR Search

An AND/OR search space is a generalization of the standard ("OR") search space that is able to exploit conditional independence structure during search (Dechter and Mateescu 2007). The AND/OR search space for a graphical model is defined relative to a *pseudo tree* that captures problem decomposition along a fixed search order.

Definition 1 (pseudo tree). *A pseudo tree of an undirected graph $\mathcal{G} = (V, E)$ is a directed tree $\mathcal{T} = (V, E')$ sharing the same set of nodes as $\mathcal{G}$. The tree edges E' form a subset of E, and we require that each edge $(i, j) \in E \setminus E'$ be a "back edge", i.e., the path from the root of $\mathcal{T}$ to j passes through i (denoted $i \leq j$). $\mathcal{G}$ is called the primal graph of $\mathcal{T}$.*

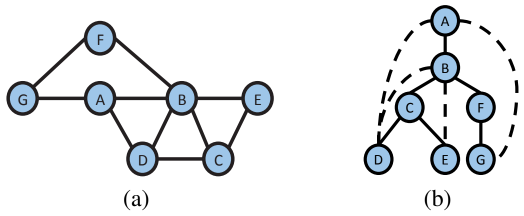
One simple way to build a pseudo tree for $\mathcal{G}$ is via depth-first search along a reverse elimination order.

Example. Fig. 1(a) shows the primal graph of a pairwise model. Fig. 1(b) shows one pseudo tree consistent with the elimination order G, F, E, D, C, B, A.

Guided by a pseudo tree, we can construct an AND/OR search tree consisting of alternating levels of OR and AND nodes for a graphical model. Each OR node s is associated with a variable, which we lightly abuse notation to denote X_s; the children of s, $ch(s)$, are AND nodes corresponding to the possible values of X_s. The root $\emptyset$ of the AND/OR search tree corresponds to the root of the pseudo tree. Let $pa(c) = s$ indicate the parent of c, and $an(c) = \{n \mid n \leq c\}$ indicate the ancestors of c (including itself) in the tree.

In an AND/OR tree, any AND node c corresponds to a partial configuration $x_{\leq c}$ of X, defined by its assignment and that of its ancestors: $x_{\leq c} = x_{\leq p} \cup \{X_s = x_c\}$, where $s = pa(c)$, $p = pa(s)$. A complete configuration $X = x$ of the model corresponds to a subtree called a *solution tree*:

Definition 2 (solution tree). *A solution tree T of an AND/OR search tree $\mathcal{T}$ is a subtree satisfying three conditions: (1) T contains the root of $\mathcal{T}$; (2) if an OR node is in T, exactly one of its children is in T; (3) if an AND node is in T, all of its children are in T.*



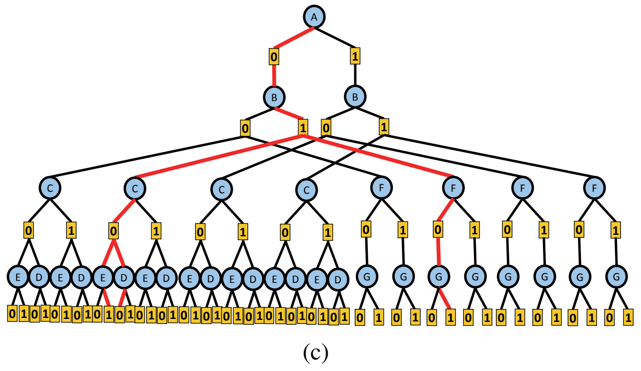


Figure 1: (a) A primal graph of a graphical model over 7 variables with unary and pairwise potential functions. (b) A pseudo tree for the primal graph. (c) AND/OR search tree guided by the pseudo tree. One solution tree is marked red.

We also associate a weight w_c with each AND node, defined to be the product of all factors f_α that are instantiated at c but not before:

$$w_c = \prod_{\alpha \in \mathcal{I}_c} f_\alpha(x_\alpha), \quad \mathcal{I}_c = \{\alpha \mid X_{pa(c)} \in X_\alpha \subseteq X_{an(c)}\}$$

For completeness, we define $w_s = 1$ for any OR node s. It is then easy to see that the product of weights on a path to the root, $g_c = \prod_{a \leq c} w_a$ (termed the *cost* of the path), equals the value of the factors whose scope is fully instantiated at c. The product of weights on any solution tree equals the value of its complete configuration.

Example. A solution tree is marked red in the AND/OR search tree shown in Fig. 1(c). According to the definition, a solution tree corresponds to one full instantiation of all the variables. Its cost is defined as product of weights of all its AND nodes, and equals the value of this configuration in the graphical model.

Finally, the purpose of the search tree is to compute some inference quantity for the model, such as the optimum $f^* = \max_x f(x)$ or the partition function $Z = \sum_x f(x)$. To this end, we associate an exact "value" v_n with each node n in the AND/OR search graph, which represents the inference task's value on the unexpanded portion of the search space below node n. The value v_n can be defined recursively in terms of its children and grandchildren as follows. We first define $v_n = 1$ for any leaf (since no part of the model remains uninstantiated). Let p be an AND node; for maximization tasks, we have

$$\text{Max:} \qquad v_p = \prod_{s \in ch(p)} v_s, \qquad v_s = \max_{c \in ch(s)} w_c v_c \qquad (1)$$

while in the case of summation, the recursion defining v_p for an AND node p is

$$\text{Sum:} \quad v_p = \prod_{s \in ch(p)} v_s, \qquad v_s = \sum_{c \in ch(s)} w_c v_c \tag{2}$$

Any search algorithm for reasoning about the model can be thought of as maintaining upper and lower bounds on these quantities at each node. In particular, for heuristic search, we assume that we have a heuristic function h_c that gives upper (or upper and lower) bounds on v_c. These heuristics typically are more accurate deeper in the search tree, and therefore their updates can be propagated upwards to the root to yield tighter bounds to the overall inference value. Any search algorithm is then defined by the order of expansion of the search tree.

Proposition 1. Any algorithm exploring the full AND/OR tree has complexity $O(b^h d^h)$, where h is the pseudo tree height, b is the max branching factor of the pseudo tree, and d is the max variable domain size. See Dechter and Mateescu (2007).

Best-first Search

We explore the power of AND/OR search with precompiled heuristics for bounding sum inference tasks. In this section, we introduce our main algorithm; we first present a simplified version that is A*-like, and then generalize to an SMA*-like version (Russell 1992) to operate with limited memory.

Beginning with only the root $\emptyset$, we expand the search tree according to the pseudo tree structure and some control strategy (depth-first, breadth-first, etc.) With best-first search, we assign a priority to each frontier node on the search tree, then expand the top priority frontier node in each iteration.

More precisely, we maintain an explicit AND/OR search tree of visited nodes, denoted $\mathcal{S}$, whose frontier nodes are denoted *OPEN*. The remaining nodes of $\mathcal{S}$ are called *internal* nodes. For an internal node n, we denote *OPEN(n)* as the set of descendants of n that are in *OPEN*. A frontier AND node of $\mathcal{S}$ is called *solved* if it corresponds to a leaf node in the pseudo tree. An internal node of $\mathcal{S}$ is solved only if all its children are solved.

For each node n in the AND/OR search tree, we denote u_n and l_n to be upper and lower bounds of v_n, initialized via pre-compiled heuristics $h_n^- \leq v_n \leq h_n^+$, and subsequently updated during search.

Given the currently expanded tree $\mathcal{S}$, we update the bounds u_n and l_n using information propagated from the frontier:

$$\begin{aligned} &\text{AND node } p\text{:} \quad u_p = \prod_{s \in ch(p)} u_s, \quad l_p = \prod_{s \in ch(p)} l_s \\ &\text{OR node } s\text{:} \quad u_s = \sum_{c \in ch(s)} w_c\, u_c, \quad l_s = \sum_{c \in ch(s)} w_c\, l_c \end{aligned} \tag{3}$$

Note that these values depend implicitly on the search tree $\mathcal{S}$.

It will also be useful to define two node-specific quantities that combine the path costs and heuristics. Let $branch(p, q)$ be all OR nodes that are siblings of some node s on the path $p < s \leq q$; these are "other" children of AND-node ancestors of q. For shorthand we use $branch(q) = branch(\emptyset, q)$ for paths from the root.

$$U_n = g_n\, u_n \prod_{s \in branch(n)} u_s, \quad L_n = g_n\, l_n \prod_{s \in branch(n)} l_s \tag{4}$$

These quantities combine the path cost and heuristic in the usual way, e.g., the path cost g_n times the current heuristic bound, u_n; the branch terms correspond to incorporating the bounds for conditionally independent subproblems for sibling subtrees at AND nodes. At the root, the dynamically updated bounds $U = U_\emptyset = u_\emptyset$, $L = L_\emptyset = l_\emptyset$ serve as anytime bounds on Z. U_n and L_n can also be understood in terms of the exact quantity

$$V_n = g_n\, v_n \prod_{s \in branch(n)} v_s, \tag{5}$$

and we show in Appendix,

Proposition 2. The quantity V_n represents the total weighted sum over all solution trees that include node n. Given a current search tree $\mathcal{S}$, for all n, we have $U_n \geq V_n \geq L_n$.

Thus, we can interpret U_n and L_n as the contributions of n to the global bounds U and L.

Priority Types

A critical element of best-first search is the priority value for each frontier node, which determines which node will be expanded next. However, some complications arise for AND/OR search, and particularly for summation queries, which we discuss here.

In typical A* search, we select the frontier node with the lowest cost incurred so far, plus an optimistic heuristic estimate of the cost to go. In our notation and for maximization, this corresponds to selecting $s \in$ *OPEN* to maximize U_s; this choice will tighten the current global upper bound U, and we refer to it as the "upper" priority. However, for summation problems it is more natural to tighten the current gap, $U - L$, which suggests selecting s to maximize $U_s - L_s$ (the "gap" priority); more precisely, in Appendix we show:

Proposition 3. Given a current search tree $\mathcal{S}$, fully solving the summation problem below a frontier node s will tighten the bound difference $U - L$ by

$$g_s\,(u_s - v_s) \prod_{t \in branch(s)} u_t + g_s\,(v_s - l_s) \prod_{t \in branch(s)} l_t \tag{6}$$

which is upper bounded by $gap(s) = U_s - L_s$.

Thus, $gap(s)$ serves as an optimistic estimate of the effect of expanding s on the global bound difference. For OR search, this essentially reduces to the priority proposed in Henrion (1991).

However, in AND/OR search, tracking the highest priority node can be difficult. In particular, both the upper and gap priorities are *non-static*: after expanding a node in *OPEN*, the priority of other nodes may change their values and relative orders. Consider two nodes which share an AND ancestor; this means that the nodes correspond to conditionally independent subproblems, and their contributions are multiplied

Algorithm 1 Best-first search for anytime sum bounds.

```
1: Initialize $\mathcal{S} \leftarrow \{\emptyset\}$ with the root $\emptyset$.
2: while termination conditions not met
3:     ExpandBest($\emptyset$, 1, 1) // find best frontier, from root
4: end while
5: function ExpandBest($n$, $U_n$, $L_n$)
6:     if $n \notin OPEN$          // not frontier; recurse down:
7:         Find $c^+$ via (7) or (8)
8:         ExpandBest($c^+$, $U_{c^+}$, $L_{c^+}$)
9:     else                        // expand frontier node:
10:        Generate children of $n$; $u_c = h_c^+$, $l_c = h_c^-$.
11:        Mark any leaves (or $u_c = l_c$) as SOLVED.
12:    end if
13:    Update $u_n$, $l_n$ via (3)
14:    if all children $c \in ch(n)$ are SOLVED
15:        Remove $ch(n)$ from $\mathcal{S}$; add $n$ to SOLVED.
16:    end if
17:    Find $c^+$ and update $U_n^+$, $L_n^+$ by (7) or (8)-(9).
18: end function
```

in (1)–(2). Since *both* branches must be solved, proving that one branch is no good tells us not to explore the other branch, and reduces its priority.

For the upper bound heuristic at a node n, it is easy to show that, if no descendant of n is updated, the relative order of n's children does not change. To see this, write $U_s = U_n(U_s/U_n)$ for any frontier node s descended from n, $s \geq n$. The second term, U_s/U_n, involves only descendants of n; thus, changes to ancestors of n can only affect U_n and hence cannot change the relative order of the descendants. Then, by maintaining at each node n,

$$U_n^+ = \max_{s \geq n, s \in OPEN} U_s/U_n$$

we can always identify the highest-priority child of n, and propagate U_c^+ upward, by computing

$$c^+ = \operatorname*{argmax}_{c \in ch(n)} U_c U_c^+ \quad \text{and} \quad U_n^+ = U_{c^+} U_{c^+}^+ / U_n \tag{7}$$

Unfortunately, the gap heuristic is less well-behaved. At a node n, the descendant gap $U_n(U_s/U_n) - L_n(L_s/L_n)$ *can* change the relative order if n's ancestors are updated and U_n and L_n change values. We elect to track the gap priority approximately, by computing

$$c^+ = \operatorname*{argmax}_{c \in ch(n)} \; U_c U_c^+ - L_c L_c^+ \tag{8}$$

$$U_n^+ = U_{c^+} U_{c^+}^+ / U_n, \qquad L_n^+ = L_{c^+} L_{c^+}^+ / L_n \tag{9}$$

This is not guaranteed to find the current highest gap priority node, but will become more accurate as the search tree grows and the U_n, L_n become more accurate (and thus stable).

The overall algorithm is given in Alg.1. Each iteration recurses from the root to the best frontier node, scoring n's children and computing U_n, L_n. At the frontier, we expand a node, and recurse back up the tree, updating the bounds u_n, l_n and descendant priority quantities U_n^+, L_n^+. We can show:

Algorithm 2 Memory-limited BFS for anytime sum bounds.

```
1: Initialize $\mathcal{S} \leftarrow \{\emptyset\}$ with the root $\emptyset$.
2: while termination conditions not met
3:     if memory OK:   $n \leftarrow$ExpandBest($\emptyset$, 1, 1)
4:     else   $n \leftarrow$RemoveWorst($\emptyset$, 1, 1)
5:     end if
6: end while
7: function RemoveWorst($n$, $U_n$, $L_n$)
8:     if $n$'s children all in OPEN  // worst removable node
9:         Remove $ch(n)$ from $\mathcal{S}$; mark $n$ in OPEN
10:    else                           // or recurse toward worst
11:        Find worst non-OPEN child $c^-$ via (10) or (11)
12:        RemoveWorst($c^-$, $U_{c^-}$, $L_{c^-}$)
13:    end if
14:    Update $c^-$ and $U_n^-$, $L_n^-$ via (10) or (11)–(12)
15: end function
16: function ExpandBest($n$, $U_n$, $L_n$)
17:     // As in Alg. 1, except:
18:     Ensure $u_n$, $l_n$, $U_n^+$, $L_n^+$ updated monotonically
19:     Update $c^-$ and $U_n^-$, $L_n^-$ via (10) or (11)–(12)
20: end function
```

Proposition 4. The time complexity of each node expansion and update in Alg. 1 is bounded by $O(h(b+d))$ where h is the pseudo tree height, b is the max branching factor of the pseudo tree, and d is the max variable domain size.

Memory-limited BFS

As is typical for best-first search, memory usage can quickly become a major bottleneck. To continue improving in memory-limited settings, we could switch to some low-memory strategy such as depth-first search (DFS). However, this is often slow to tighten the bounds.

Instead, we apply a variant of SMA* (Russell 1992), so that near the memory limit, we continue expanding nodes in a best-first way, but remove low-priority nodes from $\mathcal{S}$, in such a way that they will be re-generated once the high-priority subtrees are tightened or solved. We simply modify our updates in two ways: (1) at each node n, we also track the lowest-priority *removable* descendant of n; and (2) we force u_n, l_n, and the node priority quantities U_n^+, L_n^+ to be updated monotonically, to avoid worsening the bounds or overestimating priority when subtrees are removed and later re-generated. The resulting memory-limited best-first algorithm is shown in Alg. 2.

For convenience, we define a node as removable if its children are all in *OPEN*, and "remove" it by deleting its children and re-adding it to *OPEN*; this simplifies tracking and re-expanding removed nodes. We keep track of the smallest priority value of any removable node below n; for the upper heuristic,

$$c^- = \operatorname*{argmin}_{c \in rm(n)} U_c U_c^- \quad \text{and} \quad U_n^- = U_{c^-} U_{c^-}^- / U_n \tag{10}$$

with $rm(n) = ch(n) \setminus OPEN$, i.e., the children of n not in *OPEN*, and $U_n^- = U_n^+$ at removable nodes. For the gap priority, we again track only approximately, using

$$c^- = \underset{c \in rm(n)}{\operatorname{argmin}}\ U_c U_c^- - L_c L_c^- \tag{11}$$

$$U_n^- = U_{c^-} U_{c^-}^- / U_n, \qquad L_n^- = L_{c^-} L_{c^-}^- / L_n \tag{12}$$

Then, to remove a node, we search downward along the worst children c^-, and remove n when its children are all in *OPEN*.

Empirical Evaluation

To evaluate the effectiveness of our algorithm in Alg. 2, we compare it to a number of existing methods on four benchmarks and three different memory budgets.

Experimental Settings

Benchmarks We evaluated performance on several benchmark instance sets: CPD, a set of computational protein design problems from Viricel et al. (2016); PIC'11, a benchmark subset of 23 instances selected by Viricel et al. (2016) from the 2012 UAI competition; BN, a set of Bayesian networks from the 2006 competition[1]; and Protein, made from the "small" protein side-chains of Yanover and Weiss (2002). For CPD, BN, and Protein, we evaluated on randomly selected subsets of size 100, 50, and 50, respectively. Table 1 gives statistics on the composition of these benchmarks.

Methods Our proposed algorithm (Alg. 2), which we call **AOBFS**, is tested in two variants: using the "gap" priority (denoted **A-G**), and using the "upper" priority (**A-U**). We also compare an OR search variant, **O-G**, by selecting a chain-structured pseudo tree. For our experiments, we use weighted mini-bucket (Liu and Ihler 2011) heuristics, whose memory use is roughly controlled by a parameter called the *ibound*. For a given memory budget, we first compute the largest ibound that fits the memory budget, and then use the remaining memory for search. (While a more carefully chosen allocation strategy might be able to yield better performance, we leave this for future work.)

$\boldsymbol{Z_\varepsilon^*}$ (Viricel et al. 2016) is a recent algorithm that provides lower and upper bounds on the partition function. For a given ε, it returns bounds on $\ln Z$ whose gap is at most $\ln(1+\varepsilon)$. We adopted the parameter setup suggested by the authors: upper bound set to Ub_1, enforcing VAC at the root, and using EDAC during the rest of the search.

VEC (Dechter 1999; 2013) (Variable Elimination with Conditioning, also known as custet-conditioning) is a classical method for trading off memory with time when the induced-width of the problem is too high. The algorithm determines the highest induced-width for which it can run the variable elimination algorithm, denoted w, and then searches over a w-cutset set of nodes, applying variable elimination to each assignment of the cutset. For implementation details, see http://graphmod.ics.uci.edu/group/VEC.

MMAP-DFS (Marinescu, Dechter, and Ihler 2014) (abbreviated **M-D**) is a state-of-the-art method for marginal MAP using AND/OR search, which solves the internal summation problem exactly using depth-first search aided by weighted mini-bucket heuristics. We use it to compute the partition function by treating all variables as summation variables. Since it also uses weighted mini-bucket, we use the same *ibound* selected for the heuristics in our algorithm.

[1]http://melodi.ee.washington.edu/~bilmes/uai06InferenceEvaluation/

	PIC'11	Protein	BN	CPD
# instances	23	50	50	100
avg. # variable	104.04	99.96	838.60	15.68
avg. # factor	409.09	355.84	838.60	135.32
avg. max domain size	2.91	77.94	12.44	32.54
avg. # evidence	0	0	96.04	0
avg. induced width	16.35	11.24	32.78	14.68
avg. pseudotree depth	26.09	27.66	112.46	12.27

Table 1: Statistics of the four evaluated benchmark sets.

Viricel et al. (2016) also compared several solvers, minic2d (Oztok and Darwiche 2015), ace (Chavira and Darwiche 2008), and cachet (Sang, Beame, and Kautz 2005), but found that even with significant memory (60GB), minic2d solved about 13%, ace solved about 5%, and cachet solved 7% of the CPD instances, hence we did not include them in our evaluation.

Implementations of all methods are in C/C++ by the original authors. We used maximum time 1 hour, and tested three memory budgets: 1GB, 4GB and 16GB.

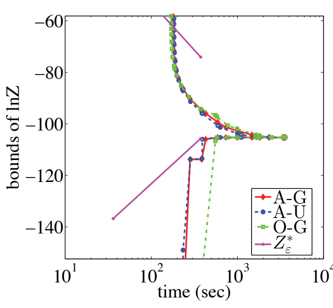
Empirical Results

The main goal of our algorithm is to provide improving, anytime bounds. While Z_ε^* is able to reduce its quality requirement (by increasing ε), it is not anytime in this sense. To simulate anytime-like behavior, we vary ε across the range $\ln(\varepsilon+1) \in [10^{-3}, 10^2]$ to provide a sequence of quality/time pairs. Fig. 2 shows the anytime behavior of our algorithm along with these Z_ε^* runs on one instance per benchmark. Generally speaking, our search process tightens the bound much more quickly and smoothly than Z_ε^* . For example, in Fig. 2(a), only very coarse bounds are produced within the time limit; in (b), the Z_ε^* bounds are loose until our proposed algorithm has essentially solved the problem (with heuristic construction comprising most of the total time); in (c) Z_ε^* did not produce any useful bound within the time limit.

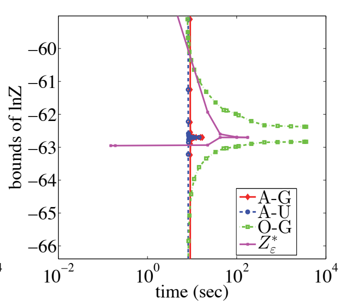
Within our algorithm's variants, we see that both AND/OR variants perform similarly, with the OR variant similar or slower (e.g., Fig. 2(b)). We find that our gap-based priority can be slower initially, but usually improves to do better than upper priority after enough time; for example, in Fig. 2(c).

Solving to fixed tolerance We evaluate the number of instances in each benchmark that can be solved to tolerance (so that the gap between the upper and lower bound is less than ϵ) within the memory and time constraints. Table 2 shows the performance of each algorithm at two thresholds, "loose" ($\epsilon = 1.0$) and "tight" ($\epsilon = 10^{-3}$). Exact methods (VEC, M-D) are included with the tight tolerance interval.

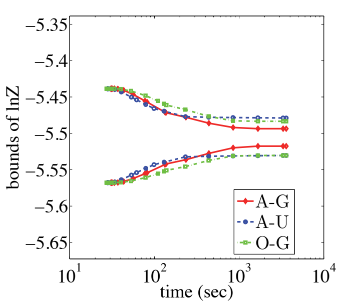
From Table 2, we can see that **A-G** solves the most instances to tight tolerance in 9 of 12 settings; **A-U** is only slightly worse, and also improves over the previous approaches. (Again, we see that **O-G** solves fewer instances, emphasizing the usefulness of exploiting conditional independence in the AND/OR space.) An exception is on CPD,



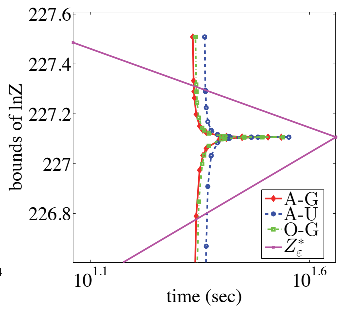
(a) PIC'11/queen5_5_4



(b) Protein/1g6x

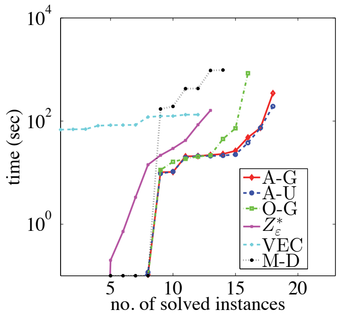


(c) BN/BN_8

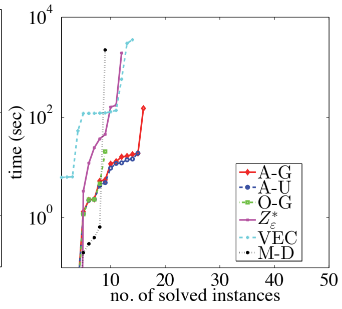


(d) CPD/237_SER

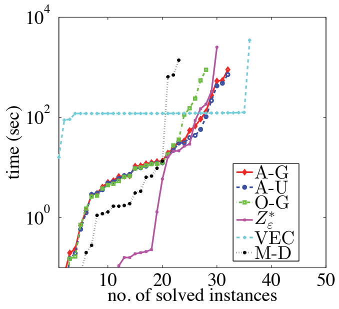
Figure 2: Anytime behavior of AOBFS on one instance per benchmark. For **A-G**, **A-U**, and **O-G**, anytime bounds on $\ln Z$ are truncated at $\ln U - \ln L = 10^{-3}$. Curves for Z_ε^* may be missing since it may not be able to produce bounds or bounds are not in a reasonable scope for $\ln(\varepsilon + 1) \in [10^{-3}, 10^2]$.



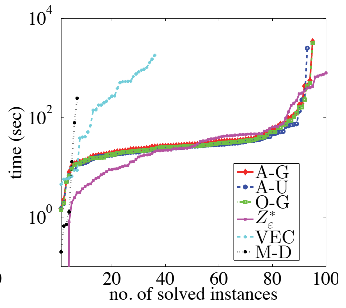
(a) PIC'11



(b) Protein

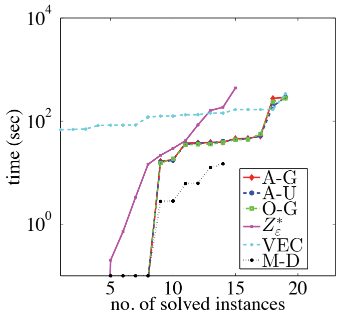


(c) BN

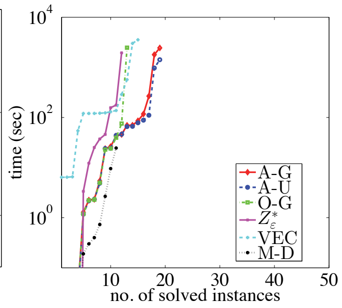


(d) CPD

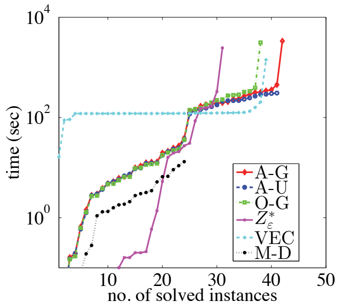
Figure 3: Number of solved (to tolerance 10^{-3}) instances with memory 1GB. The time budget is 1 hour. Sizes of four benchmarks (from left to right) are 23, 50, 50 and 100 respectively. Lower is better.



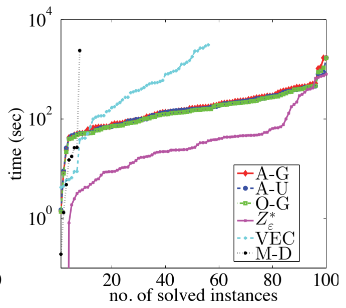
(a) PIC'11



(b) Protein



(c) BN



(d) CPD

Figure 4: Number of solved (to tolerance 10^{-3}) instances with memory 16GB. The time budget is 1 hour. Sizes of four benchmarks (from left to right) are 23, 50, 50 and 100 respectively. Lower is better.

which Z_ε^* performs slightly better at lower memory bounds. When looser bounds are acceptable, Z_ε^* solves only one additional problem at 16GB, and two additional problems at lower memory; it has difficulty taking advantage of anytime tradeoffs. In contrast, the proposed search algorithms solve significantly more problems to loose tolerance.

Fig. 3 and Fig. 4 show the time required to solve (to tolerance 10^{-3}) the instances in each benchmark with memory 1GB and 16GB respectively. Again, we find that Z_ε^* is usually faster on CPD problems, but slower than our proposed method on PIC'11 and Protein, and faster on only the easiest of the BN instances. Among exact methods, **M-D** is often faster on simple models, but then fails to solve harder models, while **VEC** is usually both slower and solves fewer models than our best-first search. We also observe that more memory often leads to better performance for all methods. Our algorithm and **VEC** generally improve more compared to other baselines with more memory.

“Loose”: $\log U - \log L < 1.0$

	PIC’11	Protein	BN	CPD
#inst.	23	50	50	100
Memory: 1GB/4GB/16GB				
A-G	**20/20/21**	**23/29/30**	**39/42/43**	**100/100/100**
A-U	**20/20/21**	**23/29/30**	**39/42/43**	**100/100/100**
O-G	19/**20/21**	12/13/16	35/37/40	**100/100/100**
Z^*_ε	14/14/15	13/13/13	30/31/31	**100/100/100**

“Tight”: $\log U - \log L < 10^{-3}$

	PIC’11	Protein	BN	CPD
#inst.	23	50	50	100
Memory: 1GB/4GB/16GB				
A-G	**18/18/19**	**16/17/19**	32/**40/42**	95/98/**100**
A-U	**18/18/19**	15/**17/19**	32/**40**/41	93/95/**100**
O-G	16/**18/19**	9/12/13	28/36/38	95/98/**100**
Z^*_ε	13/13/15	12/12/12	30/31/31	**100/100/100**
VEC	12/14/**19**	14/15/15	**36**/38/39	36/52/56
M-D	14/14/14	9/9/11	23/23/24	7/7/8

Table 2: Number of instances solved to tolerance interval for each benchmark & memory setting. Each entry contains the number of solved instances in 1 hour with memory budget 1GB, 4GB, and 16GB from left to right. The highest (most solved) for each setting is bolded. Exact methods (VEC, M-D) are compared to the “tight” tolerance 10^{-3}.

Search can sometimes effectively solve instances with very high width, if there are relatively few high-weight configurations of the model. For example, in the Protein instances, **A-G** solves instance ‘1who’ in only 12 seconds and 1GB memory, while the corresponding junction tree requires about 150GB memory; even more extreme, instance ‘2fcr’ is solved in 21 minutes and 16GB memory, while junction tree would require approximately 3.5PB.

A-G vs A-U As we observed from Fig. 2, **A-U** may outperform **A-G** early on in search, but **A-G** usually catches up. One possible explanation is that, early on, the lower bound heuristics from weighted mini-bucket are often significantly worse than the corresponding upper bounds, and thus the gap, $U - L$, is very close to U, and the priorities are similar. However, our approximate procedure may not identify the top gap-priority node; and moreover, **A-U** priority requires slightly less computation than that of **A-G**, making it more efficient during early search. However, as search proceeds and the heuristics become more accurate, focusing on nodes that have high gap, rather than merely high value, pays off; for example, **A-G** solves slightly more instances than **A-U** to tolerance in Table 2.

Conclusion

In this paper, we develop an anytime anyspace search algorithm for bounding the partition function. It is a priority-driven best-first search scheme on the AND/OR search tree based on precompiled heuristics from state-of-the-art variational bounds, and is able to improve on these bounds in an anytime fashion within limited memory resources. The AND/OR search tree enables exploiting conditional independence during search, while the heuristics and priority guide it toward critical subtrees of the search space. In experimental comparisons, our best-first algorithm outperforms existing, state-of-the-art baseline methods on multiple standard benchmarks and memory limits.

Acknowledgements

We thank Clément Viricel, Vibhav Gogate, and Radu Marinescu for sharing code/data with us. We appreciate the help from William Lam for proofreading.

This work is sponsored in part by NSF grants IIS-1526842, IIS-1254071, and by the United States Air Force under Contract No. FA8750-14-C-0011 under the DARPA PPAML program.

References

Bidyuk, B., and Dechter, R. 2007. Cutset sampling for bayesian networks. *Journal of Artificial Intelligence Research* 28:1–48.

Bidyuk, B.; Dechter, R.; and Rollon, E. 2010. Active tuples-based scheme for bounding posterior beliefs. *Journal of Artificial Intelligence Research* 39:335.

Chavira, M., and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172(6):772–799.

Darwiche, A. 2001. Recursive conditioning. *Artificial Intelligence* 126(1):5–41.

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

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

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

Dechter, R.; Geffner, H.; and Halpern, J. Y. 2010. *Heuristics, Probability and Causality. A Tribute to Judea Pearl*. College Publications.

Dechter, R. 1999. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* 113(1-2):41–85.

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

F. Bacchus, S. D., and Piassi, T. 2003a. Algorithms and complexity results for #sat and bayesian inference. In *FOCS 2003*.

F. Bacchus, S. D., and Piassi, T. 2003b. Value elimination: Bayesian inference via backtracking search. In *Uncertainty in AI (UAI03)*.

Henrion, M. 1991. Search-based methods to bound diagnostic probabilities in very large belief nets. In *Proceedings of the Seventh conference on Uncertainty in Artificial Intelligence*, 142–150. Morgan Kaufmann Publishers Inc.

Kask, K., and Dechter, R. 2001. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence* 129(1):91–131.

Kisa, D.; den Broeck, G. V.; Choi, A.; and Darwiche, A. 2014. Probabilistic sentential decision diagrams. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*.

Liu, Q., and Ihler, A. 2011. Bounding the partition function using Hölder's inequality. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*.

Liu, Q.; Fisher, III, J. W.; and Ihler, A. T. 2015. Probabilistic variational bounds for graphical models. In *Advances in Neural Information Processing Systems*, 1432–1440.

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

Marinescu, R., and Dechter, R. 2009b. And/or branch-and-bound search for combinatorial optimization in graphical models. *Artificial Intelligence* 173(16):1457–1491.

Marinescu, R.; Dechter, R.; and Ihler, A. 2014. And/or search for marginal map. In *Uncertainty in Artificial Intelligence (UAI)*, 563–572.

Marinescu, R.; Dechter, R.; and Ihler, A. 2015. Pushing forward marginal map with best-first search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 696–702.

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

Otten, L.; Ihler, A.; Kask, K.; and Dechter, R. 2011. Winning the pascal 2011 map challenge with enhanced and/or branch-and-bound.

Oztok, U., and Darwiche, A. 2015. A top-down compiler for sentential decision diagrams. In *Proceedings of the 24th International Conference on Artificial Intelligence*, 3141–3148. AAAI Press.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc.

Russell, S. 1992. Efficient memory-bounded search methods. In *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, 1–5.

Sang, T.; Beame, P.; and Kautz, H. 2005. Solving bayesian networks by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence*, volume 1 of *AAAI'05*, 475–482.

Santos, E. 1991. On the generation of alternative explanations with implications for belief revision. In *Uncertainty in Artificial Intelligence (UAI'91)*, 339–347.

Shimony, S., and Charniak, E. 1991. A new algorithm for finding map assignments to belief networks. In *P. Bonissone, M. Henrion, L. Kanal, and J. Lemmer (Eds.), Uncertainty in Artificial Intelligence*, volume 6, 185–193.

Valiant, L. 1979. The complexity of computing the permanent. *Theoretical Computer Science* 8(2):189 – 201.

Viricel, C.; Simoncini, D.; Barbe, S.; and Schiex, T. 2016. Guaranteed weighted counting for affinity computation: Beyond determinism and structure. In *International Conference on Principles and Practice of Constraint Programming*, 733–750. Springer.

Wainwright, M., and Jordan, M. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning* 1(1-2):1–305.

Weller, A., and Domke, J. 2015. Clamping improves trw and mean field approximations. *arXiv preprint arXiv:1510.00087*.

Yanover, C., and Weiss, Y. 2002. Approximate inference and protein-folding. In *Advances in neural information processing systems*, 1457–1464.

Appendix

Proof of Proposition 2

First, it is obvious that U_n and L_n are upper and lower bounds of V_n respectively. For a *solution* tree $T \in \mathbb{T}(\mathcal{S})$ where $\mathbb{T}(\mathcal{S})$ denotes the set of all solution trees of $\mathcal{S}$, its mass is

$$v_T = \prod_{c \in T} w_c \prod_{s \in T \cap OPEN} v_s \tag{13}$$

T also defines an upper bound of v_T denoted as u_T where

$$u_T = \prod_{c \in T} w_c \prod_{s \in T \cap OPEN} u_s \tag{14}$$

By taking the sum of masses over all the solution trees of $\mathcal{S}$, we obtain the exact partition function. Analogously, by taking the sum of upper bounds over all its solution trees, we can upper bound the partition function. Namely,

$$U = \sum_{T \in \mathbb{T}(\mathcal{S})} u_T \tag{15}$$

Now, for any solution tree T that contains n, g_n contributes to v_T as a multiplicative factor. The rest of v_T comes from subtrees underneath nodes in $\{n\} \bigcup branch(n)$. Thus, it is easy to verify that

$$V_n = \sum_{\{T \in \mathbb{T}(\mathcal{S}) \mid n \in T\}} v_T \tag{16}$$

i.e., V_n is the total mass of all solution trees that contain n.

Proof of Proposition 3

We can see that fully solving the subproblem underneath s will decrease U_s to

$$g_s \, v_s \prod_{t \in branch(s)} u_t \tag{17}$$

This will improve U by

$$g_s \, (u_s - v_s) \prod_{t \in branch(s)} u_t \tag{18}$$

By applying the same argument to lower bound, we know the bound difference $U - L$ will be reduced by

$$g_s \, (u_s - v_s) \prod_{t \in branch(s)} u_t + g_s \, (v_s - l_s) \prod_{t \in branch(s)} l_t \tag{19}$$

which is obviously a lower bound of $gap(s) = U_s - L_s$.