A Note on Bounding the Partition Function by Search

Eyal Dechter and Rina Dechter, An ICS Technical report

Tuesday 1st March, 2016

Abstract

We develop basic best-first and depth-first search algorithms for providing upper and lower bounds on the partition function in an anytime manner. We also develop an ϵ-gurantee depth-first search scheme for OR and AND/OR search trees that corresponds to a given graphical model.

1 Introduction

We assume a weighted directed graph G, representing a global function (generated from a probabilistic graphical model) where the weights on the arcs are numbers between zero and one. For an OR graph the cost of a solution path is the product of the weights on the arcs. For an AND/OR graph it is the product of the weights on a solution subtree (or graph). The task is to compute the sum cost over all solutions (paths or trees, respectively), denoted Z.

Starting from OR graphs we seek an anytime algorithm that generates a lower and upper bounds on Z, the partition function. We can associate with every node a value $V(n)$ which stands for the exact weighted count in the subgraph that it roots. $g(n)$ is the cost of the partial path to n. The conditional weighted count of n denoted by $F(n)$ is defined by $F(n) = g(n) \cdot V(n)$. A leaf node is "solved" if it corresponds to a full solution (a configuration of all the variables). An internal node is solved if all its child nodes are solved.

We use search algorithms that expand the search graph starting at the root of the graph. The search can be expanded depth first, breadth first or by any other control strategy. The algorithm maintains an explicit search graph of expanded nodes denoted G', whose frontier leaf nodes, called OPEN, are associated with $g(n)$ and with an upper bound $up(n)$ and a lower bound $lo(n)$ on $V(n)$. In the uninformed case $lo(n) = 0$ and $up(n) = 1$.

Given a currently expanded graph G' for OR search space, internal nodes can have their lower and upper bounds propagated from the frontier nodes, according to the following definition.

DEFINITION **1** (bound propagation for OR nodes)**.** *Given the currently explored graph G', for any node $n \in G'$ its lower and upper bounds can be computed recursively:*

- *If n is a non-solved leaf node $U(n) = 1$ and $L(n) = 0$*
- *If n is a solved leaf node $U(n) = L(n) = 1$*
- *If n is an internal node then*

$$U(n) = \sum_{s \in ch(n)} w(n, s) \cdot U(s) \tag{1}$$

and

$$L(n) = \sum_{s \in ch(n)} w(n, s) \cdot L(s)$$

labeleq1

We distinguish between static $up(n)$ and $lo(n)$ which are pre-compiled versus dynamic $L(n)$ and $U(n)$ that are updated. At a fixed point when all its upper and lower bounds were propagated from the leaves, the above equalities hold for every node in G'.

Clearly, at a fixed point, the upper and lower bounds of the root node, s_0, is the answer provided by the current graph G'.

Computation over cutset. The upper and lower bounds of G' can also be obtained by summing across any full cutset nodes of G' whenever G' is at a fixed point.

DEFINITION **2** (cutset)**.** *Given a graph G', a cutset C is any subset of nodes of G' s.t, 1. if we remove C from G' there is no path from the root to any of the (remaining) leaves of G', and 2. C does not contain more than a single node on any path in G'.*

Clearly, the root is a cutset and also the set of leaves of G' is a cutset. If G' in at a fixed point, and given any cutset C, then the total upper and lower bounds of G'.

$$U(G') = \sum_{s \in C} g(s) \cdot U(s) \tag{2}$$

and

$$L(G') = \sum_{s \in C} g(s) \cdot L(s)$$

Identifying the exact portion. Consider the set of "solved" Leaf (SL) nodes in G' which correspond to full solutions.

When $C = OPEN$ we get

$$L(s_0) = \sum_{t \in SL} g(t) + \sum_{s \in OPEN \setminus SL} g(s) \cdot lo(s) \tag{3}$$

$$U(s_0) = \sum_{t \in SL} g(t) + \sum_{s \in OPEN \setminus SL} g(s) \cdot up(s)$$

The absolute gap between the upper and lower bound is

$$AbsGap(G') = U(s_0) - L(s_0) = \sum_{s \in OPEN \setminus SL} g(s) \cdot (up(s) - lo(s)) = \tag{4}$$

$$AbsGap(G') = U(s_0) - L(s_0) = \sum_{s \in OPEN} g(s) \cdot (up(s) - lo(s)) \tag{5}$$

The ratio between the upper and lower bound is in general

$$RatioGap(G') = \frac{L(s_0)}{U(s_0)} = \frac{\sum_{s \in SL} g(s) + \sum_{s \in OPEN \setminus SL} g(s) \cdot lo(s)}{(\sum_{s \in SL} g(s) + \sum_{s \in OPEN \setminus SL} g(s) \cdot up(s)} \tag{6}$$

We will use "gap" for "AbsGap". For AND/OR search graph the explicit graph G' is an AND/OR graph. $U(s_0)$ and $L(s_0)$ are computed by propagating those bounds from leaves to root. For OR nodes the computation is as in equation 2. If n is an AND nodes then

DEFINITION **3** (Bound propagation for AND nodes)**.** *Given the currently explored graph G', for any node $n \in G'$ its lower and upper bounds can be computed recursively:*

- *If n is a non-solved leaf node then $U(n) = L(n) = 0$ (inconsistent problem)*
- *If n is an internal node then*

$$U(n) = \prod_{s \in ch(n)} U(s) \tag{7}$$

and

$$L(n) = \prod_{s \in ch(n)} L(s)$$

2 Bounded Weighted Counting by Search

Since we are interested in computing upper and lower bounds having a bounded gap that should diminish as more of the search space is explored, we can convert the problem into an optimization one.

DEFINITION **4.** *Given a weighted graph G, and given ϵ, find the smallest subgraph G' of G for which $\frac{L(s_0)}{U(s_0)} \geq 1 - \epsilon$. Or, alternatively, $U(s_0) - L(s_0) \leq \epsilon$.*

We can view the problem as finding the shortest path in a meta search space graph as follows.

DEFINITION **5** (Approximated counting as shortest path)**.** *Given a weighted AND/OR graph G, its root s_0 and a threshold ϵ we define a (meta) search graph $\tilde{G}$ as follows*

- *The initial state G_0 is the graph containing only s_0, which is the root node in G.*
- *The states of $\tilde{G}$ are explicit partial AND/OR subgraphs G' that include the root s_0.*
- *if G' is a state in $\tilde{G}$, each leaf $n \in G'$,either an AND node or an OR node, yields a child graph G'_n in which G' is augmented with $ch(n)$, having the corresponding arc weights.*
- *Each G' is associated with a lower bound $L_{G'}(s_0)$ and an upper bound $U_{G'}(s_0)$ which are the backup values propagated from the leaves.*
- *A state G' satisfies the goal conditions if its gap (absolute or ratio) is bounded below ϵ.*
- *The task is to find the shortest solution path in $\tilde{G}$. Alternatively, we seek the smallest G' satisfying the goal conditions.*

Since we are interested in an anytime algorithms we can justify the following greedy scheme: given an explicit graph G', extend it in the direction that minimize the gap function the most. We define the notion of gain as the reduction in the gap due to expanding a node n in G'.

DEFINITION **6** (Gain)**.** *The gain obtained moving from G' to G'_n is defined by:*

$$gain_{G'}(n) = gap(G') - gap(G'_n) \tag{8}$$

It can also be defined by

$$gain_{G'}(n) = gap_{G'}(s_0) - gap_{G'_n}(s_0)$$

PROPOSITION **1.** *If G' is an OR graph we have*

$$gain_{G'}(n) = g(n) \cdot \{up(n) - lo(n) - \sum_{m \in ch(n)} w(n,m)(up(m) - lo(m))\} \tag{9}$$

Thus the gain can be computed locally and efficiently for OR graphs. If we have AND/OR graphs however the computation may not be as local. In the worst-case, we will have to propagate the impact of expanding a node n to the root and compute its corresponding gap and the corresponding gain. But if we keep the updated backup value at each node in G', the gain computation will impact only nodes along the path from n to the root.

PROPOSITION **2.** *Given an AND/OR graph G' and an expansion of a node n, computing the $gain_{G'}(n)$ is linear in the depth of n (the depth of the associated variable) in the pseudo-tree.*

When the heuristic function yielding $lo(n)$ and $up(n)$ is accurate the gain will be zero for many nodes. In that case we need to break ties in favor of another measure. We choose $gain2$

$$gain2_{G'}(n) = up(n) - lo(n)$$

We can also define the relative gain.

DEFINITION **7** (Relative Gain)**.** *Given the relative cost of G' defined by*

$$RatioGap(G') = \frac{L(G')}{U(G')}$$

Once we expand node n in G' we get

$$RatioGap(G'_n) = \frac{L(G'_n)}{U(G'_n)} = \frac{L(G') + LO(n)}{U(G') + UP(n)}$$

where

$$LO(n) = (\sum_{m \in child(n)} w(n,m) \cdot lo(m)) - g(n) \cdot lo(n)$$

and

$$UP(n) = (\sum_{m \in child(n)} w(n,m) \cdot up(m)) - g(n) \cdot up(n)$$

Then

$$RationGain(n) = \frac{L(G')}{U(G')} - \frac{L(G') + \sum_{m \in child(n)} w(n,m) \cdot lo(m) - g(n)lo(n)}{U(G') + \sum_{m \in child(n)} w(n,m) \cdot up(m) - g(n)up(n)}$$

3 Algorithms

3.1 Greedy best first

Since the path in our meta search space will be shorter if the gains associated with its nodes are largest it seems that a greedy algorithm that expand a node having a largest gain should be appropriate. Algorithm "greedy-Best-First-Search" or GBFS is defined in Algorithm 1. It assumes absolute gap.

If we order the nodes in a breadth-first manner GBFS is similar to the next algorithm, called BFS (Algorithm 2, which addresses explicitly the notion of G being a graph rather than a tree(steps 7-11) and assumes no initial lower and upper-bounds.

Algorithm 1 Greedy BFS

Input: a weighted state space graph G over set of variables $X = \{X_1, ..., X_n\}$, defined implicitly. A root s_0 of G. $w(s, s')$ is the cost of the arc (s, s'), ϵ.

initialize: $G' \leftarrow \{s_0\}$. initialize $up(s_0)$ and $lo(s_o)$. Evaluate the gain in s_0.
initialize: $cost(G') \leftarrow up(s_0) - lo(s_0)$. Q holds the leaf nodes of G' ordered by their *gain* (EQ. 8).
for until G' satisfies $U(G') - L(G') \leq \epsilon$ **do**
    $n \leftarrow$ dequeue(Q).
    Expand n generating its $child(n)$.
    for each $m \in child(n)$ **do**
        compute $gain(m)$ and gain2(m) and insert $(m, gain(m))$ into Q.
        If gain is zero use $(m, gain2(m))$.
    end for
    $newcost \leftarrow cost(G') - gain(n)$
    $G' \leftarrow G' \cup child(n)$.
    $Cost(G') \leftarrow newcost$
end for
    return $G', cost(G')$.

3.2 A depth first summation algorithm with bounded error

Theorem 1. *Let T be an OR tree with arcs with nodes N and non-negative arc weights $w(n, m)$ for parent node n and child node m, such that for all nodes n, $\sum_{m \in children(n)} w(n, m) \leq 1$. We recursively define the partition function $Z(n)$ of a node n to be equal to 1 if n is a leaf node and $\sum_{m \in children(n)} w(n, m)Z(m)$ otherwise. Then the function defined above $Z_\epsilon(n, \epsilon)$ computes upper and lower bounds $lo(n)$ and $hi(n)$ such that*

$$Z(n)(1 - \epsilon) \leq lo(n) \leq Z(n) \leq hi(n) \leq \frac{Z(n)}{1 - \epsilon} \tag{10}$$

Lemma 1. *If $a \leq b \leq c$ and $\frac{a}{c} \geq (1 - \epsilon)$ then $b(1 - \epsilon) \leq a \leq b \leq c \leq \frac{a}{1-\epsilon}$*

Proof of theorem 1. The proof is by induction on the tree. If n is a leaf node then $lo(n) = hi(n) = Z(n) = 1$, which satisfies the claim. For the inductive case, n is a node whose children m satisfy the theorem. The algorithm maintains the invariant that $lo(n)$ and $hi(n)$ are upper and lower bounds on $Z(n)$ at all times. To see this note that at the initialization of $lo(n)$ and $hi(n)$ this is trivially true. For each child of n, if we change $lo(n)$ then we are are adding a term that, by the inductive assumption, is a lower bound on the corresponding term in $Z(n)$. Similarly, for $hi(n)$, if in the loop we replace a term $w(n, m)$ with $w(n, m) * hi(m)$ then we are replacing an upper bound of the corresponding term in $Z(n)$ with another (potentially tighter) upper bound.

There are two cases to consider: 1) if the loop is broken because $lo(n)/hi(n) \geq 1 - \epsilon$ then by the invariant just described above and the lemma, we are done.

Algorithm 2 Breadth-First Search (BFS) for probabilistic weighted counting

Input: a weighted state space graph $\mathcal{S} = (\mathbf{X}, \mathbf{D}, \mathbf{G})$ over set of variables $X = \{X_1, ..., X_n\}$, defined implicitly. A root s_0 of $\mathcal{S}$. $c(s, s')$ is the cost of the arc (s, s').

Output: for each node, the weight of all partial paths merge into it from the root.

initialize $OPEN \leftarrow s_0$. $q(s_0) = 1$. q(s) is the weight count for s
while OPEN is not empty **do**
    $< s, q > \leftarrow first\ in\ OPEN$ where ordering is breadth-first. Remove it from OPEN and put it in CLOSED.
    expand s, generating all its child nodes with pointers back to s.
    for each child s' of s **do**
        $r(s') \leftarrow q(s) \cdot c(s, s')$.
        if s' appears neither in OPEN nor in CLOSED, add it to OPEN. Attach a pointer from s' to s. **then**
            Assign the newly computed $q(s') = r(s')$ to s'.
        else
            update $q(s') \leftarrow q(s') + r(s')$.
            if s' is in CLOSED move it back to OPEN.
        end if
    end for
end whilereturn the sum of q's of all leaf nodes.

2) On the other hand, if the loop finishes, then we need to show that it must be the case that $lo(n)/hi(n) \geq 1 - \epsilon$. If the loop finishes then we have that $lo(n) = \sum_{m \in \text{children}(n)}(w(n, m)lo(m))$ and $hi(n) = \sum_{m \in \text{children}(n)}(w(n, m)hi(m))$. But by the inductive hypothesis, each of the children m satisfies $\frac{lo(m)}{hi(m)} \geq 1 - \epsilon$. Substituting these inequalities into the parent lower bound expression:

$$lo(n) = \sum_{m \in \text{children}(n)} w(n, m)lo(m) \tag{11}$$

$$\geq \sum_{m \in \text{children}(n)} w(n, m)(1 - \epsilon)hi(m) \tag{12}$$

$$= (1 - \epsilon) \sum_{m \in \text{children}(n)} w(n, m)hi(m) \tag{13}$$

$$= (1 - \epsilon)hi(n). \tag{14}$$

It follows that $\frac{lo(n)}{hi(n)} \geq 1 - \epsilon$. Thus, again by the lemma and invariant, the theorem holds.

□

Algorithm 3 Z_ϵ

Input: State-space node n
$\epsilon > 0$
Output: lo and hi, lower and upper bounds on the partition function of n

```
function Z_ε(n, ε)
    if n is a leaf node then
        return (1,1)
    else
        lo(n) ← 0
        hi(n) ← Σ_{m∈children(n)}(w(n,m))
    end if
    for all m ∈ children(n) do
        if lo(n)/hi(n) ≥ 1 − ε then
            break
        else
            lo(m), hi(m) ← Z_ε(m, ε)
            lo(n) ← lo(n) + w(n,m)lo(m)
            hi(n) ← hi(n) − w(n,m) + w(n,m)h(m)
        end if
    end for
    return (lo, hi)
end function
```

3.3 For AND/OR graphs

In the case of AND/OR graphs we need to multiply the lower and upper bounds at each AND node. But this will cause the bound guarantees to drop off exponentially with the number of OR children of an AND node. So to preserve the bound we can iterate sequentially through the OR children and update the ϵ as we go.

Let ϵ be the input to the algorithm, and let $\epsilon_1, \ldots, \epsilon_K$ be the K OR children $n_1, \ldots, n_K$. Then we can adjust the ϵ_k as follows:

$$\epsilon_1 \leftarrow \epsilon \tag{15}$$

$$\epsilon_i \leftarrow 1 - (1 - \epsilon_{i-1})\frac{hi(n_{i-1})}{lo(n_{i-1})} \text{ for } i > 1 \tag{16}$$

$$\text{where } (lo(n_{i-1}), hi(n_{i-1})) \leftarrow Z_\epsilon(n_{i-1}, \epsilon_{i-1}) \tag{17}$$

Proof. We need to show that if we do this then

$$\prod_i \frac{lo(n_i)}{hi(n_i)} \geq (1 - \epsilon) \tag{18}$$

This equality holds for just the first term in the above product. Suppose

this inequality holds after multiplying together the first $k-1$ terms. Then the inequality after multiplying the kth term is:

$$\left(\prod_{i=1}^{k-1}\frac{lo(n_i)}{hi(n_i)}\right)\frac{lo(n_k)}{hi(n_k)} \geq (1-\epsilon)(1-\epsilon_k) \text{ by the inductive hypothesis} \tag{19}$$

$$= (1-\epsilon)\left(1-\left(1-(1-\epsilon_{k-1})\frac{hi(n_{k-1})}{lo(n_{k-1})}\right)\right) \tag{20}$$

$$= (1-\epsilon)(1-\epsilon_{k-1})\frac{hi(n_{k-1})}{lo(n_{k-1})} \tag{21}$$

$$\geq (1-\epsilon). \tag{22}$$

(22) follows because by construction $(lo(n_{k-1}), hi(n_{k-1})) \leftarrow Z_\epsilon(n_{k-1}, \epsilon_{k-1})$. Thus by the inductive hypothesis, $\frac{lo(n_{k-1})}{hi(n_{k-1})} \geq 1-\epsilon_{k-1}$, which implies that $\frac{hi(n_{k-1})}{lo(n_{k-1})} \geq \frac{1}{1-\epsilon_{k-1}}$. □

4 A RBFS style approximation algorithm

See algorithm 5. An approximate partition function algorithm in the style of recursive best first search. Suppose we want to compute the probability mass in the subtree rooted at node n. We maintain for each child m of n an upper bound $hi(m)$ and lower bound $lo(m)$. Our goal is to minimize

$$hi(n) - lo(n) = \sum_{m \in children(n)} w(n,m)(hi(m) - lo(m)). \tag{23}$$

Our algorithm explores each child m of n so long as m is the child which contributes most to this sum. We implement this as a recursive function call; the call to the function on a child is passed the bound Δ which corresponds to the value of the next largest term in the sum.

Claim: at any point $(lo, hi) \leftarrow$ RecZ-AO(n) are lower and upper bounds on $Z(n)$.

Proof. By construction, lo and hi are initially lower and upper bounds. RecZ-AO computes $Z(n)$ exactly for leaf nodes. For non-leaf OR nodes, RecZ-AO, if each $hi(m)$ is an upper bound then

$$hi(n) = \sum_{m \in children(n)} w(n,m)hi(m) \tag{24}$$

$$\geq \sum_{m \in children(n)} w(n,m)Z(m) \tag{25}$$

$$= Z(n). \tag{26}$$

Very similarly, we can show $hi(n)$ and $lo(n)$ are upper and lower bounds for AND and OR nodes if the same holds for their children. □

Algorithm 4 RecZ

function $RecZ(n)$ ▷ top-level procedure
  Initialize $lo(n)$ and $hi(n)$
  repeat
    $\Delta \leftarrow hi(n) - lo(n)$
    $(lo(n), hi(n)) \leftarrow RecZ(n, \Delta)$
    yield $(lo(n), hi(n))$ ▷ Yield bounds to return stream for anytime retrieval
  until $hi(n) = lo(n)$
end function

function $RecZ(n, \Delta)$
  ▷ n is a node. We explore this subtree until $hi(m) - lo(m)$ is less than Δ.
  if n is a leaf node **then**
    return (1,1)
  end if

  for all $m \in$ children(n) **do** ▷ Initialize F value for each child node
    $lo(m) \leftarrow 0$ ▷ Or any precomputed lower bound
    $hi(m) \leftarrow 1$ ▷ Or any precomputed upper bound
    $F(m) \leftarrow w(n, m)(hi(m) - lo(m))$
  end for

  $Q \leftarrow$ pqueue$(\{(m, F(m)) | m \in children(n)\})$ ▷ Create priority queue Q
  while $hi(n) - lo(n) > \Delta$ **do**
    $(m, F(m)) \leftarrow pop(Q)$ ▷ Pop child with largest F(m) from priority queue
    $NextBest \leftarrow peek(Q)$ ▷ Get the F value of the next largest child in the queue
    $\Delta_m \leftarrow NextBest / w(n, m)$
    $(lo(m), hi(m)) \leftarrow RecZ(m, \Delta_m)$ ▷ Recursive call
    $hi(n) \leftarrow \sum_{m \in children(n)} w(n, m) hi(m)$ ▷ Recompute node bounds
    $lo(n) \leftarrow \sum_{m \in children(n)} w(n, m) lo(m)$
    $F'(m) \leftarrow w(n, m)(hi(m) - lo(m))$ ▷ Insert child node back into queue with new F value
    $Q \leftarrow insert((m, F'(m)), Q)$
  end while
  return (lo, hi)
end function

Algorithm 5 RecZ-AO: Recursive partition function algorithm for AND/OR trees

function RecZ-AO(n) ▷ top-level procedure
  Initialize $lo(n)$ and $hi(n)$
  repeat
    $\Delta \leftarrow hi(n) - lo(n)$
    $(lo(n), hi(n)) \leftarrow$ RecZ-AO(n, Δ)
    yield $(lo(n), hi(n))$ ▷ Yield bounds to return stream for anytime retrieval
  until $hi(n) = lo(n)$
end function

function RecZ-AO(n, Δ)
    ▷ We explore this subtree until $hi(n) - lo(n)$ is less than Δ.
  if n is a consistent leaf node **then**
    return (1,1)
  else if n is an inconsistent leaf node **then**
    return (0,0)
  end if

  for all $m \in$ children(n) **do** ▷ Initialize F value for each child node
    $lo(m) \leftarrow 0$ ▷ Or any precomputed lower bound
    $hi(m) \leftarrow 1$ ▷ Or any precomputed upper bound
    $F(m) \leftarrow w(n,m)(hi(m) - lo(m))$
  end for

  $Q \leftarrow$ pqueue($\{(m, F(m)) | m \in children(n)\}$) ▷ Create priority queue Q
  while $hi(n) - lo(n) > \Delta$ **do**
    $(m, F(m)) \leftarrow pop(Q)$ ▷ Pop child with largest F(m) from priority queue
    $NextBest \leftarrow peek(Q)$ ▷ Get the F value of the next largest child in the queue
    $\Delta_m \leftarrow NextBest / w(n,m)$
    $(lo(m), hi(m)) \leftarrow$ RecZ-AO(m, Δ_m) ▷ Recursive call
    if n is an OR node **then** ▷ Recompute node bounds
      $hi(n) \leftarrow \sum_{m \in children(n)} w(n,m) hi(m)$
      $lo(n) \leftarrow \sum_{m \in children(n)} w(n,m) lo(m)$
    else if n is an AND node **then**
      $hi(n) \leftarrow \prod_{m \in children(n)} w(n,m) hi(m)$
      $lo(n) \leftarrow \prod_{m \in children(n)} w(n,m) lo(m)$
    end if
    $F'(m) \leftarrow w(n,m)(hi(m) - lo(m))$ ▷ Insert child node back into queue with new F value
    $Q \leftarrow insert((m, F'(m)), Q)$
  end while
  return (lo, hi)
end function

Claim: RecZ-AO eventually terminates with an $lo = hi = Z(n)$.

Proof. TBD

□