Mini-Buckets: A General Scheme for Generating Approximations in Automated Reasoning

Rina Dechter*
Department of Information and Computer Science
University of California, Irvine
dechter@ics.uci.edu

Abstract

The class of algorithms for approximating reasoning tasks presented in this paper is based on approximating the general bucket elimination framework. The algorithms have adjustable levels of accuracy and efficiency, and they can be applied uniformly across many areas and problem tasks. We introduce these algorithms in the context of combinatorial optimization and probabilistic inference.

1 Overview

Bucket elimination is a unifying algorithmic framework that generalizes dynamic programming to enable many complex problem-solving and reasoning activities. Among the algorithms that can be accommodated within this framework are directional resolution for propositional satisfiability [Dechter and Rish, 1994], adaptive consistency for constraint satisfaction [Dechter and Pearl, 1987], Fourier and Gaussian elimination for linear inequalities [Lassez and Mahler, 1992], and dynamic programming for combinatorial optimization [Bertele and Brioschi, 1972]. Many algorithms for probabilistic inference, such as belief updating, finding the most probable explanation, finding the maximum a posteriori hypothesis, and computing the maximum expected utility, also can be expressed as bucket-elimination algorithms [Dechter, 1996].

The main virtues of this framework are *simplicity* and *generality*. By simplicity, we mean that extensive terminology is unnecessary for complete specification of bucket-elimination algorithms, thus making the algorithms accessible to researchers working in diverse areas. More important, their uniformity facilitates the transfer of ideas, applications, and solutions between disciplines. Indeed, all bucket-elimination algorithms are similar enough to make any improvement to a single algorithm applicable to all others in its class.

Normally, an input to a bucket-elimination algorithm is a knowledge base specified by a collection of functions or relations, over subsets of variables (e.g., clauses for propositional satisfiability, constraints and cost functions for constraint optimization, conditional probability matrices for belief networks). Bucket-elimination algorithms process variables one by one in a given order. In its first step, given a variable ordering, the algorithm partitions these functions into buckets, where the bucket of a particular variable contains the functions defined on that variable, provided the function is not defined on variables higher in that ordering. The next step is to process the buckets from top to bottom. When the bucket of variable X is processed, an "elimination procedure" is performed over the functions in its bucket, yielding a new function defined over all the variables mentioned in the bucket, excluding X. This function summarizes the "effect" of X on the remainder of the problem. The new function is placed in a lower bucket. For illustration we include *directional resolution*, a bucket-elimination algorithm similar to the original Davis-Putnam procedure for satisfiability (Figure 1) and *elim-bel*, a bucket-elimination algorithm for belief updating in probabilistic networks (Figure 2).

An important property of variable-elimination algorithms is that their performance can be predicted using a graph parameter, w^*, called the *induced width* [Dechter and Pearl, 1987], also known as *tree-width* [Arnborg, 1985], which is the largest cluster in an optimal tree-embedding of a graph. In general, a given theory and its query can be associated with an *interaction graph*. The complexity of a bucket-elimination algorithm is *time and space* exponential in the induced width of the problem's interaction graph. The size of the induced width varies with various variable orderings, so each ordering has a different performance guarantee. Although finding the optimal ordering is hard [Arnborg, 1985], heuristics and approximation algorithms are available [Robertson and

*This work was partially supported by NSF grant IRI-9157636 and by the Air Force Office of Scientific Research, AFOSR 900136, Rockwell International, and Amada of America.

Algorithm directional resolution
Input: A cnf theory φ over $Q_1, ..., Q_n$; an ordering d.
Output: A decision of whether φ is satisfiable. If it is, a theory $E_d(\varphi)$, equivalent to φ; else, an empty directional extension.
1. **Initialize:** Generate an ordered partition of the clauses, $bucket_1, ..., bucket_n$, where $bucket_i$ contains all the clauses whose highest literal is Q_i.
2. **Backward:** For $p = n$ to 1, do:
• **If** $bucket_p$ contains a unit clause, perform unit resolution. Put each resolvent in its bucket.
• **Else,** resolve each pair $\{(\alpha \vee Q_p), (\beta \vee \neg Q_p)\} \subseteq bucket_p$. If $\gamma = \alpha \vee \beta$ is empty, return $E_d(\varphi) = \emptyset$, the theory is not satisfiable; else, determine the index of γ and add γ to the appropriate bucket.
3. **Return:** $E_d(\varphi) \Longleftarrow \bigcup_i bucket_i$.

Figure 1: Algorithm *directional resolution*

Algorithm elim-bel
Input: A belief network $BN = (G, P)$, $P = \{P_1, ..., P_n\}$; an ordering of the variables, $o = X_1, ..., X_n$.
Output: The belief of X_1 given evidence e.
1. **Initialize:** Generate an ordered partition of the conditional probability matrices into buckets. $bucket_i$ contains all matrices whose highest variable is X_i. Put each observation in its bucket. Let $S_1, ..., S_j$ be the subset of variables in the processed bucket on which matrices (new or old) are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do:
For $\lambda_1, \lambda_2, ..., \lambda_j$ in $bucket_p$, do:
• (bucket with observed variable) **If** $X_p = x_p$ appears in the bucket, then substitute $X_p = x_p$ in each matrix λ_i and put the results in the appropriate bucket.
• **Else,** $U_p \leftarrow \bigcup_{i=1}^{j} S_i - \{X_p\}$. For all $U_p = u$, $\lambda_p(u) = \sum_{x_p} \Pi_{i=1}^{j} \lambda_i(x_p, u_{S_i})$. Add λ_p to the largest index variable in U_p.
3. **Return**: $Bel(x_1) = \alpha P(x_1) \cdot \Pi_i \lambda_i(x_1)$.
(Where the λ_i are in $bucket_1$, α is a normalizing constant.)

Figure 2: Algorithm *elim-bel*

Seymour, 1995].

When a problem has a large induced width, bucket elimination is unsuitable because of its extensive memory demand. In such cases, approximation algorithms should be attempted. In this paper, we present a collection of parameterized algorithms that have varying degrees of accuracy and efficiency. We demonstrate the general scheme for combinatorial optimization and for the probabilistic task of belief assessment.

2 Preliminaries

Definition 2.1 (graph concepts) *A* directed graph *is a pair, $G = \{V, E\}$, where $V = \{X_1, ..., X_n\}$ is a set of elements and $E = \{(X_i, X_j) | X_i, X_j \in V\}$ is the set of edges. For each variable X_i, $pa(X_i)$ or pa_i, is the set of variables pointing to X_i in G. A directed graph is acyclic if it has no directed cycles. In an undirected graph, the directions of the arcs are ignored: (X_i, X_j) and (X_j, X_i) are identical. An* ordered graph *is a pair (G, d) where G is an undirected graph and $d = X_1, ..., X_n$ is an ordering of the nodes. The* width of a node *in an ordered graph is the number of the node's neighbors that precede it in the ordering. The* width of an ordering d, *denoted $w(d)$, is the maximum width over all nodes. The* induced width of an ordered graph, $w^*(d)$, *is the width of the induced ordered graph obtained by processing the nodes from last to first; when node X is processed, all its earlier neighbors in the ordering are connected. The* induced width of a graph, $w*$, *is the minimal induced width over all its orderings; it is also known as the* tree-width *[Arnborg, 1985]. The* moral graph *of a directed graph G is the undirected graph obtained by connecting the parents of all the nodes in G and then removing the arrows.*

Definition 2.2 (belief network, cost network) *Let $X = \{X_1, ..., X_n\}$ be a set of random variables over multivalued domains $D_1, ..., D_n$. A* belief network *is a pair (G, P), where G is a directed acyclic graph and $P = \{P_i\}$. $P_i = \{P(X_i | pa(X_i))\}$ are the conditional probability matrices associated with X_i. An assignment $(X_1 = x_1, ..., X_n = x_n)$ can be abbreviated to $x = (x_1, ..., x_n)$. The belief network represents a probability distribution over X having the product form $P(x_1, ..., x_n) = \Pi_{i=1}^{n} P(x_i | x_{pa(X_i)})$, where x_S denotes the projection of a tuple x over a subset of variables S. A* cost network *is a triplet (X, D, C), where X is a set of discrete variables, $X = \{X_1, ..., X_n\}$, over domains $D = \{D_1, ..., D_n\}$, and C is a set of real-valued cost functions $C_1, ..., C_l$. Each function C_i is defined over a subset of variables, $S_i = \{X_{i_1}, ..., X_{i_r}\}$, $C_i : \bowtie_{j=1}^{r} D_{ij} \rightarrow R^+$. The* cost graph *of a* cost network *has a node for each variable and connects nodes denoting variables appearing in the same cost function.*

$A = a$ denotes a partial assignment to a subset of variables, A. An evidence set e is an instantiated subset of variables. We use (u_S, x_p) to denote the tuple u_S appended by a value x_p of X_p. We define $\bar{x}_i = (x_1, ..., x_i)$ and $\bar{x}_i^j = (x_i, x_{i+1}, ..., x_j)$.

Definition 2.3 (elimination function) *Given a function h defined over subset of variables, S, where $X \in S$, the functions $(min_X h)$, $(max_X h)$, $(mean_X h)$, and $(\sum_X h)$ are defined over $U = S - \{X\}$ as follows. For every $U = u$, $(min_X h)(u) = \min_x h(u, x)$, $(max_X h)(u) = \max_x h(u, x)$, $(\sum_X h)(u) = \sum_x h(u, x)$, and $(mean_X h)(u) = \sum_x \frac{h(u,x)}{|X|}$, where $|X|$ is the cardinality of X's domain. Given a set of functions $h_1, ..., h_j$ defined over the subsets $S_1, ..., S_j$, the functions $(\Pi_j h_j)$ and $\sum_J h_j$ are defined over $U = \cup_j S_j$. For every $U = u$, $(\Pi_j h_j)(u) = \Pi_j h_j(u_{S_j})$ and $(\sum_j h_j)(u) = \sum_j h_j(u_{S_j})$.*

Algorithm elim-opt
Input: A cost network (X, D, C), $C = \{C_1, ..., C_l\}$; an ordering of the variables, o; assignments e.
Output: The minimal cost assignment.
1. **Initialize:** Partition the cost components into $bucket_1, \ldots, bucket_n$. Let $S_1, ..., S_j$ be the subset of variables in the processed bucket on which the functions are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do:
For $h_1, h_2, ..., h_j$ in $bucket_p$, do:
• **If** $bucket_p$ contains $X_p = x_p$, assign $X_p = x_p$ to each h_i and put each h_i in its appropriate bucket.
• **Else,** generate the functions h^p: $h^p = min_{X_p} \sum_{i=1}^{j} h_i$. $x_p^{opt} = argmin_{X_p} h^p$. Add h^p to the bucket of the largest-index variable in $U_p \leftarrow \bigcup_{i=1}^{j} S_i - \{X_p\}$.
3. **Forward:** Assign values in the ordering o using the recorded functions x^{opt} in each bucket.

Figure 3: Algorithm *elim-opt*

Definition 2.4 (optimization, belief assessment) *Given a cost network* (X, D, C), *the* discrete optimization problem *is to find an assignment* $x^o = (x_1^o, ..., x_n^o)$, $x_i^o \in D_i$, *such that* $x^o = argmin_{\bar{x}_n} \sum_{i=1}^{l} C_i$. *Given a belief network* (G, P), the belief assessment task *for* $X = x$ *is to determine* $bel(x) = P(x|e)$.

3 Approximating optimization

In [Bertele and Brioschi, 1972], the non serial dynamic programming algorithm for the discrete optimization problem is presented. This algorithm can be rephrased within the bucket-elimination scheme, as shown in Figure 3. Given a partitioning of the cost functions into their respective buckets, algorithm *elim-opt* processes buckets from top to bottom. When processing X_p, it generates a new function by taking the minimum relative to X_p, over the sum of the functions in that bucket. The time and space complexity of *elim-opt* is exponential in the induced width w^* of the cost graph. Thus, when its induced width is not small, we must resort to approximations.

Since the complexity of processing a bucket is tied to the arity of the functions being recorded, we propose to approximate these functions by a collection of smaller arity functions. Let $h_1, ..., h_j$ be the functions in the bucket of X_p, and let $S_1, ..., S_j$ be the subsets of variables on which those functions are defined. When *elim-opt* processes the bucket of X_p, it computes the function h^p: $h^p = min_{X_p} \sum_{i=1}^{j} h_i$. Instead, we can use a brute-force approximation method to generate another function g^p by migrating the minimization operator inside the summation. That is, $g^p = \sum_{i=1}^{j} min_{X_p} h_i$. Clearly, h^p and g^p are defined on the same arguments, and since each function h_i in the sum of h^p is replaced by $min_{X_p} h_i$ in the sum defining g^p, $g^p \leq h^p$. In g^p the minimizing elimination operator $min_{X_p} h_i$ is applied separately to each function, (so g^p will never increase dimensionality) and can be moved, separately, to a lower bucket. Thus, once *elim-opt* reaches the first variable, it has computed a lower bound on the minimal cost.

This idea can be generalized to yield a collection of parameterized approximation algorithms having varying degrees of accuracy and efficiency. Instead of the elimination operator (i.e., minimization) being applied to each singleton function in a bucket, it can be applied to a coerced partitioning of buckets into mini-buckets. Let $Q' = \{Q_1, ..., Q_r\}$ be a partitioning into mini-buckets of the functions $h_1, ..., h_j$ in X_p's bucket. The mini-bucket Q_l contains the functions $h_{l_1}, ..., h_{l_r}$. Algorithm *elim-opt* computes h^p: (l indexes the mini-buckets) $h^p = min_{X_p} \sum_{i=1}^{j} h_i = min_{X_p} \sum_{l=1}^{r} \sum_{l_i} h_{l_i}$. By migrating the minimization operator into each mini-bucket we get $g_{Q'}^p = \sum_{l=1}^{r} min_{X_p} \sum_{l_i} h_{l_i}$. As the partitionings are more coerced, complexity as well as accuracy increases.

Definition 3.1 *Partitioning* Q' *is a refinement of* Q'' *iff for every set* $A \in Q'$ *there exists a set* $B \in Q''$ *such that* $A \subseteq B$.

Proposition 3.2 *If in the bucket of* X_p, Q' *is a refinement partitioning of* Q'', *then* $h^p \geq g_{Q''}^p \geq g_{Q'}^p$. □

Algorithm *approx-opt* is described in Figure 4. It is parameterized by two indexes that control the partitionings.

Definition 3.3 *Let* H *be a collection of functions* $h_1, ..., h_j$ *on subsets* $S_1, ..., S_j$. *A partitioning of* H *is* canonical *if any function whose arguments are subsumed by another belongs to a mini-bucket with one of its subsuming functions. A partitioning* Q *is an* (i, m) − *partitioning iff the following three conditions are met: the partitioning is canonical, at most* m *non-subsumed functions participate in each mini-bucket, and the total number of variables mentioned in each mini-bucket does not exceed* i.

Theorem 3.4 *Algorithm approx-opt(i,m) computes a lower bound to the minimal cost in time* $O(m \cdot exp(2i))$ *and space* $O(m \cdot exp(i))$ *where* $i \leq n$ *and* $m \leq 2^i$. □

In general, as m and i increase we get more accurate approximations. The two parameters, although dependent, allow flexibility in bounding performance.

Example 3.5 *Consider the network in Figure 5. We use ordering* $(A, B, C, D, E, F, G, H, I)$ *to which we apply both elim-opt and approx-opt(*$m = 1$*) with unbounded* i.

Algorithm approx-opt(i,m)
Input: A cost network (X, D, C), $\{C_1, ..., C_l\}$; ordering o.
Output: A lower bound on the optimal cost.
1. **Initialize:** Partition the functions into $bucket_1, \ldots, bucket_n$. Let $S_1, ..., S_j$ be the subset of variables in $bucket_p$ on which the functions are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do:
• **If** $bucket_p$ contains $X_p = x_p$, assign $X_p = x_p$ to each h_i and put each h_i in the appropriate bucket.
• **Else,** for $h_1, h_2, ..., h_j$ in $bucket_p$, generate an (i, m)-mini-bucket-partitioning, $Q' = \{Q_1, ..., Q_r\}$. For each $Q_l \in Q'$ containing $h_{l_1}, ..., h_{l_t}$, do:
Generate function h^l, $h^l = min_{X_p} \sum_{i=1}^{t} h_{l_i}$. Add h^l to the largest-index bucket in $U_l \leftarrow \bigcup_{i=1}^{j} S_{l_i} - \{X_p\}$.
3. **Forward:** For $i = 1$ to n, do:
Given $\bar{x}_{p-1}$, choose the x_p that minimize the sum functions in X_p's bucket.

Figure 4: Algorithm *approx-opt(i,m)*

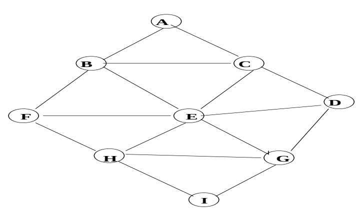

Figure 5: A cost network $C = C_1(I, H, G) + C_2(G, E, D) + C_3(H, F, E) + C_4(F, B) + C_5(E, B, C) + C_6(D, C) + C_7(B, A) + C_8(C, A) + C_9(A)$

(We omit the cost subscripts for simplicity.) The initial partitioning into buckets is
bucket(I) $= C(I, H, G)$, *bucket(H)* $= C(H, E, F)$,
bucket(G) $= C(G, E, D)$, *bucket(F)* $= C(F, B)$,
bucket(E) $= C(E, C, B)$, *bucket(D)* $= C(D, C)$,
bucket(C) $= C(C, A)$, *bucket(B)* $= C(B, A)$,
bucket(A) $= C(A)$.

Processing the buckets from top to bottom by elim-opt will generate what we denote as λ *functions:*
bucket(I) $= C(I, H, G)$
bucket(H) $= C(H, E, F), \lambda_I(H, G)$
bucket(G) $= C(G, E, D), \lambda_H(E, F, G)$
bucket(F) $= C(F, B), \lambda_G(E, F, D)$
bucket(E) $= C(E, C, B), \lambda_F(E, B, D)$
bucket(D) $= C(D, C), \lambda_E(C, B, D)$
bucket(C) $= C(C, A), \lambda_D(C, B)$
bucket(B) $= C(B, A), \lambda_C(A, B)$
bucket(A) $= C(A), \lambda_B(A)$
where $\lambda_I(H, G) = min_I C(I, H, G)$, $\lambda_H(E, F, G) = min_H(C(H, E, F) + \lambda_I(H, G))$, *and so on. The optimal cost* $min_A(C(A) + \lambda_B(A))$ *is computed in bucket(A).*

Processing by approx-opt($m = 1$*), and denoting functions that differ from those recorded by elim-opt by* γ*'s, we get*
bucket(I) $= C(I, H, G)$
bucket(H) $= C(H, E, F), \lambda_I(H, G)$
bucket(G) $= C(G, E, D), \gamma_H(G)$
bucket(F) $= C(F, B), \gamma_H(E, F)$
bucket(E) $= C(E, C, B), \gamma_F(E), \gamma_G(E, D)$
bucket(D) $= C(D, C), \gamma_E(D)$
bucket(C) $= C(C, A), \gamma_E(C, B), \gamma_D(C)$
bucket(B) $= C(B, A), \gamma_C(B), \gamma_F(B)$
bucket(A) $= C(A), \gamma_C(A), \gamma_B(A)$.

Bucket(H) contains two functions, each placed in a separate mini-bucket and each processed independently, yielding $\gamma_H(G) = min_H \lambda_I(H, G)$ *placed in bucket(G) and* $\gamma_H(E, F) = min_H C(H, E, F)$ *placed in bucket(F); then, in bucket(G) we generate* $\gamma_G(E, D) = min_G(C(G, E, D) + \gamma_H(G))$*; and so on. Finally, in bucket(A) we minimize over A the sum functions in that bucket, which provides a lower bound on the optimal cost. The first difference in elim-opt and approx-opt(m=1) occurs in bucket(H) and is visible in bucket(G), where elim-opt records a function on three variables,* $\lambda_H(E, F, G)$*, but approx-opt(m=1) records two functions, one on G ,*$\gamma_H(G)$*, and one on E and F,* $\gamma_H(E, F)$.

After *approx-opt(m=1)* processes all the buckets, we can generate a tuple in a greedy fashion as in *elim-opt*: we choose a value a of A that minimizes the sum in A's bucket, a value b of B minimizing the sum in bucket(B) given $A = a$, and so on. Although we have no guarantee on the quality of the tuple generated, we can bound its error a posteriori by evaluating its cost against the derived lower bound.

3.1 Mini-bucket heuristics and search

The mini-bucket approximations compute lower bounds of the exact quantities, and these lower bounds can be viewed as under-estimating heuristic functions in a minimization problem. We can associate with each partial assignment $\bar{x}_p$ an evaluation function $f(\bar{x}_p) = (g + h)(\bar{x}_p)$, where $g(\bar{x}_p) = \sum_{\{i | S_i \subseteq \{X_1, ..., X_{p-1}\}\}} C_i$ and $h(\bar{x}_p) = \sum_{j \in bucket_p} h_j$.

Proposition 3.6 *The evaluation function* f*, generated by* approx-opt*, provides a lower bound on the optimal cost.* □

We can conduct a best-first search using this heuristic evaluation function. When expanding the search tree, the best-first algorithm has a collection of tip nodes denoting a partial value assignment and their estimated evaluation function. At each step, the algorithm will expand a node with a minimal value of the evaluation function, f. From the theory of best-first search, we know the following: when the algorithm terminates with a complete assignment, it has found an optimal solution; as the heuristic function becomes more accurate, fewer nodes are expanded; and if we use the full bucket-elimination algorithm, best-first search will reduce to a greedy-like algorithm for this optimization task [Pearl, 1984]. Thus, mini-bucket approximations can be used to generate mechanically heuristics for best-first or branch-and-bound search.

4 Approximation of belief updating

The algorithm for belief assessment, *elim-bel* (for details see [Dechter, 1996]), is identical to *elim-opt* with two changes: minimization is replaced by summation, and summation by multiplication. Let $X_1 = x_1$ be an atomic proposition. Given a belief network (G, P) and evidence e, the problem is to compute $P(x_1|e) = P(x_1, e)/P(e)$. We can thus compute $P(x_1, e) = \sum_{x=\bar{x}_2^n} \Pi_{i=1}^n P(x_i, e|x_{pa_i})$ and normalize at the end. Consider an ordering of the variables $(X_1, ..., X_n)$ and a corresponding partition of the probability matrices into buckets. The procedure has only a backward phase (Figure 2). When processing a bucket, we multiply all the bucket's matrices, $\lambda_1, ..., \lambda_j$, defined over subsets $S_1, ..., S_j$, and then eliminate the bucket's variable by summation. The computed function is $\lambda^p : U_p \rightarrow R$, $\lambda^p = \sum_{X_p} \Pi_{i=1}^j \lambda_i$, where $U_p = \cup_i S_i - X_p$. Once all the buckets are processed, the answer is available in the first bucket of X_1. Like all bucket-elimination algorithms this algorithm's complexity (time and space) is $O(exp(w^*(d))$, where $w^*(d)$ is the induced width along d of the moral graph.

We will now apply the mini-bucket approximation to belief assessment. Let $Q\prime = \{Q_1, ..., Q_r\}$ be a partitioning into mini-buckets of the functions $\lambda_1, ..., \lambda_j$ in X_p's bucket. Algorithm *elim-bel* computes λ^p: (l indexes the mini-buckets) $\lambda^p = \sum_{X_p} \Pi_{i=1}^j \lambda_i = \sum_{X_p} \Pi_{l=1}^r \Pi_{l_i} \lambda_{l_i}$. If we, following the analog to *approx-opt* precisely, migrate the summation operator into each mini-bucket, we get $f_{Q\prime}^p = \Pi_{l=1}^r \sum_{X_p} \Pi_{l_i} \lambda_{l_i}$. This, however, amounts to computing an unnecessarily bad upper bound of λ^p, since the products $\Pi_{l_i} \lambda_{l_i}$ for $i > 1$ are replaced by their sums $\sum_{X_p} \Pi_{l_i} \lambda_{l_i}$. Instead, we can bound the product by its maximizing function. Separating the processing of the first mini-bucket from the rest, we get $\lambda^p = \sum_{X_p}((\Pi_{l_1} \lambda_{l_1}) \cdot (\Pi_{l=2}^r \Pi_{l_i} \lambda_{l_i}))$, which yields $g_{Q\prime}^p = \sum_{X_p}((\Pi_{l_1} \lambda_{l_1}) \cdot \Pi_{l=2}^r max_{X_p} \Pi_{l_i} \lambda_{l_i})$. Clearly,

Proposition 4.1 *For every partitioning Q, $\lambda^p \le g_Q^p \le f_Q^p$, and if in X_p's bucket, Q' is a refinement partitioning of Q'', then $\lambda^p \le g_{Q''}^p \le g_{Q'}^p$.* □

Algorithm approx-bel-max(i,m)
Input: A belief network $BN = (G, P)$, $P = \{P_1, ..., P_n\}$; an ordering of the variables, o; evidence e.
Output: An upper bound on $P(x_1, e)$.
1. **Initialize:** Partition the functions into buckets. Let $S_1, ..., S_j$ be the subset of variables in $bucket_p$ on which matrices are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do:
• **If** $bucket_p$ contains $X_p = x_p$, assign $X_p = x_p$ to each λ_i and put each λ_i in the appropriate bucket.
• **Else,** for $\lambda_1, \lambda_2, ..., \lambda_j$ in $bucket_p$, do:
Generate $Q' = \{Q_1, ..., Q_r\}$, an (i, m)-partitioning.
• (First bucket) For $Q_1 = \{\lambda_{1_1}, ..., \lambda_{1_j}\}$ first in Q', do:
Generate function $\lambda^1 = \sum_{X_p} \Pi_{i=1}^j \lambda_{1_i}$. Add λ^1 to the bucket of the largest-index variable in $U_1 \leftarrow \bigcup_{i=1}^j S_{1_i} - \{X_p\}$.
• For each $Q_l, l > 1$, in $Q' = \{\lambda_{l_1}, ..., \lambda_{l_j}\}$, do:
$U_l \leftarrow \bigcup_{i=1}^j S_{l_i} - \{X_p\}$. Generate the functions $\lambda^l = \max_{X_p} \Pi_{i=1}^j \lambda_{l_i}$. Add λ^l to the bucket of the largest-index variable in U_l.
3. **Output:** The product in bucket of X_1.

Figure 6: Algorithm *approx-bel-max(i,m)*

In summary, an upper bound g^p of λ^p can be obtained by processing one of X_p's mini-buckets by summation, while the rest of the mini-buckets are processed by maximization. In addition to approximating by an upper bound, we can approximate by a lower bound by applying the *min* operator to each mini-bucket or we can approximate by computing a mean-value approximation using the *mean* operator in each mini-bucket. Algorithm *approx-bel-max*, which uses the maximizing elimination operator, is described in Figure 6. Algorithms *approx-bel-min* and *approx-bel-mean* can be obtained by replacing the operator *max* with *min* and *mean*, respectively.

Theorem 4.2 *Algorithm approx-bel-max(i,m) computes an upper bound to the* belief $P(x_1, e)$ *in time* $O(m \cdot exp(2i))$ *and space* $O(m \cdot exp(i))$. □

Similarly, *aprox-bel-min* computes a lower bound on $P(x_1, e)$ and *approx-elim-mean* computes a mean-value of $P(x_1, e)$.

Remark: Algorithm *approx-bel-max* computes an upper bound on $P(x_1, e)$ but not on $P(x_1|e)$. If an exact value of $P(e)$ is not available, deriving a bound on $P(x_1|e)$ from a bound on $P(x_1, e)$ may not be easy. We can use *approx-bel-min* to derive a lower bound on $P(e)$ and use the resulting lower bound to derive an upper bound on $P(x_1|e)$.

5 Related work

The bucket-elimination framework provides a convenient and succinct language in which to express elimination algorithms across many areas. Most of these algorithms are widely known. In addition to dynamic programming [Bertele and Brioschi, 1972], constraint satisfaction [Dechter and Pearl, 1987], and Fourier elimination [Lassez and Mahler, 1992], there are variations on these ideas and algorithms for probabilistic inference in [Canning *et al.*, 1978; Tatman and Shachter, 1990; Shenoy, 1992; Zhang and Poole, 1996].

Mini-bucket approximation algorithms parallel consistency-enforcing algorithms for constraint processing, in particular those enforcing directional consistency. Specifically, *relational adaptive-consistency* is a full bucket-elimination algorithm whose approximations, *directional-relational-consistency(i,m)* ($DRC_{(i,m)}$), enforces bounded levels of consistency [Dechter and van Beek, 1997]. For example, (denoting a generic mini-bucket algorithm by *elim-approx*) directional relational arc-consistency, DRC_1, corresponds to *elim-approx*($m = 1$); directional path-consistency, DRC_2, corresponds to *elim-approx*($m = 2$); and so on.

Using mini-bucket approximations as heuristics for search parallels pre-processing by local consistency prior to backtrack search for constraint solving. In propositional satisfiability, where the original Davis-Putnam algorithm [Davis and Putnam, 1960] is a bucket-elimination algorithm, *bounded-directional-resolution* with bound b [Dechter and Rish, 1994] corresponds to *elim-approx*($i = b$).

Finally, a collection of approximation algorithms for sigmoid belief networks was recently presented [Jaakkola and Jordan, 1996] in the context of a recursive algorithm similar to bucket elimination. Upper and lower bounds are derived by approximating sigmoid functions by Gaussian functions. This approximation can be viewed as a singleton mini-bucket algorithm where Gaussian functions replace the *min* or *max* operations applied in each mini-bucket.

6 Conclusion

We present a collection of parameterized algorithms that approximate bucket-elimination algorithms. The basic idea is to partition each bucket into mini-buckets in order to control complexity. Due to the generality of the bucket-elimination framework, such algorithms and their approximations will apply uniformly across areas such as constraint optimization, satisfiability, probabilistic reasoning, and planning. Here we introduced the algorithms in the context of deterministic and probabilistic inference tasks.

Many questions need to be investigated in the context of our proposal. For example, given that there are many (i, m)-partitionings, and that each may result in different accuracy, how do we determine a good partitioning? Such questions should be investigated empirically, and will, we hope, yield domain-independent and domain-dependent heuristics.

Acknowledgements

Thanks to Irina Rish for commenting on an earlier version of the paper.

References

[Arnborg, 1985] S. A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25:2–23, 1985.

[Bertele and Brioschi, 1972] U. Bertele and F. Brioschi. In *Nonserial Dynamic Programming*. Academic Press, 1972.

[Canning *et al.*, 1978] C. Canning, E.A. Thompson, and M.H. Probability functions on complex pedigrees. *Advances in Applied Probability*, 10:26–61, 1978.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association of Computing Machinery*, 7(3), 1960.

[Dechter and Pearl, 1987] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[Dechter and Rish, 1994] R. Dechter and I. Rish. Directional resolution: The davis-putnam procedure, revisited. In *Proceedings of Knowledge Representation (KR-94)*, pages 134–145, Bonn, Germany, 1994.

[Dechter and van Beek, 1997] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science, (A preliminary version appears in CP-95 International Conference on Principles and Practice of Constraint Programming, pp. 240-257, 1995)*, 1997.

[Dechter, 1996] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference algorithms. In *Uncertainty in AI (UAI-96)*, pages 211–219, 1996.

[Jaakkola and Jordan, 1996] T. S. Jaakkola and M. I. Jordan. Recursive algorithms for approximating probabilities in graphical models. *Advances in Neural Information Processing Systems*, 9, 1996.

[Lassez and Mahler, 1992] J.-L. Lassez and M. Mahler. On fourier's algorithm for linear constraints. *Journal of Automated Reasoning*, 9, 1992.

[Pearl, 1984] J. Pearl. Heuristics: Intelligent search strategies. In *Addison-Wesley*, 1984.

[Robertson and Seymour, 1995] N. Robertson and P. Seymour. Graph minor. xiii. the disjoint paths problem. *Combinatorial Theory, Series B*, 63:65–110, 1995.

[Shenoy, 1992] P.P. Shenoy. Valuation-based systems for bayesian decision analysis. *Operations Research*, 40:463–484, 1992.

[Tatman and Shachter, 1990] J.A. Tatman and R.D. Shachter. Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 1990.

[Zhang and Poole, 1996] N. L. Zhang and D. Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research, (JAIR)*, 1996.