Recursive Best-First AND/OR Search for Optimization in Graphical Models

Akihiro Kishimoto and **Radu Marinescu**
IBM Research – Ireland
{akihirok,radu.marinescu}@ie.ibm.com

Abstract

The paper presents and evaluates the power of limited memory best-first search over AND/OR spaces for optimization tasks in graphical models. We propose *Recursive Best-First AND/OR Search with Overestimation* (RBFAOO), a new algorithm that explores the search space in a best-first manner while operating with restricted memory. We enhance RBFAOO with a simple overestimation technique aimed at minimizing the overhead associated with re-expanding internal nodes and prove correctness and completeness of RBFAOO. Our experiments show that RBFAOO is often superior to the current state-of-the-art approaches based on AND/OR search, especially on very hard problem instances.

1 INTRODUCTION

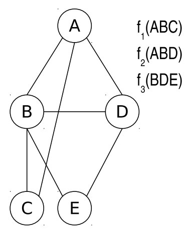
Graphical models provide a powerful framework for reasoning with probabilistic information. These models use graphs to capture conditional independencies between variables, allowing a concise knowledge representation and efficient graph-based query processing algorithms. Combinatorial optimization tasks such as MAP or marginal MAP inference arise in many applications and are typically tackled with either search or inference algorithms (Pearl, 1988; Dechter, 2003). The most common search scheme is the *depth-first branch and bound*. Its use for finding exact solutions was studied and evaluated extensively in the context of AND/OR search spaces that are sensitive to the underlying problem structure (Marinescu and Dechter, 2009a,b).

Meanwhile, *best-first* search algorithms, despite their better time efficiency than depth-first search (Dechter and Pearl, 1985), are largely ignored in practice primarily due to their inherently enormous memory requirements (Marinescu and Dechter, 2009b). Furthermore, an important best-first search property, avoiding the exploration of unbounded paths, seems irrelevant to optimization tasks in graphical models where all solutions are at the same depth (ie, the number of variables).

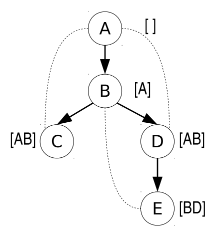
We aim at inheriting the advantages of both depth-first and best-first search schemes. We introduce RBFAOO, a new algorithm that explores the context minimal AND/OR search graph associated with a graphical model in a *best-first* manner (even with non-monotonic heuristics) while operating within restricted memory. RBFAOO extends Recursive Best-First Search (RBFS) (Korf, 1993) to graphical models and thus uses a threshold controlling technique to drive the search in a depth-first like manner while using the available memory to cache and reuse partial search results. In addition, RBFAOO employs an overestimation method designed to further reduce the high overhead caused by re-expanding internal nodes. RBFAOO is also related to the AND/OR search algorithms based on proof/disproof numbers (Allis et al., 1994) (eg, df-pn and df-pn$^+$ (Nagai, 2002)) which are very popular for solving two-player zero-sum games. However, since game solvers ignore the solution cost, they do not come with optimality guarantee. Moreover, while df-pn's completeness on finding suboptimal solutions assumes that the cache table preserves all search results of nodes previously explored (Kishimoto and Müller, 2008), RBFAOO is proven to be complete with a small cache table. We evaluate empirically RBFAOO on benchmark problems used during the PASCAL2 Inference Challenge. Our results show that RBFAOO is often superior to the state-of-the-art solvers based on AND/OR search, especially on the hardest problem instances.

2 PRELIMINARIES

We consider combinatorial optimization problems defined over graphical models, including Bayesian networks and Markov random fields (Pearl, 1988; Koller and Friedman, 2009). A *graphical model* is a tuple $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes \rangle$, where $\mathbf{X} = \{X_1, \dots, X_n\}$ is a set of variables and $\mathbf{D} = \{D_1, \dots, D_n\}$ is the set of their finite domains of values. $\mathbf{F} = \{f_1, \dots, f_r\}$ is a set of positive real-valued functions defined on subsets of variables, called *scopes* (ie,



(a) Primal graph



(b) Pseudo tree

Figure 1: A simple graphical model.

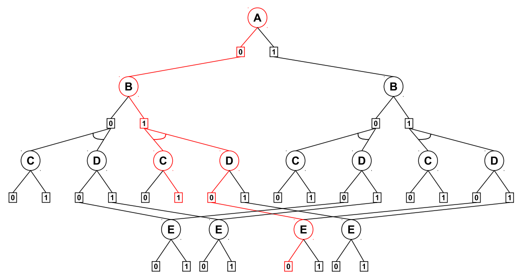


Figure 2: Context minimal AND/OR search graph.

$\forall j \; f_j : \mathbf{Y}_j \rightarrow \mathbb{R}^+$ and $\mathbf{Y}_j \subseteq \mathbf{X}$). The set of function scopes implies a *primal graph* whose vertices are the variables and which includes an edge connecting any two variables that appear in the scope of the same function. The combination operator $\bigotimes \in \{\prod, \sum\}$ defines the complete function represented by the graphical model $\mathcal{M}$ as $\mathcal{C}(\mathbf{X}) = \bigotimes_{j=1}^{r} f_j(\mathbf{Y}_j)$. In this paper, we focus on *min-sum problems*, in which we would like to compute the optimal value $\mathcal{C}^*$ and/or its optimizing configuration x^*:

$$\mathcal{C}^* = \mathcal{C}(x^*) = \min_{\mathbf{X}} \sum_{j=1}^{r} f_j(\mathbf{Y}_j) \tag{1}$$

Koller and Friedman (2009) convert the MAP task defined by $max_{\mathbf{X}} \prod_j f_j$ to log-space and solve it as an *energy minimization* (min-sum) problem to avoid numerical issues.

2.1 AND/OR SEARCH SPACES

Dechter and Mateescu (2007) introduce the concept of AND/OR search spaces for graphical models. A *pseudo tree* of the primal graph defines the search space and captures problem decomposition.

Definition 2.1. *A* pseudo tree *of an undirected graph $G = (V, E)$ is a directed rooted tree $\mathcal{T} = (V, E')$, such that every arc of G not included in E' is a back-arc in $\mathcal{T}$, namely it connects a node in $\mathcal{T}$ to an ancestor in $\mathcal{T}$. The arcs in E' may not all be included in E.*

Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$ with primal graph G and a pseudo tree $\mathcal{T}$ of G, the *AND/OR search tree* $S_{\mathcal{T}}$ based on $\mathcal{T}$ has alternating levels of OR nodes corresponding to the variables and AND nodes corresponding to the values of the OR parent's variable, with edges weighted according to $\mathbf{F}$. Identical subproblems, identified by their *context* (the partial instantiation that separates the subproblem from the rest of the problem graph), can be merged, yielding an *AND/OR search graph* (Dechter and Mateescu, 2007). Merging all context-mergeable nodes yields the *context minimal AND/OR search graph*, denoted by $C_{\mathcal{T}}$. The size of the context minimal AND/OR graph is exponential in the induced width of G along a depth-first traversal of $\mathcal{T}$ (Dechter and Mateescu, 2007).

A *solution tree* T of $C_{\mathcal{T}}$ is a subtree such that: (1) it contains the root node of $C_{\mathcal{T}}$; (2) if an internal AND node n is in T then all its children are in T; (3) if an internal OR node n is in T then exactly one of its children is in T; (4) every tip node in T (ie, nodes with no children) is a terminal node. The cost of a solution tree is the sum of the weights associated with its arcs.

Each node n in $C_{\mathcal{T}}$ is associated with a *value* $v(n)$ capturing the optimal solution cost of the conditioned subproblem rooted at n. It was shown that $v(n)$ can be computed recursively based on the values of n's successors: OR nodes by minimization, AND nodes by summation (see also (Dechter and Mateescu, 2007)).

Example 1. *Figure 1(a) shows a simple graphical model with 5 bi-valued variables $\{A, B, C, D, E\}$ and 3 functions $\{f_1(ABC), f_2(ABD), f_3(BDE)\}$, respectively. Figure 2 displays the context minimal AND/OR search graph based on the pseudo tree from Figure 1(b). The contexts of the variables are shown next to the corresponding pseudo tree nodes. A solution tree corresponding to the assignment $(A = 0, B = 1, C = 1, D = 0, E = 0)$ is highlighted.*

2.2 AND/OR SEARCH ALGORITHMS

AND/OR Branch and Bound (AOBB) (Marinescu and Dechter, 2009a,b) is a state-of-the-art informed search approach for solving optimization tasks over graphical models. AOBB explores in a *depth-first* manner the context minimal AND/OR search graph associated with the problem and therefore takes advantage of problem decomposition. During search, AOBB keeps track of the value of the best solution found so far (an upper bound on the optimal cost) and uses this value and the heuristic function to prune away portions of the search space that are guaranteed not to contain the optimal solution in a typical branch and bound manner. Most notably, AOBB guided by a class of partitioning based heuristics won the first place in the PASCAL2 competition (Otten et al., 2012).

Best-First AND/OR Search (Marinescu and Dechter, 2009b) (AOBF) is a variant of AO* (Nilsson, 1980) applicable to graphical models that explores the graph in a *best-first* rather than depth-first manner. This enables AOBF to visit a significantly smaller search space than AOBB which sometimes translates into important time savings. Extensive empirical evaluations (Marinescu and Dechter, 2009b) showed that when given enough memory AOBF is often superior to AOBB. However, in many practical situations AOBF's overhead of maintaining in memory the explicated portion of the search space is still prohibitively large. AOBB therefore remains the best alternative.

3 RECURSIVE BEST-FIRST AND/OR SEARCH WITH OVERESTIMATION

We introduce RBFAOO, a new algorithm that belongs to the class of recursive best-first search algorithms and employs a local threshold controlling mechanism to explore the context minimal AND/OR search graph in a depth-first like manner (Korf, 1993; Nagai, 2002). It can however use additional memory to cache and reuse partial search results to enhance performance. RBFAOO also leverages an overestimation technique to possibly find a suboptimal solution and then refine it to an optimal one. The latter plays an essential role in enhancing the performance by avoiding a high overhead of re-expanding internal nodes.

Before explaining RBFAOO in detail, we give an overview of the threshold controlling scheme that makes RBFAOO behave similarly to AO*. Assume that the weight from an OR node to an AND node is 1, the weight from an AND node to an OR node is 0, and a heuristic function h returns values as shown in Figure 3. Let $q(n)$, called *q-value*, be a lower bound of the solution cost at node n and $th(n)$ be RBFAOO's threshold at n. RBFAOO keeps examining the subtree rooted at n until either $q(n) > th(n)$ or the subtree is solved optimally. In Figure 3(a), RBFAOO selects B to expand, because $w(A,B) + q(B) = w(A,B) + h(B) = 3 < w(A,C) + q(C) = w(A,C) + h(C) = 5$. It sets $th(B) = w(A,C) + q(C) - w(A,B) = 4$ to indicate that C becomes the best child (ie, $w(A,B) + q(B) > w(A,C) + q(C)$ holds) if $q(B) > th(B)$. Then, RBFAOO expands B and updates $q(B)$ by using the q-values of B's children (Figure 3(b)). Because $q(B) = q(D) + q(E) = h(D) + h(E) = 3$, $q(B) \leq th(B)$ still holds. Hence, RBFAOO examines B's descendants with no backtracks to A. Assume that D is chosen to examine. RBFAOO sets $th(D) = th(B) - q(E) = 2$ to indicate that C becomes best if $q(D) > th(D)$ holds, which is equivalent to $q(B) > th(B)$, because $q(B) = q(D) + q(E)$ and $th(B) = th(D) + q(E)$. Next, RBFAOO expands D and updates $q(D) = w(D,F) + h(F) = 4$ (Figure 3(c)). Because $q(D) > th(D)$, the subtree rooted at D contains no best leaf in terms of AO*'s strategy. RBFAOO backtracks to A by updating $q(B) = q(D) + q(E) = 6$ and examines C (Figure 3(d)) with $th(C) = w(A,B) + q(B) - w(A,C) = 6$ to be able to select B when B becomes best.

RBFAOO gradually grows its search space by updating the q-values of internal nodes and re-expanding them. The overhead of internal node re-expansions is still high, even if RBFAOO does not always propagate back the q-values. For example, assume that an internal OR node n has two children c_1 and c_2, c_1 is selected to re-expand, and RBFAOO proves that c_2 becomes best to examine after expanding only one leaf that is k-steps away from c_1. If k is large, RBFAOO need to spend most of time in re-expanding internal nodes without exploring the new search space. The overestimation technique avoids this scenario by increasing the threshold while it verifies solution optimality.

3.1 ALGORITHM DESCRIPTION

Figure 4 shows the pseudo-code of RBFAOO. Let ϵ be a small number and assume $\infty - \epsilon < \infty$. In practice, a finite real number is used to represent ∞. Let δ be an empirically tuned parameter that determines the amount of overestimation. $HasNoChildren$ checks whether a node has no children (ie, terminal leaf or dead-end) or not. $Evaluate$ evaluates a terminal leaf/dead-end n and returns a pair of the cost (ie, 0 or ∞) and a Boolean flag indicating whether n is solved or a dead-end. $UnsolvedChild$ returns an unsolved child. $SaveInCache$ saves in the cache table a q-value and a flag indicating whether a node is solved optimally or not. $RetrieveFromCache$ retrieves them from the cache table. $Context$ calculates the context of a node.

When RBFAOO starts solving a problem, the threshold of the root node is set to $\infty - \epsilon$. If RBFAOO exceeds this threshold, the problem is proven to have no solution. Otherwise, RBFAOO returns the optimal solution cost to the problem. In addition, RBFAOO can be easily instrumented to recover the assignment corresponding to the optimal solution cost (this extension is omitted for clarity reasons).

Function $RBFS(n)$ traverses the subtree rooted at n in a depth-first manner. It calculates either an optimal solution cost or a lower bound by using $BestChild$ or Sum and checks if the termination condition is satisfied. If the solution optimality is guaranteed at n, $n.solved$ is set to **true**.

At an OR node, $RBFS(n)$ may find a suboptimal solution. In this case, $n.solved$ is still set to **false** and $RBFS(n)$ continues examining other children until it finds an optimal solution at n. Because the solution cost found so far is an upper bound of the optimal one, $RBFS(n)$ uses that solution cost (maintained by ub) to prune away unpromising branches and to adjust the threshold.

When $RBFS(n)$ selects c_{best}, it examines c_{best} with a new threshold. At OR nodes, $c_{best}.th$ is set to subtracting the weight between n and c_{best} from the minimum of:

1. The current threshold for n.
2. The second smallest lower bound q_2 to solve n's child with considering the weight from n to that child among a list of such lower bounds of n's children.

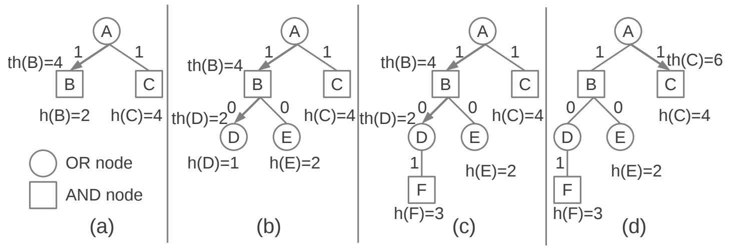


Figure 3: Snapshot of RBFAOO without overestimation.

This indicates when the current second best child becomes the best one. Additionally, a parameter δ called *overestimation rate*, that allows for returning a suboptimal solution cost is added to q_2 to avoid an excessive number of backtracks to n.

3. The upper bound of the optimal solution at n.

At AND nodes, $c_{best}.th$ is set to the sum of c_{best}'s q-value and the gap between $n.th$ and the total q-value of n's children. If $q(c_{best}) > c_{best}.th$, $q(n) > n.th$ also holds.

Let N be the number of nodes in the search space. If the search space fits into memory, AO* expands $O(N)$ nodes in the worst case. In contrast, due to node re-expansions, RBFAOO's worst-case scenario is $O(N^2)$. However, in practice, by introducing δ, RBFAOO avoids such a high node re-expansion overhead (see Section 4).

3.2 IMPLEMENTATION DETAILS

The cache table is implemented as a hash table with the Zobrist function (Zobrist, 1970) using 96-bit integers. The Zobrist function computes almost uniformly distributed hash keys by XORing precomputed random integers, each of which represents the component of a context and is commonly used in game-playing programs and in planning. Each cache table entry preserves the context of a node to avoid collisions caused by an astronomically small possibility of two different nodes having the same hash key.

When RBFAOO fills up the cache table and tries to store new results there, some cached results must be replaced. We use SmallTreeGC (Nagai, 1999), a batch-based replacement that discards $R\%$ of the table entries with small subtree sizes. We set R to 30.

Due to floating-point errors, RBFAOO is occasionally unable to expand a new leaf. We bypass this by using small error margins when comparing floating-point numbers.

As in full RBFS (Korf, 1993), q-values of children can be increased based on the current q-value of their parent. This technique generates more accurate heuristic information when cache table entries are replaced or non-monotonic heuristic functions are used. We implemented this technique and observed small performance improvement.

3.3 CORRECTNESS AND COMPLETENESS

We prove that RBFAOO is both correct and complete for solving optimization tasks defined over graphical models even with a small cache table and arbitrary cache table replacement schemes as long as certain table entries are preserved. In contrast, df-pn$^+$ may return suboptimal solutions if the contexts of nodes are used. Additionally, although its completeness on finding either one (possibly suboptimal) or no solution is proven, df-pn$^+$ must preserve all TT entries and the completeness with TT replacement schemes remains an open question (Kishimoto and Müller, 2008). The following theorems hold with/without enhancements described in Section 3.2.

Theorem 3.1 (correctness)**.** *Given a graphical model* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$, *if RBFAOO solves* $\mathcal{M}$ *with admissible heuristic function* h*, its solution is always optimal.*

Proof. Let $v(n)$ be the optimal solution cost for node n. We first prove that for any value q for node n in the cache table, $q \leq v(n)$ holds. Since different nodes with the same context are proven to be equivalent in DAGs (Marinescu and Dechter, 2009b), we denote n as Context(n) when a search result at node n is saved in the cache table. Additionally, we assume $h(n) = h(n')$ if Context(n) = Context(n').

Let $Cache_t$ be the state of the cache table immediately after the t-th save is performed in the cache table. Let $Q_t(n)$ be the value saved in $Cache_t$ for n if that value exists in $Cache_t$ or $h(n)$ if n is not preserved in $Cache_t$. By induction on t, we prove that all the entries in cache table contain values that do not overestimate optimal ones.

1. Because no result is stored in $Cache_0$, the above property holds for $t = 0$.

2. Assume that the above property holds for $t = k$. $Q_{k+1}(n)$, saved in $Cache_{k+1}$, is then calculated as:

   - If n is a terminal leaf, Evaluate(n) in the pseudo code always returns $v(n)$. $Q_{k+1}(n) = v(n)$ therefore holds.

```
// Set up for the root node
double RBFAOO(node $root$) {
  $root.th$ = $\infty - \epsilon$;
  $q$ = RBFS($root$);
  return $q$;
}
// Depth-first search with a threshold
double RBFS(node $n$) {
  // Terminal leaf/dead-end check
  if (HasNoChildren($n$)) {
    // Calculate the probability
    // for a terminal leaf or dead-end
    $(q, s)$ = Evaluate($n$);
    // Store search results
    SaveInCache(Context($n$),$q$,$s$);
    return $q$;
  }
  GenerateChildren($n$);
  // Continue search until satisfying
  // the termination condition
  if ($n$ is an OR node)
    loop {
      $(c_{best}, q, q_2, ub)$ = BestChild($n$);
      if ($n.th < q$ || $n.solved$ = true)
        break;
      // Update the threshold
      $c_{best}.th$ = min($n.th$,
                          $q_2 + \delta$,
                          $ub) - w(n, c_{best})$;
      RBFS($c_{best}$);
    }
  else
    loop { // AND node
      $q$ = Sum($n$);
      if ($n.th < q$ || $n.solved$ = true)
        break;
      $(c_{best}, q_{c_{best}})$ = UnsolvedChild($n$);
      // Update the threshold
      $c_{best}.th$ = $n.th - (q - q_{c_{best}})$;
      RBFS($c_{best}$);
    }
  // Store search results
  SaveInCache(Context($n$),$q$,$n.solved$);
  return $q$;
}

// Select the best child
double BestChild(node $n$) {
  $q = q_2 = ub = \infty$;
  $n.solved$=false;
  foreach ($n$'s child $c_i$) {
    $ct$ = Context($c_i$);
    if ($ct$ is in the cache table)
      $(q_{c_i}, s)$ = RetrieveFromCache($ct$);
    else {
      $q_{c_i}$ = $h(c_i)$;
      $s$=false;
    }
    $q_{c_i}$ = $w(n, c_i) + q_{c_i}$;
    if ($s$=true) // $c_i$ is solved
      $ub$ = min($ub, q_{c_i}$);
    if ($q_{c_i} < q$ ||
        ($q_{c_i} = q$ && $n.solved$=false)) {
      $q_2 = q$;
      $n.solved = s$;
      $q = q_{c_i}$;
      $c_{best} = c_i$;
    } else if ($q_{c_i} < q_2$)
      $q_2 = q_{c_i}$;
  }
  return $(c_{best}, q, q_2, ub)$;
}
// Calculate the total value
double Sum(node $n$) {
  $q$ = 0;
  $n.solved$ = true;
  foreach ($n$'s child $c_i$) {
    $ct$ = Context($c_i$);
    if ($ct$ is in the cache table)
      $(q_{c_i}, s)$ = RetrieveFromCache($ct$);
    else {
      $q_{c_i}$ = $h(c_i)$;
      $s$ = false;
    }
    $q = q + q_{c_i}$;
    $n.solved = n.solved \wedge s$;
  }
  return $q$;
}
```

Figure 4: Pseudo-code of RBFAOO

- If n is an internal OR node, $Q_{k+1}(n) = w(n, c_{best}) + Q_k(c_{best}) = \min_i(w(n, c_i) + Q_k(c_i))$ holds where c_i is n's child. Additionally, because $Q_k(c_i) \leq v(c_i)$, $Q_{k+1}(n) \leq \min_i(w(n, c_i) + v(c_i)) = v(n)$ holds.
- If n is an internal AND node, $Q_{k+1}(n) = \sum_i Q_k(c_i)$ where c_i is n's child. Since $Q_k(c_i) \leq v(c_i)$, $Q_{k+1}(n) \leq \sum_i v(c_i) = v(n)$ holds.

Hence, $Q_t(n) \leq v(n)$ holds in case of $t = k + 1$.

Let $Q(root)$ be a value that is about to be saved in the cache table with satisfying the termination condition of $root.solved = true$. $Q(root) \leq v(root)$ holds from the above. Additionally, because RBFAOO has traced a solution tree with the cost of $Q(root)$, $v(root) \leq Q(root)$ holds. Therefore, $Q(root) = v(root)$ holds. □

Theorem 3.2 (completeness)**.** *Let $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$ be a graphical model with primal graph G, let $\mathcal{T}$ be a pseudo tree G and let $C_{\mathcal{T}}$ be the context minimal AND/OR search graph based on $\mathcal{T}$ (also a finite DAG). Assume that RBFAOO preserves the q-values of the nodes $n_1, n_2, \cdots, n_k$ which are on the current search path and the q-values of n_i's siblings. Then RBFAOO eventually returns an optimal solution or proves no solution exists.*

Proof sketch. Let a *marked* node be a node expanded at least once by RBFAOO and an *unmarked* node be a node that has never been expanded. Denote $p_1 \subset p_2$ if the path length of p_2 is longer than that of p_1 and p_2 is identical to p_1 if p_2 is limited to the path with the length of p_1. Denote $p_1 \not\subset p_2$ unless it holds $p_1 \subset p_2$. Let $s(n,p)$ be the sum of the edge costs from the root to n via path p. Assume that $s(n,p)+q(n) < \infty - \epsilon$ holds for any path p if $q(n) \neq \infty$, which is reasonable in practical settings.

Let $q_{t,p}(n)$ and $th_{t,p}(n)$ be the q-value and threshold for t-th visit to n via path p, respectively. Assume that RBFAOO expands no unmarked nodes after expanding k unmarked nodes. Then, because the search space is finite, there are two unproven marked nodes n and m examined as follows:

1. RBFAOO starts searching downward from n via path p_1 since $q_{t_1,p_1}(n) \leq th_{t_1,p_1}(n)$ holds.
2. RBFAOO reaches m via path p_2 that satisfies $th_{t_2,p_2}(m) < q_{t_2,p_2}(m)$ and $p_1 \subset p_2$.
3. RBFAOO keeps exploring the remaining search space rooted at n via p_1 (and composed of marked nodes) and backtracks to n.
4. Continue steps (1)-(3).

If n is an AND node, satisfying $th_{t',p_3}(c) < q_{t',p_3}(c)$ immediately leads to satisfying $th_{t,p_1}(n) < q_{t,p_1}(n)$ where c is n's unproven child and $p_1 \subset p_3$. Because backtracking to n's parent contradicts step (1), n is an OR node. Additionally, $th_{t',p_3}(c) < th_{u,p_3}(c)$ holds for any unproven child c, $t' < u$ and $p_1 \subset p_3$, because the q-values of n's children are preserved in memory as described in the assumption of the theorem. With similar discussions, there is an infinite sequence $u_1, u_2, \cdots, u_k, \cdots$ that satisfies $u_i < u_j$ for $i < j$, $th_{u_i,p_2}(m) < th_{u_j,p_2}(m)$ and $th_{u_i,p_2}(m) < q_{u_i,p_2}(m)$. This indicates that m has at least one unproven child o_1 via path r_1 ($p_2 \not\subset r_1$) that contributes to increasing $q_{u_i,p_2}(m)$ and satisfying $th_{u_i,p_2} < q_{u_i,p_2}(m)$ when $q_{u_i,p_2}(m)$ is calculated. Because the search space is DAG, $q(o_1)$ is never affected by $q(m)$. With similar discussions, if no unmarked node is expanded, there is an infinite number of nodes $o_1, o_2, \cdots, o_k, \cdots$, where o_{j+1} is a child of o_j that contributes to increasing $q(o_j)$ and satisfying $th(o_j) < q(o_j)$. However, this contradicts the assumption of the finite search space. Hence, by eventually examining the whole search space, RBFAOO finds an optimal solution (see Theorem 3.1) or proves no solution. □

4 EXPERIMENTS

We empirically evaluate our proposed best-first search scheme on the MAP task in graphical models. We compare RBFAOO against the state-of-the-art depth-first and best-first AND/OR search solvers proposed recently in (Marinescu and Dechter, 2009b) and denoted by AOBB and AOBF, respectively. All competing algorithms use pre-compiled mini-bucket heuristics (Kask and Dechter, 2001; Marinescu and Dechter, 2009b) for guidance and are restricted to a static variable ordering obtained as a depth-first traversal of a minfill pseudo tree (Marinescu and Dechter, 2009a). Since AOBF cannot use an initial upper bound (obtained via local search) we also disabled its use by AOBB and RBFAOO in order to maintain a fair comparison.

Our benchmark problems[1] include three sets of instances from genetic linkage analysis (Fishelson and Geiger, 2002) (denoted `pedigree`), grid networks and protein side-chain interaction networks (denoted `pdb`) (Yanover et al., 2008). In total, we evaluated 21 pedigrees, 32 grids and 240 protein networks. The algorithms were implemented in C++ (64-bit) and the experiments were run on a 2.6GHz 8-core processor with 80GB of RAM.

We report the CPU time in seconds and the number of nodes expanded for solving the problems. We also specify the problems parameters such as the number of variables (n), maximum domain size (k), the depth of the pseudo tree (h) and the induced width of the graph (w^*). The best performance points are highlighted. In each table, 'oom' stands for out-of-memory and '-' denotes out-of-time. Note that oom for RBFAOO/AOBB indicates that the mini-bucket heuristic pre-computation procedure uses up the physical memory before search is performed.

Tables 1 and 2 show the results obtained for experiments with pedigree, grid and protein networks. For space reasons and clarity we select a representative subset from the full 293 instances. The columns are indexed by the mini-bucket i-bound which ranged between 6 and 16 for pedigrees and grids, and between 2 and 5 for proteins, respectively. All algorithms were allotted a 1 hour time limit. Algorithm AOBF(i) was allowed a maximum of 80GB of RAM while algorithm RBFAOO(i) used a 10-20GB cache table with 134,217,728 entries pre-allocated before search. The overestimation parameter δ was set to 1.

We observe clearly that RBFAOO(i) improves considerably over its competitors, especially at relatively small i-bounds which yield relatively weak heuristics. For example, on the pedigree30 instance, RBFAOO(6) with the smallest reported i-bound ($i = 6$) was 4 and 41 times faster than AOBF(6) and AOBB(6), respectively. Similarly, RBFAOO(6) solves the 75-23-5 grid in about 30 minutes and expands over 300 million nodes, while both AOBB(6) and AOBF(6) run out of time and memory, respectively. As the i-bound increases and the heuristics become more accurate thus pruning the search space more effectively, the differences in running time between the algorithms de-

[1] All instances are available at *http://graphmod.ics.uci.edu*.

Table 1: CPU time (seconds) and number of nodes expanded for `pedigree` and `grid` networks. Time limit 1 hour. RBFAOO(i) ran with a 10-20GB cache table (134,217,728 entries) and overestimation parameter $\delta = 1$.

instance	algorithm	$i = 6$		$i = 8$		$i = 10$		$i = 12$		$i = 14$		$i = 16$	
(n, k, w^*, h)		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
pedigree instances													
	AOBB		-		-		-		-		-		-
pedigree7	AOBF		oom		oom		oom		oom		oom		oom
(1068,4,28,140)	RBFAOO		-		-		-	**2210**	345204317	**1368**	216767091	**818**	144733023
	AOBB		-		-		-		-		-	1076	139749607
pedigree9	AOBF		oom		oom	1846	30506650	1379	20960401	1152	20897564	263	7682927
(1118,7,25,123)	RBFAOO	**1084**	195214857	**728**	136764248	**522**	97410715	**248**	46922921	**241**	44561263	**60**	10634230
	AOBB		-		-		-		-		-		-
pedigree13	AOBF		oom		oom		oom		oom		oom		oom
(1077,3,30,125)	RBFAOO		-		-		-		-		-	**2629**	364037130
	AOBB		-		-		-		-		-		-
pedigree19	AOBF		oom		oom		oom		oom		oom		oom
(793,5,21,51)	RBFAOO		-		-	**1753**	319268527	**834**	168262596	**226**	45738797	**378**	69780223
	AOBB	825	113195179	1450	198371250	244	34182326	63	10855277	102	17794376	**3**	107437
pedigree30	AOBF	83	2648120	103	2689106	45	1717523	24	867988	39	932986	**3**	30794
(1289,5,20,105)	RBFAOO	**20**	5435997	**19**	5401921	**14**	3840692	**5**	1406493	**6**	1691396	**3**	60479
	AOBB		-	935	125740961	107	15616376	5	885551	9	1272810	**5**	24174
pedigree39	AOBF	307	9740964	215	8073776	53	2347928	8	384757	14	607860	**5**	19960
(1272,5,20,77)	RBFAOO	**79**	19804239	**67**	16260143	**14**	3461943	**2**	480866	**4**	666873	**5**	24826
	AOBB		-		-		-		-		-		-
pedigree41	AOBF		oom		oom		oom		oom		oom		oom
(1062,5,29,119)	RBFAOO		-		-		-	**2706**	373308327	**3312**	440228598	**1517**	210630024
binary grid instances													
	AOBB		-		-		-		-		-		-
50-20-5	AOBF		oom		oom		oom		oom	1309	33138951	789	19857843
(400,2,27,97)	RBFAOO	**1163**	214829892	**736**	142564959	**385**	80803927	**232**	48848448	**120**	25641963	**66**	13994679
	AOBB		-	738	111785572	309	43858649	36	5997367	19	3234878	10	1405451
75-20-5	AOBF	2268	30767273	567	20761132	182	5685498	42	1766240	23	930839	12	442278
(400,2,27,99)	RBFAOO	**212**	46289779	**89**	19603768	**39**	8451032	**9**	2026625	**5**	1007726	**4**	511806
	AOBB		-		-		-	2206	314621887	994	144092486	67	10500198
75-22-5	AOBF		oom		oom	1123	34528523	743	22103512	313	10577016	49	1714348
(484,2,30,107)	RBFAOO	**563**	107126385	**643**	118981360	**227**	44947693	**153**	29325424	**91**	18077594	**17**	3047665
	AOBB		-		-		-		-		-	131	16039678
75-23-5	AOBF		oom		oom	1751	39532238	417	11103193	340	8092564	37	1218023
(529,2,31,122)	RBFAOO	**1860**	304935340	**1109**	198613807	**455**	87285533	**106**	20952230	**71**	13910863	**17**	2915543
	AOBB		-		-		-		-		-		oom
75-26-5	AOBF		oom		oom		oom		oom		oom		oom
(676,2,36,134)	RBFAOO		-		-		-		-	**3005**	394135020		oom
	AOBB		-		-	1479	195188949	560	74507590	51	7366618	102	13921196
90-23-5	AOBF	970	32478634	376	12937697	289	10087022	91	3169720	52	1944481	33	1224632
(529,2,31,116)	RBFAOO	**277**	52920346	**105**	20736738	**71**	14460847	**18**	3602104	**9**	1810658	**9**	1390189
	AOBB		-		-		-		-		-	1647	186283089
90-26-5	AOBF	1016	25948278	1108	29700313	505	16035732	552	16728882	457	12983459	159	4413795
(676,2,36,136)	RBFAOO	**241**	44068170	**183**	33284922	**68**	12738955	**65**	12176988	**49**	9170451	**30**	5180019

crease. In terms of the size of the search spaces explored, we see that AOBF(i) typically expands the smallest number of nodes, as expected. RBFAOO(i) expands more nodes that AOBF(i), due to re-expansions, but in many cases it expands significantly fewer nodes than AOBB(i) which translates into important time savings. We also notice that RBFAOO(i) and AOBB(i) have a relatively small overhead per node expansion. On the other hand, the computational overhead of AOBF(i) is much larger. It is caused primarily by maintaining an extremely large search space in memory and, secondly, because the node values are typically updated all the way up to the root. Most notably, RBFAOO(i) was the only algorithm that could solve the most difficult instances in these benchmarks (eg, pedigrees 7, 13, 19 and 41, as well as grid 75-26-5). This demonstrates the benefit of expanding nodes in best-first rather than depth-first manner as well as using efficiently a bounded amount of memory, thus overcoming the most critical limitation of AOBF(i). Finally, the results on the protein networks show a similar pattern, namely RBFAOO(i) improves considerably over both AOBB(i) and AOBF(i) for relatively small i-bounds. This is important because, unlike the pedigrees and grids, these problems have very large domains (81 values) and therefore the mini-bucket heuristics could only be compiled for small i-bounds. Figure 5 which plots the normalized total CPU time as a function of the i-bound summarizes the running time profile of the competing algorithms across the benchmarks we considered.

In Table 3 we report on five additional very difficult genetic linkage analysis networks. The mini-bucket i-bound was set to 20 in this case. We see again that RBFAOO is the best performing algorithm closing all instances within the 100 hour time limit. In contrast, AOBB could solve only one instance while AOBF ran out of memory. For example, on the type4-120-17 instance, RBFAOO was nearly 3 orders of magnitude faster than AOBB, while expanding 3 orders of magnitude fewer nodes.

We summarize next the most important additional factors that could help improve RBFAOO(i)'s performance.

Impact of caching: Table 4 shows the average performance of algorithm RBFAOO(i) (as CPU time in seconds, number of nodes expanded, and number of problem instances solved) as a function of available memory, across all three benchmarks. The columns are indexed by the cache table size used, namely *very small* (10-20MB), *small*

Table 2: CPU time (seconds) and number of nodes expanded for `protein` networks. Time limit 1 hour. RBFAOO(i) ran with a 10-20GB cache table (134,217,728 entries) and overestimation parameter $\delta = 1$.

instance	algorithm	$i = 2$		$i = 3$		$i = 4$		$i = 5$	
(n, k, w^*, h)		time	nodes	time	nodes	time	nodes	time	nodes
	AOBB		-		-	2218	65175805		oom
pdb1a3c	AOBF		oom		oom		oom		oom
(144,81,15,32)	RBFAOO	**1915**	45513907			**344**	259261		oom
	AOBB	129	2919570	**8**	2694	**204**	1302		oom
pdb1aac	AOBF	2851	3195539	11	6264	205	3072		oom
(85,81,11,21)	RBFAOO	**51**	1148212	**8**	1492	**204**	783		oom
	AOBB	996	55994055	2672	162495198	**16**	1593	**136**	4767
pdb1acf	AOBF		oom		oom	17	4021	139	10464
(90,81,9,22)	RBFAOO	**22**	987416	**56**	2553896	**16**	1090	137	3212
	AOBB	259	7770890	134	3806154	657	3312980	2552	274955
pdb1ad2	AOBF	831	1250161	394	715399	1109	1049637	2595	135265
(177,81,9,33)	RBFAOO	**36**	1227741	**42**	858899	**585**	1218780	**2543**	113780
	AOBB	4	177150	31	75051	610	1474224		oom
pdb1ail	AOBF	80	66207	47	15817	1728	928375		oom
(62,81,8,23)	RBFAOO	**2**	78677	**30**	16427	**599**	1311325		oom
	AOBB		-	6	154434	260	9348036	**236**	24412
pdb1atg	AOBF		oom	38	119195	632	1196429	247	30072
(175,81,12,39)	RBFAOO	**620**	24347033	**4**	71446	**32**	430747	**236**	11545

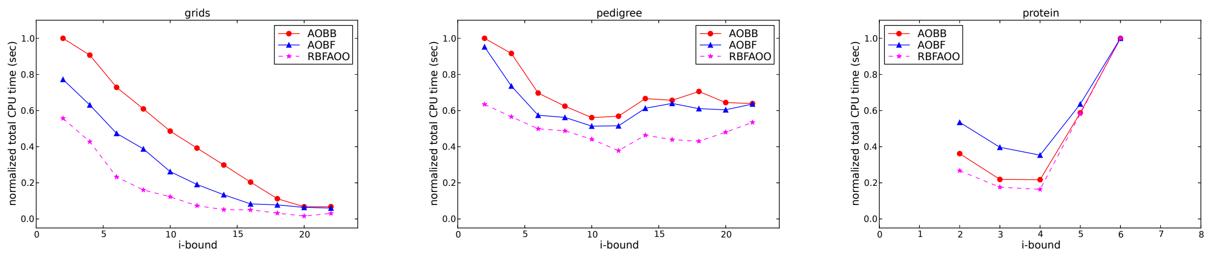


Figure 5: Normalized total CPU time as a function of the i-bound.

Table 3: CPU time (seconds) and node expansions for previously unsolved linkage networks. Time limit 100 hours.

instance	algorithm	time	nodes
	AOBB		-
type4b-100-19	AOBF		oom
(7308,5,29,354)	RBFAOO	**107258**	15157422871
	AOBB	162196	5473951156
type4b-120-17	AOBF		oom
(7766,5,24,319)	RBFAOO	**218**	3388901
	AOBB		-
type4b-130-21	AOBF		oom
(8883,5,29,416)	RBFAOO	**312887**	43341893185
	AOBB		-
type4b-140-19	AOBF		oom
(9274,5,30,366)	RBFAOO	**270856**	28653407450
	AOBB		-
largeFam3-10-52	AOBF		oom
(1905,3,36,80)	RBFAOO	**129633**	12826083707

(100-200MB), *medium* (1-2GB) and *large* (10-20GB), respectively. We see that, as expected, as more memory is available, the performance improves considerably, namely more problem instances are solved while the running time and size of the search space decrease significantly. The best results were obtained with the 10-20GB cache.

Impact of overestimation: Figure 6 plots the CPU time, node re-expansion rate (as the ratio of the number of nodes re-expanded to the total number of expansions) and percentage of problem instances solved by RBFAOO(i) as a function of the overestimation rate δ, across all benchmarks. We see that RBFAOO(i) without overestimation (ie, $\delta = 0$) performed rather poorly and was outperformed considerably by AOBB(i) and AOBF(i), respectively. This was due to a relatively large number of node re-expansions. On grids, for example, more than 78% of the nodes were actually re-expanded for $\delta = 0$ compared to only 11% re-expansions for $\delta = 1.2$. However, as δ increases the re-expansion rate decreases but the CPU time starts to increase due to explorations of unpromising search spaces. Therefore, we obtained the overall best performance for $\delta = 1$.

Impact of heuristics quality: Based on our empirical evaluation we noticed that RBFAOO(i) was superior to its competitors especially for relatively inaccurate heuristics (which are typically obtained for smaller i-bounds) and on the hardest problem instances. This is important because it is likely that for these types of problems it may only be possible to compute rather weak heuristics given limited resources (eg, `type4` instances in Table 3).

5 RELATED WORK

Algorithms based on proof and disproof numbers (Allis et al., 1994) have been dominating AND/OR search techniques and successfully applied to many game domains (eg, (Nagai, 2002; Kishimoto and Müller, 2005; Schaeffer et al., 2007)). See (Kishimoto et al., 2012) for a comprehensive literature review.

Table 4: Average CPU time (seconds), number of nodes expanded and number of problem instances solved by RBFAOO(i) with different cache sizes. Time limit 1 hour. $i = 10$ for grids and pedigrees, $i = 4$ for protein networks.

	10-20MB			100-200MB			1-2GB			10-20GB		
benchmark	time	nodes	solved	time	nodes	solved	time	nodes	solved	time	nodes	solved
grids	1928	496571526	15/32	1685	321943731	18/32	1257	203898268	22/32	1220	173080755	22/32
pedigree	1416	366052904	12/21	1350	277666996	13/21	1180	223258293	14/21	1155	188200810	14/21
protein	574	35439167	208/240	509	27532421	213/240	443	23135969	217/240	396	20521511	222/240

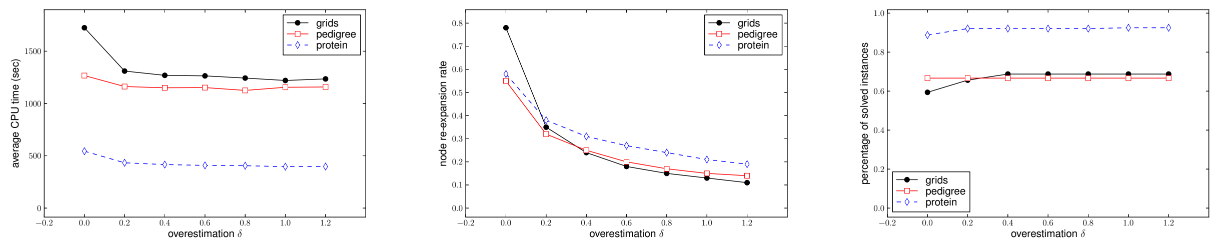


Figure 6: Average CPU time (seconds), node re-expansion rate and percentage of instances solved by RBFAOO(i) as a function of the overestimation rate δ. Time limit 1 hour. $i = 10$ for grids and pedigrees, $i = 4$ for protein networks.

A proof/disproof number estimates the difficulty of proving that the first/second player wins in a partially built tree. The proof number of node n is defined as the minimum number of leaf nodes that must be expanded to prove that the first player wins at n. A node with a smaller proof number is assumed to be easier to prove a win for the first player. In contrast, the disproof number of n is the minimum number of leaf nodes that must be expanded to prove that the second player wins at n. A node with a smaller disproof number is assumed to be easier to prove a win for the second player.

Depth-First Proof-Number Search (df-pn) (Nagai, 2002) is a depth-first reformulation of Best-First Proof-Number Search (PNS) (Allis et al., 1994) enhanced with a so-called transposition table (TT), a cache table preserving the search effort for the expanded nodes. While preserving PNS' leaf selection strategy, df-pn empirically re-expands fewer internal nodes than PNS that always restarts from the root the procedure of finding a promising leaf to expand. Besides, df-pn runs using a small amount of space limited by the TT size in practice, although whether df-pn is complete or not with a limited amount of TT still remains an open question. As in RBFS (Korf, 1993), df-pn introduces thresholds to limit the search depth of depth-first search. Df-pn updates the thresholds of a node by taking into account when the search tree rooted at that node contains none of the most promising leaf nodes chosen by PNS' best-first strategy. The df-pn$^+$ algorithm (Nagai, 2002) generalizes df-pn by introducing evaluation functions to heuristically initialize proof and disproof numbers and a weight in each edge to decrease the overhead of node re-expansions (Kishimoto and Müller, 2005).

Other related work includes MAO* (Chakrabati et al., 1989), memory-limited AO*. Although MAO* can run under a similar memory limit to RBFAOO, it needs a specific strategy to discard examined nodes from memory. In contrast, RBFAOO can leverage arbitrary TT replacement strategies including SmallTreeGC (Nagai, 1999), which is empirically most effective in solving games. Additionally, by incorporating ideas behind RBFS and the overestimation technique, RBFAOO has much smaller overhead to update node values than MAO* and AO*.

6 CONCLUSION

The paper presents RBFAOO, a limited memory best-first AND/OR search algorithm for solving combinatorial optimization defined over graphical models. RBFAOO belongs to the Recursive Best-First Search family of algorithms and therefore uses a threshold controlling mechanism to guide the search in a depth-first like manner. It also employs a flexible caching scheme to reuse partial search results as well as an overestimation mechanism to further reduce the internal node re-expansions. We prove correctness and completeness of the algorithm. We evaluate RBFAOO empirically on a variety of benchmarks used during the PASCAL2 Inference Challenge. Our results show that RBFAOO is often superior to current state-of-the-art solvers based on AND/OR search, especially on the most difficult problem instances.

For future work we plan to extend RBFAOO to use dynamic variable orderings, an initial upper bound obtained via local search and soft arc-consistency based heuristics. One possibility we are currently investigating is to implement RBFAOO on top of the `toulbar` solver (de Givry et al., 2005). Since many interesting real-world problems are still too hard to solve exactly, we also plan to convert the algorithm into an anytime best-first search scheme.

REFERENCES

L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

P. Chakrabati, S. Ghose, A. Acharya, and S. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 3(41):197–221, 1989.

S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted csps. In *International Joint Conference in Artificial Intelligence (IJCAI)*, 2005.

R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *In Journal of ACM*, 32(3): 505–536, 1985.

M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(1): 189–198, 2002.

K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 129(1-2):91–131, 2001.

A. Kishimoto and M. Müller. Search versus knowledge for solving life and death problems in Go. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 1374–1379, 2005.

A. Kishimoto and M. Müller. About the completeness of depth-first proof-number search. In *Computers and Games 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 146–156, 2008.

A. Kishimoto, M. Winands, M. Müller, and J.-T. Saito. Game-tree search using proof numbers: The first twenty years. *ICGA Journal, Vol. 35, No. 3*, 35(3):131–156, 2012.

D. Koller and N. Friedman. *Probabilistic Graphical Models*. MIT Press, 2009.

R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

R. Marinescu and R. Dechter. AND/OR branch-and-bound search for combinatorial optimization in graphical models. *Artificial Intelligence*, 173(16-17):1457–1491, 2009a.

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

A. Nagai. A new depth-first search algorithm for AND/OR trees. Master's thesis, Department of Information Science, University of Tokyo, 1999.

A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, The University of Tokyo, 2002.

N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, 1980.

L. Otten, A. Ihler, K. Kask, and R. Dechter. Winning the PASCAL 2011 MAP challenge with enhanced and/or branch-and-bound. Technical report, School of Information and Computer Science, University of California, Irvine, 2012.

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

J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

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

A. L. Zobrist. A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1970.