Parallel AND/OR Search for Marginal MAP

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

Adi Botea*
Eaton
Dublin, Ireland
adibotea@eaton.com

Abstract

Marginal MAP is a difficult mixed inference task for graphical models. Existing state-of-the-art algorithms for solving exactly this task are based on either depth-first or best-first *sequential* search over an AND/OR search space. In this paper, we explore and evaluate for the first time the power of *parallel* search for exact Marginal MAP inference. We introduce a new parallel shared-memory recursive best-first AND/OR search algorithm that explores the search space in a best-first manner while operating with limited memory. Subsequently, we develop a complete parallel search scheme that only parallelizes the conditional likelihood computations. We also extend the proposed algorithms into depth-first parallel search schemes. Our experiments on difficult benchmarks demonstrate the effectiveness of the parallel search algorithms against current sequential methods for solving Marginal MAP exactly.

Introduction

Graphical models provide a powerful framework for reasoning about conditional dependency structures over many variables. The Marginal MAP (MMAP) query asks for the optimal configuration of a subset of variables that has the highest marginal probability. More specifically, MMAP distinguishes between maximization variables (called MAP variables) and summation variables, and it is more difficult than either max- or sum-inference tasks alone primarily because summation and maximization operations do not commute, forcing processing along constrained variable orderings that may have significantly higher induced widths (Dechter 1999). This implies larger search spaces for search algorithms or larger messages when using message-passing schemes.

In general, MMAP is NP^{PP}-complete and it can be NP-hard even on tree structured models (Park 2002). Still, MMAP is often the appropriate task where hidden variables or uncertain parameters exist. It can also be treated as a special case of the more complicated frameworks of decision networks (Howard and Matheson 2005; Liu and Ihler 2013).

State-of-the-art exact algorithms for MMAP are based on either depth-first or best-first search over an AND/OR search space that is sensitive to the underlying problem structure. Specifically, AOBB is a recent algorithm that traverses the context minimal AND/OR search graph in a depth-first manner and uses a heuristic evaluation function to prune the search space in a typical branch and bound fashion (Marinescu, Dechter, and Ihler 2014). AOBF is a memory intensive algorithm that explores the AND/OR search space in a best-first rather than depth-first manner (Marinescu, Dechter, and Ihler 2015). This enables AOBF to visit significantly smaller search spaces than AOBB which sometimes translates into important time savings as well as considerably fewer conditional likelihood evaluations. The recursive best-first search algorithm called RBFAOO uses a threshold controlling mechanism to drive the search in a depth-first like manner and thus is able to operate within restricted memory (Marinescu, Dechter, and Ihler 2015). These algorithms are most effective when guided by heuristics based on weighted mini-bucket elimination which is enhanced with cost-shifting schemes (Liu and Ihler 2011; Marinescu, Dechter, and Ihler 2014).

Up until now, all these advanced search-based MMAP solvers were developed primarily as *sequential search* algorithms. One way to extract substantial speed-ups from the hardware and further improve the performance is to resort to *parallel processing*. Indeed, parallel search has been successfully applied in several AI areas, such as planning (Kishimoto, Fukunaga, and Botea 2013), satisfiability (Chrabakh and Wolski 2003), and game playing (Campbell, Hoane, and Hsu 2002; Silver et al. 2016). Solving graphical models in parallel, especially in shared-memory multi-core architectures, has been explored recently in the context of pure MAP queries only (Kishimoto, Marinescu, and Botea 2015).

In this paper, we explore and evaluate the power of shared-memory parallel search for solving MMAP exactly, which to the best of our knowledge has not been attempted before. More specifically, we develop SPRBFAOO-MMAP, a new parallel recursive best-first AND/OR search algorithm for MMAP. The algorithm maintains a single cache table which is shared amongst the threads so that each thread can effectively reuse the search effort performed by the others. In addition, the algorithm associates with each node a so-called virtual q-value that captures both the estimated cost

*This work was performed while the author was affiliated with IBM Research, Ireland.

of the subproblem below it as well as the number of threads currently working on that subproblem. This mechanism ensures an effective and balanced exploration of the search space. Although we can show that SPRBFAOO-MMAP is correct whether or not is a complete search scheme remains open. Therefore, we also introduce a complete parallel search scheme called SPRBFAOO2-MMAP that uses a single master thread to conduct the search over the MAP variables in a recursive best-first search manner and parallelizes only the conditional likelihood computations using a set of slave threads. Subsequently, we extend the proposed parallel best-first search algorithms into parallel depth-first search schemes. Our empirical evaluation on various difficult benchmarks for MMAP demonstrates conclusively that the new parallel AND/OR search schemes improve considerably over current state-of-the-art sequential AND/OR search approaches, in many cases leading to considerable speed-ups (up to 8-fold using 12 threads) especially on the harder problem instances.

Background

A *graphical model* is a tuple $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, where $\mathbf{X} = \{X_i : i \in V\}$ is a set of variables indexed by set V and $\mathbf{D} = \{D_i : i \in V\}$ is the set of their finite domains of values. $\mathbf{F} = \{\psi_\alpha(\mathbf{X}_\alpha) : \alpha \in F\}$ is a set of discrete non-negative real-valued factors indexed by set F, where each ψ_α is defined on a subset of variables $\mathbf{X}_\alpha \subseteq \mathbf{X}$, called its scope. Specifically, $\psi_\alpha : \Omega_\alpha \rightarrow \mathbb{R}_+$, where Ω_α is the Cartesian product of the domains of each variable in $\mathbf{X}_\alpha$. *primal graph* whose vertices are the variables and whose edges connect any two variables that appear in the scope of the same factor. The model $\mathcal{M}$ defines a factorized probability distribution on $\mathbf{X}$:

$$P(\mathbf{x}) = \frac{1}{Z} \prod_{\alpha \in F} \psi_\alpha(\mathbf{x}_\alpha) \quad (1)$$

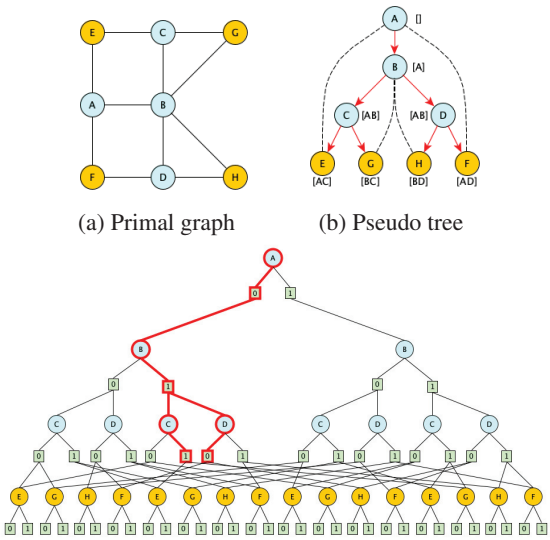
The *partition function*, $Z = \sum_{\mathbf{x}} \prod_{\alpha \in F} \psi_\alpha(\mathbf{x}_\alpha)$, normalizes the probability.

Let $\mathbf{X}_M = \{X_1, ..., X_m\}$ be a subset of $\mathbf{X}$ called *MAP variables* and $\mathbf{X}_S = \mathbf{X} \setminus \mathbf{X}_M$ be the complement of $\mathbf{X}_M$, called *SUM variables*. The Marginal MAP (MMAP) task seeks an assignment $\mathbf{x}^*_M$ to variables $\mathbf{X}_M$ having maximum probability. This requires access to the marginal distribution over $\mathbf{X}_M$, which is obtained by summing out variables $\mathbf{X}_S$: Therefore, $\mathbf{x}^*_M = \text{argmax}_{\mathbf{x}_M} \sum_{\mathbf{x}_S} \prod_{\alpha \in F} \psi_\alpha(\mathbf{x}_\alpha)$.

AND/OR Search Spaces

Significant recent improvements in search for MMAP inference have been achieved by using AND/OR search spaces, which often capture problem structure far better than standard OR search methods (Dechter and Mateescu 2007; Marinescu, Dechter, and Ihler 2014). The AND/OR search space is defined relative to a *pseudo tree* of the primal graph, which captures problem decomposition.

Definition 1 (pseudo tree)**.** *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 in E' is a back-arc in $\mathcal{T}$ connecting a node in $\mathcal{T}$ to one of its ancestors. The arcs in E' may not all be included in E.*



(a) Primal graph (b) Pseudo tree

(c) AND/OR search graph

Figure 1: A simple graphical model.

For MMAP, we need to restrict the collection of pseudo trees to *valid* ones only. Specifically, given a graphical model $\mathcal{M}$ with primal graph G, a pseudo tree $\mathcal{T}$ of G is *valid for the MAP variables* $\mathbf{X}_M$ if $\mathcal{T}$ restricted to $\mathbf{X}_M$ forms a connected *start pseudo tree* $\mathcal{T}'$ having the same root as $\mathcal{T}$ (i.e., $\mathcal{T}'$ has the same root and is a connected subgraph of $\mathcal{T}$).

Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ with primal graph G and valid 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 *edge weights* extracted from the original functions $\mathbf{F}$ (for details see (Dechter and Mateescu 2007)). Identical sub-problems, identified by their *context* (the partial instantiation that separates the sub-problem from the rest of the problem graph), can be merged, yielding an *AND/OR search graph*. Merging all context-mergeable nodes yields the *context minimal AND/OR search graph*, denoted $C_\mathcal{T}$. The size of $C_\mathcal{T}$ is exponential in the induced width of G along a depth-first traversal of $\mathcal{T}$ (also known as the *constrained* induced width).

Definition 2 (solution subtree)**.** *A* solution subtree $\hat{x}_M$ *of $C_\mathcal{T}$ relative to the MAP variables $\mathbf{X}_M$ is a subtree of $C_\mathcal{T}$ restricted to $\mathbf{X}_M$ that: (1) contains the root of $C_\mathcal{T}$; (2) if an internal OR node $n \in C_\mathcal{T}$ is in $\hat{x}_M$, then n is labeled with a MAP variable and exactly one of its children is in $\hat{x}_M$; (3) if an internal AND node $n \in C_\mathcal{T}$ is in $\hat{x}_M$ then all its OR children which denote MAP variables are in $\hat{x}_M$.*

Each node n in $C_\mathcal{T}$ can be associated with a *value* $v(n)$ capturing the optimal marginal MAP value of the conditioned sub-problem rooted at n, while for a sum variable it is the likelihood of the partial assignment denoted by n. Clearly, $v(n)$ can be computed recursively based on the values of n's successors: the OR nodes labeled by MAP variables (resp. SUM variables) perform maximization (resp. summation), while the AND nodes perform multiplication.

Example 1. *Figure 1 shows a graphical model with 8 bivalued variables and 11 binary factors, where the MAP and sum variables are* $\mathbf{X}_M = \{A, B, C, D\}$ *and* $\mathbf{X}_S = \{E, F, G, H\}$*, respectively. Figure 1a is the primal graph while Figure 1b is a valid pseudo tree whose MAP variables form a start pseudo tree (dashed lines denote back-arcs). Figure 1c displays the context minimal AND/OR search graph based on the pseudo tree (the contexts are shown next to the pseudo tree nodes). A solution subtree corresponding to the MAP assignment (A=0,B=1,C=1,D=0) is shown in red.*

Exact Sequential Search Schemes for MMAP

The current best performing exact methods for solving MMAP are based on either *depth-first* or *best-first* sequential search over an AND/OR search space. Specifically, *depth-first AND/OR Branch and Bound* (AOBB) (Marinescu, Dechter, and Ihler 2014) explores in a *depth-first* manner the context minimal AND/OR search graph while keeping track of the value of the best solution found so far (a lower bound on the optimal MMAP value). It 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. *Best-First AND/OR Search* (AOBF) (Marinescu, Dechter, and Ihler 2015) is a variant of AO* (Nilsson 1980) that explores the context minimal AND/OR search 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 as well as considerably fewer conditional likelihood evaluations. *Recursive Best-First AND/OR Search* (RBFAOO) (Marinescu, Dechter, and Ihler 2015) extends Recursive Best-First Search (RBFS) (Korf 1993) to MMAP queries and uses a threshold controlling mechanism to drive the search in a depth-first like manner. The algorithm may operate in linear space, however, for efficiency, it may use a fixed size cache table to store some of the nodes (based on contexts). In practice, RBFAOO outperformed dramatically both AOBB and AOBF on a variety of benchmarks.

Weighted Mini-Bucket Heuristics

The effectiveness of the above search algorithms greatly depends on the quality of the upper bound heuristic function that guides the search. More specifically, they use a recently developed weighted mini-bucket (WMB) based heuristic (Liu and Ihler 2011) which can be pre-compiled along the reverse order of the pseudo tree (Kask and Dechter 2001; Marinescu, Dechter, and Ihler 2014). First, WMB improves the naïve mini-bucket bound (Dechter and Rish 2003) with Hölder's inequality. For a given variable X_k, the mini-buckets Q_{kr} associated with X_k are assigned a non-negative *weight* $w_{kr} \geq 0$, such that $\sum_r w_{kr} = 1$. Then, each mini-bucket r is eliminated using a powered sum operator, $(\sum_{X_k} f(X)^{1/w_{kr}})^{w_{kr}}$. The cost shifting scheme (or reparameterization) is performed across mini-buckets to match the marginal beliefs (or "moments") to further tighten the bound. The single-pass message passing algorithm yields a scheme denoted by WMB(i), where i is called the i-bound and controls the accuracy of the approximation.

Parallel AND/OR Search for MMAP

In this section, we present the first parallel AND/OR search algorithms for solving MMAP exactly in shared memory environments. We extend a previous parallel search approach called SPRBFAOO that was introduced recently for pure MAP queries (Kishimoto, Marinescu, and Botea 2015). The idea is to search the context minimal AND/OR graph in parallel using multiple threads and associate both MAP and SUM nodes with so-called virtual q-values which capture the estimated cost of the corresponding subproblems as well as the number of threads working on them. This ensures an effective and balanced exploration of the corresponding subspaces (Kaneko 2010). More importantly, since we parallelize the search below SUM nodes our approach is directly applicable to counting or sum-inference tasks as well.

Parallel Recursive Best-First Search

Algorithm 1 describes our parallel recursive best-first AND/OR search approach for MMAP which we denote hereafter by SPRBFAOO-MMAP for consistency. The algorithm initiates parallelism by spawning a number of threads that all start from the root node of the search space and run in parallel (lines 1–5). The threads share one cache table, which allows them to effectively reuse the results of each other. A cache table entry corresponds to a node n and contains the following information: a *q-value* $q(n)$ that is an upper bound on the optimal value of node n and is typically provided by the heuristic evaluation at the node (i.e., the WMB(i) value), a flag $n.solved$ indicating whether n is solved optimally, a *virtual q-value* $vq(n)$, a best known solution cost $\beta(n)$ for node n, the number of threads currently working on n, and a lock. Two threshold values $n.th$ and $n.thlb$ are maintained for each node n and they may differ from one thread to another. More specifically, $n.th$ indicates the next best upper bound to node n, whereas $n.thlb$ gives a lower bound on the best solution cost known for n so far and is only relevant to OR nodes labeled by MAP variables. When accessing a cache entry, threads lock it temporarily for other threads. Ctxt(n) identifies the context of n, which is further used to access the corresponding cache entry. The methods SaveCache() and ReadCache() store and, respectively, retrieve from cache the following values associated with the node context: $q(n)$, $n.solved$, $vq(n)$ and $\beta(n)$, respectively. Algorithms 2 and 3 show how the best child node and, respectively, an unsolved child node is selected for expansion.

Function RBFS(n) (lines 6-41 in Algorithm 1) describes the procedure invoked on each thread. When a thread examines a node n, it first increments in the cache the number of threads working on that node (line 7). Then it decreases $vq(n)$ by dividing it with $\zeta > 1$, and stores the new value in the cache (line 8). The virtual q-value $vq(n)$ is initially set to $q(n)$ and, as more threads work on solving n, $vq(n)$ decreases due to the repeated divisions by ζ. In effect, $vq(n)$ reflects both the estimated cost of node n (through its $q(n)$ component) and the number of threads working on n. Computing $vq(n)$ this way allows us to dynamically control the degree to which threads overlap when exploring the search space. When a given area of the search space is more promising than others (i.e., a node with a larger vq value), more than

Algorithm 1: SPRBFAOO-MMAP

Input: Graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, pseudo tree $\mathcal{T}$, heuristic $h(\cdot)$, $\mathbf{X}_M = \mathbf{X} \setminus \mathbf{X}_S$, N CPU cores

1 **Function** SPRBFAOO-MMAP():
2 **forall** *i from 1 to N* **do**
3 $root.th = 0$; $root.thlb = -\infty$
4 Launch RBFS($root$) on a separate thread
5 **return** optimal cost (as $root$'s q-value in cache)
6 **Function** RBFS(n):
7 IncrementNrThreadsInCache(Ctxt(n))
8 DecreaseVQInCache(Ctxt(n))
9 **if** $ch(n) = \emptyset$ **then**
10 $q = 1$; $solved = true$
11 SaveCache(*Ctxt(n), q, solved, q, q*)
12 DecrementNrThreadsInCache(Ctxt(n))
13 **return**
14 GenerateChildren(n)
15 **if** n *is an OR node* **then**
16 **while** *true* **do**
17 **if** n *is labeled by MAP variable* **then**
18 $(c_{best}, vq, vq_2, q, \beta) =$ BestChild(n)
19 $n.thlb = \max(n.thlb, \beta)$
20 **if** $vq < n.th \vee q \leq n.thlb \vee n.solved$ **then**
21 **break**
22 $c_{best}.th = \max(n.th, vq_2/\delta)/w_{(n,c_{best})}$
23 $c_{best}.thlb \leftarrow n.thlb/w_{(n,c_{best})}$
24 **else if** n *is labeled by SUM variable* **then**
25 $(q, vq, \beta) =$ Sum(n)
26 **if** $vq < n.th \vee n.solved$ **then break**
27 $c_{best} =$ UnsolvedChild(n); $c_{best}.th = 0$
28 RBFS(c_{best})
29 **else if** n *is an AND node* **then**
30 **while** *true* **do**
31 $(q, vq, \beta) =$ Prod(n)
32 $n.thlb = \max(n.thlb, \beta)$
33 **if** $vq < n.th \vee n.solved$ **then break**
34 $(c_{best}, q_{c_{best}}, vq_{c_{best}}) =$ UnsolvedChild(n)
35 $c_{best}.th = vq(c_{best}) \cdot (n.th/vq)$
36 $c_{best}.thlb = q(c_{best}) \cdot (n.thlb/q)$
37 RBFS(c_{best})
38 **if** $n.solved \vee$ *NrThreadsCache*$(Ctxt(n)) = 1$ **then**
39 $vq \leftarrow q$
40 DecrementNrThreadsInCache(Ctxt(n))
41 SaveCache(*Ctxt(n), q, n.solved, vq, bs*)

Algorithm 2: BESTCHILD(n)

Input: node n (OR node labeled by MAP variable)

1 $q = q_2 = vq = vq2 = \beta = -\infty$ and $n.solved =$ **false**
2 **forall** n*'s child* c_i **do**
3 **if** $Ctxt(c_i) \in Cache$ **then**
4 $(q_{c_i}, s_{c_i}, vq_{c_i}, \beta_{c_i}) =$ ReadCache($Ctxt(c_i)$)
5 **else**
6 $q_{c_i} = h(c_i)$; $s_{c_i} =$ **false**
7 $vq_{c_i} = h(c_i)$; $\beta_{c_i} = -\infty$
8 $q_{c_i} = w_{(n,c_i)} \cdot q(c_i)$; $vq_{c_i} = w_{(n,c_i)} \cdot vq(c_i)$
9 $\beta = \max(\beta, w_{(n,c_i)} \cdot \beta_{c_i})$
10 **if** $q_{c_i} > q \vee (q_{c_i} = q \wedge \neg n.solved)$ **then**
11 $n.solved = s_{c_i}$; $q = q_{c_i}$
12 **if** $vq_{c_i} > vq \wedge \neg s_{c_i}$ **then**
13 $vq_2 = vq$; $vq = vq_{c_i}$; $c_{best} = c_i$
14 **else if** $(vq_{c_i} > vq_2 \wedge \neg s_{c_i})$ **then** $vq_2 = vq_{c_i}$
15 **return** $(c_{best}, vq, vq_2, q, \beta)$

Algorithm 3: UNSOLVEDCHILD(n)

Input: node n

1 **forall** n*'s child* c_i **do**
2 **if** $Ctxt(c_i) \in Cache$ **then**
3 $(q_{c_i}, s_{c_i}, vq_{c_i}, \beta_{c_i}) =$ ReadCache($Ctxt(c_i)$)
4 **else**
5 $q_{c_i} = h(c_i)$; $s_{c_i} =$ **false**
6 $vq_{c_i} = h(c_i)$; $\beta_{c_i} = -\infty$
7 **if** $\neg s_{c_i}$ **then**
8 $vq_{best} = vq_{c_i}$; $q = q_{c_i}$
9 $c_{best} \leftarrow c_i$; $\beta \leftarrow \beta_{c_i}$
10 **return** $(c_{best}, vq_{best}, q, \beta)$

one thread are encouraged to work together within that area. On the other hand, when several areas are roughly equally promising, threads should diverge and work on different areas. Indeed, the tests on lines 20, 26 and 33 prevent a thread from working on a node n if $vq(n) < n.th$. A smaller $vq(n)$, which increases the likelihood that $vq(n) < n.th$, may reflect a less promising node (i.e., small q-value), or many threads working on n, or both. Therefore, the virtual q-values provide an automated and dynamic mechanism for tuning the number of threads working on solving a node n based on how promising that node is.

Following the expansion of the current node n, the node values vq, q and β are updated using the values of its children (lines 18, 25, 31). The thread backtracks to n's parent if at least one of the following conditions hold: $vq(n) < n.th$, discussed earlier; $q(n) \leq n.thlb$ i.e., a solution containing n cannot possibly beat the best known solution; or the node is solved. The solved flag is true iff the node value has been proven to be optimal. Notice that we prune with $n.thlb$ only at OR nodes labeled by MAP variables.

The best successor c_{best} of a node is selected as follows. At OR node n labeled by a MAP variable, c_{best} is the child with the largest vq among all children not solved yet (see method BESTCHILD). At OR nodes labeled by summation variables as well as at AND nodes, any unsolved child can be chosen (see method UNSOLVEDCHILD). Subsequently, the thresholds of node c_{best} are updated and the algorithm recursively processes c_{best}.

Notice that when updating the threshold $n.th$ we use the overestimation parameter $\delta > 1$ (line 22) to minimize the node re-expansion rate (Marinescu, Dechter, and Ihler 2015) as well as the vq- instead of the q-value to obtain the thread coordination mechanism presented earlier. In contrast, the threshold $n.thlb$ is updated in a consistent way with $\beta(n)$

and the current q-values of n and its children (lines 19, 23, 32 and 36, respectively).

When a thread backtracks to n's parent, if either n's solved flag is set or no other thread currently examines n, the thread sets $vq(n)$ to $q(n)$ (lines 38–39). Finally, the thread decrements in the cache the number of threads working on n (line 40) and saves in the cache the recalculated $vq(n)$, $q(n)$, $\beta(n)$, and the solved flag (line 41).

To discuss whether or not search can return an optimal solution tree *cost*, we extend to parallel search the notion of memory requirement linear in the search depth and the branching factor along the path, which is introduced in (Korf 1993; Kishimoto, Marinescu, and Botea 2019). An additional structure linear to the number of MAP variables is necessary to return an actual assignment.

Definition 3. *Given a search problem and an amount of memory available, we say that the* reasonable memory requirement *is satisfied if, for any path that could be explored in the search, the memory for each thread is sufficient to store all nodes along the current paths, together with all their siblings.*

If t is the number of threads, b is the maximum branching factor, and d is the maximum depth of a path explored in the problem at hand, we need at least $O(bdt)$ memory. Under the reasonable memory requirement, it follows that:

Theorem 1 (correctness)**.** *Algorithm SPRBFAOO-MMAP guided by an admissible heuristic returns optimal solutions.*

Proof. The optimal solution cost $v(n)$ is calculated by using the q-values rather than the vq-values in the cache table. It is therefore sufficient to prove that all the q-values in the cache table are always admissible.

Let T_k be the state of the cache table immediately after the k-th save is performed in the cache table. Let $q_k(n)$ be the value saved in T_k for n if that value exists in T_k or $h(n)$ if n is not preserved in T_k. We prove this by induction on k in an analogous way to (Akagi, Kishimoto, and Fukunaga 2010; Kishimoto, Marinescu, and Botea 2015).

T_0 has only admissible values since no result is stored.

Assume that the above property holds for T_k. Then, $q_k(n) \geq v(n)$ holds for any node n. The q-value $q_{k+1}(n)$, saved in T_{k+1}, is then calculated as: (1) If n is a terminal leaf node in the SUM tree then $q_{k+1}(n) = v(n)$ holds; (2) If n is an internal OR node in the SUM tree, then $q_{k+1}(n) = \sum_i q_k(c_i) \geq \sum_i v(c_i) = v(n)$, where c_i is n's child; (3) If n is an internal AND node in the SUM tree, then $q_{k+1}(n) = \prod_i q_k(c_i) \geq \prod_i v(c_i) = v(n)$, where c_i is n's child; (4) If n is an internal OR node in the MAP tree, then $q_{k+1}(n) = w(n, c_{best}) \cdot q_k(c_{best}) = \max_i(w(n, c_i) \cdot q_k(c_i))$ holds where c_i is n's child c_{best} is the best child of n. Additionally, because $q_k(c_i) \geq v(c_i)$, $q_{k+1}(n) \geq \max_i(w(n, c_i) \cdot v(c_i)) = v(n)$ holds; (5) If n is an internal AND node in the MAP tree $q_{k+1}(n) = \prod_i q_k(c_i) \geq \prod_i v(c_i) = v(n)$ holds, where c_i is n's child. Hence, T_{k+1} contains only admissible values. □

Although SPRBFAOO-MMAP is correct, its completeness remains open. We conjecture that SPRBFAOO-MMAP is also complete, and leave the analysis as future work.

A Complete Parallel Search Scheme

We next describe a scheme called SPRBFAOO2-MMAP that is both correct and complete for MMAP. The idea is to use one *master* thread to manage the search over the MAP variables, and a set of *slave* threads that share one cache table and are dedicated to solving only the conditioned summation subproblems. More specifically, when the master reaches a conditioned summation subproblem, it signals the slaves to start searching the subproblem in parallel. Then, the master waits for the slaves to finish and subsequently propagates back the results to update the q-values of the nodes labeled by MAP variables. These steps are repeated until an optimal solution is found.

All slaves start searching at the same root node of the conditioned summation subproblem. When a slave expands a node it reorders the children generated randomly to encourage diversification and focus the search on regions that have not been examined by other threads. The shared cache table is used to alleviate the duplicated search effort performed by the threads. The algorithm can easily be implemented on top of SPRBFAOO-MMAP, but we omit the details for space reasons.

The following lemma which is a modification to Lemma 7 in (Kishimoto, Marinescu, and Botea 2019) holds. The proof of Theorem 3 below requires the lemma.

Lemma 2. *Assume that SPRBFAOO2-MMAP examines a node n with $th(n) = b$, where $th(n)$ is a threshold at node n. Then, the master/slave of SPRBFAOO2-MMAP backtracks to n in finite time with either $b > q(n)$ and/or n is marked as solved/deadend.*

Theorem 3 (correctness and completeness)**.** *Algorithm SPRBFAOO2-MMAP guided by an admissible heuristic is correct and complete.*

Proof sketch. The correctness is proven in a similar way to Theorem 1. Therefore, we give a proof sketch to the completeness. The master thread of SPRBFAOO2-MMAP conducts a sequential recursive best-first AND/OR search (RBFAOO) over the MAP variables. From Theorem 3.2 in (Kishimoto and Marinescu 2014) we know that RBFAOO search for the pure MAP task is complete.

We prove that SPRBFAOO2-MMAP eventually expands a new SUM node in finite time. We assume that no new unsolved SUM node is expanded after a certain time has passed. Let $lp(n)$ be the length of the longest path to node n from the root. Let S be a set of unsolved SUM nodes which have been expanded before but hold $th(n) \leq q(n)$ for each node $n \in S$ an infinite number of times.

Let $n_{max} = \text{argmax}_{n \in S} lp(n)$. Since n_{max}'s children are visited at most a finite number of times, there exists an constant value k (stemming from children's q-values) which holds $q(n_{max}) \geq k$. Let m be a node where SPRBFAOO2-MMAP starts descending during no new node expansion. Because of the reasonable memory requirement, m is always preserved in memory. In addition, Lemma 2 requires $th(m)$ to continue to be decreased. This leads to $th(n_{max}) \leq k$ in a finite number of visits to n_{max}. Hence, n_{max} needs to be expanded with $th(n_{max}) \leq k$. Lemma 2 indicates that $k >$

instance	AOBB*	RBFAOO	SPAOBB	SPRBFAOO	SPRBFAOO2	SPAOBB2
p01_2	0.02	0.02	**0.00** (5)	0.01 (4)	0.01 (2)	0.01 (2)
p01_3	0.09	0.08	**0.02** (6)	**0.02** (4)	**0.02** (4)	**0.02** (5)
p01_4	0.31	0.32	0.06 (6)	**0.05** (6)	0.07 (5)	0.07 (4)
p01_5	1.03	1.02	0.17 (6)	**0.16** (6)	0.23 (5)	0.22 (5)
p01_6	3.34	3.35	0.47 (7)	**0.45** (7)	0.67 (5)	0.7 (5)
p01_7	10.75	10.7	1.48 (7)	**1.43** (8)	2.07 (5)	2.04 (5)
p01_8	34.99	35.38	**4.28** (8)	4.54 (8)	6.58 (5)	6.58 (5)
p01_9	191.95	197.85	34.26 (6)	**34.00** (6)	39.99 (5)	40.54 (5)
p01_10	811.13	796.84	124.65 (7)	**122.63** (6)	164.72 (5)	161.78 (5)
p01_11	oot	oot	**502.45** (-)	502.64 (-)	1123.21 (-)	1105.94 (-)
p01_12	oot	oot	1965.02 (-)	**1935.61** (-)	oot	oot
overhead ρ			6.77%	6.90%	12.51%	12.37%

Table 1: CPU time (sec) for solving the `planning` instances. Time limit 1 hour, 12 CPU cores. "oot" is "out of time".

domain	AOBB*	SPAOBB			RBFAOO	SPRBFAOO		
	P = 20%							
	#solved	#solved	speedup	overhead ρ	#solved	#solved	speedup	overhead ρ
pedigree	9	10	(3/7/11)	15.21%	9	10	(5/7/10)	9.93%
pdb	13	19	(4/8/12)	21.95%	13	19	(4/7/12)	25.48%
	P = 50%							
pedigree	22	25	(2/5/12)	51.14%	24	25	(2/6/17)	20.76%
pdb	71	90	(1/8/63)	38.89%	69	90	(1/6/10)	32.21%
	P = 80%							
pedigree	71	77	(1/5/16)	68.97%	71	77	(1/5/8)	33.76%
pdb	171	181	(1/8/437)	168.02%	169	182	(1/5/12)	73.57%

Table 2: Number of solved instances (#solved), speedup (min/avg/max) and average search overhead (ρ) for the `pedigree` and `pdb` domains. Time limit 1 hour, 12 CPU cores.

$q(n_{max})$ holds and/or n_{max} is marked as solved/deadend in finite time, which is a contradiction to the assumption. □

We note that the `daoopt` algorithm introduced recently for solving pure MAP queries in parallel (Otten and Dechter 2012) uses a similar master-slaves architecture, but is fundamentally different from SPRBFAOO2-MMAP because: 1) it is a distributed memory scheme that does not allow any communication between the slave processes, 2) caching information and bounds are not exchanged between the slave processes, and 3) it assumes a particular distributed grid computational environment (called condor). In contrast, SPRBFAOO2-MMAP is a shared-memory algorithm suitable for current multi-core architectures where cache table entries (and therefore bounds) are shared between the slave threads. Therefore, a direct comparison with `daoopt` is outside the scope of this paper. Extending our approach to distributed memory environments is one direction of future work that we are currently investigating.

Parallel Depth-First Branch-and-Bound Search

Algorithm SPRBFAOO-MMAP uses an overestimation technique (via parameter $\delta > 1$ in line 22) to encourage the search to keep examining the current subtree so that the number of node re-expansions is minimized. Hence, setting δ to a large value effectively turns the search into a depth-first branch and bound search. Therefore, two parallel depth-first search schemes called SPAOBB-MMAP and SPAOBB2-MMAP can be obtained from algorithms SPRBFAOO-MMAP and SPRBFAOO2-MMAP, respectively, by using $\delta = \infty$, which effectively sets the threshold $n.th$ to 0.

Experiments

We evaluate empirically the performance of the proposed parallel search schemes on standard benchmark problems for MMAP. All algorithms were implemented in C++ (64-bit) and the experiments were run on a 2.0GHz IBM Cloud virtual machine with 12 cores and 64GB of RAM.

Our benchmark set includes three domains from genetic linkage analysis (`pedigree`), computational protein design (`pdb`) and probabilistic conformant planning (`planning`), respectively. Since the original `pedigree` and `pdb` problems are pure MAP tasks (Elidan, Globerson, and Heinemann 2012), we generated 5 synthetic MMAP instances for each pure MAP instance by randomly selecting $P\%$ of the variables to act as MAP variables, where we chose $P \in \{20, 50, 80\}$. Intuitively, the three P values correspond to hard ($P = 20$), medium ($P = 50$) and easy ($P = 80$) conditioned summation subproblem. Therefore, we created 110 pedigree and 200 protein instances for each value of P (a total of 930 instances). The `planning` instances are derived from probabilistic conformant planning with a finite time horizon (Lee, Marinescu, and Dechter 2016). Specifically, we considered instances corresponding to the probabilistic version of the classical slippery gripper planning problem with different time horizons (Kushmerick, Hanks, and Weld

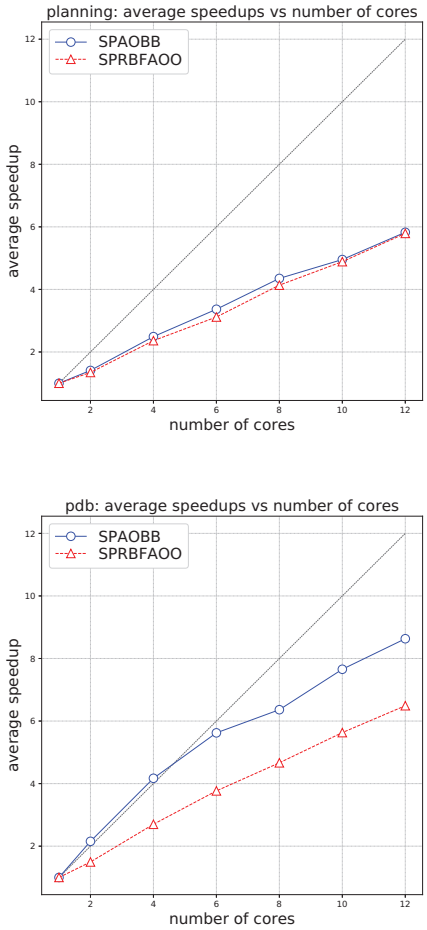


Figure 2: Average speedups for the first 9 planning instances in Table 1 (left) and 50 difficult pdb instances (right) with varying numbers of cores. Time limit 1 hour.

1995). Computing an optimal policy to the planning problem is equivalent to solving exactly a MMAP query over the dynamic Bayesian network (DBN) encoding of the original planning problem (Lee, Marinescu, and Dechter 2016).

We compare our parallel algorithms SPAOBB-MMAP and SPRBFAOO-MMAP[1] against their sequential search counterparts, AOBB* and RBFAOO. Algorithm AOBB* is obtained from RBFAOO by setting $\delta = \infty$ and therefore its behavior is different from the original AOBB (Marinescu, Dechter, and Ihler 2014). In addition, we ran the two complete variants, namely SPAOBB2 and SPRBFAOO2. The parallel search algorithms ran with 12 threads each and all competing algorithms used the same heuristic function WMB(i) with the i-bound set to 10 for pedigree and planning domains, and to 2 for the pdb domain, respectively. The time limit was set to 1 hour and all algorithms used a 10GB cache table to record partial search results. When the cache table is full, the algorithms use the Small-TreeGC strategy (Nagai 1999) to discard $R\%$ entries with small subtrees. As in (Akagi, Kishimoto, and Fukunaga 2010; Kishimoto and Marinescu 2014), we set R to 30. For numerical stability, our implementation solves MMAP as a minimization problem in log space as suggested in (Marinescu et al. 2018) and therefore we used parameters $\delta = 1$ and $\zeta = 0.01$, respectively.

Table 1 shows the CPU time in seconds obtained on the planning instances. The number displayed in parenthesis next to the CPU time denotes the speedup with respect to the corresponding sequential search algorithm. We also compute the parallel search overhead (Marsland and Popowich 1975), denoted by ρ, and average it over those instances that were solved by both sequential and parallel search. Specifically, ρ is defined as $\rho = 100 \cdot \left(\frac{\#par}{\#seq} - 1\right)$, where $\#par$ and $\#seq$ denote the number of nodes expanded by the parallel and the sequential search algorithm. Intuitively, ρ can explain the overhead incurred by parallel search compared with sequential search due to duplicated search effort (i.e., expanding the same node multiple times by different threads). record

Both SPAOBB and SPRBFAOO improve considerably over their sequential counterparts, in some cases achieving up to an 8-fold speedup. Furthermore, both parallel algorithms solve two additional problem instances (p01_11 and p01_12) within the 1-hour limit. Their average search overhead is less than 7% which explains in part the effectiveness of parallel search on this domain.

SPAOBB2 and SPRBFAOO2 are competitive only on the easier problem instances. However, on the more difficult ones they suffer due to a much higher search overhead, lock overhead as well as poor load balancing due to blocking the main thread during the parallel conditional likelihood computations. This is why we did not run these two algorithms on the pedigree and pdb domains.

Table 2 summarizes the results in the pedigree and pdb domains. We show the number of problem instances solved (#solved), the speedup and the average search overhead (ρ) with respect to sequential search. The second column gives the domain size k, the average constrained induced width w_c^* and the average induced width of the conditioned summation subproblems w_s^*, respectively. The table is divided into three horizontal blocks each corresponding to a MAP ratio value P. The numbers shown in the speedup columns indicate the minimum, average and maximum speedups obtained for that particular benchmark. The parallel search algorithms SPAOBB and SPRBFAOO consistently solve more problem instances compared with the corresponding sequential search ones. They are also considerably faster, in some cases achieving on average up to 8-fold speedups (e.g., pdb with $P = 50\%$). We also see that as the ratio of MAP variables increases, the search overhead increases as well, and this may explain the relatively mild slow down observed on the corresponding problem instances. We notice several outliers (i.e., superlinear speedups), especially in the case of SPAOBB, but these are most likely due to the actual information (in particular the lower bounds) written in the cache table by the different threads which in turn allows for more effective pruning of the search space. For completeness, in Figure 3 we

[1]We drop the -MMAP suffix from all names, for simplicity.

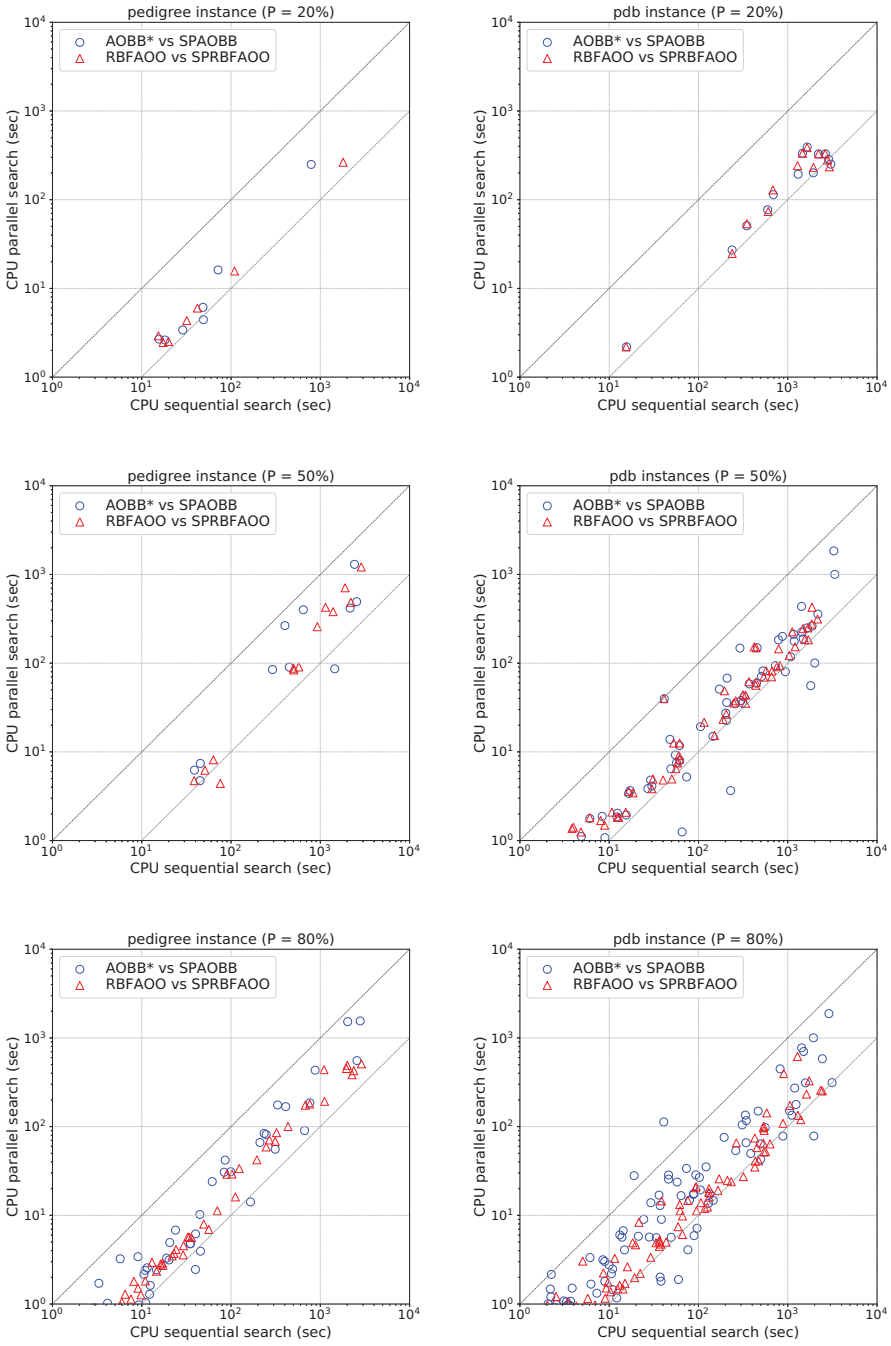


Figure 3: CPU time (sec) for sequential vs parallel search on `pedigree` (left) and `pdb` (right) instances. Time limit 1 hour.

also plot the running times (on a log scale) of the sequential versus parallel search algorithms, clearly showing the benefit of parallel search.

In Figure 2 we plot the average speedup obtained by SPAOBB and SPRBFAOO, on the subset of 9 `planning` instances (p01_2 to p01_10) as well as on 50 difficult `pdb` instances for increasing numbers of CPU cores. Note that all the instances considered were solved by both sequential and parallel search algorithms. The speedups obtained are consistent with the behavior observed in the 12-core setup.

Conclusion

In this paper, we presented a new shared-memory parallel recursive best-first AND/OR search scheme for solving MMAP exactly. The algorithm uses the so-called virtual q-values to

enable the threads to work on promising regions of the search space with effective reuse of the search effort performed by other threads. Subsequently, we proposed a complete parallel scheme that conducts the search over the MAP variables in a single master thread and solves the conditioned summation subproblems in parallel using a set of slave threads. We also extended the proposed parallel best-first search algorithms into parallel depth-first search schemes. Our experiments demonstrated that the new parallel search algorithms improved considerably over current state-of-the-art sequential AND/OR search approaches, in many cases leading to considerable speed-ups (up to 8-fold using 12 threads) especially on hard problem instances.

References

Akagi, Y.; Kishimoto, A.; and Fukunaga, A. 2010. On transposition tables for single-agent search and planning: Summary of results. In *Symposium on Combinatorial Search (SoCS)*, 1–8.

Campbell, M.; Hoane, A.; and Hsu, F. 2002. Deep Blue. *Artificial Intelligence* 134(1–2):57–83.

Chrabakh, W., and Wolski, R. 2003. Gradsat: A parallel SAT solver for the Grid. Technical report, University of California at Santa Barbara.

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

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

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

Elidan, G.; Globerson, A.; and Heinemann, U. 2012. PASCAL 2011 probabilistic inference challenge. http://www.cs.huji.ac.il/project/PASCAL/.

Howard, R., and Matheson, J. 2005. Influence diagrams. *Decision Analysis* 2(3):127–143.

Kaneko, T. 2010. Parallel depth first proof number search. In *AAAI Conference on Artificial Intelligence*, 95–100.

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

Kishimoto, A., and Marinescu, R. 2014. Recursive best-first AND/OR search for optimization in graphical models. In *Uncertainty in Artificial Intelligence (UAI)*, 400–409.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence* 195:222–248.

Kishimoto, A.; Marinescu, R.; and Botea, A. 2015. Parallel recursive best-first and/or search for exact map inference in graphical models. In *Advances in Neural Processing Information Systems (NIPS)*, 928–936.

Kishimoto, A.; Marinescu, R.; and Botea, A. 2019. Depth-first memory-limited AND/OR search and unsolvability in cyclic search spaces. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1280–1288.

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

Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76(1-2):239–286.

Lee, J.; Marinescu, R.; and Dechter, R. 2016. Applying search based probabilistic inference algorithms to probabilistic conformant planning: Preliminary results. In *Proceedings of International Symposium on Artificial Intelligence and Mathematics (ISAIM)*.

Liu, Q., and Ihler, A. 2011. Bounding the partition function using Hölder's inequality. In *Int'l Conference on Machine Learning (ICML)*, 849–856.

Liu, Q., and Ihler, A. 2013. Variational algorithms for marginal MAP. *Journal of Machine Learning Research* 14:3165–3200.

Marinescu, R.; Lee, J.; Dechter, R.; and Ihler, A. 2018. AND/OR search for marginal MAP. *Journal of Artificial Intelligence Research* 63(1):875–921.

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

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

Marsland, T., and Popowich, F. 1975. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 7(4):442–452.

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

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

Otten, L., and Dechter, R. 2012. A case study in complexity estimation: Towards parallel branch-and-bound over graphical models. In *Uncertainty in Artificial Intelligence (UAI)*, 665–674.

Park, J. 2002. MAP complexity results and approximation methods. In *Uncertainty in Artificial Intelligence (UAI)*, 388–396.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529:484–489.