Propagating Distributions Up Directed Acyclic Graphs

Eric B. Baum and Warren D. Smith
NEC Research Institute 4 Independence Way Princeton NJ 08540
Email: eric and wds @research.NJ.NEC.COM

July 9, 1998

Abstract

In a previous paper, the authors considered game trees as graphical models. Adopting an evaluation function that returned a probability distribution over values likely to be taken at a given position, they described how to build a model of uncertainty and use it for utility directed growth of the search tree and for deciding on a move after search was completed. In some games, such as chess or Othello, the same position can occur more than once, collapsing the game tree to a Directed Acyclic Graph (DAG). This induces correlations among the distributions at sibling nodes. The present paper discusses some issues that arise in extending our algorithms to a DAG. We give a simply described algorithm for correctly propagating distributions up a game DAG, taking account of dependencies induced by the DAG structure. This algorithm is exponential time in the worst case. We prove that it is #P complete to correctly propagate distributions up a game DAG. We suggest how our exact propagation algorithm can yield a fast but inexact heuristic.

1 Introduction

Recently there has been considerable interest in the use of directed graphical models for inference and modeling in problems involving uncertainty, cf. [Jensen 1996]. In playing a game, one typically searches a subtree of the game tree in order to reduce one's uncertainty about which move to make. The authors have recently explored the use of a probabilistic model in this procedure[Baum and Smith 1997]. Instead of using an evaluation function returning a scalar value as in standard game programs, we used an evaluation function that returns a probability distribution over the possible values of a position. Assuming independence of the distributions at the leaves of the search subtree, we built a model of our uncertainty. We described how to use this model for utility directed growth of the search tree, and for the choice of move after the tree is grown.

Our algorithm is an example of use of a directed graphical model, but is simpler in (at least) two respects than the general case. First, the graphs we explored had no loops, and second, in a general graphical model the nodes take values from a distribution that could depend in an arbitrary way on the distribution at connected nodes. In game trees, there is a natural notion of "causality": the leaves have values (or probability distributions of values), and the distributions of values taken by child nodes determine the distributions of their parents through "negamax" (or equivalently "min-max") as will be discussed below. Because of these simplifications, we were able to describe near linear time algorithms. In this paper we discuss relaxing the first of these simplifications, and thus the extension of our methods to more general Directed Acyclic Graphs (DAG's).

In games such as chess or Othello, the same position can occur more than once in a game tree, which thus collapses to a DAG. Competitive programs for such games generally use a hash table to efficiently spot

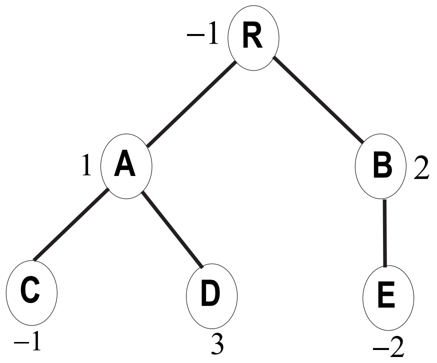

Figure 1: The figure shows a search tree rooted at position R. From R one can move to positions A and B. From A one can move to positions C and D. Leaves C,D, and E are respectively assigned values $-1, 3$, and -2. From these, the values associated to positions A, B, and R are computed by the negamax algorithm described in the text.

recurrences of previously evaluated positions, and then one need neither valuate nor store that node twice. This idea is equally valid in our formalism[1].

The new feature for our methods when we allow DAG's comes from the correlation between distributions at different nodes. In a DAG there may be nodes with common ancestors. In our previous work we assumed that the distributions at leaves were independent, and this implied that the distributions at all sibling nodes were independent. The present paper assumes that the distributions at the sources of the DAG are independent, and then gives an algorithm which propagates distributions up a game DAG taking correct account of all the dependencies then induced by the DAG structure. Although conceptually simple, this algorithm is, unfortunately, exponential time in the worst case. We also show that it is #P complete to correctly propagate distributions up a game DAG, so that there is no algorithm for efficiently propagating distributions up a game DAG unless P=NP. The intractability of propagation of distributions on general Bayes nets was previously known[Cooper 1990]. Our result is stronger in showing that propagation is intractable even when the dependence of the value of a node on that of its neighbors is restricted to negamax. We suggest an approach by which our exact (but slow in the worst case) propagation algorithm can yield fast but inexact heuristics.

§2 reviews the handling of distributions on search trees. §3 gives our new results about DAGS. §4 suggests a plausible approach to acceptably fast, but inexact propagation of distributions on game DAG's, and discusses how standard distribution propagation algorithms would fare in the game application.

2 The Model

In this section we review search trees, the introduction of distribution valued evaluation functions, and the propagation of distributions up trees.

In playing a game, one typically grows a search tree (figure 1) by looking ahead. The present position is R, or root, and one has expanded a portion of the game tree looking ahead (down). If the exact value of each leaf position were known, and assuming that both players knew those values, the value at each of the other nodes would then be determined by the negamax algorithm[2]. This determines the value of a node to be

[1] Some subtleties in the use of hash tables are mentioned in [Baum and Smith 1997]. Note in particular that our algorithm iteratively expands the most utilitarian leaves. The utility of expanding a leaf depends on how knowledge gained from appending successor positions to the search tree may impact move choice and later expansion decisions. Note that in a DAG the "influence function" at a node is the sum of the influence functions at the tree nodes it represents, so that one accounts for the utilities arising from different paths to the root[Baum and Smith 1997].

[2] Incidentally, we speak of the values of the leaf positions as "causing" the values of the positions at the internal nodes because the values of the internal nodes are in fact *defined* from the values of the source positions by the negamax algorithm. For example, a game theoretic won position is, by definition, one from which one has a winning move, i.e. a move that takes one to a position from which one's opponent's moves all lose for him.

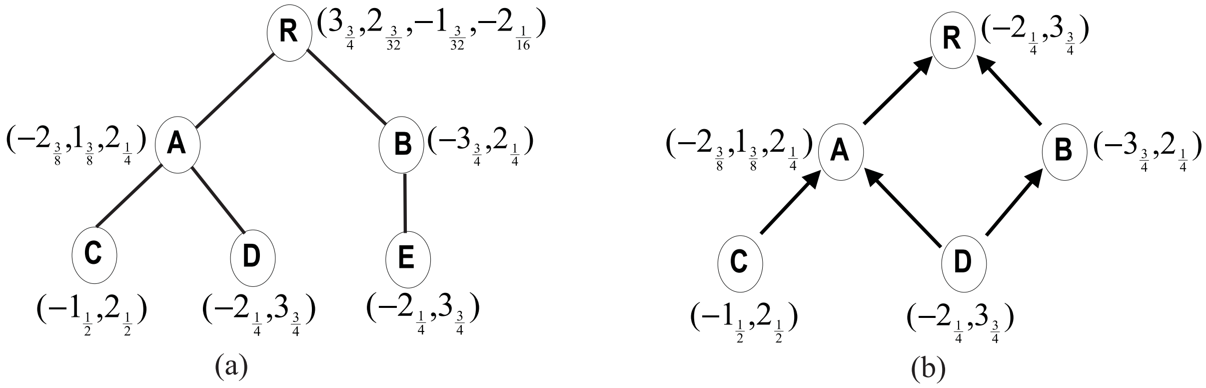

Figure 2: Figure (a) shows a search tree rooted at position R. From R one can move to positions A and B. From A one can move to positions C and D. Looking ahead from position R, we have assigned probability distributions to the leaves C, D, and E using an evaluation function. The distributions associated to the nodes are given as tuples of values, with subscripts encoding their respective probabilities. For example, node C takes value -1 with probability $1/2$ and value 2 with probability $1/2$. We have then calculated distributions at the nodes A, B, and R using the nega-max rule, described in the text, assuming that the distributions at the leaves are independent. For example, the joint probability that C has value -1 and D has value 3 is $3/8$. In this event, A will have value equal the maximum of the negatives of the values of its children, or $\max(1, -3)=1$. Now it turns out that the board position E is identical to the board position D, although these positions were reached by different sequences of moves from R. This is represented with the DAG of figure (b) where nodes D and E have been merged. Now the distributions at A and B are no longer independent, since they have a common parent, which leads to a different distribution for R. **Nota Bene:** It is customary when talking about a game tree to refer to "children" as being positions reached by moving from "parents". However, in the DAG, the arrows flow the other way, following the direction of *causality*: the value of the position at node A e.g. is determined by the values of the positions C and D. In the text we have used the language natural to the DAG representation, wherein the successor positions in the game tree are called the "parents", so we speak of D as the parent of A, but alternatively speak of A as a successor of D.

the maximum of the negative of the values of its successor positions. This negamax algorithm is isomorphic to the alternative max-min algorithm. Both simply assume that what is good for a player is bad for his opponent, and recursively value the nodes on the assumption the player makes his optimal choice according to the valuation.

Usually, one does not have the computational resources to search to terminal positions of the game, and this introduces uncertainty into the values of the leaves. Computer game programs adopt some form of evaluation function that estimates the expected values of leaf positions. For example, the evaluation function might be a neural net trained to predict game outcome. Standard game playing algorithms do not handle the uncertainty in a principled way, but simply use these estimates as if they were exact values for the leaves, and propagate them using negamax. We discussed in [Baum and Smith 1997] how this leads to errors. Instead, we proposed to adopt an evaluation function that associates to each leaf a probability distribution that estimates the probability a position takes will acquire various values[3]. See figure 2. The distribution associated to a given source typically depends on features of the position– e.g. in chess it might depend on the pawn structure and the material balance. This evaluation function is typically prepared by training from game data.

[3]These values are not game theoretic win or loss, but rather our expected payoff if we play from the position. For a discussion, see [Baum and Smith 1997].

Our evaluation function returns a distribution written as a weighted sum over point masses[4]

$$\rho^{(\eta)}(x) = \sum_i p_i^{(\eta)} \delta(x - x_i^{(\eta)}). \tag{1}$$

Here $\rho^{(\eta)}(x)$ is the probability distribution giving the probability node η has value x. p_i^{η} is thus the probability that node η has value x_i. δ is the Dirac delta function.

We assume that the distributions at the leaves (also called sources) are probabilistically independent. Note that this does not imply the *means* of the distributions are similar or dissimilar, any more than the means of sources for any other Bayes net. In the stereotypical causal net(c.f. [Jensen 1996]p10) a source for "earthquake occurred" and "burglary occurred" are deemed independent (absent evidence regarding the value of their descendants) yet the mean value of each source is low: earthquakes and burglaries are rare events. Likewise when we look ahead from position R to estimate the values at positions C and D, our estimate may well be that these similar positions have similar expected values. We are assuming that our *uncertainties* in the values of these positions are uncorrelated. We are assuming (as is common in game playing algorithms) that we have no additional information about the value of nodes internal to the search tree.

Now, as shown in figure 2b, the search tree often collapses to a search DAG, because of transpositions: i.e. different nodes of the tree may correspond to the same board positions.

We assume that the value of each source is drawn from its associated probability distribution. We further assume (effectively) that a player later reaching any position internal to the search DAG (i.e. below the root) would then be able to search below the leaves, which search would give more information about the value of the leaves. Our *model* is that the player would then know the exact value of the leaves (chosen from the associated distribution) and would move accordingly. This induces a probability distribution at all the internal nodes: the probability that internal node X has value x is the joint probability that the sources are in some configuration y yielding x by negamax propagation. See figure 2. For further motivation and discussion of this approach see [Baum and Smith 1997]. For a discussion of techniques for training such distribution based evaluation functions, and experimental results, see [Smith et al 1997].

It is equivalent to look at this in the ensemble picture[Baum and Smith 1997], see figure 3. Instead of one DAG with a probability distribution at each node, there is equivalently an ensemble of DAG's, each with a scalar value at each leaf. Each DAG represents one configuration, i.e. assignment of values to the leaves. Each DAG has values assigned to all internal nodes as the negamax of their successors (parents in the DAG). The DAG's each have a probabilistic weight, given by the product of the weights of the leaves. The distribution induced on an internal node as in the preceeding paragraph is the probability that if you choose a DAG from the ensemble, it assigns a given value to the node. Unfortunately the number of DAG's in the ensemble grows exponentially with the number of leaves, so the ensemble picture is not very useful for computation.

Note that the distribution at R depends importantly on recognizing the DAG structure. If we simply unwrapped the DAG of figure 2b into the tree of figure 2a (so that node D was treated as two different leaves with identical distributions) we would treat the distributions at A and B as independent. This leads to a distribution at node R different from the correct value. Since the Bayesian algorithm proposed in [Baum and Smith 1997] uses such distributions in decisions about where to search and which move to make, these differences can importantly affect play. In this case, for example, treating the graph as a tree leads to a dependence of the distribution at R on the distribution at C, so that expanding C would be found to have some utility, whereas we will see below that R's distribution depends only on D when correctly calculated in the DAG.

Recall the efficient algorithm for propagating (independent) distributions on a tree[Palay 1985],[Baum and Smith 1997]. One first defines two different types of cumulative distribution functions (CDF's):

$$\text{Falling CDF}: \quad \overline{c}^{(\eta)}(x) = \sum_{i,\, x_i^{(\eta)} \geq x} p_i^{(\eta)} = \int_x^{\infty} \rho^{(\eta)}(u)\mathrm{d}u = \mathrm{Prob}\left(\mathrm{value}(\eta) \geq x\right) \tag{2}$$

[4]One could imagine a more general form for such distributions, but assuming this form is essential to our algorithms.

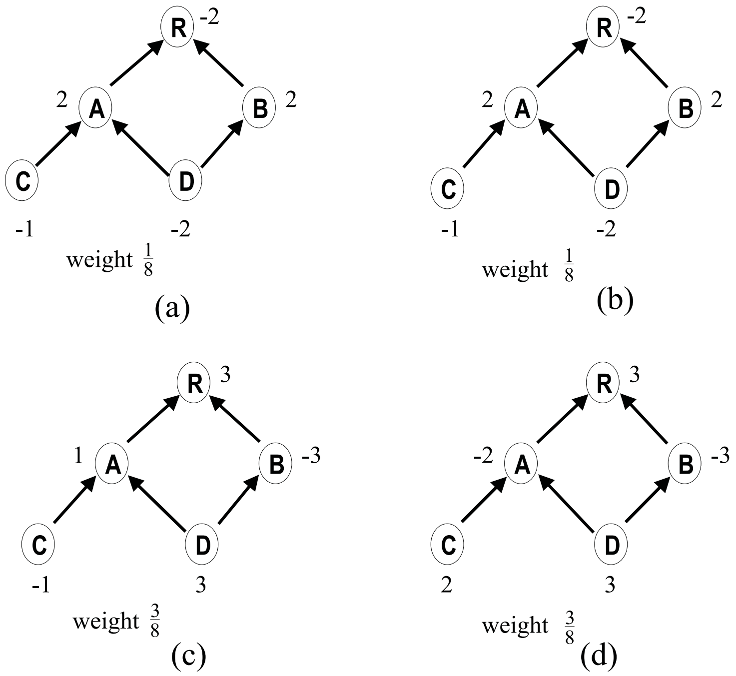

Figure 3: The DAG of figure 1b is equivalent to an ensemble of 4 DAG's, with scalar values associated to each node, and a weight associated to each DAG. The values at the leaves C and D are drawn from the distribution associated with nodes A and B. The weight of the DAG is the product of the probabilities of the values of C and D. The values assigned to the nodes A, B, and R in each DAG in the ensemble are given by the max of the negatives of the values of their children. Choose a DAG from the ensemble of figure 2 according to its weight. The distributions assigned to nodes A, B, and R in figure 1b gives the probability this DAG assigns a given value to these nodes.

$$\text{Rising CDF}: \quad \underline{c}^{(\eta)}(x) = \sum_{i,\, x_i^{(\eta)} \le x} p_i^{(\eta)} = \int_{-\infty}^{x} \rho^{(\eta)}(u)\mathrm{d}u = \mathrm{Prob}\left(\mathrm{value}(\eta) \le x\right). \qquad (3)$$

Because our ρ is a sum of point masses, these CDF's are "staircase" functions. Evidently, there is a bijection between staircase CDF's and distributions of the form of equation 1. Now one may efficiently calculate the distribution at a position from the distributions at its successors using the relation[Palay 1985]:

$$\underline{c}^{(\mathrm{position})}(-x) = \prod_{i=1}^{b} \overline{c}^{(\mathrm{successor}_i)}(x) \qquad (4)$$

This relationship holds because, under the definition of negamax, a position's value is less than $-x$ if and only if all its successors' values are above x, and because on a tree, the distributions at the successors are independent. This equation leads[5] to an $O(N \log b)$ time algorithm for computing a position's distribution, given its successors', where N is the total number of point masses in all the successors' distributions together, and b is the number of successors. From this, one gets an $O(N_T d_L \overline{\log b})$ time algorithm for computing the distributions at *all* nodes in a search tree, where N_T is the total number of spikes in all leaf distributions, d_L is the average depth of the leaves, and $\overline{\log b}$ is the log of the geometric mean branching factor.

3 Results on DAGS

The last section described how to valuate the nodes of a tree. We desire now to calculate the distributions at each node in the DAG, where now the distributions at siblings can no longer be assumed independent. First note

Lemma 1. *The CDF at any node in a DAG is a multilinear function of the CDF's at the leaves.*

Proof. This is evident in the ensemble viewpoint of figure 3. The probability distribution at any node is given by a sum over the different configurations of its leaf descendants, with configurations weighted by their probabilities. But each term in this sum is a multilinear function of the leaf probabilities, since we have assumed the leaf distributions are independent. QED.

Now we have

Theorem 2. *To propagate CDF's up a DAG, first use equation 4 to naively yield the CDF at each node as a polynomial function of the CDF's at the leaves. Then expand that polynomial into a sum of monomials, and wherever in this formula any leaf CDF $c_i(x)$ appears raised to a power greater than one, simply replace the power by 1.*

Example. See figure 4.

Proof. By the lemma above, the CDF $c_i(x)$ at node i is a sum of 2^L monomials (where there are L leaves), i.e. it can be written as $\sum_{S \subseteq \{1,\ldots,L\}} a_S \prod_{j \in S} c_j(-1^d x)$ where d is the height of node i above leaf j. (Note, the players alternate turns, so we never have two paths from a leaf to the same position of differing parities.[6]) We must only determine the coefficients a_S. There are 2^L such coefficients, which can thus be determined by interpolating the function at 2^L points. At the 2^L assignments where we set the $c_j(x)$ to be 1 or 0, we have $c_j^n = c_j$ and thus the formula given in the theorem is correct by inspection. QED.

[5]There are two ways to achieve $O(N \log b)$ runtime. First, we could do $\log_2 b$ passes, at each pass combining pairs of child distributions and thus reducing the number of such distributions by a factor of 2 so that after the final pass only one would remain. Each such pair-combining step (which involves computer code similar to "ordered list merge") would run in time linear in the number of point masses in the two pairs. The second method would combine all b child distributions at once, using a so-called "heap" data structure to facilitate finding the next point mass to process in $O(\log b)$ time rather than the naive $O(b)$.

[6]A position is defined both by board position and by the player to move: two positions with identical board position but different players to move are different. In games where players may pass, such as Go, typically positions may further be importantly distinct depending on whether the last move was a pass: the Go game ends when the players pass consecutively. There may be other information distinguishing otherwise identical positions in principle. In chess, when a position is repeated 3 times, it is a draw. Thus a position which has recurred twice before is different from one which has not previously recurred in the very real sense that moves from it have different consequences. This is typically neglected in representing the search tree as a DAG, but this neglect gives rise to a problem called the "graph history interaction"([Palay 1985] p 131) that in practice causes errors in computer play.

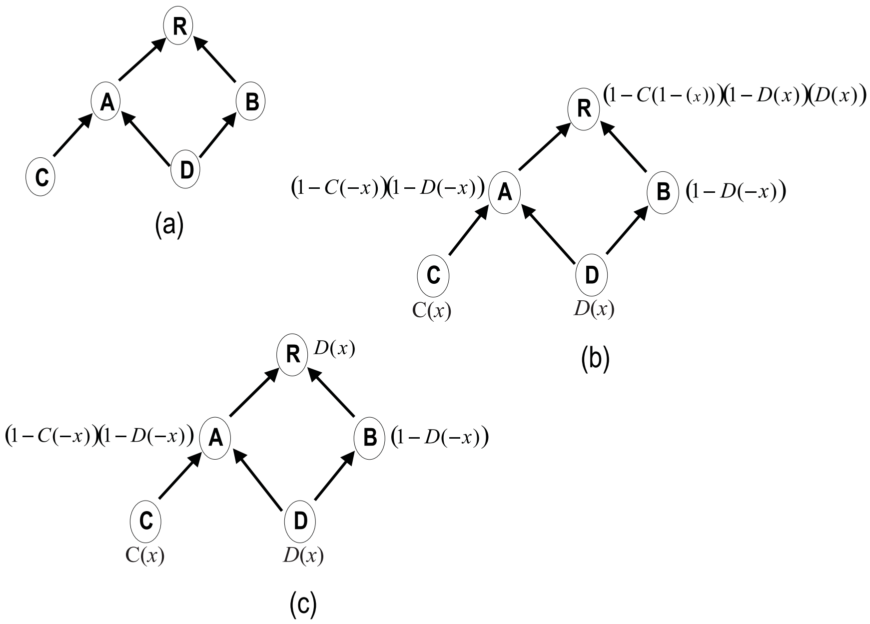

Figure 4: From the negamax DAG of figure 3a, assigning falling CDF's $C(x)$ to leaf C and $D(x)$ to leaf D, we naively calculate from equation 4 the falling CDF's assigned to leafs A, B, and R as shown in figure 3b. (Note that $1-\overline{c}(x) \equiv \underline{c}(x)$.) Linearizing at the root $(1-(1-CD)(1-D) = CD+D-CD^2 \to CD+D-CD = D)$ takes care of the dependency and yields figure 3c. That this is correct is evident from the alpha-beta cutoff structure in the ensemble viewpoint. Node C cannot influence the root in minimax propagation as either the value of node D is greater than that of node C, in which case C disappears at max node A, or D is smaller, in which case C disappears at min node R.

Unfortunately the prescription offered is not efficiently computable for minimax DAG's with a large number L of sources because the number of terms in the formula is 2^L. The theorem below shows that one can not generally compute the distributions in time polynomial in the number of leaves[7].

Theorem 3. It is #P-hard to evaluate the mean value of the root of a pure negamaxing game DAG whose leaves are probability distributions. The result holds even if the leaf probability distributions are all independent 50-50 coin flips (i.e. assume value 0 with probability 1/2, 1 with probability 1/2), the DAG has only negamaxing nodes, and has depth 2, and no node's branching number (except for the root node's) exceed 2.

Proof. Regard the leaves as Boolean (0-1) variables arising from i.i.d. coin flips. The minimax value of a 2-level DAG, whose root is MIN and whose depth-1 nodes are MAX nodes, is then an AND of the ORs of the children of the level-1 nodes. Let the depth-1 nodes have branching factor ≤ 2. Then determining the probability that the root assumes value 1 is precisely (except for a normalization by a factor of 2^L, where the DAG has L leaves) the problem of counting satisfying assignments in "monotone 2-SAT," which was shown to be #P-complete in [Valiant 1979]. QED.

Note that we can approximate the exact CDF's at all nodes, taking full account of the DAG dependencies, by Monte Carlo evaluation. (See also [Karp et al 1989].) This may be useful in evaluating our move choice, but we believe is likely to be too slow in convergence to be useful in deciding how to expand the tree. The reason for this is the following. Gameplayers are interested in changes to the utility coming from expanding N leaves, where N is a large number. Hence we are interested in changes in utility of order 1/N caused by any given leaf. Hence our Monte Carlo calculation must sample from regions in the ensemble space with volume of order 1/N. But of course it takes time N to write down any given configuration, so the fastest one might imagine a naive[8] Monte Carlo calculation being computed[9] is $\Omega(N^2)$ which would be unacceptable for game applications, where alpha-beta programs often grow trees involving billions of leaves.

4 Discussion

Alternatively, for *small* DAG's our exact prescription is feasible. This suggests the use of a "local heuristic" node-valuation method in which the dependencies arising from the "local" structure of a game DAG are treated exactly, but where we continue to pretend that the CDF's at "far away" DAG nodes are independent. One could, for example, use the propagation algorithm of Theorem 2 to calculate the CDF at each node exactly in terms of the CDF's of its descendants d levels deeper in the DAG. In the limiting case of $d = 1$ we neglect all DAG structure, and obtain our usual linear time propagation algorithm for trees. For finite $d > 1$ we compromise, using an algorithm that accounts for local DAG structure. The size d that is acceptably fast in practice will depend on the branching factor of the nodes and the DAG structure.

There are, of course, numerous algorithms in the literature for propagating probability distributions up DAG's, notably the junction tree algorithm[Lauritzen and Spiegelhalter 1988][Jensen 1996], the arc reversal algorithm[Shachter 1986], and conditioning [Pearl 1986]. These algorithms are more general than ours, since they do not assume negamax propagation, but therefore do not attempt to profit from the special nature of game tree propagation. They are exponential time in the worst case, but may be practical in some circumstances, so it is interesting to ask how they might fare in practice on game DAG's.

In many games there is relatively little DAG structure– so taking account of it would be *relatively* easy, but equally relatively uninteresting. In Othello, for example, use of a Hash table typically speeds up alpha-beta algorithms by a factor of only about 2 [Buro 1997], indicating a relatively uninteresting DAG structure. Even here, however, it seems unlikely that standard techniques (or our new algorithm in its exact form)

[7] Unless P=NP=#P. #P -hard problems are at least as hard as NP-Complete problems. An easy corollary is that it is also #P-hard even to determine the *sign* of the mean value of a game DAG.

[8] One might imagine a more sophisticated Monte Carlo procedure which evaluated the utility of leaf L by sampling uniformly directly from the region in which the value of leaf L is relevant and then computed the integral by some procedure analogous to that of [Karp et al 1989], but we have been unable to construct a procedure along these lines which is efficient and rapidly mixing.

[9] Actually the situation is worse than this for two reasons. First we must sample from such regions many times because of the inherent noise in the Monte Carlo procedure. Second, in a game like chess we are interested in move accuracies at least of order 1/50 (if we would like to make 50 moves without expecting an error) and so are interested in changes in utility of order 1/(50N) from expanding a given leaf .

would be tractable. The junction tree algorithm, for example, attaches tables to cliques in the junction graph (cf [Jensen 1996] for a discussion) of size the product, over the nodes in the clique, of the number of the states of the variables at each node. The use of such a large matrix is an example where working for general dependencies of one variable on its parents, rather than exploiting the special nature of negamax propagation, may create problems. The number of states of variables in the game DAG is large: proportional to the number of sources that are ancestors of the node, with a proportionality factor that can be of order 10[Smith et al 1997]. The size of the largest cliques will be large. A node in the DAG has parents for every following position in the game tree. For some nodes (e.g. the root, or in losing positions) where one must consider every legal move, this can be 20 or larger, which will result in cliques of at least size 20. Moreover the DAG is of depth about 22, and contains loops of circumference up to 44 which will involve adding additional chords to the junction graph.

For chess, the DAG structure is much richer yet than for Othello, and thus there is much more interest in exploiting it. Use of a hash table typically lowers the effective branching ratio of alpha-beta chess programs from about 6 to about 4 [Buro 1997]. In other words, the alpha-beta program grows a DAG of perhaps 10^6 sources, which would have perhaps 100 times as many leaves if unwrapped into a tree. This DAG is depth, say, 11 (and sometimes could be considerably deeper for our Bayesian algorithm, searching the most utilitarian lines), and has loops of circumference up to twice its depth. It will in practice have nodes with 50 or more parents, and thus cliques in the junction graph at least of size 50.

Note that to compete with alpha-beta algorithms (not using probability distributions), the propagation step must be reasonably fast: i.e. taking time growing *linearly* in the size of the DAG, and with an acceptable proportionality constant so that propagation of probability distributions occupies only a reasonable constant fraction of total thinking time. It seems unlikely that any algorithm will suffice in practice for tractable, *exact* propagation of probability distributions on such graphs, but some version of the heuristic we suggested above (e.g. for small enough d) could achieve acceptable speed. Only extensive experimentation, however, would determine whether a useful level of accuracy could be obtained if the heuristic were tuned for sufficient speed.

Acknowledgement: We thank the referees for comments on the draft.

References

[Baum and Smith 1997] Eric B. Baum, Warren D. Smith: A Bayesian Approach to Relevance in Game Playing, AI Journal 97: 195-242 (1997).

[Smith et al 1997] Warren D. Smith, Eric B. Baum, Charles Garrett, Rico Tudor: Experiments with a Bayesian Game Player, submitted for Publication, also at http://www.neci.nj.nec.com:80/homepages/eric/eric.html

[Buro 1997] Buro, M., personal communication.

[Cooper 1990] Cooper,F. G. “The computational complexity of probabilistic inference using Bayesian belief networks”. AIJ 42:393-405, 1990.

[Lauritzen and Spiegelhalter 1988] Lauritzen, S. L., D. J. Spiegelhalter (1988) “Local computations with probabilities on graphical structures and their application to expert systems (with discussion)”, J. Royal Statistical Soc. Ser B 50 pp157-224.

[Jensen 1996] Jensen, F. (1996). *An Introduction to Bayesian Networks.* London: UCL Press.

[Karp et al 1989] Richard M. Karp, Michael Luby, and Neal Madras: Monte-Carlo Approximation Algorithms for Enumeration Problems, Journal of Algorithms 10 (1989) 429-448

[Palay 1985] A.J. Palay: Searching with probabilities, Pitman 1985

[Pearl 1986] Pearl, J. (1986) “A constraint-propagation approach to probabilistic reasoning”, in *Uncertainty in Artificial Intelligence*, eds. L. M. Kanal and J. Lemmer, pp 357-370, Amsterdam: North Holland.

[Shachter 1986] Shachter, R. D. (1986) "Evaluating influence diagrams", Operations Research 34(6) pp871-82.

[Valiant 1979] Leslie G. Valiant: The complexity of enumeration and reliability problems, SIAM J. Computing 8 (1979) 410-421