Best Play for Imperfect Players and Game Tree Search; part I - theory

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

REVISED

September 4, 1995

And with a pseudocode appendix by Charles Garrett.

Abstract[1]**.**

The point of game tree search is to insulate oneself from errors in the evaluation function. The standard approach is to grow a full width tree as deep as time allows, and then value the tree as if the leaf evaluations were exact. This has been effective in many games because of the computational efficiency of the Alpha-beta algorithm. But as Bayesians, we want to know the best way to use the inexact statistical information provided by the leaf evaluator to choose our next move. We add a model of uncertainty to the standard evaluation function. Within such a formal model, there is an optimal tree growth procedure and an optimal method of valuing the tree. We describe how to optimally value the tree within our model, and how to efficiently approximate the optimal tree to search. Our tree growth procedure provably approximates the contribution of each leaf to the utility in the limit where we grow a large tree, taking explicit account of the interactions between expanding different leaves. Our algorithms run (under reasonable assumptions) in linear time and hence except for a small constant factor, are as time efficient as Alpha-beta. In a given amount of time our algorithm can thus, at least in principle, grow a tree several *times* as deep as Alpha-beta along the lines judged most relevant, but at a cost of expanding sufficiently less along lines judged less relevant so that a small constant factor fewer nodes are searched in total. Our algorithm will apportion a greater fraction of its thinking time to more relevant moves, and will appropriately weigh the relevance of each leaf in choosing its move once it is finished searching. Empirical evidence of the efficacy of this approach is presented in part 2.

1 Introduction

The standard method for computers to play games like chess, first suggested by Shannon[32] , is to grow a full width game tree as deeply as time permits, heuristically assign a numerical evaluation to each leaf, propagate these numbers up the tree by minimax, and choose as the "best move" the child of the root with the largest number. In practice, this computation is speeded up dramatically by the "Alpha-beta" algorithm, which allows one to prune off large sections of the search tree while still rigorously guaranteeing to evaluate

[1]Portions of the abstract and introduction were previously excerpted in [5]

the full width tree of given depth. A number of heuristic improvements such as "move ordering", "iterative deepening" and killer tables have been suggested which in practice allow Alpha-beta to achieve nearly its theoretical limit speed up, i.e. to search in a given time nearly twice as deep a tree as would a naive tree searcher. Alpha-beta, with heuristic improvements, is the search engine in virtually every high-performance game program.

However, it is worth asking whether this is conceptually the right approach. Can it be optimal to set out to evaluate a full width tree? Surely there must be some way to use information gained as the search proceeds to decide to search deeper along more relevant avenues and less deep in other directions? And while minimax is the correct way to combine exact leaf values, is there no better way to combine noisy estimates? Historically a number of authors have asked questions such as these (see below for a partial review). Indeed such questions seem central to understanding reasoning, the original motivation for studying computer game playing. But Alpha-beta is such an efficient algorithm that it has been hard to beat.

Now the whole point of search (as opposed to just picking whichever move looks best) is to insulate oneself from errors in the evaluation function. When one searches below a node, one gains more information and one's opinion of the value of that node may change. Such "opinion changes" are inherently probabilistic. They occur because one's information or computational abilities are unable to distinguish different states, e.g. a node with a given set of features might have different values. We thus propose to adopt a probabilistic model of one's uncertainty. Given such a model, there is a well defined answer to how to combine information in any given partial search tree. And there is a decision theoretic answer to which avenues should be explored further to optimize one's chances of winning. We call this approach BPIP – Best play for imperfect players.

To implement this approach one must adopt some definite probabilistic model. The model we adopt, which we call DFISA, involves an evaluation function which, rather than returning a single number estimating the value of the position, also returns a probability distribution giving the likelihood of opinion changes in that number if the node were searched deeper. We assume that these distributions are independent[2]. We are then able to specify a procedure which (a) values the tree correctly, taking into account all the information at our disposal, including the probabilistic distribution information which is not available to the standard approach, (b) recursively estimates which is the most utilitarian set of leaves to expand, taking into account for example the impact of current leaf expansion decisions on future leaf expansion decisions, and (c) under reasonable assumptions has only a small constant factor overhead, and hence is competitive with Alpha-beta in terms of raw efficiency.

The mean of our distribution valued evaluation function is an ordinary evaluation function, but our distribution gives the probability of various deviations if the node is searched deeper. Such an evaluation function may be readily learned. In essence one may empirically measure the likelihood of various opinion changes as a function of various features. Under our assumption that these distributions are independent, BPIP yields a certain formula[25] for propagating these distributions up the tree which associates to each node η in the tree the probability node η's negamax value is x given that a value is assigned to each leaf from *its* distribution. After we are done expanding the tree, the best move is the child of the root whose distribution has lowest mean. Note that we take means at the child of the root *after* propagating, whereas the normal (Shannon) approach takes the mean at the leaves before propagating, which throws away information.

The main motivation for calculating all these exact distributions is, however, to allow "utility guided tree growth." We model the expansion of a leaf as selection of one value from its distribution. Within this model, one may formally describe the optimal decision-theoretic strategy for growing a search tree; c.f. Appendix D. Unfortunately this strategy seems computationally intractable.

The practical solution we propose is to grow the search tree by repeatedly expanding the leaves with an appropriately defined largest expansion relevance. The "immediate expansion utility" of a leaf is the expected gain that would accrue if one expanded that one leaf and then chose one's move, rather than choosing one's move with no expansion. Expanding according to immediate expansion utility (similar to the "metagreedy" idea of [28]) is not good enough because of its neglect of one's ability to expand other leaves

[2]Note: we are *not* assuming that the probabilities of winning at different nodes are independent, as have [26, 14]. We are assuming that the *errors* in our estimates of these quantities are independent. Tests of the validity of our independence assumptions in several games are reported in Part 2.

before moving. For example, frequently a "conspiracy" among many leaves is necessary to affect the best move choice [22]. We may similarly define the immediate expansion utility of any subset of the leaves. In fact, we can compute in ≈linear time the decision theoretic utility U of expanding the entire game tree. This gives a natural condition for termination of the search: stop searching and make your move when the cost of computer time outweighs the utility of further search. This condition may be used to divide thinking time between moves, allocating more time to moves where the decision is more difficult and more important.

The leaf relevance measure we call ESS is defined as the expected absolute change in U when we expand leaf L. We have three arguments why this is a natural measure. First, it can be proven to approximate within a factor of two the total contribution δ^L of leaf L to *all* the probabilistically weighted 'conspiracies' it could participate in. The true decision theoretic utility of expanding a given leaf depends on the interactions with possible other leaf expansions in the future. δ^L directly measures the relevance of expanding leaf L in the limit where you will search a large enough tree before moving to leave little remaining expansion utility, i.e. where your play is accurate. Arguably this is the operative limit for modern game playing programs. Second, the *absolute* change in U values rises and falls in U, when you expand L, equally. This is intuitively reasonable since both are valuable. A fall in U takes you closer to moving. A rise in U means your previous understanding was flawed in a relevant way. Third, the ESS can be seen as the best estimate of the *a posteriori* change in the expected utility when you expand leaf L. These arguments are explained in detail in §7.

This, then, is our proposal. Use a trained evaluation function to assign a probability distribution to each leaf. Propagate these distributions up the tree according to certain formulae yielding a probability distribution at each node. Compute the ESS values of each leaf and expand the leaves whose ESS values lie above some percentile. Keep iterating this procedure, i.e. re-evaluating the whole tree and expanding another set of leaves, until the utility of further growth is smaller than the estimated time-cost it would take, then output the best move. We propose various algorithmic devices to do this efficiently, e.g. the "gulp trick," certain multilinearity lemmas, and our "influence function" methods. Assume that it takes at least about $d \log b$ computer steps to evaluate a position, where b is the geometric mean branching factor and d is the mean depth of the leaves in the final tree. This is entirely reasonable for complex games like Chess, where evaluation is relatively expensive, but may fail in games like Kalah, where evaluation is cheap. If this assumption holds, then our entire move-finding procedure will run in time depending only linearly on the size of the *final* search tree. (If this assumption fails, then BPIP gives up a logarithmic factor.)

The constant factors in our time bounds are small. Thus for chess assume that we tune our algorithm to search three times as deep along the lines judged most relevant as Alpha-beta. Then if our algorithm and Alpha-beta explore for the same amount of time, the tree we grow will be up to three times as deep, but contain about half as many leaves, as that of Alpha-beta.

Experiments playing BPIP against Alpha-beta in a variety of games are reported in Part 2. In our experiments, BPIP beat strong Alpha-beta programs in several games. This is the first time to our knowledge that an alternative algorithm has beaten strong Alpha-beta programs under realistic competitive conditions[3]. BPIP's advantage in our experiments increases rapidly with the complexity of the game and the amount of time allotted to each contestant.

1.1 Relationship to Previous work

There is an extensive history of proposals to selectively grow trees in the most interesting directions, dating back to Shannon's original paper[32]. We briefly review here the relationship to the present paper of some modern proposals. The work of Palay[25], Russell and Wefald[28], McAllester[22], and Rivest[27] is discussed in more detail in §12.

Palay was the first author to propose the use of distribution valued evaluation functions and also proposed the same equations for propagation of probability distributions that we do. We have improved the use of these equations by providing a rapid, low storage algorithm for a class of distributions, and showing that they result from the BPIP philosophy for node valuation. Palay did not appreciate this philosophy: indeed

[3] And see §12.3.

his book does not propose choosing a move based on means of distributions after propagation (cf. §12.4). Palay's main interest was in guiding tree growth, but according to very different criteria than the present paper.

I. J. Good mentioned some vague notions of utility in tree searching in his "5 year plan" [16]. Russell and Wefald[28] gave a clearer proposal of what optimal search might mean. They also first proposed the use of a utility based stopping condition[4], and they used evaluation functions which return distributions. However they did not propose or maintain a Bayesian model of one's uncertainty. Instead, they attempted to calculate directly the utility of expanding individual leaves, which they did in the "metagreedy" approximation. This is oblivious to interactions of different leaf expansions. They valued the tree using negamax. We also remark in §12.3 that their algorithm required time superlinear in the number of leaves.

McAllester first enunciated the notion of conspiracy number and gave an interesting algorithm for constructing high conspiracy number trees. His approach basically considered the expansion relevance of a leaf to be inversely proportional to the size of the single smallest conspiracy it could participate in, independent of how probable that conspiracy might be.

Rivest gave an ingenious algorithm which roughly speaking considered the expansion relevance of a leaf to be the partial derivative of the root's minimax value with respect to that leaf, if the "max" function were replaced by a differentiable approximation. This approach is fascinating, but somewhat ad hoc.

Several proposals to extend particular search lines based on heuristic considerations have been incorporated in modern Alpha-beta programs. "Singular extensions", "Threat extensions", and "PV extensions" taken together were estimated to add 86 USCF rating points to the Deep Thought Chess Machine[3]. "Quiescence search" [6] is an important feature of every strong Chess program, although much less important in most other games. Similar extensions as well as Alpha-beta style cutoffs occur automatically in our procedure, and also in the algorithms of Russell and Wefald [28]. Our approach of stating a Bayesian model of search, and then giving a provably efficient algorithm approximating Best Play for Imperfect Players can thus be seen as formalizing this line of research.

A number of authors have discussed probabilistic tree valuation procedures. Pearl[26] proposed a probabilistic propagation scheme we discuss in more detail in §3.3. Hansson and Mayer[12] advocated probabilistic propagation taking into account dependencies and evaluations at internal nodes in the search tree. Baum[4] remarked that the value of a node depends not only on the values of its children but also on an estimate of the 'extra information' one would have if the node were the root of a search tree, and proposed a propagation scheme taking into account that the amount of extra information will vary with depth in the search tree.

1.2 Outline of rest of the paper

§2 gives some motivation for our approach and gives a sample procedure for finding evaluation functions in the form we need. §3.1 and §3.2 give two views of our tree valuation philosophy. §3.3 clarifies the difference between our philosophy and two competitors, "plain negamaxing" and "naive probability." §4 describes how we propagate distributions up trees. The next 5 sections describe how we grow our tree. §5 describes the gulp trick which is instrumental to our time efficiency. §6 describes the "influence function" methods which are also critical. §7 covers briefly our "leaf relevance" measure for determining which leaf is most interesting to expand next. The proof of our relevance approximation theorem is in Appendix B. §8 gives a condition for terminating search and making a move. §9 puts the algorithm all together. §10 reviews what happens when the game tree is a DAG instead of a tree. We give both a procedure for exactly propagating distributions up a DAG, and complexity results showing this is hard to do efficiently. §11 describes how our framework allows partial expansion of nodes. §12 is a brief review of previous work. §13 is a conclusion and states the main open question. Appendix A gives numerical examples illustrating Sections 3, 6, and 7. Appendix C gives pseudocode for our algorithms and describes how influence function techniques can be used to compute our preferred leaf relevance measure efficiently. Appendix D describes the true (but not efficiently computable) decision theoretic optimal leaf to expand. Appendix E proposes a solution to a

[4]Their condition used the metagreedy utility of a single leaf, since they had no estimate of the total utility left in the tree.

problem called "Graph History Interaction", first noticed by Palay ([25] p 131), which arises in attempting to use transposition tables in games like chess.

The companion paper ("part II" [34]) reports experimental results on games including Kalah, "mod-9 Connect4," and Othello.

Our BPIP-DFISA techniques work not only for these 2-player games of perfect information, but also for games such as backgammon in which chance plays a role, and in 1-player games such as solitaire or Rubik's cube.

2 Evaluation function error

When the evaluation function assigns to a position a value (e.g. win probability) of .8, deeper search may indicate that the value of the position is .6 or 1.0. We propose instead of associating a single number with a position to have an evaluation function which returns a probability distribution. One can think of this as giving both a value (the mean of our distribution) like an ordinary evaluation function does and a distribution of error. Note in practice that the distributions as well as the mean values are position (feature) dependent, and also are not always unimodal. Thus substantial extra position dependent information is contained in our form of evaluation function beyond simply the mean. We will use this information in two ways. (1) It will help us to decide which move is best. (2) Because it expresses how a position is likely to change when we expand it, it will help us to value which nodes to expand.

The distributions at the leaves will be expressed as a sum of point masses p_i at abscissae x_i. Here we mean that if we expand the leaf L to some depth, say 6, evaluate the leaves using an ordinary evaluation function, and propagate by negamax the result back up to L, that leaf L will be found to have value x_i with probability p_i. In addition to the generalized density $\rho_m(x)$, it will be convenient to associate two kinds of *cumulative distribution functions* (CDFs) with each node m [25]:

$$\overline{c}^{(m)}(x) = \sum_{i,\, x_i^{(m)} \geq x} p_i^{(m)} = \mathrm{Prob}\,(y \geq x | y \text{ is distributed as above}) \tag{1}$$

$$\underline{c}^{(m)}(x) = \sum_{i,\, x_i^{(m)} \leq x} p_i^{(m)} = \mathrm{Prob}\,(y \leq x | y \text{ is distributed as above})\,. \tag{2}$$

Observe these CDF's are "staircase" functions.

Here is one 3-step method for producing such an evaluation function.

Step 1. Design an (ordinary) evaluation function E_1 to predict the expected desirability of the final game outcome. One way to create such a function is as follows. Divide the space of game positions into a number of classes, or "bins." Also, find a certain number of positional "features" $x_1, x_2, \ldots x_k$ which you think are correlated to desirability. Obtain a vast number of well-played games and assign the positions arising in them into your bins, and compute their feature vectors. In each bin perform a least-squares fit of a function of $x_1, x_2, \ldots x_k$ to the desirability of the final outcome of the game, starting from that position. The result[5] is a piecewise linear E_1.

Step 2. Again divide the space of game positions into bins. In each bin record the results of statistical experiments (from playing vast numbers of games using E_1 with negamax) indicating the probability distribution of the additive perturbations on E_1 that arise from backing it up by a deeper search, such as depth-6 negamax. Each bin will then contain a set of data points. This set may be thought of as a staircase CDF probability distribution, each jump of which corresponds to a single data point, and in which all jumps have equal magnitude.

[5] Lee and Mahajan proposed an elegant-sounding quadratic fitting procedure as an alternative to a linear least-squares fit [21]. Unfortunately, if each feature is either "antisymmetric" (from the viewpoint of the other player, that feature has negated value) or "symmetric" (same value from opponent's viewpoint as from player's) then it may be seen that all the quadratic terms in Lee & Mahajan's evaluation function actually will cancel (!), so any nonlinearity that remains will be 100% statistical noise. This effect apparently happened in Lee and Mahajan's program "Bill," since two matrices given on page 32 of [20] have elements identical to within 1%...

Step 3. *Compress* the staircase-CDF probability distributions in each bin, into staircase CDFs optimally approximating the original staircases, but having only a small number of stairsteps ([19] and see part II).

By adding regurgitated compressed bin data to E_1, one obtains an evaluation function E_2 which returns staircase probability distributions[6].

3 Philosophy of Distributions at Nodes

There are two alternative but mathematically equivalent ways of regarding our approach to tree valuation. Both of them lead to the recurrences (3-5) of §4. The first, the "ensemble view," is more straightforward to describe. The second, which we dub "BPIP-DFISA," is philosophically more pleasing and makes evident some approximations that might have been overlooked in the ensemble view. We describe these in the next two subsections.

3.1 The Ensemble View

We have a tree T. At each leaf L we have a probability distribution $P_L(x)$ returned by an evaluation function as described. The true value of the leaf is viewed as being drawn from this distribution. So assuming that the distributions at the leaves are independent, we define an ensemble of trees. This ensemble contains one tree for each possible configuration of leaf values one could get by drawing from the associated distribution at each leaf, and each such tree is probabilistically weighted by the product of the sizes of all the point masses in its leaves. (A numerical example is worked out in Appendix A.)

We now associate a probability distribution $P_n(x)$ to each node in our tree T as follows. If a tree is chosen from the ensemble, $P_n(x)$ is the probability that the negamax value of n in this tree will be x. §4 will tell how to calculate this efficiently.

We have assumed that the distributions at the leaves are independent. This will not be exactly the case. However we feel dependencies are not likely to be as important here as in some probabilistic game tree propagation schemes. If two sibling positions are assigned by an ordinary evaluation function a value .5, meaning that the probability of winning in that position is .5, it could be unwise to assume that these are independent and assign a probability of winning of .75 to their parent,as is sometimes advocated[26]. Our distributions, however, are over the *error* in the evaluation function, or to put it another way, are distributions over changes in our opinion about the position that would arise from future expansion. Such a distribution of errors in the evaluation function, would seem to be inherently less correlated than the evaluation function itself.

3.2 BPIP-DFISA

When it was first remarked that any evaluation function returns probabilistic information[26], the question arose whether minimax is the correct way to propagate this information up a tree. One might instead propose a rule for combination of probabilities[26]. This reasoning correctly asserts that a position with 100 alternative moves, each of which has independently .1 probability of leading to a won game, is almost certainly a won position. Unfortunately this offers little solace to the player in this position, if he cannot figure out which moves lead to wins, for he must choose one particular move, and then loses with probability .9. Reasoning along these lines, one discovers that in attempting to value the nodes in a search tree, one must take account of the extra information one will have when playing at the position[4]. A position at depth 9, which has two children each estimated to be independently winning with probability 1/2, should be regarded as being a won position with probability 3/4 if we will have sufficient information to decide the

[6] We do not intend to imply that this suggested procedure is a particularly new or good way to design a CDF-returning evaluation function. It is merely *a* way. Other ideas that might be useful include piecewise polynomial fits, automated tree-classifier schemes, and replacing the discrete "bins," each containing a CDF, with a CDF whose parameters (step abscissae and heights) depend continuously on positional features. See part II for experience with such ideas.

status of the two children when we get there, but should be regarded as being a won position with probability 1/2 if we will gain no useful information. This line of thought leads us to the following recursive definition.

Definition. Best play for imperfect players (BPIP):

1. The value of the root is the max of the negated values of its children.
2. The value of a non-root, non-leaf interior node η of S, is the sum, over m, of (the probability that[7] in BPIP starting from η, the player on move will make move m) times (the value of child m).
3. The value of a leaf node is the expectation value of the desirability of the final score which would be obtained in BPIP starting from this node.
4. "Best play" means to pick the child of the root with the greatest value, as your move.

Observation. Best play for imperfect players actually *is* the best possible strategy for picking your next move, in the Bayesian sense that it maximizes the expected desirability of the eventual game result.

In order to make BPIP concrete, one needs a *probabilistic model of extra information* so that one can estimate the probabilities in item 2 of the definition of BPIP. If one chooses a sufficiently tractable such model, then it becomes possible to actually compute BPIP. The model of extra information which the present paper utilizes is as follows:

Definition. Depth-free independent staircase approximation (DFISA). The evaluation function, applied to a game position η, returns two things. The first is the expected desirability E of the final position which would be reached in actual play between the present two players onward from η. The second thing returned by the evaluation function is the probability distribution P of opinion changes. The statement '$P(x) = y$' means that upon deeper search of η, the probability is y that we would change our opinion of the expected desirability of η from E to x. The word "staircase" reflects the fact that we will demand that this probability distribution have a staircase CDF. The "depth-free approximation" we will now make is to *suppress* any level-dependence[8] or opponent-dependence in the opinion-change distributions. That is, we assume that the probability of various opinion changes at a given node is independent of whether we expand that node 1 level or many levels[9]. We also assume, as our final approximation, that the opinion-change distributions at different leaves are *independent*.

3.3 How BPIP differs from two previous tree-valuation philosophies

There are 3 competing philosophies of how to use heuristic evaluation values at the leaves of our search tree to decide what is the best move. The first (Shannon) is plain Negamaxing. The leaf evaluations are scalars. The second, which we call Naive Probability Update (NPU), is to have the evaluator return a probability distribution over the space of possible game outcomes[10]. For example, in a 2-outcome game, it would just return "the probability that this position is a win, with perfect play." One then propagates these probabilities up the tree assuming all probabilities are independent. The *same* propagation formulas 3-5 are used as in BPIP, a fact which has led some unwary readers of previous drafts to conclude erroneously that BPIP is NPU. NPU is equivalent to the probability propagation proposal of Pearl[26], experimentally studied by Chi and Nau[14] and others. NPU yields at each interior node η of the tree, an estimate of the probability that η is a win with perfect play. You then move to the child of the root with the largest chance of being a win.

Both these philosophies are wrong in principle. Negamaxing is not the best way to use probabilistic information; indeed much available information is thrown out (since the evaluator is scalar) before one even

[7] This *is* a probability since your move choice, if you reached η, would be based on extra, currently inaccessible, information, gained from search beneath η. This extra information can only be modeled probabilistically.

[8] See [4] for an attempt to take level-dependence into account in the context of scalar valued evaluation functions.

[9] Otherwise, we would have had to deal with distributions of distributions of distributions.

[10] N.B. the BPIP evaluator returns a probability distribution over the real numbers, not over game outcomes. $P(x)$ is the probability that upon searching a little deeper, we would change our opinion of that node's value (i.e. expectation value of game result with BPIP play) to x.

starts negamaxing[11]. NPU computes the wrong thing since we don't care about the probability that a position is a win for God, we care about the probability that *we* will be able to win it. NPU also makes much too drastic independence assumptions, as seen in experiments reported in Part 2. Finally, NPU is not the 'correct' way to combine the probabilistic estimates in a game tree in principle, even if one grants independence, because it neglects the issue of 'extra information'[4], as illustrated by the examples in the first paragraph of §3.2.

NPU and negamaxing are both degenerate special cases of BPIP-DFISA. If the BPIP-DFISA's evaluator's opinion changes are trained at infinite lookahead, then BPIP-DFISA reduces to NPU. If the evaluator is trained at zero lookahead so that there are no opinion changes, then BPIP-DFISA reduces to negamax.

4 Propagating Distributions Up Trees

The node distributions discussed in the previous section may be efficiently computed by propagating staircase CDF's up the tree. The formula for the CDF of a parent node in terms of its b children is[25]

$$\underline{c}^{(\text{parent})}(-x) = \prod_{i=1}^{b} \overline{c}^{(\text{child}_i)}(x) \tag{3}$$

if we are doing negamaxing,

$$\underline{c}^{(\text{parent})}(x) = \prod_{i=1}^{b} \underline{c}^{(\text{child}_i)}(x) \tag{4}$$

if we are doing Maxing, and

$$\underline{c}^{(\text{parent})}(x) = \sum_{i=1}^{b} \rho_i \underline{c}^{(\text{child}_i)}(x) \tag{5}$$

if we have a nondeterministic b-way "dice-roll" where the dice probabilities are ρ_i.

Theorem. Formulas 3-4 are equivalent to the ensemble picture in §3.1 and to BPIP-DFISA (§3.2).

Proof Sketch. In BPIP-DFISA, due to our model that at any deeper level of exploration, extra information would cause the probability distributions to shrink to a single one of their spikes, the move probabilities mentioned in item 2 of the BPIP definition, are simply the probabilities that this child's spike will have greater value than its siblings's, which is a certain product of CDFs... leading to the validity of formula 3 at one level above the leafs. The validity at succesively higher levels follows by induction. The ensemble picture is readily seen to be a consequence of equations 3-4. QED.

Note that if we assign, say, a k spike probability distribution to each of N leaves in a depth d tree, then we have a total of Nk spikes in the leaves, and at most Nk spikes in the nodes at any height h above the leaves, so that the total storage to propagate the CDF's exactly to the root is $O(Nkd)$, where d is the average depth of the leaves. By the use of an algorithm resembling binary list merge, the total time may be bounded by $O(Nkd\log b)$ where b is the geometric mean branching factor. This means that we don't need to make any approximations in propagating the CDF's. We will see that this is important in evaluating which leaves to expand.

Note that if the distributions at the leaves are all delta functions, i.e. the evaluations are exact, then our algorithm reduces to negamax.

5 Incremental growth of stored subtree. "Gulp" trick.

The next several sections will describe how we propose to grow the search tree. The first key technique is the gulp trick.

[11] A dramatic illustration of this is a tree all of whose leaves had the *same* expectation value. Although negamaxing would be helpless, BPIP would generally find a unique best move.

One may pay for a lot of sins by iteratively expanding the most "relevant" f of the leaves (f is some constant with $0 < f \leq 1$) in one gulp, and then re-valuating the entire tree from scratch. If the final tree has a typical branching factor b bounded above 1 (essentially always true, in practice), then the total work to do all the iterations will be dominated (except for a slowdown by a constant factor C) by the work on the single final iteration.

More precisely, if the number of leaves of the current tree is L, then the number of leaves in the next tree (after the gulp) will be gL, where the "effective growth factor" g is given by $g = 1 + (b-1)f$. The slowdown factor C, assuming the runtime cost of valuating a tree grows as least as quickly as its number of leaves, will be

$$C \leq 1 + g^{-1} + g^{-2} + \ldots = \frac{g}{g-1} = 1 + \frac{1}{(b-1)f}. \tag{6}$$

If one parametrizes f in terms of a positive constant κ by $f \equiv \frac{b^{1/\kappa}-1}{b-1}$, then after d gulps one has $b^{d/\kappa}$ leaves. This will allow the iterative gulp procedure to examine some nodes which lie κ times deeper than naive negamaxing with a constant-depth cutoff, could go. For example, with $b \approx 38$ (chess) and $\kappa = 6$ we have $C \approx 2.2$. For $\kappa > \ln b$ we find $C \approx \frac{1}{2} + \frac{\kappa}{\ln b}$.

Making f larger decreases the runtime but also decreases the control we have over the search. Presumably there is an optimal f which best balances these two effects. It may be found empirically.

The gulp trick makes it easy to design efficient algorithms which both grow the tree, and valuate the tree nodes, in sensible ways. Sometimes it is still possible to get an efficient re-valuation algorithm, even if one insists in expanding the tree one leaf at a time. This can be done if one only needs to re-value the nodes on the length-d *path* from the expanded leaf up to the root, and if, by cleverly updating stored information somehow, one can then still efficiently find the most "relevant" leaf of the new tree to expand next. See §12.1 and §12.2 for a discussion of previous work in this vein. Every such one-at-a-time algorithm will still pay a slowdown factor *at least* proportional to the depth d of the leaf being expanded, compared to the cost of expanding the same number of leaves using the gulp trick.

It is of course conceivable that the utility of expanding a gulp might be much less than the utility of expanding the same number of leaves one at a time. For example, this would occur if expanding few most relevant leaves in the gulp made the rest of the leaves much less relevant. See Part 2 for empirical evidence regarding the efficacy of our gulping procedure.[12].

6 Influence Functions

The "influence function" of a node η is defined to be the additive amount that some scalar quantity F near the root will change by if the value of η is perturbed by t while the other node values remain fixed. In negamaxing of single numerical values, the graph of the influence function consists of three line segments, the outer two being constant and the inner one being of slope 1. The t-values of the corners are the "α-β window" [26]. In this section we describe how we may associate to each jump in the CDF at each node in our search tree an influence coefficient. Let $inco_F^\eta[i]$ be the influence coefficient at the i-th jump at some node η for function F. For example F might be the mean of the distribution at the root. Then if one were to change the jump heights at η by some amounts $\delta[i]$, the change in F would be $\sum_i inco_F^\eta[i] \cdot \delta[i]$.

In fact, we will see how to compute the influence coefficients for all the nodes in a total amount of time dominated by the time it takes to valuate the nodes, provided the scalar function near the root is chosen appropriately. The method is based on noticing that each probability distribution at each node is a multilinear function of its children's vectors. Thus in particular if only *one* child node's value-vector is perturbed, the others remaining fixed, its parent's value is a *linear* function of that child. Thus each parent-child tree edge is associated with a linear transformation. By composing all the linear transformations of

[12] Another heuristic which could grow deeper trees than straightforward gulping is the following. Pick a set of leaves possibly to be included in the Gulp, ordered from most to least relevant. As you expand these leaves, consider their children and approximate their relevance. This must be done by determining the dependence of the leaf expanded on the child and the original relevance of the leaf. Also insert the children into the gulp, if sufficiently relevant. We have not further explored this idea because the Gulp trick is sufficiently powerful for most uses.

paths down from the root, we obtain the influence function for each node. Assuming that F is scalar and linear, then composition of the linear transformations on the root-node path will collapse down to a single dot product with a coefficient vector of the same dimensionality as that node's vector, namely $\sum_i inco_F^\eta[i] \cdot \delta[i]$.

We thus have two subtasks when endeavoring to compute the influence functions$_F$ at all leaf nodes. First, we have to compute the special starting linear transformation. That is we have to compute the influence coefficients at the root. We will describe how to do this in detail for a particular function F in Appendix C. This will allow us there to compute the expansion relevance of each leaf efficiently.

Second, we have to compose the linear transformations at tree nodes. This is done in a top-down manner. Assume we have the distribution at node η, with each jump labelled with its influence coefficient, and the child it came from. The child label was attached on the way up when we computed the distribution at η. We desire now to compute the influence coefficients for η's children.

What happens to F if, say, the jump at x_0 in child 1's CDF is perturbed[13] by δ? We then have that $\overline{c}^1(x)$ is increased by δ for $x \leq x_0$. In that case, from equation 3 we see that the corresponding jump j at $-x_0$ in the parent's CDF is perturbed by an amount[14] $\delta M_j = \delta \prod_{i=2,\ldots,b} \overline{c}_i$. Here M_j is the child-to-parent jump size magnification factor associated with that jump, and is most easily computed in practice by taking the ratio of the parent jump to the child jump. Also any jump of height p in $\underline{c}_{\text{parent}}(x)$ for $x > -x_0$ *not* due to a jump in child 1, will be perturbed by an amount $p\frac{\delta+\overline{c}_1(-x)}{\overline{c}_1(-x)} - p = \delta p/\overline{c}_1(-x)$. The jumps in the parent at $x > -x_0$ which *are* due to child 1 are unaffected by alteration of child 1's jump at x_0. Hence, the influence function coefficient I of the jump in the child's CDF which caused jump j in the parent is given by:

$$I^j_{\text{child}} = M_j \cdot I^j_{\text{parent}} + \sum_{k=\text{jumps in parent at } x>-x_j \text{ due to siblings}} I^k_{\text{parent}} p_k^{\text{parent}} / \overline{c}_{\text{child}}(-x_k) \tag{7}$$

We give an explicit numerical example of the computation and use of influence coefficients in Appendix A. We may efficiently compute using equation 7 as follows. Assuming the jumps at η are stored in a sorted list, we consider them in order of decreasing abscissa. As we consider each jump in the parent, we accumulate the $\underline{c}_i(x)$ of the corresponding child, and some quantities which allow us to compute the sum in equation 7, and we assign an influence coefficient to the corresponding jump in the child. The key idea is that by the use of various "running totals" we evaluate all the sums we need in linear total time. Appendix C gives pseudo-code.

If N is the number of leaves in the tree, and say each leaf has a k spike distribution, the total time and storage needed to do this top-down propagation is bounded by $Nkd + T_0$, since the time and storage needed at any level of the tree to find all the influence functions at all the nodes of the next level, is $O(Nk)$. Here T_0 represents the time needed to find the initial starting linear transformations at the children of the root; for the particular F's we consider in the paper, $T_0 = O(Nk)$.

Note that these influence function methods are critical to our time bounds. There will generally be $\Omega(Nk)$ spikes in the CDF's near the root. Any naive approach to find the expansion relevance of a particular leaf would then take time of order Nk. But this means that valuing all N leaves would take time of order N^2, which is unacceptable.

[13] Note that this is a fiction, since the sum of all jumps no longer adds to 1. Nonetheless, F is still a linear function of the jump heights at η, and we can formally compute its change. A real perturbation of the jump heights at η would have involved perturbing more than one jump height, with the sum of the perturbations equal to zero. If, say, we perturbed the jump at x_0 by δ and the jump at $x_1 < x_0$ by $-\delta$, $\overline{c}^1(x)$ would be increased by δ for $x_1 < x \leq x_0$. There is in fact some freedom in how we define the influence coefficients as they will only be used for "legal" perturbations.

[14] The situation is slightly complicated if there are two children with jumps at x_0. In this case one may correctly treat the jump in the parent as composed of two separate components, one at $x_0 + \epsilon$, in the limit where $\epsilon \to 0$. If the scalar function F at the root is an ensemble integral of the scalar value of the root, then the change in it when the distribution at a node is changed is a well defined quantity, independent of any questions of tiebreaking. Any consistent tiebreaking definition will give equally valid influence coefficients. Note however that different definitions will lead to different influence coefficients. One must use the same convention when calculating the influence coefficients as when applying them.

7 Expansion Importance

In this section we will give the notion of expansion utility which will be used to guide our search. As has been discussed, we associate to each leaf a probability distribution describing the probability the leaf will be found to have a given value if it is expanded, and we propagate these probability distributions up to the children of the root. Accordingly we may in principle estimate how the distribution at a child of the root will change if we expand a given leaf or set of leaves. Thus in principle we can calculate which leaves are most interesting to expand, then expand them via a gulp. The present section will devise a suitable notion of expansion utility for this purpose, which is efficiently computable.

7.1 A formal model

To focus ideas it will help to consider the following 'metagame', which formalizes the problem of choosing which leaves to expand. We have a game tree with N leaves. At each leaf there is a probability distribution over the reals consisting of a finite set of spikes. A value (unknown to us) is assigned to each leaf chosen from its distribution. Now we play the metagame as follows. We are allowed to point at k leaves and for each of these leaves we are told the leaf's value. Then we choose a move at the root and are rewarded with the true value of this move (computed according to the game tree where each leaf has its true value).

In an actual game tree when we expand a leaf the probability distribution at that leaf generally gets sharper, but we are assuming that it actually sharpens to a spike. Some justification for this assumption of drastic shrinking, and hence of DFISA, is provided by the results of [33]. In any case we are only using this assumption to decide which order to expand leaves in. If we continue to expand to some depth below a node, we will in practice achieve substantial sharpening of the distribution at that node. It is accordingly hard to envision how the procedure of iteratively expanding the leaf we would most like to see sharpen to a spike could in practice seriously distort our order of leaf expansion.

Now we would like a strategy for playing this metagame: which leaves should we expand? Note that there are two variations of this metagame. In the first we must choose our set of k leaves before any are expanded, and in the second each leaf is expanded before we choose the next to expand. The situation in the tree growth algorithm we propose is midway between these two, due to our use of the gulp trick.

7.2 Expansion Utility of Leaf Subsets

Suppose one currently believes the best move is move 1. We may efficiently compute the expected utility of all future expansion. The *expected utility of all future expansion* is by definition the probability we will win[15], if before moving we expand all leaves until they are terminal positions, minus the probability we will win if we move immediately with no further expansion. Let m denote the number of children of the root, and, letting child #1 be the current favorite among these, we have

$$U_{\text{all leaves}} = \int_{-\infty}^{\infty} \left[\int_{x}^{\infty} (y-x)\rho_1(y)dy \right] \left[\prod_{2\leq j\leq m} \overline{c}^{(j)}(x) \right] \left[\sum_{2\leq i\leq m} \frac{\rho_i(x)}{\overline{c}^{(i)}(x)} \right] dx \tag{8}$$

if the root is negamaxing. A similar formula applies if the root is maxing.

Analogously we can compute the expected utility of expanding any set of leaves, under the assumption that we move immediately after expanding these leaves. So we can write down an algorithm for solving the metagame: compute the expansion utility for every set of k leaves, and pick whichever one has largest utility. This unfortunately is infeasible because there are too many subsets of size k. One can also write down a similarly impractical algorithm to solve the sequential version, see Appendix D.

In order to proceed, therefore, we assume that there is some measure of the 'relevance' of a leaf such that the relevance of a set of leaves can be determined as the sum of the relevances of its elements. We have

[15] For clarity we discuss the case of 2 outcome games. Our ideas generalize: for games with more outcomes, change, e.g., "the probability of winning" to "the expected desirability of the final game outcome" throughout.

examples (cf. appendix A) showing that no such additive relevance can exist. Nonetheless we make such a pretense and hope it is often approximately true. Because of our assumption of "additive relevance" the question of which version of the formal metagame we are considering will not arise.

One candidate that naturally might be proposed for the relevance of a leaf is the "single leaf expansion utility" (SLEU), i.e. the expected utility of expanding a leaf, assuming we made our move choice immediately after expanding it[16]. SLEU's may be efficiently computed using influence function methods. Unfortunately, this is a poor approach because it ignores conspiracies involving more than one leaf. Put another way, it neglects future expansion. This is most evident if the tree has a "utility conspiracy number" larger than 1, i.e. if all the SLEUs are *zero*. This can easily happen if no single leaf expansion will be enough to change one's opinion of which move is the best. Even when the conspiracy number is 1, however, we still argue below that the SLEU is not a good measure of leaf relevance.

7.3 The large expansion limit

Re-expressing this in the ensemble picture, we have that

$$U_{\text{all leaves}} = \sum_{C} [m_1 - \min(m_1, m_2, ..., m_n)] = \mathbf{E}(\text{negamax value of move 1}) + \mathbf{E}(\text{root value}). \tag{9}$$

Notation: C is the set of all configurations in the ensemble. In sums over configurations it is understood that each summand is to be weighted with the appropriate probabilistic weight for that configuration; these weights always sum to 1. We assume there are n children of the root, among which WLOG child 1 is the current favorite, and m_i for $1 \leq i \leq n$ will denote the negamax value of child i in configuration C.

The far right side of equation 9 is a convenient way to calculate $U_{\text{all leaves}}$.

Similarly the immediate expansion utility of any subset S of the leaves is

$$U_S = \sum_{C_S} \left[\sum_{C_{\overline{S}}} m_1 - \min \sum_{C_{\overline{S}}} (m_1, m_2, ..., m_n) \right] \tag{10}$$

(More notation: C_S is the set of all configurations of the subset S of the tree leaves, and $\overline{S}$ is the complement of S, i.e., all the leaves other than the ones in S, so $C = C_S \times C_{\overline{S}}$.) which when S is a singleton is the SLEU measure, and U_L is computable reasonably efficiently for all leaves L simultaneously by using influence function techniques.

The problem with SLEU which makes it a poor leaf relevance measure is that it regards each leaf expansion in isolation, ignoring the interactions with other leaf expansions.

In practice we expect one will continue expanding until the total remaining utility of expansion has gotten quite low. If we are going to expand $k \gg 1$ leaves, we are interested in achieving the largest gain possible from expanding a set of k leaves, and benefit from expanding just one leaf is not overly relevant to this. So instead, it might be reasonable to estimate gain in the opposite limit; that is to expand first the leaf which would be the one we would least like to leave, if we were going to expand all but one leaf.

We thus go to the "large expansion limit" where we take expectations assuming a large number of leaves have already been expanded. Thus instead of U_S, we would prefer $\not{U}_S$, the expected utility of expanding set S *after* $\overline{S}$ has already been expanded:

$$\not{U}_S = \sum_{C_{\overline{S}}} \left[\min \sum_{C_S} (m_1, m_2, ..., m_n) - \sum_{C_S} \min(m_1, m_2, ..., m_n) \right]. \tag{11}$$

Although at first sight $\not{U}_S$ looks new, we may rewrite the expression above as

$$\not{U}_S = U_{\text{all leaves}} - \sum_{C_{\overline{S}}} \left[\sum_{C_S} m_1 - \min \sum_{C_S} (m_1, m_2, ..., m_n) \right] = U_{\text{all leaves}} - U_{\overline{S}}; \tag{12}$$

[16] The procedure of expanding the leaf with optimal SLEU is akin to the procedure advocated by [28] in that leaves are valued as if there was no possibility for future expansion. They call this approach "metagreedy."

thus $\not{U}_S$ is just the difference between the utility of expanding all leaves and the utility of expanding all leaves other than S.

The trouble with $\not{U}_S$ is that we do not know how to compute (or approximate) it efficiently[17] with influence function techniques, for singleton S.

7.4 Our proposal: Expected Step Size (ESS)

Fortunately, there is a natural leaf relevance measure that is closely related to $\not{U}_S$, which we call δ^S. Although we also don't know how to compute δ^S efficiently for singleton S, we do know how for a fourth measure V_S for which we can prove an

Approximation Theorem. If L is a leaf, then

$$V_L \leq \delta^L \leq 2V_L. \tag{13}$$

Furthermore, if n, the number of spikes in the probability distribution at leaf L, is 2, then $V_L = \delta^L$. Also, if $p_1 + p_n = \alpha$, then $V_L \alpha^{-1} \geq \delta^L$.

Thus our final proposal is to use V_L as the relevance of expanding leaf L. (Numerical example – see appendix A.) The proof of this theorem is in Appendix B. But we have gotten ahead of ourselves; we still have not defined V_S and δ^S. We will now motivate and provide these definitions.

A natural quantity to examine in determining the expansion relevance of a leaf is how $U_{\text{all leaves}}$ changes when you expand it.

Let

$$Q^i \equiv \int_0^\infty P(\text{Some move is better than move } i \text{ by } q) q dq \tag{14}$$

We call Q^i the expected utility with respect to move i. If our current favorite move is move 1, we have $U_{\text{all leaves}} = Q^1$. Exactly similarly to the derivation of equation 9, we have

Lemma.

$$Q^i = \langle \text{root value} \rangle - \langle v^i \rangle. \tag{15}$$

where v^i is the value (e.g. probability of winning) if we make move i, and $\langle\rangle$ denotes ensemble averaging.

Corollary.

$$Q^i - Q^j \equiv \langle v^j \rangle - \langle v^i \rangle. \tag{16}$$

When we expand any leaf, the expected change in Q^i is zero. This is evident because Q^i before the expansion is defined as the average of the values it takes after the expansion. If the current favorite move changes after the expansion, say from 1 to 2, then $U_{\text{all leaves}}$ is redefined so that after the expansion it equals Q^2. It is easy to see that the expected change in $U_{\text{all leaves}}$ when we expand a leaf is precisely the negative of the leaf's SLEU.

Consider how $U_{\text{all leaves}}$ changes as we expand all the leaves in our meta-game. It starts with some value U_0. As we expand each leaf, it makes a random change, chosen from a certain, readily calculable distribution. For most of these expansions, the utility conspiracy number will be greater than 1 and the mean change in U will accordingly be 0, since the expansion cannot affect the current favorite move. As we expand leaves there will occasionally be a case where the utility conspiracy number is 1 and the mean change will equal the negative of the SLEU of that leaf. Of course when we actually expand this leaf U may increase or decrease, but the average change in it is negative. When we have expanded all leaves, the utility U of further expansion will by definition be zero. This convergence to 0 of course shows that our assumption that the leaf "relevances" are independent is not strictly true. The means of these steps must change as we go along to guarantee arrival at exactly 0.

[17] Alternatively, one might try to devise a leaf relevance measure such as Importance$(L) = \sum_{S,\, L \in S} U_S / |S|$ based on combining the utilities of many sets S. But we don't know how to compute these things efficiently either.

It would therefore be wrong to model the contributions of the leaves toward the total additive change in $U_{\text{all leaves}}$ as *independent* 0-mean steps of a random walk, and thus conclude that the variance in Q^1 would be the right leaf relevance measure[18].

At last we are ready to define V_L, our prefered notion of leaf relevance, to be the average absolute length of the step in Q^1 (if 1 is our current favorite move) i.e.

$$V_L \equiv \sum_{i=1}^{n} p_i |Q_L^1(i) - U_{\text{all leaves}}| \tag{17}$$

Here $Q_L^1(i)$ is the value of Q^1 after we expand leaf L, assuming it takes value x_i in its distribution, which occurs with probability p_i. We call this quantity the expected step size, or ESS. By the use of influence functions for Q^1, we can efficiently compute V_L at all leaves L simultaneously. The suprising fact that $Q_L^1(i)$ is equal to I_i^L, the influence coefficient of jump i in leaf L, simplifies this computation (see Appendix C).

As motivation for this definition: Assume our current favorite move is move 1. There may be some configurations in the ensemble in which 1 is not the best move. V_S thus tells us how much the ensemble sum over such configurations, weighted by the amount move 1 is inferior in each configuration, would be affected by knowledge of leaf set S's true value: in the notation of §7.3

$$V_S = \sum_{C_S} \left| \sum_{C_{\overline{S}}} [m_1 - \min(m_1, m_2, ..., m_n)] - U_{\text{all leaves}} \right| = \sum_{C_S} \left| Q^1(S) - Q^1 \right|. \tag{18}$$

Another motivation: We have already seen that decreases in the utility of future expansion caused by expanding a leaf, have much to do with the relevance of expanding that leaf, and it seems reasonable to treat increases and decreases in the utility of future expansion as equally valuable information. Increases reveal new and surprising information making future expansion more promising, while decreases take us closer to being able to move. Also, increases really are decreases, if you look at it from an *a posteriori* point of view. The reader may have found it counter-intuitive that the "utility of future expansion" can increase when we expand a leaf. The leaf (game position) always had a certain value, and we only became aware of it when we expanded the leaf. Expanding a leaf always yields information, so how can the remaining information be more valuable than the remaining information (a superset) was before the expansion? In fact, the future expansion has not actually become more valuable, but our previous estimate of it, i.e. $U_{\text{all leaves}}$ defined by equation 8, was in error. The SLEU values a leaf by summing the signed changes between $U_{\text{all leaves}}$ computed before the expansion and after. But this cancels inherently meaningless positive quantities against potentially meaningful negative ones. If you were going to make a monetary bet with a friend that the world will be destroyed next Monday, fair odds should reflect the *a posteriori* value of money on Tuesday in the two eventualities, and likewise it makes no sense to decide which nodes to expand based on *a priori* estimates. If we define the *a posteriori* utility with respect to leaf 1 before leaf L is expanded as the maximum of what equation 14 computes before the leaf is expanded and what it computes after (reasonable since this utility cannot increase) then the leaf of maximal V_L is equivalent to expanding the leaf which has highest expected drop in *a posteriori* remaining utility.

Finally, we have our Approximation Theorem connecting V_L and δ^L, where we define: The relevance δ^S of a leaf subset S is the expectation value of how much Q^1 changes when we expand S, i.e. $\delta^S = |\Delta_S Q^1|$, assuming we have already expanded $\overline{S}$.

This assumption means that we are working in the large expansion limit, so that δ^S takes the interactions among different leaf expansions into account. Thus δ^S tells us how much leaf set S contributes to the possibility we will change our mind about the best move in this limit.

[18] Another problem with the variance is that it seems to overvalue low probability high value perturbations, because it is a sum of squares. Note that the variance in Q is a more sensible measure than the variance in U because the latter actually penalizes leaves for having non-zero SLEU.

Indeed δ^S may be described with exactly the same words as our description of V_S in the sentences before equation 18, except that we now are working in the large expansion limit. The analogue of equation 18 is

$$\delta^S = \sum_{C_S} \sum_{C_{\overline{S}}} \left| m_1 - \min(m_1, m_2, ..., m_n) - \sum_{C_S} [m_1 - \min(m_1, m_2, ..., m_n)] \right| . \tag{19}$$

δ^S is very similar to $\not{V}_S$, but with the these two differences: (a) we are using Q^1 instead of $U_{\text{all leaves}}$, (b) we are summing the *absolute values* of $\Delta_S Q^1$.

To motivate (a), we imagine that we are in some intermediate regime (the "medium expansion limit"?) in which 1 is still the favorite move so that we may disregard the distinction between Q^1 and $U_{\text{all leaves}}$. In fact Q^1 and the medium expansion limit may be more interesting than $U_{\text{all leaves}}$ and the large expansion limit because it isn't clear we are going to do enough expansion to change our favorite move. Perhaps we should first worry about the utility of expansion with respect to the current favorite move, and only later, if and when this favorite changes, should we worry about the utility of expansion caused by potential further favorite changing.

To motivate (b), we could use the same argument as before ("it seems reasonable to treat increases and decreases... as equally valuable information"), but, if you didn't like that argument, here is another: if the absolute value signs were removed from the definition of δ^S, then it would be 0, and since if $\sum_i x_i = 0$ then $\sum_i |x_i| = 2 \sum_{i,\ x_i>0} x_i$, our sum of absolute values is really just the same as the sum of only the terms corresponding to utility *decreases*, except for an overall factor of 2 which has no effect.

We've now made it plausible that δ^S is a statistically good measure of the expansion relevance of a subset S of our tree leaves, and our approximation theorem showing that $V_L \approx \delta^L$ within a factor of 2, where V_L is efficiently computable for all leaves L, means that it is also algorithmically good.

Finally, let us make a few pragmatic remarks about our approximation theorem and our preference for ESS over SLEU.

One case of interest in practice is leaves with 3 spike distributions. In this case the bound would be $(3/2)V_L \geq \delta^L$ if the center spike contained no more than a third of the distribution. In general, cases which approach the upper bound of 2 have their weight concentrated at their central spike, and accordingly have low δ^L as well as low V_L, and are not particularly relevant leaves to expand. Thus as a general rule, V_L can be expected to be a better practical predictor of δ even than the theorem demands. Also note that if δ^L really were an additive leaf relevance, and if we expanded a set S of leaves whose relevance was in fact half as large as the optimal leaves in some disjoint set S', we could still make up for this by simply expanding twice as many leaves, so in the worst case the fact that we can only approximate δ^L would only cost us a factor of 2 in time.

V_L may not be a good measure of which leaf to expand if we do not expect to be able to expand many more leaves, since it counts heavily on the ability to exploit information gained by future expansion. If there is a probability p that we will move after expanding the next leaf, it might make heuristic sense to expand the leaf of greatest $pU_L + (1-p)V_L$ or some such[19]. But since we expand a large gulp of leaves at a time, it's likely we never need to worry about this.

8 Termination of growth. The "cost of time."

A natural termination condition is simply to terminate growth when the expected utility of such growth is less than the cost of the CPU time required to carry it out. The expected utility of future growth can be estimated by using $U_{\text{all leaves}}$. So we now need to estimate the "cost of CPU time" relative to the cost of losing the game.

[19]Also: In the event that one knows, before expanding a node, how many children $C \geq 1$ it is going to have [which is the case, in some games, at no extra cost] then it makes sense to expand the node L with the greatest value of V_L/C so that we get the most utility for our computer time expenditure.

In practice this would be accomplished with a "penalty function." For example, suppose that one must make 40 moves in 2 hours of thinking time, or else forfeit the game. Then one could use as the "cost of time" per second, cm/t, where m is the number of moves which one still has left to make in the time t that will be left after doing next the growth stage, and c is a positive constant chosen empirically.

What penalty function is the right one? Here is a derivation of one formula. We start with the table given by Newborn [23] of the probability $p(d)$ that the the chess machine `Belle`, when searching d ply deep, will prefer a different move to the one it prefers at $d-1$ ply. (This data came from 447 searches performed by `Belle` while extending its opening book into the early middle game, and the error bars on each $p(d)$ estimate are about $\pm .02$. We also give figures from W.D. Smith's warri program `w1` for 3523 selfplay searches performed with 29-42 seeds in play, with data only collected on 1 game ply out of 10 in an effort to assure independence among searches. In the `w1` data, any time a move was preferred over *any* move it had not been preferred to before, that was counted as an opinion change, even if `w1`'s favorite move remained unchanged.)

d	3	4	5	6	7	8	9	10	11	12	13	14	15	16	19	20
`Belle` $p(d)$			.331	.331	.277	.295	.260	.226	.177	.181						
`w1` $p(d)$	$\frac{2633}{3523}$	$\frac{2133}{3523}$	$\frac{1895}{3523}$	$\frac{1621}{3522}$	$\frac{1580}{3521}$	$\frac{1335}{3520}$	$\frac{1178}{3519}$	$\frac{923}{3518}$	$\frac{872}{3508}$	$\frac{718}{3430}$	$\frac{769}{3069}$	$\frac{389}{1755}$	$\frac{118}{729}$	$\frac{29}{309}$	$\frac{2}{39}$	$\frac{0}{20}$

It is not clear what formula should be used to describe $p(d)$; David Levy once suggested $c_1 d^{-2}$ but we prefer the Szabos's [35] suggestion $\exp(-c_1 - c_2 d)$, for some positive real constants c_1, c_2. Assume a depth-d search requires time B^d and assume that any favorite move change represents some constant[20] utility increment $(\Delta U)_{\text{typ}}$. Then if one is given an infinitesimal extra amount $\mathrm{d}t$ of time to use in some future search, the extra utility one can expect to be able to derive from this is $(\Delta U)_{\text{typ}}(\mathrm{d}t) \cdot p(\log_B t)/t$, that is, proportional to $p(\log_B t)/t$. Similarly, if we are given an infinitesimal extra amount $\mathrm{d}t$ of time to use in a total of m future searches (each one with $\mathrm{d}t/m$ extra time), the total utility we may expect to derive from that is proportional to $\frac{m}{t} p(\log_B[\frac{t}{m}])$.

So, presumably we should stop searching and move when

$$\frac{U_{\text{gulp}}}{t_{\text{gulp}}} < c_6 p(\log_B[\frac{t}{m}]) \frac{m}{t} = \begin{cases} c_3 \frac{m/t}{\log_B(t/m)^2} & \text{according to Levy} \\ c_4 (\frac{m}{t})^{1+c_5} & \text{according to Szabo} \end{cases} \tag{20}$$

where t_{gulp} is the estimated time to do the next gulp, U_{gulp} is the utility estimate for that gulp, t is the time that would then remain to make the next m moves in, and the c_i are positive real constants ($c_5 = c_2 / \ln B$). The Szabo and Belle data on chess would imply $c_5 = 0.085 \pm 0.04$, and the `w1` Warri data would yield $c_5 = 0.15 \pm 0.02$.

Caveat. Because of credos such as "if you have more time left than your opponent, strive to complicate the position," good time management is more complicated than just selecting a good penalty function: It should incorporate the opponent's time and an estimate of the complexity of the positions that are going to be searched in the future, and the "constant" c_6 in (20) should probably really be an increasing function of this estimate.

9 Putting it all together: Algorithm and time analysis

The final combined algorithm which we recommend is roughly as follows.

1. Start with a search tree S consisting of the root and its m children.

2. Compute estimated bound on expansion utility $U_{\text{all leaves}}$; if less than estimated cost of time for further expansion, then goto step 8. In particular if $m \le 1$, this would happen since $U_{\text{all leaves}} = 0$.

3. Compute the utility influence functions of the L leaf nodes of S.

[20] It could also depend upon depth, and if this dependence were assumed to be exponential, then the Szabo version of (20) would remain unchanged, although the constants might change. More generally if the dependence on depth were $f(d)$, then replace $p(x)$ in the below by $p(x)f(x)$.

4. Find Expected Step Sizes (ESS) for all leaves.
5. Mark a fraction f top most-utilitarian (largest ESS) leaves.
6. Expand marked leaves of tree S, use the evaluator to provide probability distributions for each of the new leaves, and (re)calculate the CDF's at all nodes using equations 3-5.
7. Goto step 2.
8. Find the child of the root with the greatest expected desirability, return it as the best move, and exit.

Pseudo-code for the important parts of this is is given in Appendix C. Here f, the fraction of leaves to expand, is some constant with $0 < f \leq 1$, as explained in §5. The running time of algorithm, except for a small constant slowdown factor $C = O(1)$, will be dominated by the running time of the last iteration (see §5).

We assume that the total time it takes for the evaluation function to evaluate a node and return a result is $v \times \mathrm{Ev}$, where the "result" consists of a vector of v different numbers and Ev is some amount. We let V denote the total number of numbers output by the evaluation function on all the calls to it that we make. Thus the total time spent inside the evaluation subroutine is $V \times \mathrm{Ev}$. We also assume that finding the n moves one can make in a given position takes $O(1+n) \cdot \mathrm{Mv}$ time and making or unmaking a move takes $O(\mathrm{Mv})$ time, where Mv is some constant value. (This assumption is usually realistic if the game has a "board" of bounded size.) Let b be the geometric mean branching factor and let d be the average depth of the leaves on our (final partial) tree S. Let $|S|$ and L be respectively the number of nodes and the number of leaves in S.

Now Step 1 is trivial. Step 6 (which has larger, or comparable runtime and space consumption to steps 2, 3, and 4) takes runtime $O(|S|\mathrm{Mv} + V\mathrm{Ev} + Vd\log b)$ and consumes storage $O(Vd)$. Step 5 takes $O(L)$ time and storage by the use of a linear-time selection algorithm [15] [8]. All iterations of the "gulping" loop (steps 2-7 inclusive) are dominated by the final iteration, up to the small slowdown factor $C = O(1)$ (see §5), leading to the final bound on the total running time and storage of:

$$\mathrm{runtime} = O\left((\mathrm{Ev} + d\log b)V + \mathrm{Mv}|S|\right) \qquad \mathrm{storage} = O(dV + |S|). \tag{21}$$

As was pointed out by a referee, we have here allowed ourselves a $d\log b$ factor of the sort we disallowed in previous comments on the limitations of one-at-a-time algorithms (§5). *But*, if Ev and Mv are decently large, i.e. of the same order as (or larger than) $d\log b$ (which will probably happen in complicated games such as chess), then our procedure's runtime, including all "constant factors," will be linear and entirely comparable to the runtime of a naive tree searcher[21].

10 What if your game tree isn't a tree? DAGs

Game trees are of course not necessarily trees, but may instead (in games in which "transpositions" are possible) be Directed Acyclic Graphs, also known as DAGs.

It is generally possible to save storage and time in DAG games. One uses a "hash table" to efficiently spot recurrences of previously evaluated positions, and then one need not valuate nor store that node twice.

[21] And if this $d\log b$ term actually did loom large – as might happen in simple games – it could be gotten rid of by compressing probability distributions as we go up the tree. In our exact scheme, with L leaves each with k spikes, by the time one propagates up to the root, the resulting distribution could have nearly kL spikes. But, with distribution compression, this would have been replaced by an approximate distribution with some smaller number, e.g. $2^{-d}kL$ if we compress by a factor of 2 each depth, of spikes. With algorithms of this sort, the $d\log b$ term in the runtime would be replaced by $\log b$, and the dV term in the space consumption would be replaced by V. The costs incurred would be: (a) sacrifice of exactness, (b) it is not clear what the best compression schemes (and ratios) are, and the compressor would have to be compatible with a scheme for propagating linear influence coefficients back down the tree, where these influence coefficients would now have to take into account the effect of the compressor. It is certainly possible to find compression schemes with the right linear properties (one example: replace every pair of two consecutive spikes by a single spike at their averaged location and with their summed masses), though.

This idea is an old one and is used in all the competitive chess programs, and it will reduce the workload by roughly at least as large a factor as the reduction in leaf count. It is equally valid in our formalism[22].

Aside from the issue of saving time, though, there is the issue of getting the right answer. In DAGs, our independence assumptions are assuredly incorrect, since some tree nodes may in fact be the *same* DAG node (or merely have a common descendant) and are therefore statistically dependent. We now describe how equations (5-7) may be modified to correctly propagate CDFs up a DAG. Our model here is that we still assume that the CDFs at the leaves are independent, but we now take correct account of all dependencies induced by the DAG structure in propagating the CDFs at the leaves to the root. First note

Lemma. The CDF at any node in a DAG is a multilinear function of the CDFs at the leaves.

Proof. This is evident in the ensemble viewpoint. The probability distribution at any node is given by a sum over the different configurations of its leaf descendants, weighted by their relative 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. To propagate CDFs up a DAG, first use equations 5-7 to naively yield the CDF at each node as a polynomial function of the CDFs at the leaves, and then, after expanding into a sum of monomials, 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. For the following negamax DAG

```
                 R
             A       B
         C       D
```

we naively calculate the following CDFs using equation 5:

```
                (1-(1-C(x))(1-D(x)))(D(x))
       (1-C(-x))(1-D(-x))            1-D(-x)
 C(x)                        D(x)
```

but linearizing at the root ($DC + D^2 - CD^2 \to D$) takes care of the dependency and yields:

```
                             D(x)
       (1-C(-x))(1-D(-x))            1-D(-x)
 C(x)                        D(x)
```

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 D is greater, in which case C disappears at max node A, or D is smaller, in which case C disappears at min node R.

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. the products $\prod_{j\in\{1,\dots,L\}} c_j(-^d x)$ where d is the height of node i above leaf j and the product is

[22]Some subtleties in the use of transposition tables: 1. Since we store the game-DAG, as compared to game-playing programs which use essentially no game-tree storage, our hash table entries need to contain virtually no information – only a single pointer. 2. For the same reason, we can painlessly take advantage of "bonus depth" gotten from occasional transpositions to positions that were searched (in the other line) more deeply, *both ways.* 3. Alpha-Beta programs need to store upper and lower *bounds* on position-values in the hash table, and sometimes these bounds need to be refined further upon use of that table entry. We have no such difficulties, but if we are using influence functions and leaf expansion utilities, we can run into the problem that DAG nodes lead multiple lives when the DAG is regarded as a tree. Hence, each DAG node may need to store several influence functions at once... and when deciding whether to expand a DAG leaf in the next gulp, we must somehow appropriately combine the expansion utility values from each multiple life. The "correct" way to do this is to store only the *sum* of all of a DAG node's multiple influence functions and regard it as "the" influence function of that DAG node. (The idea is that this is basically functionally equivalent to not compressing the tree into a DAG at all.) Then each DAG node has a unique utility value. 4. As Palay ([25] page 131) observed in a different context: in games such as chess where repetitions of position are possible, the DAG is not really a DAG (since it is not Acyclic), and a position must include a repetition counter as part of the definition of "position." Complications stemming from this are discussed and a resolution proposed in Appendix E.

taken over all subsets of the leaves. (Note, the players alternate turns, so we never have two paths from a leaf to the same position of differing parities.) We must only determine the coefficients of these terms. The formula given in the theorem is evidently correct at the 2^L assignments where we set the $c_j(x)$ to be 1 or 0, since in this case correlations are unimportant. QED.

Unfortunately the formula offered is not efficiently computable for minimax DAGs. It is possible to take account of DAG dependencies correctly using the ensemble viewpoint, but this requires runtime exponential in the number of leaves. One therefore asks whether it is possible to propagate distributions in DAGs in a smaller running time – bounded by a polynomial of the number of leaves. The theorem below shows that the answer is "no[23]."

Theorem. 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 outvalencies (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 [37]. QED.

Note that we can approximate the exact CDFs at all nodes, taking full account of the DAG dependencies, by Monte Carlo evaluation. (See also [17].) 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. We 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[24] Monte Carlo calculation being computed[25] is $\Omega(N^2)$ which would be unacceptable.

Alternatively, for *small* DAGs our exact prescription is feasible. This suggests the use of a "local heuristic" BPIP node-valuation method in which the dependencies arising from the "local" structure of a game DAG are treated exactly, but we continue to pretend that the CDFs at "far away" DAG nodes are independent. We have not experimented with either transposition tables in BPIP, or this heuristic.

11 On memory savings and Partial expansion of nodes

There are two main disadvantages of the procedure of §9 as compared with the standard chess playing methods.

First, our methods store the entire search tree in memory, whereas the standard Alpha-beta approach has tiny storage requirements (although a large transposition table is often used).

But this problem can largely be eliminated by using our scheme with an evaluation function that performs an ordinary Alpha-beta search to, say, depth 3. Such a "hybrid" scheme would only store the "inner core" of the tree (nodes at distance ≥ 3 from the leafs), which is much smaller, and could also take advantage of special purpose Alpha-beta and chess evaluation hardware.

[23] Unless P=NP=#P. Incidentally, an easy corollary is that it is also #P-hard even to determine the *sign* of the BPIP mean value of a game DAG.

[24] 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 [17], but we have been unable to construct a procedure along these lines which is efficient and rapidly mixing.

[25] 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 for instance, 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 changes in utility from expanding a given leaf of order 1/50N.

A related memory saving idea is "2-stage BPIP" in which a shallow "outer" BPIP search is called by the inner BPIP search as its evaluation function. Memory is saved because the outer search trees are *not* kept. Indeed, if the inner and outer search trees have roughly equal depths d, then the storage needs of a depth $d + d$ 2-stage BPIP searcher, in nodes, will be a small constant (≈2-10) times the *square root* of the storage needs of a 1-stage BPIP searcher going to depth $2d$. (Distribution compression would be needed at the boundary between stages to realize this space savings.) Of course, 2-stage BPIP will have to pay for these huge space savings with speed reductions, because the outer BPIP trees, not having been stored, have to be recalculated. But it seems that the speed reductions will only be a constant factor (≈ 3)! This is because during each "gulp," certain leaves of the inner tree will have to be re-searched at higher depth (in which case the higher depth search will utilize time swamping the time used by the previous search – same principle as the "gulp trick") or not (in which case we can simply retrieve the result of the previous outer search from a store) – either way, we are OK. There are various flavors of the 2-stage idea that one could consider; in the fanciest, the outer BPIP trees are themselves built and rebuilt by an iterative gulping procedure and with influence coefficients propagated down from values stored at their roots (and ultimately from the root of the inner tree).

Second, the standard methods often employ certain highly selective, and highly effective, searches near the leafs of their trees. We are thinking in particular of "capture searches" [6] and of the "null move heuristic" [11]. Capture searches recursively expand only the capturing moves from a node and replace the other moves mentally by a mythical "just sit there" move. The "null move" heuristic falsely pretends that the null move is a legal chess move, and uses a shallower than usual search beneath the null move to generate plentiful and cheap Alpha-beta cutoffs. These methods are considering only a very small subset of the most important moves out of affected nodes. Unfortunately our approach, as described so far, always generates either all or none of the children of any given node, and thus might suffer in comparision[26]. But we need not be so procrustean; we have found a very nice way to allow partial expansion within our framework, which we shall now describe.

Nodes instead of being always either fully expanded or unexpanded, can now more generally be partially expanded. The children of a partially expanded node are of two kinds: children which we've seen, and those we haven't. The entire set of the unseen children of a partially expanded node, is replaced mentally by a single "mythical" child.

"Expansion" of a leaf node used to mean: agglomerate all its children onto the tree. It now means:

- If the node is "real:" agglomerate some subset of its children onto the tree, the subset being the next batch of "next most plausible kind of moves" chosen by a move generator that outputs moves in decreasing plausibility order.
- If the node is "mythical:" agglomerate the next batch of its *parent's* children onto the tree (possibly causing the mythical node to disappear, or its evaluation to change).

Thus in chess, captures might be the first batch, checks the second, and the rest are the third and final batch. Expanding only one child at a time (i.e. batch size= 1) is another possibility.

The distribution at a node is obtained by combining (in the usual way) the distributions of its known children, and (if the node is only partially expanded) an additional distribution from the mythical child, and representing the entire set of as-yet unexpanded children. The ESS-value of a leaf node is also obtained in the usual way: "mythical" leaf nodes are treated exactly the same as "real" ones for both these purposes.

One could train evaluation functions to return distributions for the unexpanded children set. Among additional "bin indices" (cf. §2) might be how many batches of children have already been expanded, and the maximum plausibility score among the untried moves.

It is mildly worrysome that we have gotten a little further from the model that "expanding" a node means "replacing its CDF with a single value" which had motivated our influence function and expansion utility computations, but such is life.

[26] It will only suffer by a constant factor, since, if the branching factor is b, then any tree generated by a clever algorithm with partial node expansion, will be contained within the tree gotten by appending all children of every non-leaf tree node, and this latter tree will be at most b times bigger. Still, since in chess $b \approx 30$, this could be a big constant factor!

Also, we remark that it is also possible to regard the operation of calling a more sophisticated (and slower) heuristic evaluator than usual, to evaluate a node (thus gaining more information than usual) as an alternative notion of node "expansion." The usual machinery will then allow one to decide which nodes to apply this operation to. (The question of whether to use the slow evaluator on some node, or to expand it in the usual manner, would have to be decided heuristically and is not one our techniques are going to answer.)

12 Tree-shaping algorithms by previous authors

We now list some tree-shaping techniques, proposed by previous authors, which require storage of the whole tree S.

12.1 Conspiracies

D. McAllester [22] suggested a method of "conspiracies" in the negamax tree-searching philosophy. A set of n leaves of S is called an *n-conspiracy* if some arbitrarily violent change in the evaluation functions at these nodes is possible (i.e. they are not true terminal positions of the game) that would result in a change in the negamaxed value of the root. By a sophisticated algorithm which walks up and down the tree, one may recursively expand a member of the smallest current conspiracy. One continues until either S is unwieldily large, or the *conspiracy number*, i.e. the cardinality of the smallest conspiracy, is greater than some desired value. McAllester's tree-shaping method was found experimentally to work well in tactical chess middlegames by J. Schaeffer [30], but it worked badly in "positional" chess and in forced mates in which moves near the end of the forced mate line were "any"s.

We now remark that large conspiracy number occurs even if one makes no special effort to shape the tree. Uniform trees with branching factor $b \geq 2$, and depth $2k$, (having $L = b^{2k}$ leaves) whose leaves all are losses, have conspiracy number $\sqrt{L}$. We can prove (omitted) that for b-uniform depth-d trees with b fixed and *random* 0-1 values at the $L = b^d$ leaves (i.e. with all 2^L assignments of values being equally likely) that almost surely the conspiracy number will grow more quickly than $L^{1/2-\epsilon}$ where ϵ is any positive real. A similar result holds in Schrüfer's [31] more realistic probabilistic model of b-uniform depth-d game trees with Boolean leaf values. Indeed, Schrüfer's theoretical results can be viewed as proving that, in his model, exponentially large conspiracy numbers occur if and only if negamax search is not pathological. We also conjecture that exponentially large conspiracy numbers will almost surely occur in uniform trees whose leaf values are real numbers ordered according to a random permutation.

McAllester's algorithm had assumed conspiracy number bounded above by a constant. It is possible to modify McAllester's algorithm to allow exponentially large conspiracies, but then, for trees with branching factor $b \geq 4$ and random 0-1 leaf evaluations, the runtime would be a superlinear power of the tree size [proof omitted, but easy].

McAllester only considered leaf-value changes of $\pm\infty$, and did not have a notion that larger changes were somehow less likely than smaller changes. Also in McAllester's approach, as you start to evaluate the members of a conspiracy, you will often partway through find that the values you get are inconsistent with that conspiracy actually changing the root, and you will turn to expanding some different conspiracy. By comparison, note that our measure ESS doesn't just value a leaf in terms of its contribution to one conspiracy, but in terms of its contribution to the whole ensemble average. So in a sense we take account of the integral over all (probabilistically weighted) conspiracies which a leaf participates in, and expand the leaf which is most likely to give useful information as we expand.

12.2 Rivest's suggestion

R. Rivest [27] suggested a method where the "max" in negamaxing is replaced by an L_p mean. The root value then depends *differentiably* on the leaf values. One may find the gradient of the root value with respect to the leaf values, by applying the chain rule. The leaf corresponding to the largest element in the gradient vector, which is presumably the most important one, may then be expanded to obtain a larger tree S. Rivest had an ingenious algorithm which walked up and down the tree and found this most relevant leaf incurring an overhead cost of order db (and as Rivest pointed out this can be reduced further to $O(d)$). Using a poor evaluation function in the game of "connect-4," Rivest found experimentally that his algorithm beat Alpha-beta with constant-depth cutoff when they both examined roughly the same number of nodes, but due to its computational overhead lost to Alpha-beta when a 5 second per turn time constraint was imposed. We speculate that if Rivest had used a better evaluation function, his experimental results would have shifted more toward his favor, since Rivest uses the evaluation function both to value nodes and to decide which nodes to expand, whereas Alpha-beta uses the evaluation function only in the former capacity.

The great advantages of Rivest's algorithm are its simplicity, speed, and its entirely robust behavior when confronted with trees with high conspiracy number. Some drawbacks to Rivest's algorithm are as follows.

1. When deciding which leaves to expand, Rivest only considers the ones which affect the root's value the most, and not the ones which might affect the choice of move. The difference is illustrated by a tree in which there is only one legal move. Rivest's algorithm would do a large search[27].

2. Without any probabilistic information about leaves, Rivest cannot realize that "quiet" leaves are unlikely to change value on further expansion, while "noisy" leaves will change value substantially.

3. Rivest's method for deciding which leaf to expand is ad hoc and does not try to incorporate estimates of the results of future expansion.

12.3 Russell and Wefald

S. Russell and E. Wefald ([28] chapter 4) present a game-tree searching algorithm called "MGSS*." This algorithm involves many ideas similar to our own, and helped to stimulate our research. Russell and Wefald use a measure of the "utility" of expanding leaves versus the cost of the time required to do so, to decide when to stop searching. Their evaluation function returns both a value and a variance. They then approximately propagate a normal distribution (of appropriate mean and variance) up the tree from each leaf to value its expansion utility in a greedy approximation. They grow a tree by expanding at each step the leaf with highest "utility". In Othello tournaments between MGSS* and Alpha-beta ("AB") using a high quality evaluation function, Russell and Wefald found the following results

	wins	nodes	sec.
MGSS*	24.5	3666	40
AB[2]	7.5	2501	23
MGSS*	22	6132	68
AB[3]	10	9104	82
MGSS*	20	12237	170
AB[4]	12	42977	403
MGSS*	16.5	21155	435
AB[5]	15.5	133100	1356
MGSS*	17	45120	1590
AB[6]	15	581433	6863

Russell/Wefald results. (Note: the number in []'s is the depth of AB search.)

[27] Of course, Rivest could test for this particular special case. The possibility of this quick fix in no way detracts from our larger point that Rivest's statistical goal is the wrong one.

which seems to be an impressive victory for MGSS*. Unfortunately, we've recently found out from Russell that MGSS*'s Alpha-beta opponent in these experiments had move ordering turned off. An AB[6] Othello program that we wrote, which *does* have decent move ordering heuristics, does ≈ 130000 board evaluations per game. So assuming our count of "board evaluations" is roughly the same as Russell's count of "nodes," a better move ordering for Russell could have gotten AB[6] quality information in AB[5] time, which would make MGSS* and AB[] about equal at equal time budgets, with MGSS* still having the advantage at equal node counts. While this is less impressive than it at first seemed, it still gives us grounds for optimism about BPIP, since BPIP's runtime only grows linearly with node count.

Russell and Wefald also experimented with MGSS2, a version permitting partial node expansion, and got good preliminary results (table p109 of [28]). The logic behind MGSS2 seems to us seriously flawed, however. We omit discussion here.

Russell and Wefald's MGSS* results are evidence for the power of the ideas of "expansion utility" and the use of probability distributions at leaves to describe value changes after expansion. Since MGSS* considers only one node at a time, it is perhaps not as susceptible as our proposal to difficulties arising from node corellations. A critique of MGSS* as compared with the algorithm of the present paper follows.

1. The worst case run time of their 1-at-a-time leaf expansion scheme is apparently a quadratic function of tree size n, and its typical run time behaves as a good fit to $n^{1.5}$ (based on data in the table above).

2. MGSS* only approximately propagates probability distributions.

3. MGSS* decides on its move choice, after it has finished growing the tree, based simply on the negamax valuation where single values are propagated.

4. MGSS* assumes the leaf distributions are normal, when experimentally we have found multimodal distributions.

5. MGSS* becomes completely helpless (terminates) if the move choice conspiracy number ever exceeds 1. Furthermore we have argued in §7.4 that the metagreedy approximation they make is a poor guide to the leaf relevance even when the conspiracy number is 1.

12.4 Palay

Palay [25] proposed a probability based method propagating staircase CDFs with the same formula (3) we use. However, Palay [p. 46] only allows his stairsteps to have abscissae selected from a fixed set X of 20 points, $X = \{x_1, x_2, \ldots x_{20}\}$, which he did, according to a private communication, to "limit the storage requirements," apparently not realizing that the total space requirement was small §4; and he also performs an interpolation procedure [p. 47] based on the error function to smooth out his CDFs between stairsteps.

Berliner[7] had proposed the B* search which grew a search tree in an attempt to *prove* that some move is strictly better than all others. The leaf evaluator returned (lower bound, upper bound) intervals. Berliner used heuristics to decide between attempting to "prove best" and attempting to "disprove rest." Palay's first proposal[24] was to use probabilistic information in place of such heuristics to guide the search for such a proof, and he experimentally showed that his use of such information led to improvements over a similar proof-seeking algorithm which did not use probabilistic information.

Later in [25] (see p. 72), Palay relaxed his goal of proving that some move was better than all the alternatives, replacing it with the goal of stopping searching when the confidence that some move "dominated" all the alternatives, exceeded some fixed threshhold. He then considered [p. 75] the possibility of decreasing this threshhold from 1 to 0 as more time was expended, which would force eventual termination

Our point of view is rather different than Palay's. Palay does not use the BPIP valuation philosophy. For example, if forced to terminate search before one move 'dominates', he proposes (p75) heuristics for move choice different than choosing the move with highest mean value. His search is motivated by an attempt to gain confidence that he is choosing the best move, and not by any decision theoretic notion of utility. A game in which there are two equally good best moves illustrates the difference in goals. This can easily happen if two lines of play transpose. Palay's procedure is oblivious to the lack of utility in attempting to prove one line is better than the other[28]. The present paper's method would soon realize that if one of the moves were better, it could only be an amount smaller than the time cost of ascertaining which, and hence it would terminate.

Incidentally there is an amusing technical pitfall in demanding that the probability that some move "dominates" another be greater than some threshhold x. Let $g = (\sqrt{5}-1)/2 \approx 0.618$ and consider the three probability distributions: A is 0 with probability 1, B is -2 with probability $1-g$ and $+1$ with probability g , and C is -1 with probability g and $+2$ with probability $1-g$. Then C dominates B with probability g, B dominates A with probability g, and A dominates C with probability g! This is a dramatic illustration of the fact that one should take the Bayesian viewpoint, ignore notions of "dominance probability," and ask for the move with largest expectation value.

Another drawback to Palay's approach is the following. Rather than determine which leaf of the tree was best to expand (under his criteria), Palay marched down from the root at each node choosing the child most likely to be best until he reached a leaf. As he realized (p.13 & 85 [25]) this greedy procedure need not pick the globally best leaf, nor even a good approximation. Palay was forced into this expedient, however, because examining all the leaves to find the best would have caused his runtime to grow superlinearly. We avoided such problems by using gulps.

Palay [25] implemented his algorithm, which he called "PSB*", to solve tactical chess problems. He compared it to Belle, a 130 Knode/sec Alpha-beta machine [10]. In order to get a fair comparison, considering the different platforms, Palay "projected" the performance PSB* would have if implemented into 567× faster (Belle-speed) hardware. His conclusion was "the PSB* algorithm is not able to perform at the same level as Belle; however it is not far behind."(p149).

13 Conclusion: Doing the wrong thing superbly well

The Shannon approach to playing games involves growing a large subtree of the full game tree, evaluating the leaves, and using negamax. We have improved this by improving the evaluation at the leaves, improving the propagation scheme[29], and improving the tree growth scheme. We do this with little computational overhead. The biggest disadvantage of our approach over the standard one is our storage requirements, but we have outlined (§11) how these requirements can be drastically reduced, at the cost of a constant factor slowdown and some extra programming effort, by "2-stage" or "hybrid" ideas.

For experimental results on a variety of games see the companion paper [34].

We conclude with the remark that this still has little to do with how humans play games. The computer science approach (which we are attempting to perfect) has since Shannon basically regarded a game as defined by its game tree. But what makes a game interesting is that it has a low complexity, algorithmically efficient description apart from the game tree. For example, Go is defined by 9 rules on an n by n board. Any procedure which only accesses the underlying simplicity of a game in the form of an evaluation function is inherently doing the wrong thing. One can exhibit chess positions in which reasonable evaluation functions will not notice progress for 20 ply, but which are soluble by human novices. The main open question is how to go beyond the evaluation function picture of games.

[28] See footnote 27.

[29] To gain intuition about when BPIP propagation is likely to beat the standard negamaxing approach "AB," imagine the BPIP distributions at the N children of the root. For the sake of the present argument suppose these distributions look qualitatively like normal distributions: they are unimodal and have exponentially dying tails. In that case, BPIP will yield better performance than AB if the separations between the means of the distributions of the two best children are small compared to their peakwidths, i.e. if the peaks overlap a lot. In such a case there is a chance of order one half that AB will suggest the wrong BPIP move. BPIP would also perform better than AB if the distributions weren't unimodal. On the other hand, when the separations between peaks are much larger than the peakwidths, BPIP and AB will usually suggest the same moves.

A naive reader might imagine that our approach does not gain the speedup effect of "Alpha-beta pruning." This is an illusion: if we use utility-guided tree growth, we get similar, and arguably better, pruning effects, and anyway Alpha-beta also takes runtime linear in the number of (non-cut-off) leaves which it actually examines, just like our approach.

Acknowledgements: We thank C.W. Gear for assistance with the proof of the lemma in Appendix B.

References

[1] T. Anantharaman, M. Campbell, F. Hsu: Singular extensions; adding selectivity to brute force searching, Artificial Intelligence 43 (1990) 99-109

[2] Thomas S. Anantharaman: A Statistical Study of Selective Min-Max Search in Computer Chess, (PhD thesis, Carnegie Mellon University, Computer Science Dept.) May 1990, CMU-CS-90-173

[3] Thomas S. Anantharaman: Extension heuristics, ICCA Journal 14,2 (June 1991) 47-65.

[4] E. B. Baum: On optimal game tree propagation for imperfect players. In Proceedings of the Tenth National Conference on Artificial Intelligence. American Association for Artificial Intelligence 1992.

[5] E. B. Baum: How a Bayesian approaches games like chess. In Games: Planning and Learning, Papers from the 1993 Fall Symposium, Technical Report FS-93-02, AAAI Press, Menlo Park CA. (1993) pp 48-50.

[6] D.F. Beal: A generalized quiescence search algorithm, Artificial Intelligence 43 (1990) 85-98

[7] Hans Berliner: The B* search algorithm: A best first proof procedure, Artificial Intelligence 12 (1979) 23-40

[8] M. Blum, R. Floyd, V. Pratt, R. Rivest, R. Tarjan: Time Bounds for selection, J. Computer System Sci. 7 (1973) 448-461

[9] M. Campbell: The graph-history interaction; on ignoring position history, Proc. ACM National Conf. (1985) 278-280.

[10] J.H. Condon and K. Thompson: Belle chess hardware, 45-54 in Advances in computer chess 3, M.R.B. Clarke ed. Pergamon 1982.

[11] Chr. Donninger: Null move and deep search, ICCA Journal 16,3 (Sept. 1993) 137-143.

[12] O. Hansson and A. Mayer: Heuristic search as evidential reasoning. In *Proceedings of the fifth Workshop on Uncertainty in Artificial Intelligence*, Windsor, Ontario.

[13] D. S. Nau: Pathology on game trees revisited, and an alternative to minimaxing, AI 21 (1983) 224-244.

[14] Chi, P-C, D. S. Nau: Comparison of the Minimax and Product Back-up Rules in a Variety of Games, in *Search in Artificial Intelligence*, eds. L. Kanal and V. Kumar, Springer Verlag, New York,(1989) pp451-471.

[15] R. Floyd and R. Rivest: Expected time bounds for selection, Commun. ACM 18,3 (March 1975) 165-173

[16] I.J. Good: A five year plan for automatic chess. Machine Intelligence 2 (1968) 89-118

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

[18] Feng-hsiung Hsu: Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess, (PhD thesis) Tech. Rept. CMU-CS-90-108 (Feb 1990) School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213

[19] Han La Poutre and Warren D. Smith: Approximation of staircases by staircases, Work in progress, NECI, 4 Independence Way, Princeton NJ 08540.

[20] Kai-Fu Lee and Sanjoy Mahajan: The development of a world class Othello program, Artificial Intelligence 43 (1990) 21-36

[21] Kai-Fu Lee and Sanjoy Mahajan: A pattern classification approach to evaluation function learning, Artificial Intelligence 36 (1988) 1-25

[22] D.A. McAllester: Conspiracy numbers for min max search, Artificial Intelligence 35 (1988) 287-310

[23] Monroe Newborn: A hypothesis concerning the strength of chess programs, ICCA Journal 8,4 (1985) 209-215.

[24] A.J. Palay: The B* search algorithm – new results, Artificial Intelligence 19 (1982) 145-163.

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

[26] J. Pearl: Heuristics, Addison-Wesley 1984

[27] R.L. Rivest: Game tree searching by min max approximation, Artificial Intelligence 34 (1988) 77-96

[28] S. Russell and E. Wefald: Do the Right Thing, MIT Press 1991 (see especially chapter 4)

[29] J. Schaeffer: Conspiracy numbers, Artificial Intelligence 43 (1990) 67-84

[30] J. Schaeffer: Experiments in search and knowledge, PhD thesis, Dept. of Computer Science, University of Waterloo, 1986.

[31] G. Schrüfer: Presence and Absence of Pathology on Game Trees, in D.F. Beal, ed., *Advances in Computer Chess* **4**, (Pergamon, Oxford, 1986) pp 101-112.

[32] C.E. Shannon: Programming a computer for playing chess, Philos. Magazine 41,7 (1950) 256-275

[33] Warren D. Smith: Fixed point for negamaxing probability distributions on regular trees, NECI technical report

[34] Warren D. Smith, Charles Garrett, Eric Baum, Rico Tudor: Best Play for Imperfect Players and Game Tree Search; part II - experiments.

[35] Alex & Barbara Szabo: The technology curve revisited, ICCA Journal 11,1 (1988) 14-20

[36] Robert E. Tarjan,: Data structures and network algorithms, SIAM 1983

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

14 Appendix A: Numerical example

Consider the tree of figure A1A.

```
     R                  -3                  -3                  4
  A    B    C        6      5      3      6      5      3     -4     -3     -1
 D E  F G  H I     4 -6   -5  0   1 -3   9 -6   -5  0   5 -3   4  7   6  3   1  8

                        0.13577              0.04099             0.00050
Figure A1A.                          Figure A1B.
```

If the leaves $\{D, E, F, G, H, I\}$ are assigned the 2-spike probability distributions from the lines labeled D-I of table 1 below (note: $-3_{0.95}\ 8_{0.05}$ means "-3 with probability 0.95 and $+8$ with probability 0.05"), then the remaining nodes R,A,B,C will acquire the BPIP probability distributions in the first 4 lines of that table and a 64-tree ensemble springs into being. ($64 = 2^6$, the "2" arising since each leaf has 2 spikes and the "6" because there are 6 leaves.) Three elements of this ensemble are shown in figure A1B above, along with their corresponding probabilistic masses. (Thus $0.13577 = .62 \times .67 \times .65 \times .79 \times .67 \times .95$ and $0.04099 = .38 \times .67 \times .65 \times .79 \times .33 \times .95$.)

Node	Probability Distribution	Mean	Negamax of leaf means
R	$-3_{0.41373}\ 0_{0.17599}\ 1_{0.02080}\ 3_{0.04843}\ 4_{0.20122}\ 5_{0.01443}\ 7_{0.12540}$	0.67977	-1.15000
A	$-7_{0.1254}\ -4_{0.2046}\ 6_{0.6700}$	**2.3238**	1.7100
B	$-3_{0.0735}\ 0_{0.2765}\ 5_{0.6500}$	3.0295	**1.1500**
C	$-5_{0.0165}\ -1_{0.0335}\ 3_{0.9500}$	2.7340	2.4500
D	$4_{0.62}\ 9_{0.38}$	5.90	5.90
E	$-6_{0.67}\ 7_{0.33}$	-1.71	-1.71
F	$-5_{0.65}\ 6_{0.35}$	-1.15	-1.15
G	$0_{0.79}\ 3_{0.21}$	0.63	0.63
H	$1_{0.67}\ 5_{0.33}$	2.32	2.32
I	$-3_{0.95}\ 8_{0.05}$	-2.45	-2.45

Table 1: BPIP probability distributions and mean values (and negamax values)

Notice (from the mean values of the BPIP distributions, giving the expectation values of each nonroot node with BPIP play) that the best move, according to BPIP, is "A." However, negamaxing the mean values of the leaves would tell us that the best move is "B."

The utility of expanding all the leaves is 3.00357, as we may see from the first entry of table 2 below giving four different measures (cf. §7) of the relevance of expanding each of the 64 possible subsets of the leaves. This particular entry could also have been calculated with the aid of the theorem (§7.4) that $U_{\text{all leaves}} = \langle R\rangle + \langle A\rangle$, where A is the favorite move, i.e. $3.00357 = 0.67977 + 2.32380$.

We may also see from this table that, e.g. the best leaf L to expand (largest U_L) if we can only expand one, is $L = E$, and if we can only expand 2 leaves, they should be E and F since $U_{EF} = 2.97708$ is the largest utility among all 2-leaf sets. Similarly the uniquely best 3-leaf, 4-leaf, and 5-leaf sets are EFI, EFGI, and DEFHI.

Note that the best 5-leaf set does not contain the best 4-leaf set. This is a symptom of the fact that no simple "additive relevance" generally exists which orders the relevances of leaf subsets correctly. To make this precise: we claim that there do not exist 6 real numbers $\{D, E, F, G, H, I\}$, which obey the inequalities $E+F+H > F+G+I$, $E+G+H > D+F+I$, et cetera corresponding to the inequalities $U_{EFH} > U_{FGI}$, $U_{EGH} > U_{DFI}$ obeyed by the U_S in the table, where we ignore nonstrict inequalities and only use sets S of size 3. This is easily shown by solving this linear program with the simplex algorithm. The analagous linear programs arising from either $\not{U}$, δ, or V in place of U are also infeasible.

The best strategy for expanding 2 leafs *sequentially* (cf. Appendix D), is different from the best strategy (EF) if we have to choose the pair. This is because if, when we expand E, we find it was 7, then A definitely becomes a better move than B, the only question is whether it is better than C. Hence B's descendants F and G would be utterly irrelevant at that point, and the best second node to try would be H (which at that point actually would be unable, acting alone, to alter the best move, but it could do so by cooperating with D and I and at any rate would be the only leaf which could even affect the value of C at all). On the other hand, if E turned out to be -6, then A would definitely not be the best move, and it turns out that the best leaf to expand at that point to help in deciding between B and C would be F. Thus in summary, the best 2-leaf sequential strategy is E, then either F or H, depending on the result.

One weakness of the SLEU measure U_L of leaf expansion relevance is made evident by the fact that two of our leaves, D and H, have zero SLEU, but in fact, by acting in cooperation with other leaves, D and H *can* affect the move choice. In the present example the other three measures $\not{U}_L$, V_L, and δ^L are never zero. In our experience $\not{U}_L$ is zero less often than U_L, but V_L and δ^L are zero even less often still. (Zero is not necessarily bad per se; all four measures are zero – correctly – if applied to leaves which can't ever affect move choice.)

But of these four measures, V_L is the only one we know how to compute efficiently for large trees, which is why we recommend it.

You may have observed that $\delta^S = V_S$ whenever S was a single leaf (emboldened lines of table 2). This is because our approximation theorem (§7 and Appendix B) assures us that the approximation $\delta^S \approx V_S$ will be exact for leaves with 2 spikes. If leaf G's probability distribution were replaced by the 3-spike distribution $0_{0.79}, 3_{0.11}, 6_{0.10}$, then we would have had $V_G = 0.33926 \neq \delta^G = 0.34241$.

In fact, this approximation seems to be remarkably good for all 64 S, and that, empirically, seems to be common. This empirical observation motivated us to see that the approximation theorem of §7.4 may be extended to cover the cases where $|S| > 1$ (see the end of appendix B), which explains this.

The influence coefficients for Q^A (which is the same as $U_{\text{all leaves}}$ since the favorite move is A) for each probability mass in each node's distribution, are in table 3 below.

Let's check to see that we are getting the right answers. We can calculate $U_{\text{all leaves}} = 3.0036$ from node A: $3.0036 = .2046 \times 0.0165 + 0.6700 \times 4.4779$, or from node H: $3.0036 = 0.67 \times 2.9576 + 0.33 \times 3.0969$.

Suppose we perturb the distribution in node G by changing the probability 0.79 to 0 and of course also changing the probability 0.21 to 1, which is what would happen were we to expand G and discover its value was 3. In that case, the predicted change in Q^A is $\Delta Q^A = 0.79 \times 3.5440 - 0.79 \times 2.8599 = .5404$. Upon recalculating table 3 for the entire (new) tree we find that $Q^A_{\text{new}} = 3.5440$, which sure enough is $3.0036 + .5404$. This sort of checking procedure is very useful for debugging one's influence coefficient code. In the present case, this perturbation happened to change the BPIP favorite move from A to B so that $\Delta Q^A \neq \Delta U_{\text{all leaves}}$. The new value of $U_{\text{all leaves}}$ is $Q^B_{\text{new}} = 3.4202$. This *increase* in the utility of future expansion caused by an expansion is ridiculous and shows the silliness of using a priori instead of a posteriori estimates. This does not really mean that the utility of expansion has gone up by .4166; rather it means that our previous estimate of the utility of expansion, 3.0036, was too low. It should have been 3.5440, i.e. Q^A given that the value of G is 3. The *a posteriori* expected utility of future expansion has in fact gone down by .1238 since our favorite move has changed.

15 Appendix B: Proof of ESS Approximation Theorem

Theorem. Let δ^L denote the ensemble expectation of how much knowing the exact value of leaf L allows us to improve our prediction of Q^1, given that we have expanded all leaves but L. Then

$$2V_L \geq \delta^L \geq V_L. \tag{22}$$

Furthermore, if n, the number of spikes in the probability distribution at leaf L, is 2, then $V_L = \delta^L$. Also, if $p_1 + p_n = \alpha$, then $V_L \alpha^{-1} \geq \delta^L$.

(For definitions of Q^i, V_L, $\sum_C$, etc. see §7. "How much we can improve our prediction of Q^1 by expanding L" = "how much Q^1 changes when we expand L" = $|\Delta_L Q^1|$.)

Proof. We omit throughout the superscript L on δ^L to improve readability. Let C be some configuration of all the leaf values in the ensemble. Let

$$Q^1_C(x^L) \equiv \text{Amount best move is better than move 1 by in configuration } C \tag{23}$$

In this definition Q^1_C depends on the state C of all the leaves, and we have made explicit its dependence on the value x^L of leaf L by writing it as $Q^1_C(x^L)$. We define $Q^1_C(x^L_i)$ to be the value of $Q^1_{C'}$ where C' is the configuration C except that leaf L has value x^L_i.

We are interested in the error we would make in predicting $Q^1_C(x^L)$ if we knew all leaf values other than L, and averaged over the value of leaf L. We call this error δ_C. Then

$$\delta_C \equiv |Q^1_C(x^L) - \sum_{i=1}^{n} p_i Q^1_C(x^L_i)| \tag{24}$$

The expectation value of δ is

$$\delta = \sum_C |Q^1_C(x^L) - \sum_{i=1}^{n} p_i Q^1_C(x^L_i)| = \sum_i p_i {\sum_C}' |Q^1_C(x^L_i) - \sum_{j=1}^{n} p_j Q^1_C(x^L_j)| \tag{25}$$

$$\geq \sum_i p_i | {\sum_C}' (Q^1_C(x^L_i) - \sum_{j=1}^{n} p_j Q^1_C(x^L_j))| = \sum_i p_i | {\sum_C}' Q^1_C(x^L_i) - Q^1_{C'}| = V_L \tag{26}$$

where by ${\sum}'_C$ we mean a sum over the configurations of all leaves but leaf L. This proves the lower bound.

Define

$$f(x^L_i, C) \equiv p_i \left(Q^1_C(x^L_i) - \sum_{j=1}^{n} p_j Q^1_C(x^L_j) \right) \tag{27}$$

Then we have the following facts.

(1) $f(x^L_i, C)/p_i$ is for any given leaf L, either monotonically decreasing in i or monotonically increasing in i. Which of these two it is depends only on the leaf and not on the configuration C. In fact it depends only on whether the player on move is the same or opposite at L and at the root, and on whether L is a descendant of move 1 or of some other child of the root. The reason why this is true is because of the monotonic nature[26] of the influence functions for minimax. Without loss of generality we will from now on assume f is monotonically increasing.

(2) $\sum_{i=1}^{n} f(x^L_i, C) = 0$.

Now there will generally be some set of configurations C_0 for which $Q^1_C(x^L_i)$ is independent of i, due to Alpha-beta cutoffs, and for these $f(x^L_i, C) = 0$. Then define the subset of configurations C_1 as those for which $f(x^L_1, C) < 0$ and $f(x^L_2, C) \geq 0$. similarly define the subset of configurations C_j for $j = 2, \ldots, n-1$ for which $f(x^L_j, C) < 0$ and $f(x^L_{j+1}, C) \geq 0$. For $i = 1, \ldots, n$ and $j = 1, \ldots, n-1$ let

$$A_{ij} \equiv \sum_{C \in C_j} f(x^L_i, C) \tag{28}$$

Then it follows that

$$\delta = \sum_{i,j} |A_{ij}| \tag{29}$$

and

$$V_L = \sum_i |\sum_j A_{ij}| \tag{30}$$

Also from facts (1) and (2) we have that for all j, $\sum_i A_{ij} = 0$ and for all j, A_{ij}/p_i is monotonically increasing. By the Lemma soon to come we conclude $2V_L \geq \delta$.

When $n = 2$ there is only one column in A and evidently $V_L = \delta^L$. If $p_1 + p_n = \alpha$, then from the monotonicity property (1) we have: $\sum_j |A_{1j}| + \sum_j |A_{nj}| \geq \alpha \sum_{ij} |A_{ij}|$ and we conclude $V_L \alpha^{-1} \geq \delta^L$. Q.E.D.

Lemma. For any matrix A_{ij} with $n+1$ rows and n columns, with A_{ij} not positive for $j < i$ and not negative for $i \geq j$, having the properties: (1) for all j, $\sum_i A_{ij} = 0$ and (2) there exists a vector $p_i > 0$ such that for all j, A_{ij}/p_i is monotonically increasing, it holds that

$$\sum_i |\sum_j A_{ij}| \geq \frac{1}{2} \sum_{i,j} |A_{ij}| \tag{31}$$

and there exist nontrivial matrices which approach this bound.

Proof. There exists a row I such that for $i > I$, $\sum_j A_{ij}/p_i \geq 0$ and hence $\sum_j A_{ij} > 0$, and for $i \leq I$, $\sum_j A_{ij} \leq 0$. Let J=I−1. Then the matrix A can be divided up as in figure 1.

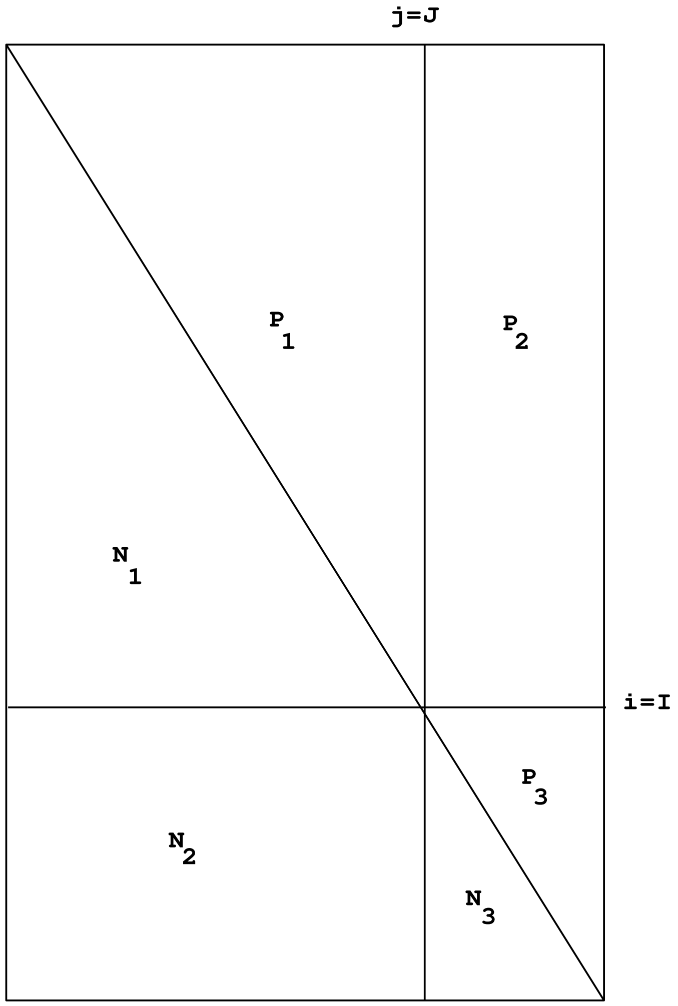

Figure 1: A schematic of the matrix A_{ij}.

We have marked regions in the matrix in the figure. All elements in regions marked P_i are non-negative and all elements in regions marked N_i are non-positive. Let P_i denote the sum of all the elements in the region marked P_i, and N_i denote the negative of the sum of all the elements in the region N_i. We have

$$\sum_{ij} |A_{ij}| = P_1 + P_2 + P_3 + N_1 + N_2 + N_3 \tag{32}$$

and

$$\sum_i |\sum_j A_{ij}| = P_1 + P_2 - N_1 + N_2 + N_3 - P_3 \tag{33}$$

Now observe that

$$P_1 = N_1 + N_2 \tag{34}$$

and

$$N_3 = P_1 + P_2 \tag{35}$$

because the column sums are zero. Hence

$$\sum_i |\sum_j A_{ij}| = 2N_2 + 2P_2 \tag{36}$$

and

$$\sum_{ij} |A_{ij}| = 2N_2 + 2P_2 + 2N_1 + 2P_3 \tag{37}$$

But now we observe that $P_2 \geq N_1$. This follows because the sum of lowest row in N_1 plus the sum of the lowest row in P_2 is no less than zero. Hence by the monotonicity the sum of any row in N_1 is no greater in absolute value than the sum of the same row in P_2. Likewise $N_2 \geq P_3$. Hence

$$\sum_i |\sum_j A_{ij}| \geq \frac{1}{2} \sum_{i,j} |A_{ij}| \tag{38}$$

To saturate this consider the matrix

$$\begin{bmatrix} X(1+p_3/p_2) & Xp_1/p_2 \\ -X & X \\ -Xp_3/p_2 & -X(1+p_1/p_2) \end{bmatrix} \tag{39}$$

in the limit as X goes to infinity, p_3 and p_1 go to zero, with $p_2 = 1$. Q.E.D.

Extension: We have recently realized that this theorem may be extended to show that

$$2V_S \geq \delta^S \geq V_S \tag{40}$$

for all leaf-subsets S, not just those of cardinality 1. We use the definitions of V_S and δ^S in §7.4, equations 18 & 19. Fact 1 has to be rewritten so that $f(x_i^L, C)/p_i$ is replaced by $f(S_i, C)/p_i$ where leaf set S assumes value-vector number i with probability p_i and the value-vectors are numbered in the correct order for the desired monotonicity properties to hold.

16 Appendix C: Pseudocode for BPIP Algorithms, including computing the ESS

By Charles Garrett[30].

Our experimental code implementing BPIP has diverged significantly from the pseudo-code in the older BPIP paper drafts. Herein, I describe what the current version of the code is doing.

[30]Authors's note: Charles Garrett, in the course of programming work on part II, simplified and corrected the pseudocode we had originally provided in the draft version of this paper, so with his permission we are incorporating his pseudocode in this appendix. Certain details which are important for replacing factors of b by $\log b$ in the runtime analysis of §9, but unimportant for understandability, have been supressed and are buried inside lines such as "find jump in child giving rise to ith jump in parent." Suffice it to say that by the use of log time priority queues ("heaps")[36], binary-merge-sort-like algorithm structure, and/or storage of appropriate pointers, such things could be accomplished in $O(\log b)$ time instead of the $O(b)$ of brute sequential search. For games with small b, it may not be worth bothering.

16.1 Influence

The negamax formula for CDF's is:

$$\underline{c}^{(\text{parent})}(-x) = \prod_{i=1}^{n} \overline{c}^{(\text{child}_i)}(x) \tag{41}$$

where n is the number of children, $\underline{c}$ is the probability that the node has a value $\leq x$ and $\overline{c}$ is the probability that the node has a value $\geq x$.

What is the impact of changing child a's CDF by δ at the position x_0? First, $\overline{c}^a(x)$ is increased by δ for $x \leq x_0$. This causes a change at the corresponding jump j at $-x_0$ in the parent's CDF.

$$\begin{aligned}
\Delta p^{(\text{parent})}(-x_0) &= \Delta \underline{c}^{(\text{parent})}(-x_0) && (42)\\
&= (\overline{c}^a(x_0) + \delta) \prod_{i \neq a} \overline{c}^i(x_0) - \prod_{i} \overline{c}^i(x_0) && (43)\\
&= \delta \prod_{i \neq a} \overline{c}^i(x_0) && (44)\\
&= \delta \frac{\underline{c}^{(\text{parent})}(-x_0)}{\overline{c}^a(x_0)} && (45)
\end{aligned}$$

Next, we have to consider the effect on jumps in the parent at locations $x < x_0$. Note that when we say $x < x_0$, we include jumps at $x = x_0$ arising from children that come earlier according to whatever tie-breaking scheme we adopt. Those jumps which were caused by the same child a are unchanged, because the δ terms cancel. But jumps caused by other children are affected. Consider a jump in the parent at x caused by child $b \neq a$. The jump sizes before and after increasing $p_i^a(x)$ by δ are:

$$\begin{aligned}
p^{(\text{parent})}(-x) &= p^b(x) \prod_{i \neq b} \overline{c}^i(x) && (46)\\
p'^{(\text{parent})}(-x) &= p^b(x)\,(\overline{c}^a(x) + \delta) \prod_{i \neq a,b} \overline{c}^i(x) && (47)
\end{aligned}$$

and so the change in the parent jump size is:

$$\begin{aligned}
\Delta p^{(\text{parent})}(-x) &= \delta p^b(x) \prod_{i \neq a,b} \overline{c}^i(x) && (48)\\
&= \delta \frac{p^{(\text{parent})}(-x)}{\overline{c}^a(x)} && (49)
\end{aligned}$$

Combining one term of the form (45) with a summation over terms of the form (49), we have the total influence coefficient for jump j in the child a:

$$I_a^j = I_{\text{parent}}^j \frac{\underline{c}^{(\text{parent})}(-x_j)}{\overline{c}^a(x_j)} + \sum_{\substack{k,\, x_k < x_j, \\ \text{due to non}-a}} I_{\text{parent}}^k \frac{p^{(\text{parent})}(-x_k)}{\overline{c}^a(x)} \tag{50}$$

which is equation 7 of §6.

In the actual program, we store the value of $\underline{c}(x)$ at each search tree node. When we need $p(x)$ or $\overline{c}(x)$, we take advantage of the fact that our probability distributions are discrete and calculate:

$$\begin{aligned}
p(x) &= \underline{c}(x) - \underline{c}(x_{\text{prev}}) && (51)\\
\overline{c}(x) &= 1 - \underline{c}(x_{\text{prev}}) && (52)
\end{aligned}$$

where x_{prev} is the abscissa to the left of x. If there is no abscissa to the left of x, then $\underline{c}(x_{\text{prev}}) = 0$.

I have been trying to create pseudo-code for the BPIP algorithms, but I usually find that if it is "pseudo", then it sweeps many things under the rug. Therefore, I am currently trying to include the real C++ code with some formatting and a few changes to enhance clarity.

Here are the basic data structures to which the code refers:

```
struct Stair {
        float abscissa;
        float cdf;
        float prob;
        float inco;
};
```

A Stair represents one jump j, where abscissa is x_j, cdf is $\underline{c}_j$, prob is p_j, and inco is the influence coefficient of the jump.

```
struct Distribution {
        unsigned int n;
        Stair* s;
};
```

A Distribution is a probability distribution or CDF having n jumps, which are stored in the array s.

```
struct Tree {
public:
        Distribution dist;
        byte nchild;
        Tree* child;
        double ESS;
};
```

A Tree is one node (internal or external) of a search tree. dist is the staircase distribution for the position, nchild is the number of children in the child array or 0 if this is a leaf node and ESS is the expected step size if this is a leaf node.

The following procedure is passed nchild Distributions and calculates the Negamax of them:

```
void Negamax(Distribution* result, int nchild, Distribution** d) {
      double last_cdf[MAXBRANCH] = {1, 1, ...  1};
      int result_len = Total number of abscissas in all children;
      /* Allocate the space for the parent jumps */
      Stair* ParentJumps = new Stair[result_len];
      /* prod is the current c_(parent) */
      double prod = 1.0;
      for (Each jump i in parent) {
          Stair ChildJump = Jump in child giving rise to the ith jump in the parent;
          int index = Index of child containing ChildJump;
          ParentJumps[i].abscissa = -ChildJump.abscissa;
          ParentJumps[i].cdf = prod;
          /* As we move one step in the child its contribution to the parent cdf changes from
          c̄(x_prev) to c̄(x), or if we have reached the last jump in the child, then the
          entire product goes to 0. */
          if (ChildJump is not the last jump in the child) {
             prod *= (1.0 - ChildJump.cdf) / last_cdf[index];
```

```
            last_cdf[index] = 1.0 - ChildJump.cdf;
        } else {
            prod = 0.0;
        }
    }
    /* Copy the parent distribution into the current Distribution object where it will be stored. */
    result->n = result_len;
    result->s = ParentJumps;
}
```

PropInfluence computes the influence coefficients of child jumps once the influence of the parent jumps are known:

```
void PropInfluence(Distribution* parent, int nchild, Distribution** d) {
    /* from_later[i] will contain the total influence of jumps to the right (i.e. the second
    summand in eqn. (50)) */
    double from_later[MAXBRANCH] = {0, 0, ...  0};
    double last_cdf[MAXBRANCH] = {1, 1, ...  1};
    /* Work from largest to smallest parent abscissa */
    for (Each jump i in parent) {
        Stair ChildJump = Jump in child giving rise to the ith jump in the parent;
        int index = Index of child containing ChildJump;
        ChildJump.inco = parent->s[i].inco * parent->s[i].cdf /
                         last_cdf[index] + from_later[index];
        for (Each child c of the parent) {
            if (c did not cause jump i in parent) {
                from_later[c] += parent->s[i].inco * parent->s[i].prob /
                                 last_cdf[c];
            }
        }
        last_cdf[index] = 1.0 - ChildJump.cdf;
    }
}
```

16.2 Utility

The utility of expansion, $U_{\text{all leaves}}$ is "the probability we will win, if before moving we expand all leaves until they are terminal positions, minus the probability we will win if we move immediately with no further expansion." If there are just two moves from the root, the favorite and the unfavorite, with probability distributions, D_{f} and D_{u} (these distributions are measured with respect to the person on move at the root), we have:

$$\langle U_{\text{all leaves}} \rangle = \sum_{j \in D_{\text{f}}} \sum_{k \in D_{\text{u}}, x_k > x_j} (x_k - x_j)\; p_j\; p_k \tag{53}$$

or, in words, the sum over all possible ways for the unfavorite child to be better than the favorite, times the probability of that event.

The influence coefficient of a jump j or k is the factor multiplying p_j or p_k in (53):

$$I_{\text{f}}^{j} = \sum_{k \in D_{\text{u}}, x_k > x_j} (x_k - x_j)\; p_k \tag{54}$$

$$I_{\text{u}}^{k} = \sum_{j \in D_{\text{f}}, x_j < x_k} (x_k - x_j)\; p_j \tag{55}$$

In practice, we will have one favorite move with distribution $\overline{c}^{\mathrm{f}}$ and multiple unfavorite moves, $\overline{c}^{\mathrm{u}_1}$, $\overline{c}^{\mathrm{u}_2}$, ..., and the distributions will be taken with respect to the person on move after 1 ply of search. We can reduce this situation to the two move case and swap the person on move using the negamax rule:

$$D_{\mathrm{f}}(-x) = \overline{c}^{\mathrm{f}}(x) \tag{56}$$

$$D_{\mathrm{u}}(-x) = \prod_i \overline{c}^{\mathrm{u}_i}(x) \tag{57}$$

If there are m jumps in the favorite distribution, and n jumps in the unfavorite, the utility equation, (53), threatens to consume $O(mn)$ time since it contains two nested summations. But we can easily change this to require only $O(\max(m,n))$ time.

$$\sum_{j\in D_{\mathrm{f}}}\sum_{k\in D_{\mathrm{u}},x_k>x_j}(x_k-x_j)\;p_j\;p_k = \sum_{j\in D_{\mathrm{f}}}\left[\sum_{k\in D_{\mathrm{u}},x_k>x_j}(x_k-x_j)\;p_k\right]\;p_j \tag{58}$$

$$= \sum_{j\in D_{\mathrm{f}}}\left[\sum_{k\in D_{\mathrm{u}},x_k>x_j}x_k\;p_k\;-\;x_j\sum_{k\in D_{\mathrm{u}},x_k>x_j}p_k\right]\;p_j \tag{59}$$

Now that we have removed all of the j terms from the summations over k, we can precompute the sums over k starting at each jump in D_u and store the results in arrays. Then we can have a single loop over j and advance the the array index over the unfavorite jumps such that x_k is always the least abscissa with $x_k > x_j$. In fact, two of the summations are already available to us as the CDFs of the favorite and unfavorite child distributions:

$$\sum_{i\in D_{\mathrm{f}},x_i\le x_j} p_i = \underline{c}^{\mathrm{f}}(x_j) \tag{60}$$

$$\sum_{i\in D_{\mathrm{u}},x_i\ge x_k} p_i = \overline{c}^{\mathrm{u}}(x_k) \tag{61}$$

The precomputed arrays we will use are:

$$\mathrm{mean}_{\mathrm{f}}[j] = \sum_{i\in D_{\mathrm{f}},x_i\le x_j} p_i x_i \tag{62}$$

$$\mathrm{mean}_{\mathrm{u}}[k] = \sum_{i\in D_{\mathrm{u}},x_i\ge x_k} p_i x_i \tag{63}$$

and in terms of these arrays, the necessary computations become:

$$\langle U_{\text{all leaves}}\rangle = \sum_{j\in D_{\mathrm{f}}} p_j\,[\mathrm{mean}_{\mathrm{u}}[k'] - x_j\overline{c}^{\mathrm{u}}(x'_k)] \tag{64}$$

and

$$I_{\mathrm{f}}^j = \mathrm{mean}_{\mathrm{u}}[l] - x_j\overline{c}^{\mathrm{u}}(x_l) \tag{65}$$

where l is the least jump in D_{u} s.t. $x_l > x_j$, and

$$I_{\mathrm{u}}^k = x_k\underline{c}^{\mathrm{f}}(x_r) - \mathrm{mean}_{\mathrm{f}}[r] \tag{66}$$

where r is the rightmost jump in D_{f} s.t. $x_r < x_k$.

This is the algorithm for computing the utility and influence coefficients at the children of the root.

```
/* favm is the favorite move index and root is the root of the search tree. */
double Utility(unsigned favm, Tree* root) {
        Distribution* dfav = Negated version of the favorite child's distribution;
        Distribution* dbad = Negamax of the unfavorite children's distributions;
        /* Fill in the sum and mean arrays (62) - (63) */
        double* mean_u = new double[dbad->n];
        double* mean_f = new double[dfav->n];

        double mlast = 0.0;
        for (Each jump BadJump in dbad) {
            mlast += BadJump.abscissa * BadJump.prob;
            mean_u[i] = mlast;
        }

        mlast = 0.0;
        for (Each jump GoodJump in dfav) {
            mlast += GoodJump.abscissa * GoodJump.prob;
            mean_f[i] = mlast;
        }

        /* Compute the utility (64) and influence coefficients of the favorite child (65)
        in one pass through the jumps in the favorite child. */
        double utility = 0.0;
        for (Each jump GoodJump in dfav) {
            int FirstBad = Index of the smallest abscissa in dbad which is > GoodJump.abscissa;
            /* If we get to the end of the unfavorite jumps there will be no more contributions to utility. */
            if (FirstBad > dbad->n) break;
            /* dbad→s[FirstBad].cdf is c̲(x_FirstBad) so 1 - dbad→s[FirstBad-1].cdf is c̄(x_FirstBad). */
            double influ = mean_u[FirstBad] -
                           GoodJump.abscissa * (1.0 - dbad->s[FirstBad-1].cdf);
            utility += GoodJump.prob * influ;
            GoodJump.inco = influ;
        }

        /* Now compute the influence coefficients of the unfavorite child (66)
        in one backward pass through the jumps in the unfavorite child. */
        for (Each jump BadJump in dbad from largest to smallest abscissa) {
            int LastGood = Index of the largest abscissa in dfav which is < BadJump.abscissa;
            if (LastGood < 0) break;
            /* We place the influence coefficients of the unfavorite child into dbad and then propagate
            them into the children tree (see below). */
            BadJump.inco = BadJump.abscissa * dfav->s[LastGood].cdf - mean_f[LastGood];
        }

        /* PropInfluence will compute the influence of the unfavorite children. */
        unfav->PropInfluence(nchild - 1, Array of unfavorite children);

        return utility;
}
```

16.3 ESS

For the finale we will illustrate how to compute the ESS values of all leaves together in linear time. We can simplify the calculation of ESS at a leaf by the following observation.

The influence coefficient I_i^L is the ratio between a change δ in the jump p_i and the resulting change in $U_{\text{all leaves}}$, $I_i^L = \Delta U_{\text{all leaves}}/\delta$. The quantity $Q_L^1(i)$, introduced in §7.4 of the BPIP paper is "the value of $[U_{\text{all leaves}}]$ after we expand leaf L, assuming it takes value x_i in its distribution." We claim that $I_i^L = Q_L^1(i)$.

This is a consequence of the fact that the utility is linear in the jump probabilities at a single node. The equivalence was first suspected after many numerical experiments indicated that it always held. The following argument is sufficient to establish the claim, although there are probably better ways to do it.

Let U be the utility of expanding all leaves before some perturbation to a leaf and U' the utility afterwards. Consider the perturbation which collapses the leaf L to the single spike i, the perturbation to jump i is $1-p_i$ and the perturbation to all other jumps j in L is $-p_j$. From the definition of I_i^L, we have:

$$Q_L^1(i) - U = U' - U = (1-p_i)I_i^L - \sum_{j\neq i} p_j I_j^L \tag{67}$$

$$= I_i^L - \sum_j p_j I_j^L \tag{68}$$

Next, consider the formal perturbation in which all jumps j in leaf L become 0. Due to the product of CDFs formula (41) all jumps in all ancestors would formally become 0 so that the utility would also formally become 0.

$$0 - U = -\sum_j p_j I_j^L \tag{69}$$

Combining (67), (68) and (69) proves the claim.

Since we have the I_i^L values directly after the downward influence propagation phase, we can compute the ESS values from the equation:

$$ESS = \sum_j \left| I_j^L - U \right| p_j \tag{70}$$

And here is the code we use to compute it:

```
void ComputeESS(Tree* t, double U) {
    if (Not a leaf node) {
      for (Each child c) {
          c.ComputeESS(U);
      }
    } else {
      t->ESS = 0.0;
      /* This is a leaf node, so compute (70) */
      for (Each jump LeafJump in distribution) {
          t->ESS += abs(LeafJump.inco - U) * LeafJump.prob;
      }
    }
}
```

17 Appendix D: Optimal Play for Sequential Metagame

We describe the optimal strategy for a player with infinite computational resources playing the formal metagame described in §7.1 in the version where one gets to sequentially expand k leaves, i.e. before expanding the i-th leaf one is told the result of the $(i-1)$-th expansion.

Let T be a tree in the metagame, i.e. a tree with probability distributions at each leaf. Let Le be the set of leaves in the tree T. For $L \in Le$, let its distribution be over some set of values v_i occuring with probability p_i. Denote by $T|_{L=v_i}$ the tree with the distribution at leaf L replaced by a single spike at v_i. Assume one has l expansions to perform in the metagame. Call the expected value of this metagame $EV_l(T)$. Then

$$EV_l(T) = \max_{L \in Le}\Big(\sum_i p_i^L EV_{l-1}(T|_{L=v_i})\Big). \tag{71}$$

Now the optimal leaf to expand first is whichever is the argument of the max in equation 71. Note that $EV_0(T)$ is the value of the mean of the distribution at the current favorite move.

See Appendix A for a numerical example.

18 Appendix E. Graph History Interaction and suggested cure

The interaction of transposition tables with position repetition in chess causes a subtle problem. (The problem cannot arise in Othello, Go, or Connect-4, in which repetition of position is impossible.) This problem has been called the "Graph History Interaction" (GHI) [9], and was apparently first pointed out by Palay ([25] page 131-2). GHI stems from the fact that the definition of 'position' in chess in fact includes the whole history of the game, since a three time repeated board position is defined to be a draw. When you attempt to store only the board position in a transposition table, and discard the historic information, mistakes can (and in practice sometimes do) result. If you attempt to keep the full history, however, it is not clear how to benefit from the use of transposition tables.

We discuss an example. In the tree of figure 2 (figure 5-14 in Palay's book), the chessboards at nodes 2 and 12 are identical, and so are the boards at 7 and 8. (Here "chessboard" means the placements of the men on the squares, castling and en passant information, and also whose move it is.) A tree search algorithm with transposition table might first search the descendants of node 1 and then those of node 2. When searching the descendants of node 2, node 8 would be encountered, and recognized to have the same chessboard as node 7. Node 8's value might then be retrieved from the trans table and no further search done. However, this would be an error. Node 12, when searching the descendants of node 2, should be regarded (since it is a repetition of node 2) as a "draw with 100% certainty." But when searching the descendants of node 1, node 12 is not a draw. Hence node 12 is really 2-valued. Hence its ancestors, nodes 9,10,11, are also 2-valued. Alternatively, the compression of the tree into a DAG in the first place, was wrong, since node 12 is really two different nodes. Hence it was incorrect to retrieve node 8's value from the trans table entry for node 7, and the use of trans tables here quite possibly would cause incorrect play.

One desires an approach yielding the time and storage savings possible from transposition tables, but also avoiding incorrect play arising from GHI. Campbell [9] suggested some simple rules for restricting the entries that may be placed in the transposition table, and proved that the use of his rules inside an Alpha-beta tree valuator would completely eliminate one type of GHI, which he called "draw first." Campbell's methods would not cure the second kind of GHI (called by him "draw last") but Campbell argued that no method like his could hope to do so. It was plausible to Campbell that in iteratively deepened Alpha-beta using his heuristics, remaining GHI problems might not be very serious and runtime would not be greatly increased.

We will now suggest an approach completely avoiding incorrect play stemming from GHI, but only suitable for algorithms which, like ours, store the whole search DAG and grow it using gulps. With each parent-child edge e in the DAG, associate the following subset of chessboards in the DAG:

$$S(e) \equiv \text{Ancestors}(e) \cap \text{Descendants}(e). \tag{72}$$

We claim that a DAG is legitimate (i.e. no nodes need to be split) if and only if, for all nodes c which have more than one parent p, $S(e_{pc})$ does not depend on p. (We have used "e_{pc}" to mean the edge from parent node p to child node c.)

Our method is as follows. When building the DAG, each time a new leaf node B is created: if B is a repetition, valuate it as "draw," otherwise, look in the trans table to see if any node A already exists, which

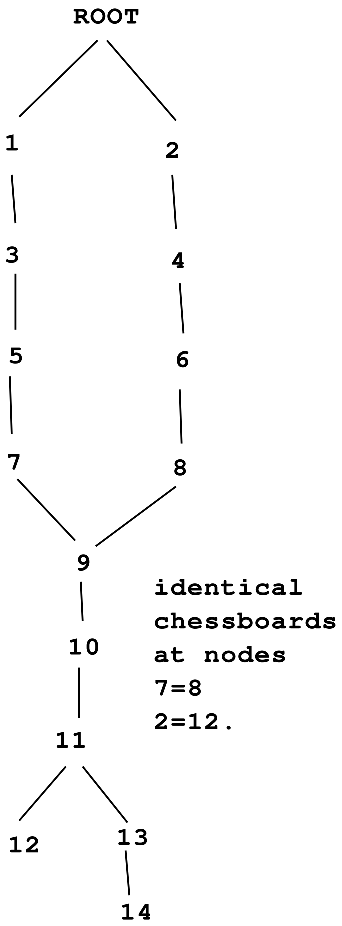

Figure 2: A fragment of a search DAG in chess.

has the same chessboard as B. If so, merge A and B into a single DAG node, if the resulting merged node c satisfies the $S(e)$ test. If no such A exists and B is not a repetition of one of its ancestors, valuate it normally.

DAGs built in this manner, although initially flaw-free, can gradually develop flaws as deeper search uncovers repeated positions. After each leaf expansion "gulp," we exhaustively investigate the current DAG to see if every node c with multiple parents passes the $S(e)$ test. If a node c is found which fails the test, it is *split* into several nodes c', c'', ,... having distinct sets $S(e_{pc})$. (Usually only two nodes will be required.) One then needs to rebuild the entire tree of descendants of each of these new nodes by further search, except for that one of them whose tree of descendants was already valuated correctly. In these new rebuilt trees, the appropriate repeated nodes will be (re)valuated correctly: as a draw, if they repeat an ancestor, or not, if not. However, for the moment we will *not* do any of this rebuilding, but will merely leave the new nodes childless, counting on later gulps to find their children. (Alternatively, one could consider immediately copying all the trees down to the points where repetitions occurred, and then re-valuate those nodes.) After each such node-split the testing and splitting is continued as long as necessary until the entire DAG is again flaw-free.

How efficient is this apparently brutish procedure? Observe that the set of ancestors of an edge e_{pc} is the union of the sets of ancestors of edges e_{xp}, and p itself. This, together with the (fixed) set of ancestors of the root, suffices to define these ancestor sets for each edge e in the DAG. Similarly the set of descendants of an edge e_{pc} is the union of the sets of descendants of edges e_{cy}, and c itself, and leaves have no descendants. These observations, together with the observation that taking the union or intersection of two sets, stored as sorted lists, may be accomplished by a list merge in linear time, allows one to see that the entire checking procedure may be accomplished in a time depending linearly on the sum of the sizes of all the ancestor and descendant sets for each of the edges in the DAG. This sum is easily seen to be bounded by $|T|d$ where T is the number of nodes in the search *tree* (i.e. before compression into a DAG) and d is its depth. Now inspecting equation 72, note that we may define "Descendants$'(e)$" to be those descendants of e which actually are repetitions of a position and similarly "Ancestors$'(e)$" as only those ancestors of e which actually got repeated later, and then use the primed quantities in equation 72 in place of the unprimed ones. The relevant nodes (possible repetitions of chessboards at a depths different by an even number ≥ 4) may be found by use of a hash table.

In practice, this "prime trick" will probably be enough of a time and space savings to make this procedure feasible. Repetition nodes should be rare since these lines of play will usually be cut off as one of the players would not prefer them, or pruned, since there is no need to investigate a sure draw any deeper. In the worst case, however, we have nothing better to fall back on than the $|T|d$ bound. On the other hand, our BPIP algorithms, if implemented without any transposition tables, would run in time of order $|T|d$. Hence putting in transposition tables with GHI-immunity will not increase runtime by more than a small constant factor, even in the pathological worst case, over the alternative of not using trans tables at all. Hopefully in practice, the runtime will in fact greatly decrease. The cure outlined above is only suitable for tree searching algorithms which store the entire search DAG and grow it using gulps. Implementation in a low storage algorithm would probably be too costly.

Subset S	Expansion Utility U_S	$\not{U}$ $=\mathbf{E}$(U) ($\overline{S}$ pre-exp'd)	ESS V_S	δ^S	Subset S	Expansion Utility U_S	$\not{U}$ $=\mathbf{E}$(U) ($\overline{S}$ pre-exp'd)	ESS V_S	δ^S
IHGFED	**3.00357**	**3.00357**	**1.97856**	**1.97856**	HGFED	2.97708	2.77138	1.97560	1.97560
IHGFE.	3.00019	3.00357	1.97856	1.97856	HGFE.	2.97708	2.77138	1.97560	1.97560
IHGF.D	1.21925	0.81535	1.22706	1.22957	HGF.D	1.04713	0.81535	1.05556	1.11564
IHGF..	1.21925	0.81535	1.22706	1.22953	HGF..	1.03383	0.81535	1.05556	1.11479
IHG.ED	2.29853	1.96974	1.97560	1.97560	HG.ED	2.26335	1.78924	1.97560	1.97560
IHG.E.	2.29515	1.96974	1.97560	1.97560	HG.E.	2.26335	1.78924	1.97560	1.97560
IHG..D	0.29404	0.02649	0.43539	0.48323	HG..D	0.08126	0.00668	0.22696	0.28551
IHG...	0.25689	0.02649	0.43539	0.48319	HG...	0.02600	0.00668	0.22696	0.28467
IH.FED	3.00027	2.97757	1.97856	2.07012	H.FED	2.97708	2.74668	1.97560	2.04806
IH.FE.	2.99689	2.93847	1.97856	2.07012	H.FE.	2.97708	2.70953	1.97560	2.04806
IH.F.D	1.21433	0.74021	1.22706	1.22957	H.F.D	1.04713	0.70842	1.05556	1.11442
IH.F..	1.21433	0.74021	1.22706	1.22953	H.F..	1.03383	0.70842	1.05556	1.11357
IH..ED	2.19160	1.96974	1.97560	1.97560	H..ED	2.18822	1.78674	1.97560	1.97560
IH..E.	2.18822	1.96974	1.97560	1.97560	H..E.	2.18822	1.78674	1.97560	1.97560
IH...D	0.23219	0.02649	0.27043	0.27081	H...D	0.02046	0.00500	0.06160	0.06245
IH....	0.23219	0.02649	0.27043	0.27077	H....	**0**	**0.00500**	**0.06160**	**0.06160**
I.GFED	2.99857	3.00357	1.97856	2.00206	.GFED	2.97708	2.77138	1.97560	1.97560
I.GFE.	2.99857	2.98311	1.97856	2.00206	.GFE.	2.97708	2.77138	1.97560	1.97560
I.GF.D	1.21683	0.81535	1.22706	1.22943	.GF.D	1.03383	0.81535	1.05556	1.05995
I.GF..	1.21683	0.81535	1.22706	1.22930	.GF..	1.03383	0.81197	1.05556	1.05739
I.G.ED	2.29515	1.96974	1.97560	1.97560	.G.ED	2.26335	1.78924	1.97560	1.97560
I.G.E.	2.29515	1.95644	1.97560	1.97560	.G.E.	2.26335	1.78924	1.97560	1.97560
I.G..D	0.29404	0.02649	0.43539	0.48309	.G..D	0.06510	0.00668	0.22696	0.22953
I.G...	0.25689	0.02649	0.43539	0.48296	.G...	**0.02600**	**0.00330**	**0.22696**	**0.22696**
I..FED	2.99689	2.97757	1.97856	2.07012	..FED	2.97708	2.74668	1.97560	2.04806
I..FE.	2.99689	2.92231	1.97856	2.07012	..FE.	2.97708	2.70953	1.97560	2.04806
I..F.D	1.21433	0.74021	1.22706	1.22943	..F.D	1.03383	0.70842	1.05556	1.05813
I..F..	1.21433	0.74021	1.22706	1.22930	..F..	**1.03383**	**0.70504**	**1.05556**	**1.05556**
I...ED	2.18822	1.96974	1.97560	1.97560	...ED	2.18822	1.78432	1.97560	1.97560
I...E.	2.18822	1.95644	1.97560	1.97560	...E.	**2.18822**	**1.78432**	**1.97560**	**1.97560**
I....D	0.23219	0.02649	0.27043	0.27056	....D	**0**	**0.00338**	**0.00257**	**0.00257**
I.....	**0.23219**	**0.02649**	**0.27043**	**0.27043**	($\emptyset$)	0	0	0	0

Table 2: Expansion relevance measures for leaf subsets.

Node	(Location, Probability; Influence coefficient) triples	Mean	ESS
A	$(-7, 0.1254; 0)$, $(-4, 0.2046; 0.0165)$, $(6, 0.6700; 4.4779)$	**2.3238**	
B	$(-3, 0.0735; 6.0555)$, $(0, 0.2765; 4.1011)$, $(5, 0.6500; 2.1916)$	3.0295	
C	$(-5, 0.0165; 7.5746)$, $(-1, 0.0335; 4.7885)$, $(3, 0.9500; 2.8612)$	2.7340	
D	$(4, 0.62; 3.0056)$, $(9, 0.38; 3.0002)$	5.90	0.0026
E	$(-6, 0.67; 4.4779)$, $(7, 0.33; 0.0102)$	-1.71	1.9756
F	$(-5, 0.65; 2.1916)$, $(6, 0.35; 4.5115)$	-1.15	1.0556
G	$(0, 0.79; 2.8599)$, $(3, 0.21; 3.5440)$	0.63	0.2270
H	$(1, 0.67; 2.9576)$, $(5, 0.33; 3.0969)$	2.32	0.6160
I	$(-3, 0.95; 2.8612)$, $(8, 0.05; 5.7079)$	-2.45	0.2704

Table 3: Influence coefficients and other BPIP data for nonroot nodes.