 Rev. November, 1992

Game Search

Consider a game, such as chess, checkers, or tick-tack-toe in which two players compete by alternating moves on a board. Games have proved to be one of the great challenges for computer programmers in the past twenty five years. Although the computational problem to be solved is very well defined, the amount of computation required under any naive approach to computer game playing is prohibitively large. Computer scientists specializing in complexity theory are fond of statements like "even if every electron in the universe was a computer performing a billion operations per second this computation would not be done before the end of the universe". Intelligence, however, is the art of performing intractable calculations efficiently and computer game playing has now reached the point where only good grand masters can still beat the best computer chess programs. It seems likely that within a couple years the best chess player in the world will be a computer. The success of computer chess provides a clear example of intelligence over intractability.

Part of the story behind the success of computer chess involves advances in computer hardware. But hardware advance is not the whole story. Recent progress in computer chess has required new algorithms as well as new hardware — without the new idea of "singular extensions" the currently level of hardware would still be playing somewhere at the master level rather than threatening the world champion. There has also been recent progress in developing a new general theory of heuristics for controlling search in games. Before exploring these recent developments however, one should start with a solid understanding of the classical techniques underlying game playing programs.

1 Static Board Evaluators

In any finite two person adversarial game there is a well defined set of optimal moves at each position and a well defined outcome for the game under the assumption that each player makes optimal moves. However, it is not possible to determine the optimal moves in a game as large as chess by searching all possible lines of play. Chess programs only examine some of the possible lines of play and, even for the lines they do examine, they do not examine these lines to the end of the game. Useful, but incomplete, search can be done with a *static board evaluator*.

A static board evaluator is a procedure that can be applied to any board position and returns a number representing a heuristic estimate of who is ahead and by how much. For example, in the game of chess one can assign point values to each piece on the board. The sum over all of one player's pieces of the point value of the piece provides a quantitative measure of the *material strength* of that player. The difference between the material strength of one player and the material strength of the opponent is called the *material advantage* of that player and is measured in points where pawns are assigned one point. It is common for chess players to say that white (the player moving the white pieces) has a two point (or two pawn) material advantage. A simple static board evaluator for chess can be given by the the material strength of the white pieces minus the material strength of the black pieces. Note that, under this static board evaluator, positive values favor white while negative values favor black.

Designing good static board evaluators for particular games is an art. One interpretation of static evaluators is that they should represent *maintainable invariants* of the position. Consider the game of nim. In a simple version of nim there is a pile of sticks and each player, on his or her turn, must remove either one or two sticks. The objective of the game is to remove the last stick. Call the players A and B. Suppose that it is A's move and that the number of sticks remaining is equal to either one or two mod three. Clearly if there were exactly one or two sticks remaining then A would win. Furthermore, A can maintain the statement that each time A moves the number of sticks remaining is either one or two mod three. If the number remaining equals one mod three then A takes one stick leaving zero mod three. After B's move the number remaining will again be either one or two mod three. If there are two mod three remaining then A takes two leaving zero mod three and the cycle repeats. In this game there exists a perfect static evaluator — if the number remaining is one or two mod three then the player to move can win, if the number remaining is zero mod three then the player to move can be forced to loose.

Maintainable invariants give properties of a far distant future position. For example, a "passed pawn" is a pawn such that no opponent pawn is between the given pawn and the queening rank. Once a passed pawn has been created it is fairly likely that the existence of a passed pawn can be maintained as an invariant by the player possessing the pawn. The existence of a passed pawn becomes important late in the game. Material advantage is also, to a large extent, a maintainable invariant that has real significance in the end game. Although it is not true in general that a material advantage can be maintained until the end game, a material advantage is likely to be maintainable and hence serves as a useful heuristic static feature. Provable invariants are used by human chess players but not, as yet, by computer chess programs. For example the distance between a pawn and its queening square, minus the distance between the pawn and the opposing king, is an invariant that can often be maintained by the opposing king. By observing that this invariant can be maintained one can, in some cases, quickly prove that the pawn can not become a queen.

2 Minmax Values

In any game playing program based on a static evaluator there is a *max player* and a *min player* where static values grater than zero indicate that the max player has an advantage and numbers less than zero indicate that the min player has an advantage. The larger the distance from zero, the greater the advantage. If the static value is an accurate indicator of who is ahead in any given position, then the max player should try to reach positions that maximize the static value and the min player should try to reach positions that minimize the static value.

The set of positions reachable from a given start position form a tree whose root is the start position and such that the children of any position consist of the positions reachable by a single move. This tree will be called the *game tree* and the positions in this tree will be called nodes. A node where the max player is to move will be called a max node, and a node where the min player is to move will be called a min node. The children of a max node are min nodes and vice versa. The transition from a node to a child node is called a *ply* in the search tree.[1]

The static evaluation function can be applied at any node in the search tree. However, search can be used to compute "better" values at nodes. For example, at a max node one can assume that max player will move to the child position that maximizes the static value of the child. In this case the value of a max position is better estimated as the maximum of the static value of the child positions. But each child position is a min node and the value of the child position is better estimated as the minimum of the static values of its children. This leads to the following definition of the d-ply minmax value of a position.

> **Definition:** The 0-ply minmax value of a node is defined to be the static value of that node. The d-ply minmax value of a max node is the maximum of the $(d-1)$-ply minmax value of its children nodes. The d-ply minmax value of a min node is the minimum of the $(d-1)$-ply minmax value of its children nodes.

The most fundamental game playing algorithm involves computing the d-ply minmax value of an initial position and then moving to the child position which has the best $(d-1)$-ply minmax value. This algorithm was first proposed by Alan Turing in the 1950's.

[1] In chess terminology a "move" involves an action from both players so that a move is two ply.

3 Duality

In game search there is a simple duality principle. Consider a statement such as "the minmax value of a min node is the minimum of the minmax values of its children". Each such statement has a *dual statement* in which every occurrence of min is replaced by max (and vice versa), every occurrence of minimum is replaced by maximum (and vice versa), and the sign of any inequality test is reversed ($x < y$ becomes $y < x$). The dual of the above statement is "the minmax value of a max node is the maximum of the minmax values of its children". For any definition or lemma about games there is a dual definition and lemma. If we define a min strategy to be a tree satisfying certain conditions, then we will also define a max strategy to be a tree satisfying the dual conditions. If we prove that every min strategy satisfies a certain property, then every max strategy will satisfy the dual property.

Many authors adopt a "negamax" formulation of game search in which every node is a maximizing node and the sign of the minmax values is reversed at each ply of the search tree. The use of negamax terminology allows every definition and lemma to be stated once — in minmax terminology every definition and lemma can be given twice, once in the original form and once in the dual form. Although negamax terminology technically reduces the number of definitions and lemmas, I find that it obscures the fundamental minmax nature of game search.

The following sections are all formulated in minmax terminology. However, each definition and lemma is only given in one of its two dual forms. The reader should keep in mind that every definition and lemma has a dual form.

4 α-β Pruning

Consider the simple two-ply tree shown in figure 1. In this game tree there are three options from each position. Suppose we are interested in computing the 2-ply minmax value of this position. The 1-ply minmax value of the first

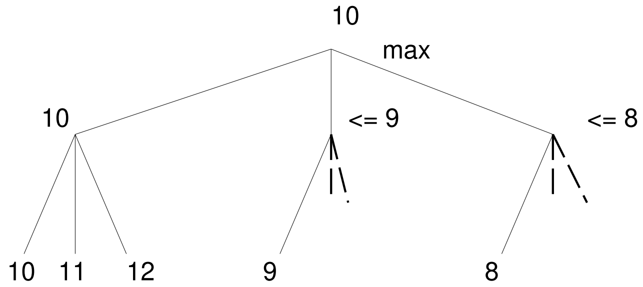

Figure 1: A simple 2-ply game tree.

child of the root position is 10. Now consider the second child of the root position. Since the root position is a max node, all the children of the root position are min nodes. By examining only one option at the second child we know that the minmax value of the second child is less than or equal to 9. Since the first child has a value of 10, and the second child is now known to be less than 9, we can ignore the other options at the second child. A similar analysis holds for the third child. In this example we can determine (exactly) the minmax value of the root position without examining all of the options at the second and third child of the root. This is a simple example of α-β pruning in the computation of d-ply minmax values.

The pruning exhibited in figure 1 could still be done if the search in a search of depth greater than 2 ply. Suppose that we were interested in the 10-ply minmax value. Provided that the minmax value of the first child of the root is 10, and that the minmax values of the first options at the second and third children are less than 10, then the other options at the second and third children can be ignored. Note that this greatly reduces the number of "grand children" of the root that must be examined. If the branching factor at each ply is b, then the number of grand children examined is reduced from b^2 to $2b-1$.

α-β pruning is a special case of the general technique of bounding an objective function. In figure 1 the max playing is trying to find a move from the root position that maximizes the minmax value. As soon as a value of 10 is established for the first move, the max player is only interested in values greater than 10. This lower bound on the objective function prunes future search.

During a search, a lower bound on minmax values establishes a value that is considered to be a win for the min player. While searching the second and third children of the root position in figure 1 any value of 10 or less can be considered as a "win" for the min player. To construct a general procedure for computing minmax values using α-β pruning we start with a simple procedure for computing the minmax value relative to two bounds on the static values. The lower bound is a value that is considered to be a win for the min player and the upper bound is a value that is considered to be a win for the max player.

Procedure `MINMAX(NODE, DEPTH, MIN-WIN, MAX-WIN)`

1. If `DEPTH` is zero then return `STATIC-VALUE(NODE)`
2. If `NODE` is a max node then
   (a) If there exists a child `NODE2` of `NODE` such that

   ```
   MINMAX(NODE2, DEPTH - 1, MIN-WIN, MAX-WIN)
   ```

   is greater than or equal to `MAX-WIN` then return `MAX-WIN`.

   (b) Otherwise, return the max of

   ```
   MINMAX(NODE2, DEPTH - 1, MIN-WIN, MAX-WIN)
   ```

   where `NODE2` is a child of `NODE`.
3. If `NODE` is a min node then perform the dual of step 2.

Note that if there exists a winning move at the node being examined then the other nodes need not be examined. This is the essence of α-β pruning. In the original formulation of the procedures the parameters `MIN-WIN` and `MAX-WIN` were called α and β respectively. Also note that if a winning move is found, it is not necessarily the best winning move, there may be some other winning move that has a better minmax value. If this procedure returns the value `MAX-WIN` then the true minmax value of the position is known to be *at least* as large as `MAX-WIN` but it may actually be larger. Similarly, if the value `MIN-WIN` is returned then the actual minmax value may be smaller than `MIN-WIN`. However, if the minmax value returns is properly between `MIN-WIN` and `MAX-WIN` then it equals the true minmax value of the position.

> **Lemma:** If the above procedure is applied to a max node with depth argument d in a tree with branching factor b, then it can (in the best case) return the value `MAX-WIN` after examining only $b^{\lfloor \frac{d}{2} \rfloor}$ leaf nodes (a leaf node in a d-ply search is a node d ply from the root position). Applied to a max node, it can return the value `MIN-WIN` after examining only $b^{\lfloor \frac{d+1}{2} \rfloor}$ root positions.
>
> **Proof:** Suppose that the procedure is applied to a max node and returns the value `MAX-WIN`. At each max node encountered during the search it is possible that the procedure examines only a single child of that position and returns the value `MAX-WIN`. If only a single child is generated from each max node, then the number of leaf nodes generated is $b^{\lfloor \frac{d}{2} \rfloor}$. Similarly, if the procedure returns `MIN-WIN` then it is possible that only one child is generated at each min node. In this case the number of leaf nodes generated is $b^{\lfloor \frac{d+1}{2} \rfloor}$.

Using techniques described below, α-β pruning can in practice be used to compute exact minmax values by searching trees of order $b^{\frac{d}{2}}$ rather than trees of order b^d. Note that $b^{\frac{d}{2}}$ is *much* smaller than b^d. b^d is the square of $b^{\frac{d}{2}}$ and in practical chess searches b^d is a factor of millions larger than $b^{\frac{d}{2}}$. Another way of looking at the difference between b^d and $b^{\frac{d}{2}}$ is that α-β pruning allows the search to done at twice the depth that would other wise be possible. Yet another way at looking at it is that α-β pruning reduces the effective branching factor at each ply to the square root of the original branching factor.

Before discussing how this dramatic reduction in search is achieved in practice, we consider the optimal search trees implicit in the above lemma. When the procedure returns `MAX-WIN`, and only considers one move from each max node, the the tree searched is called a *max strategy*.

> **Definition:** A *d-ply max strategy* is a subset of the search tree that includes the root node, includes *exactly one* child of any included max node less then d ply form the root, and includes all children of any min included node less than d ply from the root.

A max strategy should be viewed as a "plan of action" or "strategy" for the max player. To prove that the game is a win for the max player it suffices to give a plan of action, a strategy, for the max player that guarantees a win. For any max node that can be reached by the (uncontrollable) choices of the min player, one must specify a single winning move for the max player. A min strategy is defined as the dual notion. The above lemma about the minmax procedure is equivalent to the statement that in a tree with branching factor b, and whose root node is a max node, a d-ply max strategy contains $b^{\lfloor \frac{d}{2} \rfloor}$ leaf nodes while a min strategy contains $b^{\lfloor \frac{d+1}{2} \rfloor}$ leaf nodes.

Rather than ask how one can compute the minmax value of node in the search tree we might ask how one could *prove* that the minmax value equaled a given value v. To prove that the minmax value equals v it suffices to show that if we consider v to be a winning value for the max player, then the max player has a winning strategy, *and* if we consider v to be a winning value for the min player then the min player also has a winning strategy. In other words, the max player can guarantee a value of v or larger, and the min player can guarantee a value of v or smaller. So a proof that the minmax value of the root position is v consists of two strategies — a max strategy that guarantees a value of v or larger and a min strategy that guarantees a value of v or smaller. If the minmax value of the root position is in fact v, then two such strategies always exist. In practice, α-β pruning reduces the searched portion of the tree to be not significantly more than the union of a min strategy and a max strategy. This is the smallest possible proof that the the d-ply minmax value of the root position is the returned value.

The above procedure was given parameters `MIN-WIN` and `MAX-WIN` and simply passed these parameters along on recursive calls. The full α-β procedure is a minor optimization of the above procedure.

Procedure α-β`(NODE, DEPTH, MIN-WIN, MAX-WIN)`

1. If `DEPTH` is zero then return `STATIC-VALUE(NODE)`
2. If `NODE` is a max node then
   (a) Set `BEST-VALUE` to `MIN-WIN`.
   (b) For each child tt NODE2 of `NODE`
      i. Let `CHILD-VAL` be

         α-β`(NODE2, DEPTH - 1, BEST-VALUE, MAX-WIN)`.

      ii. If `CHILD-VAL` is greater than `BEST-VALUE` then set `BEST-VALUE` to `CHILD-VALUE`.
      iii. If `BEST-VALUE` $\geq$ `MAX-WIN` then return `BEST-VALUE`.
   (c) Return `BEST-VALUE`.
3. If `NODE` is a min node then do the dual of step 2.

The only difference between this procedure and the one given earlier is that in the recursive call in step 2bi, the best value seen so far is given as the winning min value. This does not interfere with a correct answer being returned because at the recursive call at step 2bi we are only interested in values greater than the best value seen so far. Pruning is done in step 2biii where the iteration over the children is terminated as soon as a winning child is found. To compute an exact minmax value for a root position one simply calls the α-β procedure using $-\infty$ and $+\infty$ as the original `MIN-WIN` and `MAX-WIN` parameters. One can verify that this procedure will generate the pruning shown in figure 1.

> **Lemma:** It is possible that the set of nodes examined during an application of the α-β procedure, with `MIN-WIN` and `MAX-WIN` parameters $-\infty$ and $+\infty$ respectively, is the union of a min strategy and a max strategy.
>
> **Proof:** Suppose that at each node the move that is ultimately the best move at that node is tried first. In this case the the sequence of first moves leading from the root position to the first

> leaf node examined is in an optimal line of play — each move is the best possible for the player involved. The sequence of optimal moves will be called the *principle line* of play. The static value of the leaf node found down the principle line equals the ultimate minmax value of the root position (given that the principle line is defined by the sequence of optimal moves). Let v be the static value of the leaf node down the principle line. Each node on the principle line will be examined with `MIN-WIN` and `MAX-WIN` parameters of $-infty$ and $+infty$ respectively. Consider a max node on the principle line. Since the `MAX-WIN` value at this node is $+\infty$ all children will be examined. The first child examined will be assigned the value v. The remaining children will be examined with the value v treated as a winning value for the min player. Given that the first child is optimal, the min player can achieve a value of v or less for all the alternative children. This means that all the remaining children are considered to be wins for the min player. A search that returns `MIN-WIN` need only examine a min strategy. This means that *at each non-principle child of a max node on the principle line, the tree searched under that child is a min strategy.* Dually, *at each non-principle child of a min node on the principle line, the tree searched under that child is a max strategy.* Now consider the tree consisting of the principle line plus all nodes that are descendants of non-principle children of some max node on the principle line. This tree is a min strategy. Dually, consider the tree that consists of the principle line plus all nodes that are descendants of non-principle children of min nodes on the principle line. This tree is a max strategy. All nodes in the α-β search tree are in one of these two trees. So the entire α-β tree is the union of a min strategy and a max strategy.

The intersection of a min strategy and a max strategy consists of a single line of play — the line of play that is taken if each play follows their respective strategies. This implies that there is exactly one leaf node that is a member of both the min and max strategy (the leaf node on the principle line). So the total number of leaf nodes examined by the α-β procedure in a d-ply search can be as small as $b^{\lfloor \frac{d}{2} \rfloor} + b^{\lfloor \frac{d+1}{2} \rfloor} - 1$. This is order $b^{\frac{d}{2}}$ rather than order

b^d as would be done by a nonpruning minmax search.

Of course the optimal behavior is only achieved under an optimal ordering of the children at each node of the search. In practice α-β search is used with iterative deepening. Each position visited is stored in a hash table (the transposition table) and the best move from that position found by the previous search is stored in the hash table. This means that when the procedure does a $(d+1)$-ply search the best move at each position found by the d-ply search is known. This information from the previous search is used to order the children. This results in essentially optimal ordering and essentially optimal performance of α-β pruning.

5 Conspiracy Theory

In practical chess programs search is not done to a uniform depth. When a node at a depth d is reached, rather than simply apply the static board evaluator, a complete α-β search is performed below that depth but restricted to capture moves, i.e., moves that remove a piece from the board. The branching factor of capture moves is much smaller than the complete branching factor and since a piece is removed on each move the branching factor of capture moves rapidly diminishes in a search tree of captures. When pruned with α-β, it is practical to compute a complete capture move search at the nodes that would normally be considered leaves of the search tree. Another common way in which search is done to a nonuniform depth involves "check extensions". A move that places the opponent's king in check does not count as a ply of search — search is carried at least one ply deeper than normal under any checking move.

Searching capture and checking moves more deeply than other moves greatly improves the performance of chess programs. However, the heuristics used to determine when a line of play should be examined deeply seem to be chess specific. Are there general game-theoretic principles for determining which lines of play should be examined more deeply. Conspiracy theory is an attempt to provide such principles. Conspiracy theory starts by asking how much information does a search tree really provide about the ultimate minmax value of the root node? Intuitively, deep search trees provide more information than shallow search trees. Can this intuition be formalized? If we can construct a measure of the information provided, then perhaps we can construct search trees whose shape is such that they provide a lot of information.

An incomplete search tree is defined to be a subset of the descendants of a given root node such that every descendant included in the incomplete tree is connected to the root by included nodes and every included node is either a leaf node of the incomplete tree, in which case none of its children are included, or an interior node of the incomplete tree, in which case all of its children are included. There is no restriction on the shape of an incomplete tree. Each node in a finite incomplete tree can be assigned a minmax value by assigning the leaf nodes their static value and assigning each interior node either the min or the max of the values of its children. Any incomplete search tree assigns some minmax value to the root node. A game playing program grows some incomplete search tree and then selects a move based on the minmax values assigned by that search tree.

Suppose that instead of selecting a move, we were concerned with determining the true minmax value of a given root position. The problem of selecting moves can be reduced to the problem of determining values — if we can determine values accurately then by determining the values of the children of the root position we can select the best move. How much information does a given incomplete search tree provide about the true minmax value of the root node? As search continues, and leaf nodes are expanded, the minmax value of nodes can change. Intuitively, a partial search tree provides a lot of information about the root value if it is very unlikely that the root value will change as the search continues.

Consider the 2 ply incomplete search tree shown figure 2. This simple tree has four leaf nodes with values -1, 0, 0, and 1. If the value of the third leaf node were changed to $+\infty$, the minmax value of the min node immediately above this leaf would change from 0 to 1 and the root value would also change from 0 to 1. In order to get the minmax value of the root to be larger than 1 at least two leaf nodes would have to change. Changing the third leaf node to $-\infty$ would change the minmax value at the root to -1. Making the

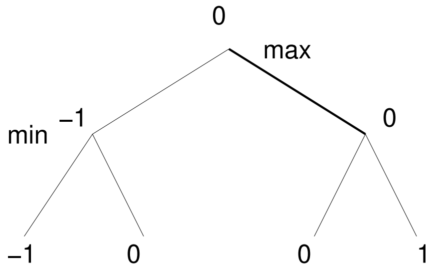

Figure 2: A simple 2 ply search tree.

root value smaller than -1 would require changing at least two leaf nodes — perhaps the first and third. If it seems likely that one leaf can change, but unlikely that two leaf nodes will change, then we can say that the root value for the tree shown in figure 2 is likely to be in the interval $[-1, 1]$. In a large incomplete search tree one may be forced to change a very large number of different leaf nodes before one can bring about a desired change in the root value.

> **Definition:** Given an incomplete search tree. If $\mathcal{C}$ is a set of leaf nodes such that by changing the value of the nodes in $\mathcal{C}$ one can change the value of a node n to v, then we say that $\mathcal{C}$ is a *set of conspirators* capable of assigning n the value v. Given a node n in the incomplete tree, and a nonnegative integer c, we define $\uparrow V_{n,c}$ to be the largest value that can be assigned to node n by a set of c or fewer conspirators. $\uparrow V_{n,0}$ is just the minmax value of n in the incomplete tree. Dually, $\downarrow V_{n,c}$ is the smallest value that can be assigned to n by a set of c or fewer conspirators.

I will often write $\uparrow V_c$ rather than $\uparrow V_{n,c}$ when the node involved is clear from context. Let $\uparrow V_0, \uparrow V_1, \uparrow V_2, \cdots$ be the values achievable by the specified number of conspirators at the root node. These values can be interpreted as plausible upper bounds on the true value of the root node. The more conspirators that are required to achieve a value the less likely that value is of being achieved. If conspiracies of size 4 are considered to be likely, but conspiracies of size 5 are considered to be unlikely, then the root value is likely to be in the interval $[\downarrow V_4, \uparrow V_4]$. A tree growth procedure can be defined in terms of this bound sequence and the following definition.

> **Definition:** Given a "goal value" V, a *critical leaf node* for achieving V is any leaf node that included in a minimal size conspiracy set that can achieve V.

For each conspiracy size c there is a corresponding range of values $[\downarrow V_c, \uparrow V_c]$ achievable with c conspirators. Let Δ be an "acceptable uncertainty" in the true value of the root. For some values of c the size of the interval $[\downarrow V_c, \uparrow V_c]$ will be less than or equal to the acceptable uncertainty Δ. We would like to grow the tree in such a way that the uncertainty is acceptable even for large values of c. This observation motivates the following tree growth procedure which is parameterized by the acceptable uncertainty Δ.

Tree Growth Procedure:

1. Let c be the smallest non-negative integer such that $\uparrow V_c - \downarrow V_c > \Delta$.
2. If $\uparrow V_c - \uparrow V_0 \quad > \quad \downarrow V_0 - \downarrow V_c$ then expand a critical leaf node for achieving $\uparrow V_c$.
3. Otherwise expand a critical leaf node for achieving $\downarrow V_c$.

This procedure is motivated by the intuition that the critical leaf nodes are the most important leaf nodes for determining if the root value will change as leaf nodes are expanded. The growth procedure can be iterated until the number of conspirators selected at step 1 is greater than some specified "unlikely" conspiracy size.

Of course we have not yet specified a method of computing the bounds of the form $\uparrow V_c$ and $\downarrow V_c$, and we have not specified any method of finding the critical leaf nodes. Both of these problems can be solved in a computationally efficient way. First we concentrate on computing the bounds of the form $\downarrow V_c$ and $\uparrow V_c$. For any finite incomplete search tree there will be some sufficiently large conspiracy size c such that $\uparrow V_c = +\infty$ (consider the case where c equals

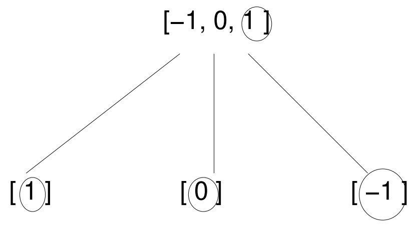

Figure 3: A max node over leaf nodes.

the total number of leaf nodes in the incomplete tree). This implies that, at each node, there is only a finite number of *finite* bounds of the form $\uparrow V_c$.

> **Definition:** The *upper bound sequence* of a node in an incomplete tree is the finite sequence $\uparrow V_0, \uparrow V_1, \cdots, \uparrow V_n$ of all *finite* values of the form $\uparrow V_c$. The *lower bound sequence* is the finite sequence $\downarrow V_m, \cdots, \downarrow V_1, \downarrow V_0$ of all finite values of the form $\downarrow V_c$.

Note that $\uparrow V_0$ and $\downarrow V_0$ both equal the minmax value of the node and that this value is included in both bound sequences. This implies that every bound sequence includes at least one value — both the upper and lower bound sequence of a leaf node consists of the single value which is the minmax value of the node. Given that leaf nodes have well defined bound sequences, the bound sequence of internal nodes can be directly computed from the bound sequences of their children. Consider a max node all of whose children are leaves as shown in figure 3. Any single child can act alone to give the max node a value of $+\infty$. Thus the upper bound sequence of this max node consists of the single value $\uparrow V_0$, the minmax value of the node. Now consider computing the value of $\downarrow V_c$ for this max leaf node. The greatest reduction in value achievable by c of its leaf node children is achieved by setting the c largest children to $-\infty$. If c is less than the total number of children, this will result in a value at the max node equal to the $c+1$'st best child. Thus the value of $\downarrow V_c$ equals the value of the $c+1$'st best child. In figure 3 the bound sequence of the max node is written in the form $[\downarrow V_m, \cdots, \downarrow V_1$,V,$\uparrow V_1, \cdots, \uparrow V_n]$ where V is the minmax value of the node.

Now consider the internal max node shown in figure 4. The children of this max nodes are themselves internal nodes with nontrivial bound sequences.

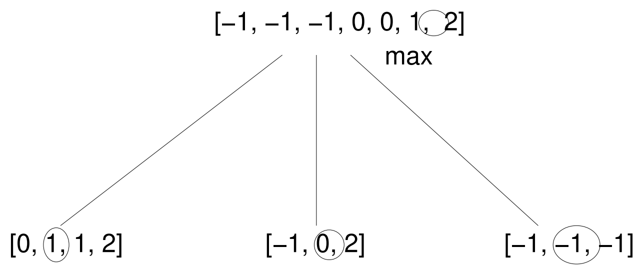

Figure 4: A max node over internal nodes.

Bound sequences of the form $[\downarrow V_m, \cdots, \downarrow V_1, \mathrm{V}, \uparrow V_1, \cdots, \uparrow V_n]$ are given at each node and a circle has been placed around each minmax value. To compute $\uparrow V_c$ of the max node one can observe that increasing the max node requires increasing one of its children nodes, and increasing any single child suffices. The maximum increase achievable with c conspirators is achieved by increasing the child with the maximum value of $\uparrow V_c$. Thus the value of $\uparrow V_c$ at the root position equals the maximum over all child nodes of $\uparrow V_c$ of the child. Computing the lower bounds $\downarrow V_c$ is a little tricky. Consider the node shown in figure 4. The minmax value of the first child acts as a "safety net" for the minmax value of the root position — the minmax value of the root position can not be decreased without changing the safety net value of the first child. To decrease the root value requires changing some leaf node under the first child. But changing the value of a single leaf under the first child, even changing that leaf to $-\infty$, can only reduce the first child to 0. This "backup value" or "second safety net" on the first child will prevent the minmax value of the parent node from dropping below 0. Changing a single leaf node can reduce the of the root to 0. Moving the minmax value of the max node below 0 requires eliminating the safety net of 0 at the first child. This can be done by changing a second leaf under the first child. Changing two leaves under the first child can reduce the value of the first child to $-\infty$. However, the minmax value of the second child now acts as a third safety net, and the max value still does not drop below 0. Changing a leaf under the second child (the third leaf changed) drops the value of the second child to -1. This value of -1 acts a fourth safety net on the value of the max node. By changing three leaf nodes we have reduced the max node value to -1. In general, reducing the value of the max node to a value v requires the removal of every safety net (lower bound value) in the child nodes which is greater than v. A given value can appear more than once in a bound sequence. In fact a bound sequence can be viewed as a multiset of values, i.e., a set in which an element can be included more than once. The lower bound sequence of a max node can be computed using a multiset union operation.

> **Bound Sequence Lemma:** The multiset of lower bounds of a max node equals the multiset union of the lower bound multisets of its children nodes.

Note that the minmax value of each node is included in the lower bound multiset. In figures 3 and 4 the lower bound sequence of the max node simply consists of all values that appear in lower bound sequences of the children nodes.

We now have a general way of computing the bound sequences of a max node from the bound sequences of its children nodes — the elements of the upper bound sequence are the maximum of the corresponding sequence elements of the children nodes, and the lower bound sequence is the multiset union of the children lower bound sequences. Note that if some child has only two finite upper bounds, then the max node can have only two finite upper bounds. In general, the length of the upper bound sequence of a max node equals the minimum length of the upper bound sequences of its children. One the other hand, the length of the lower bound sequence of a max node equals the sum of the lengths of the lower bound sequences of its children nodes. The duals of these statements hold at a min node.

Now we turn to the question of how to find critical leaf nodes. The first step is to narrow the set of candidates leaf nodes in terms of the following definition.

> **Definition:** A *high goal value* is any value larger than the root minmax value. A low goal value is any value lower than the root minmax value.
>
> **Definition:** A node n will be called *relevant* to achieving a high goal value $\uparrow V$ if n, and every parent of n, has a minmax value

less than $\uparrow V$. Dually, a node node will be called relevant to achieving a low goal value $\downarrow V$ if n, and every parent of n, has a value greater than $\downarrow V$.

All critical leaf nodes n for achieving $\uparrow V_c$ must be relevant in the above sense — if some parent of n already had a value of $\uparrow V_c$ or greater then nothing would be gained by changing the value of n. The set of nodes relevant to achieving a high goal value $\uparrow V$ forms a subset of the incomplete tree that is similar to a min strategy.

Lemma: Let $\uparrow V$ be a high goal value. If n is an internal max node that is relevant to achieving $\uparrow V$ then all of the children of n are also relevant to achieving $\uparrow V$. If n is an internal min node that is relevant to achieving $\uparrow V$ then at least one of the children of n is relevant to achieving $\uparrow V$.

The above lemma follows directly from the definition of relevant nodes and from the way in which minmax values are computed. The lemma establishes a clear connection between the set of nodes relevant to achieving $\uparrow V$ and the notion of a min strategy. In a min strategy all children of an included max node must be included and exactly one child of an included min node must be included. In the set of nodes relevant to achieving $\uparrow V$, rather than select a particular move at each min node we include all the "acceptable" moves at each min node. If there is only one acceptable move at each min node, then the set of relevant nodes forms a min strategy (but in this case a single conspirator can act alone to achieve $\uparrow V$). The above lemma motivates the following definition.

Definition: The set of nodes relevant to achieving a high goal value $\uparrow V$ will be called the *min defense* against $\uparrow V$. The max defense against a low goal value is defined as the dual notion.

The critical nodes for achieving a high goal $\uparrow V$ are all members of the min defense against $\uparrow V$, and the critical nodes for achieving a low goal $\downarrow V$

are all values of the max defense against $\downarrow V$. The critical nodes are the "weaknesses" in these defenses. To find the critical nodes in these defenses we introduce a notion of *depth*. Depth is normally defined as the number of ply between the node in question and the root node. I will call this classical notion *ply-depth* to distinguish it from the following.

> **Definition:** For any high goal value $\uparrow V$, and any given node n in the min defense against $\uparrow V$, the *conspiracy depth* of node n with respect to $\uparrow V$, denoted $d(n, \uparrow V)$ is the the least number of (co-conspirator) leaf nodes such that by changing the value of n, and changing the value of the co-conspirator leaf nodes, all parents of n (including the root node) can be changed to the goal value $\uparrow V$. The depth of nodes in the max defense against a low goal $\downarrow V$ is defined as the dual notion.

I will often write $d(n)$ rather than $d(n, \uparrow V)$ or $d(n, \downarrow V)$, when the goal value is clear from context. Intuitively, conspiracy depth of a node n is the amount of help that n requires in changing the root value to V — the more help required, the deeper (and the less critical) the node. Although conspiracy depth is different from classical ply depth, conspiracy depth is related to ply depth in that the conspiracy depth of the root node is always 0, and conspiracy depth sometimes increases, but never decreases, as we descend into the tree. The precise manner in which conspiracy depth increases as we descend is governed by the lemma given below.

> **Definition:** For a goal value V, and a node n in incomplete search tree we define $C(n, V)$ to be the least number of leaf nodes that must be changed to convert the value of n to V.
>
> **Lemma:** If $\uparrow V$ is a high goal value then $C(n, \uparrow V)$ equals the smallest c such that $\uparrow V_c \geq \uparrow V$.
>
> **Lemma:** Let n be a node in the min defense against a high goal value $\uparrow V$ and let n' be a child of n that is also a member of the min defense against $\uparrow V$. The conspiracy depth, $d(n')$, of the child (relative to the goal $\uparrow V$) can be computed from the conspiracy depth $d(n)$ of the parent as follows.

- If n is a max node then $d(n')$ equals $d(n)$.
- If n is a min node then $d(n')$ equals $d(n)$ plus the sum of $C(n'', \uparrow V)$ of all siblings n'' of n' such that n'' is also a member of the min defense against $\uparrow V$.

The last lemma above follows directly from the observation that a child of a max node requires no help from other nodes in raising the parent value to $\uparrow V$. However, a child of a min node requires the help of its siblings to raise the value of its parent node. The above lemma, and the corresponding dual lemma, have the following corollary.

> **Corollary:** Depth in the min defense only increases at min ply and depth in the max defense only increases at max play.

The following lemma directly relates to the notion of a critical leaf node to the notion of of depth.

> **Lemma:** The critical leaf nodes for the high goal value $\uparrow V$ consist of those leaf nodes in the min defense against $\uparrow V$ that have a the smallest conspiracy depth relative to $\uparrow V$.

Because values can be repeated in the upper bound sequence, the value $\uparrow V_c$ might be achievable with fewer than c conspirators. However, since some set of c (or fewer) leaf nodes can achieve $\uparrow V_c$, there must be some leaf node with a conspiracy depth of $c - 1$ or less. The following lemma provides a way of finding these shallow leaf nodes in the min defense.

> **Lemma:** For any node n in the defense against a goal value V, the conspiracy depth of the shallowest leaf node in the defense beneath n equals $C(n, V) + d(n, V) - 1$.

.

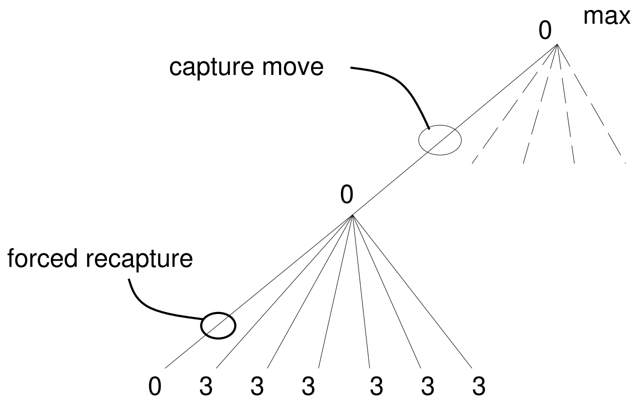

Figure 5: A forcing offensive move.

The above lemma can be used to find critical leaf nodes by starting at the root node and descending into the tree always being careful to select nodes that have shallow leaf nodes beneath them as determined by the above lemma.

We now return to the analysis of capture moves in chess. Consider the two ply tree fragment shown in figure 5. This tree fragment is taken to be part of the min defense for avoiding some high goal value. The upper ply is a max ply and the lower ply is a min ply. Since this is part of a min defense, we can consider the max move to be "offensive" and the min response to be "defensive". The minmax value of every node is shown next to the node. These static values can be interpreted as indicating that the max move was a capture that increased the static value from 0 to 3 by capturing a piece. The leftmost min move was a recapture that restored the static value to 0. However, all the alternatives that min might consider to the recapture move leave the static value at 3. The offensive capture forces a particular defensive response. If the goal value $\uparrow V$ is less than or equal to 3, then there is only one child of the min node that is a member of min defense — we say that min response is forced. Whenever a min response in the min strategy is forced (relative to a goal value $\uparrow V$) the conspiracy depth of the node resulting from the forced move is no deeper than the conspiracy depth of its parent. This leads to the following observation.

Definition: A min node in the min defense for avoiding a high goal value $\uparrow V$ is said to be *a forced ply* if only one of the children of that node is a member of the min defense against $\uparrow V$.

Lemma: Conspiracy depth does not increase at forced ply.

Since capture moves tend to force a particular response (or at least greatly reduce the number of responses that are members of the defense tree) the conspiracy depth does not increase at the defensive ply that is a response to a capture move (or at least does not increase much). This means that capture moves will be searched more deeply (under the classical notion of ply-depth) than noncapture moves (which tend not to force a response). In general, the min defense against $\uparrow V$ will be searched more deeply (in classical ply depth) down portions of the tree where min's moves are forced. A similar analysis applies to moves that place the oppenent's king in check.

6 Transcendent Game Principles

The conspiracy theory procedure given above is not necessarily the best possible tree growth procedure. However, conspiracy theory provides certain qualitative statements that probably apply to *any* game search procedure. These qualitative statements transcend the particular mathematical formalism of conspiracy theory. These qualitative statements are also intended as conjectures about human chess playing.

Principle 1. The work done by any game search procedure can be divided into work done exploring a min defense against high values and work done exploring a max defense against low values.

We observed earlier that the α-β search procedure grows a min strategy and a max strategy. The vast majority of nodes search are in one or the other, but not both, of these strategies. Thus, one can view an α-β search procedure as dividing its time between the two tasks mentioned in the above general principle. This general principle is even more evident in Berliner's B* algorithm which has not been discussed here. In the conspiracy theory tree growth procedure the procedure divides its time between expanding critical nodes for achieving $\uparrow V_c$ and time for expanding critical nodes for achieving $\downarrow V_c$. In the former case it is exploring the min defense against high values and in the latter case the max defense.

> **Principle 2.** Depth, as a measure of relevance to the root position, should be measured differently when exploring the min defense than when exploring the max defense — depth in the min defense is independent of depth in the max defense.

This principle has not been incorporated in to many existing chess programs. Although the best programs use a modified notion of depth based on ideas like "singular extensions", they assign a single depth to nodes that are in both the min and max defense. Nodes that are in both the min and max defense should be given two separate depth measures, one as a depth in min defense and one as a depth in the max defense.

> **Principle 3.** While the procedure is exploring the min defense against high values, the min player should be viewed as the defensive player and the max player as the offensive player. The roles are reversed when exploring the max defense against low values.

This principle is more intuitive than technical. When people play chess they tend to think in terms of offensive moves and defensive moves. By incorporating the offense/defense terminology into the search procedure we provide a connection between the moves being examined by the procedure and our intuitions about offensive and defensive thinking. For example, when a series of moves is being examined deeply because one player is forcing the moves of the other player, the player whose moves are being forced is viewed as the defensive player and the player selecting the positions that lead to forced responses is viewed as the offensive player. This terminology is consistent with the lines that are explored deeply by the conspiracy tree growth procedure. The offense/defense terminology also provides a neutral way of classifying the nodes in the search tree — the same terminology can be used form the min and max defenses.

Principle 4. Depth only increases at defensive ply.

This principle has been incorporated into the singular extension method currently used to modify classical ply depth in the current best computer chess programs. Actually, depth is still incremented at both offensive and defensive ply, but the analysis used to modify depth is only invoked at defensive ply.

Principle 5. Depth does not increase when the defensive move is forced.

This expresses the basic intuition that forced lines should be explored more deeply than unforced lines. However, the principle specifies that in order for the lines to be searched more deeply, the forced moves should occur at defensive ply.

Principles 2, 4 and 5 can be mathematically derived from conspiracy theory. These general principles may be useful in guiding the construction of game playing programs even if the particular tree growth procedure derived from conspiracy theory turns out not to work as well as other approaches. Some version of classical α-β techniques using depth measures guided by the above principles may provide the best performance in practice.

7 Problems

1. Consider a minmax game tree in which each max node has a branching factor of $b_{\max}$ and each min node has a branching factor of $b_{\min}$. Now consider an α-β search in which the best move is tried first at each ply and so that an optimal α-β search tree is generated. In a d-ply tree starting at a max node how many leaf nodes will be examined? Your answer should be given in terms of the parameters d, $b_{\max}$, and $b_{\min}$.

2. Consider the minmax game tree described in problem 1. However, rather than a perfect α-β, consider a "full" d-ply tree, i.e., one which contains all nodes d-ply or less from the root. Furthermore, suppose that every leaf node

in this tree has the same static value. What is the conspiracy depth of a leaf node in this tree with resect to an target value larger than the root value? What is the conspiracy depth of a leaf node in this tree with respect to a target value smaller than the root value? Both answers should be given in terms of d, $b_{\max}$ and $b_{\min}$.

3. Consider a partial search tree. Each leaf node has a static value that may or may not be equal to its true value. Assume that the true value of a leaf node is either its static value, $+\infty$, or $-\infty$. Also assume that all static values are finite. Furthermore assume that the probability that the true value is $+\infty$ equals the probability that the true value is $-\infty$ which equals $\epsilon < 1/2$. Also assume that the true values of the leaf nodes are independent of each other. Now let j be an internal node in the tree and let v be a value that is greater than the minmax value of j. Define $P_r(j \geq v)$ to be the probability that true value of j is at least v where the true value of j is to be computed by minmax from the true values of the leaf nodes. (We will ignore the fact that the probability distribution over the true values of internal nodes is quite different form the probability distribution over the true values of leaf nodes.)

Part a. Show that for an internal node j and for a fixed value v greater than the minmax value of j (as computed from static values of the leaf nodes) the probability $P_r(j \geq v)$ can be written as a polynomial in ϵ. Do not try to compute any particular polynomials. (Hint: use induction on the size of the search tree below j.)

Part b. For small values of ϵ we are more concerned with the order of the polynomial than with the coeficients. If $P_r(j \geq v)$ is quadratic or cubic in ϵ then it seems unlikely that the value of j will be larger than v. Since ϵ is small, the "order" of the polynomial is defined to be the smallest power of ϵ in the polynomial. Give recurrance relations for computing the order of the polynomial for $P_r(j \geq v)$ form the order of the corresponding polynomials of the children of the node j. You will need different recurrance relations depending on whether j is a min or max node. Justify your answer.

Part c. Let $C_\uparrow(j,\ v)$ be the least number of leaf nodes needed to increase the value of j to v. If the minmax value of j is already v or greater then we define $C_\uparrow(j,\ v)$ to be 0. If the minmax value of j is less than or equal to v then $C_\uparrow(j,\ v)$ equals $C(j,\ v)$ as defined in the notes. Give recurrance relations for computing $C_{\uparrow}(j,\ v)$ from the values $C_{\uparrow}(k,\ v)$ where k is a child of j. The recurrance relations will be different at min and max nodes. Justify your answer.

Your answers to parts b and c should be the same. This implies that the order of $Pr(j \geq v)$ equals $C_{\uparrow}(j,\ v)$.

3. Consider a position in a game of chess where white can place black's king in check so that black has only a single legal response. Let p be the position from which white can move to place black's king in check and let p' be the position that results from the only legal response for black — note both players must move to get from position p to position p'. Describe two different situations in which the conspiracy depth of p' is the same as the conspiracy depth of p. Also describe a situation in which the consipiracy depth of p' is greater than the conspiracy depth of p. For each situation specify whether depth is being measured in a strategy for white or in a strategy for black and specify which player should be viewed as the defensive player and which as the offensive player.