Chapter 2

Game Tree Search

Computer programs play games, such as checkers and chess, by considering different combinations of possible moves for itself and possible replies by the opponent. Each legal combination of move and counter-move is a possible line of play. Ultimately, the most desirable line of play, often called the *principal variation* (PV), is decided upon. The systematic searching of different combinations is a well-known problem solving strategy and, in effect, the program is trying to look ahead at the possible continuations of the game. Intuitively, as the search is deepened, the amount of analysis increases and the chance of overlooking a strong move or a blunder decreases. Therefore, the depth of search is an important determinant of the quality of the program's play.

2.1 Game Trees and Minimax Search

To support the systematic analysis of different lines of play, the program builds a game tree. The *root* of the tree is the current position on the game board, a *node* in the tree is a legal board position and a *branch* is a legal move. The side which has the choice of move at the root position will be referred to as the *player* and the other side as the *opponent*.

The notion of *ply* describes the distance between nodes and is similar to the general notion of the *depth* of a tree. The immediate children of the root are said to be at ply 1, or are one ply away from the root. Although the ideal is to search lines of play until the end of the game, practical limitations on time and computing resources may limit the depth of search. *Leaf nodes* exist where the lines of play are not searched further and represent the program's look-ahead *horizon*. *Terminal nodes* are leaf nodes where the program is able to analyze to the end of the game, possibly with the aid of a database of positions of known values, such as an *endgame database* [SCT92].

In two player zero-sum games, such as checkers, an advantage for the player implies an equal and opposite disadvantage for the opponent. *Minimax search* models the give-and-take nature of these games. A heuristic *evaluation function* is used to assign a value to a leaf node that is not a terminal node. Positive numbers represent positions advantageous for the player, and negative numbers represent disadvantageous positions. From the point of view of the opponent, these node values are negated because the opponent is the adversary.

The interior nodes are assigned values through a minimizing-maximizing procedure that *backs up* the leaf values towards the root of the tree. It is assumed that the player always plays towards a leaf node with a high value, so an interior node of the player is assigned the arithmetic maximum value of its immediate child nodes. Similarly, the interior nodes of the opponent are assigned the minimum of its child nodes. Because the two sides alternate turns at moving, the interior nodes at even plies are always maximums and nodes at odd plies are always minimums. Note that the minimax value of the tree is a function of the leaf nodes and not of the root position itself. In fact, the principal variation leads to the leaf node that gives the root position its value. If different leaf nodes can have the same value, then there may be multiple principal variations, each backing up the same value to the root position.

The naive method of determining the minimax value of the root of the game tree is to visit all of the leaf nodes and apply the recursive backup procedure at all interior nodes. Clearly, this is a computationally expensive task as the depth d of the tree increases. In fact, the number of nodes increases exponentially with respect to the *branching factor*, the average number of branches from a node. Adapting the notation and terminology of previous authors (for example, [KnM75, MaC82]), the *Number of Bottom Positions* (NBP) [SlD69] or leaf nodes in the *full minimax tree* of uniform width w is:

$$\mathrm{M}_{\mathrm{NBP}}(w,d) = w^d$$

Formula 2.1 — Leaf Nodes in Full Minimax Tree
(M is for minimax, w is uniform width, d is fixed depth)

The total number of leaf and interior nodes in the full minimax tree is called the *Node Count* (NC) (terminology from [Sch86]) and is given by:

$$\mathrm{M}_{\mathrm{NC}}(w,d) = \sum_{i=1}^{d} w^i = \frac{w^{d+1}-1}{w-1}$$

Formula 2.2 — Node Count in Full Minimax Tree
(M is for minimax, w is uniform width, d is fixed depth)

Analytically, it is more convenient to consider only the leaf nodes as given by the NBP measure since the equations are simpler and the number of leaf nodes provides a lower bound on the number of nodes in the tree. Empirically, the NC measure is also useful.

Figure 2.1 shows an example of a full minimax tree of uniform width 2 and fixed depth 3. Note that, in practice, game trees are rarely of fixed depth and uniform width. However, Formulas 2.1 and 2.2 still provide useful measures of the size of the full minimax game tree.

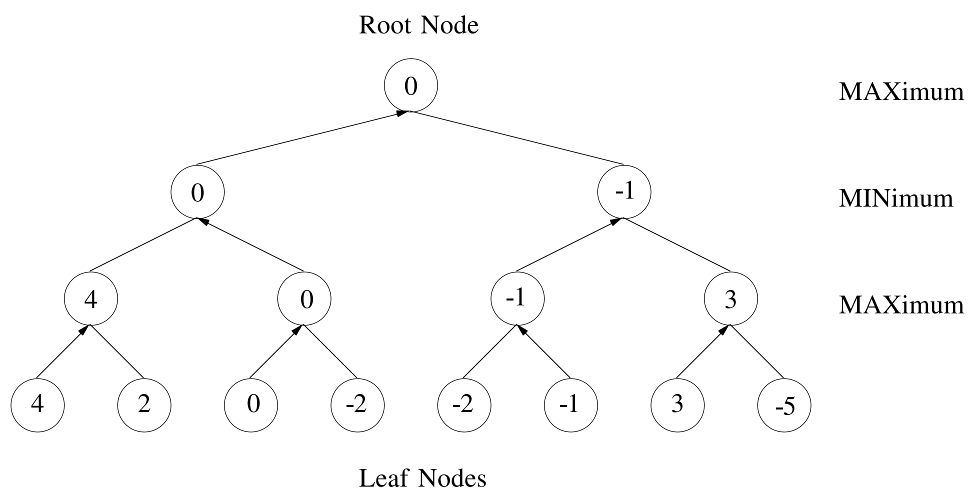

Figure 2.1 — Full Minimax Tree
(w = 2, d = 3. Arrows show path of values backed up from the leaf nodes.)

It is theoretically possible to determine the minimax value of a given game tree without visiting all of the nodes that naive minimax search visits. In fact, there are several algorithms designed to search a minimax tree without building a full minimax tree. Of them, the well-known Alpha-Beta algorithm remains the most popular. The number of leaf nodes in the smallest possible Alpha-Beta game tree or *minimal tree* is given by Formula 2.3 [SlD69,KnM75].

$$\mathrm{MT}_{\mathrm{NBP}}(w,d) = w^{\left\lceil \frac{d}{2} \right\rceil} + w^{\left\lfloor \frac{d}{2} \right\rfloor} - 1$$

Formula 2.3 — Leaf Nodes in Minimal Tree
(MT is for minimal tree, w is uniform width, d is fixed depth)

Of course, Formula 2.3 is still exponential with respect to depth, but it is a significant improvement over the full minimax tree.

2.2 Sequential Alpha-Beta Search

Alpha-Beta is a depth-first search algorithm for minimax trees that computes the correct value of the root position without necessarily visiting all of the tree nodes. Subtrees that are provably irrelevant to the value of the root position may be ignored or cut off for a savings in the search effort. In fact, Alpha-Beta is a special case of the more general class of branch-and-bound algorithms [KnM75]. In the terminology of [MaP85],[1] nodes where all of the subtrees are searched are called *ALL nodes* and nodes where some of the subtrees can be ignored are called *CUT nodes*. A *cutoff* is said to occur at CUT nodes. To detect cutoffs, Alpha-Beta maintains and updates a *search window*, (α,β). Node values strictly within the window, that is greater than α and less than β, are still relevant to the search. Node values outside the window are pruned. Alpha-Beta pruning is not a heuristic because it logically eliminates parts of the game tree and still computes the correct minimax value of the root position.

As an example of Alpha-Beta pruning, consider Figure 2.2. The minimax game tree is identical to the tree in Figure 2.1.

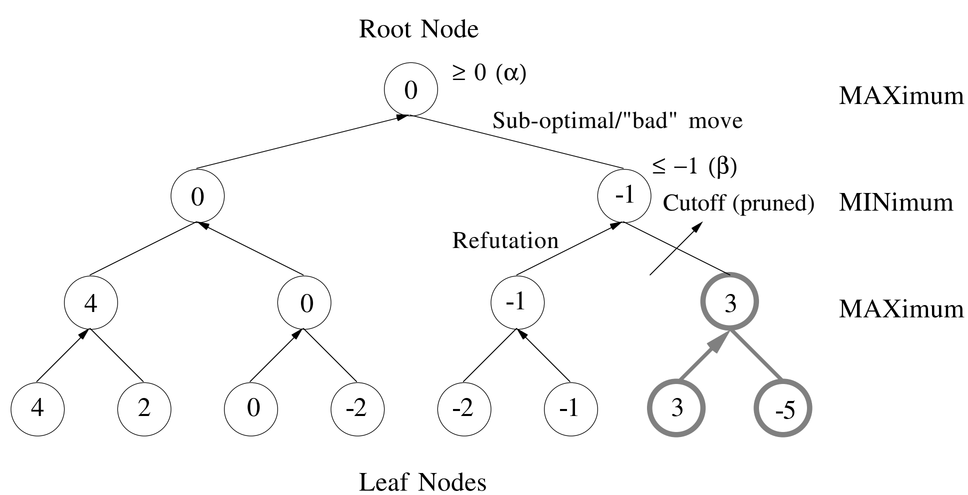

Figure 2.2 — Example of Alpha-Beta Pruning

Assuming a left-to-right order in the depth-first search, the left subtree of the root node returns a value of 0, which is used as the α value, or lower bound, for the search of the right subtree of the root node. When the -1 value is backed up to the MINimum ply, it becomes the β value,

[1] Although this paper uses the terminology in the context of parallel search, they apply equally well to sequential search.

or upper bound, for the search. However, since $\alpha \geq \beta$, the search window is empty and the indicated subtree can be cut off or pruned. In game tree terminology, the move searched just before the cutoff is called the *refutation*, and the move that led to the cutoff node is sub-optimal, or informally, a "bad" move which has been refuted.

A variety of search algorithms are presented in this thesis in a Pascal-like pseudo-code to improve the clarity of the discussion. The pseudo-code is based on the presentation by Schaeffer [Sch87] and others (for example, [MaC82]). The program header in Figure 2.3 should be logically added to all of the pseudo-code to follow. Although the use of global variables is generally discouraged, their advantages will become apparent in later chapters.

```
PROGRAM
    GameTreeSearch;

VAR { Global Variables }
    TreeDepth : integer;                                { Depth of search tree to build }
    Width   : array[ 1..MAX_DEPTH ] of integer;         { Branching factor}
    Moves   : array[ 1..MAX_DEPTH ] of
              array[ 1..MAX_WIDTH ] of moveType;        { Moves leading to children }
    Index   : array[ 1..MAX_DEPTH ] of integer;         { Index of next move }
    Alpha   : array[ 1..MAX_DEPTH ] of integer;         { Alpha bound }
    Beta    : array[ 1..MAX_DEPTH ] of integer;         { Beta bound }
    Score   : array[ 1..MAX_DEPTH ] of integer;         { Best score so far }
    Best    : array[ 1..MAX_DEPTH ] of integer;         { Index of best move }

BEGIN { Program }
```

Figure 2.3 — Game Tree Search Program Header
(Global Variables Pseudo-Code)

The pseudo-code for the Alpha-Beta algorithm is shown in Figure 2.4. The details of some of the functions, `Evaluate()`, `legalMoves()` and `orderMoves()`, have been omitted. Also, if `p` is a position and if `move` is a legal move, then the notation `p.move` refers to the position reached by playing `move` in `p`.

The pseudo-code includes the common negamax and fail-soft enhancements to the basic Alpha-Beta algorithm. The *negamax framework* swaps and negates the bounds of the search window and also negates the returned value. It is equivalent to the minimizing-maximizing formulation of the original Alpha-Beta algorithm, but is more convenient because it always computes the arithmetic maximum instead of alternating with the arithmetic minimum. The pseudo-code also contains the *fail-soft* enhancement introduced by Fishburn [Fis81]. By initializing the value of `Score[ ply ]` to $-\infty$, the value returned by fail-soft Alpha-Beta is the maximum score of the subtrees visited, even if that value is less than the alpha bound itself. If no cutoff occurs, `Score[ ply ]` contains the value of the best subtree. As we will see later, this bounds information can be useful in certain search strategies. Although technically a misnomer, the fail-soft version of Alpha-Beta is often what is intended when referring to the Alpha-Beta algorithm, including in this thesis.

```
FUNCTION
    AlphaBeta( p : position; alpha, beta, ply : integer ) : integer;

VAR
    index, value : integer;                     { Index and value of current move }
    move : moveType;                            { Current move being searched }

BEGIN { AlphaBeta }

{ Check if at leaf node }
    if ( ply = TreeDepth ) then                 { At leaf node? }
        return( Evaluate( p ) );                { Yes.  Statically evaluate }

{ Generate legal moves and order them }
    Width[ ply ] := legalMoves( p, Moves[ ply ] );      { Store moves in Moves[] }
    if ( Width[ ply ] = 0 ) then                { Any legal moves in this position? }
        return( Evaluate( p ) );                { No.  In checkers, this is a loss }
    orderMoves( Moves[ ply ] );                 { Yes. Do move ordering }

{ Set up global variables }
    Index[ ply ] := 1;                          { Start with best ordered subtree }
    Alpha[ ply ] := alpha;
    Beta[ ply ] := beta;
    Score[ ply ] := −∞;                         { Initialize as per fail-soft }
    Best[ ply ] := UNKNOWN;

{ Search each subtree }
    while ( Index[ ply ] ≤ Width[ ply ] ) do
      begin { Recurse }                         { Search each subtree }
        index := Index[ ply ];                  { An unique move leads to the subtree }
        move := Moves[ ply ][ index ];
        Index[ ply ] := Index[ ply ] + 1;       { Advance index }

        value :=                                { Negamax the value and parameters }
            −AlphaBeta( p.move, −Beta[ ply ], −Alpha[ ply ], ply + 1 );

        if ( value > Score[ ply ] ) then        { If subtree is better, remember  it }
          begin { Update Best }
            Score[ ply ] := value;
            Best[ ply ] := index;
          end;  { Update Best }

        if ( Score[ ply ] ≥ Beta[ ply ] ) then          { Cutoff? }
            Index[ ply ] := ∞;                          { Yes.  Force exit from loop }
        Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );   { Improve window? }
      end;  { Recurse }

    return( Score[ ply ] );                     { Done. Return best score }
END;  { AlphaBeta }
```

Figure 2.4 — Negamax Version of Fail-soft Alpha-Beta Algorithm
(Boldface shows points of recursion or where the recursion ends.)

2.3 Move Ordering

Achieving quick cutoffs at CUT nodes through *move ordering* is one key to the efficiency of the algorithm. For example, in Figure 2.2, if the subtree that was pruned had been searched before its sibling (to the left), then there would not have been a cutoff. Ideally, a move that causes the cutoff should be searched first.

In a *best ordered* (or *perfectly ordered*) game tree, the best move is searched first and a cutoff occurs immediately after it, if it is a CUT node. At an ALL node, the best move gives the node its value. Assuming uniform width and fixed depth, if a game tree is best ordered at ALL nodes and at CUT nodes, Alpha-Beta builds the minimal tree. In a *worst ordered* tree, the subtrees are ordered from worst to best, thus the best subtree is searched last. The cutoff occurs only after all of the subtrees have been searched at the CUT node giving no savings in search effort. Alpha-Beta builds the full minimax tree if the moves are worst ordered. *Randomly ordered* game trees, where the best subtree is equally likely to be in any order, have also been studied. However, they do not represent what is observed in practice [Hsu90].

A game tree is *strongly ordered* if the first subtree searched is best 70 percent of the time and if the best subtree is in the first quarter of the branches 90 percent of the time [MaC82]. Although the classifications are non-analytical, it is generally agreed that the trees of efficient search implementations are strongly ordered. We can summarize the relationship between games trees built with different move orderings by [MaC82]:

Minimal Tree < Strongly Ordered Tree < Randomly Ordered Tree « Full Minimax Tree

Figure 2.5 — Relative Sizes of Game Trees with Different Move Orderings
(Minimal trees are best ordered. Worst ordered trees are full minimax trees.
Uniform width and depth is assumed.)

Several heuristics are commonly used to build strongly ordered game trees in practice. As with most rules of thumb, there are cases in which the heuristics fail. Furthermore, there is often no analytical characterization of the probability that the heuristic is correct in a given situation.

One heuristic combines iterative deepening with transposition tables. *Iterative deepening* is a technique that builds successively deeper game trees. The move ordering information gathered by building the game tree of depth n is used to help order the moves for the game tree of depth $(n + s)$, where s is the step between iterations. The best move for a node can be stored in a *transposition table*, which is hashed according to the board position represented by the node. During the next search iteration, the transposition table is queried and the suggested best move from the previous iteration is searched first. The heuristic is that the best move of the shallower search is likely to be the best move of the deeper search. More generally, iterative deepening is also used to regulate the time taken to decide on the best move for the root position and the transposition table also allows identical subtrees, created by transposing lines of play, to be cached and searched only once.

The *history heuristic* [Sch86,Sch89b] orders all of the moves at a node and not just the best move. It uses tables to accumulate a history of how often a move is shown to be best. The history score of a move is increased each time it is empirically shown to be best. Therefore, moves can be sorted according to their history scores. The heuristic is that the best moves at nodes in other parts of the tree have a history of being best and are likely to be best at the current node as well. If the transposition table cannot suggest a possible best move because the table entry is non-existent or has been overwritten, the history heuristic can suggest a best move.

Of course, there is no guarantee that the suggested best move of either the transposition table or history heuristic is actually best. In those cases, the ordering of the sibling moves by the history heuristic is in decreasing empirical probability of being the actual best move. In practice, the suggested best move of the transposition table is usually searched first and then the remaining moves are searched in the order suggested by the history heuristic. It is important to order all of the moves at a node. In the context of Figure 2.4, the function `orderMoves()` is considered to implement a move ordering mechanism, such as transposition tables supplemented with the history heuristic.

The popular *killer heuristic* [SlA77] is a special case of the history heuristic. There are several other move ordering mechanisms also in use [Sch89b,Mar92].

2.4 Aspiration Windows

The window (α,β) with which the search of a node is initiated is referred to as the *full window*. A search window of $(-\infty,+\infty)$ declares that all node values are relevant to the search since all possible node values fall inside the *infinite window*. However, it is possible to initiate a search with an artificially narrowed *aspiration window*. With window-based minimax algorithms, there is a well-known relationship between the size of the search window and the size of the search tree generated [Sch86]. Generally, the smaller the window, the smaller the tree that is searched. Intuitively, if the window represents the parts of the search space that are of interest, then a smaller search window will build a smaller tree because parts of the search space have already been declared *a priori* to be of no interest. Within certain limitations, aspiration windows can be used to search trees more efficiently.

With Alpha-Beta, the returned minimax value is correct only if it falls within the initial search window. If the minimax value is not inside the initial window, then the search is considered incomplete. The *aspiration search* generated an improper cutoff and did not search the part of the tree that determines its minimax value. Therefore, although there may be strong heuristic reasons to eliminate some node values from consideration *a priori*, there are no algorithmic justifications for using anything less than a full window. An aspiration window is a heuristic gamble that the final minimax value lies within the smaller search window, and can therefore be found with a smaller search tree.

One example of using an aspiration window is at the root of the game tree. If the previous iteration of iterative deepening returned a value of v for the root position, the next search can be initiated with a finite aspiration window covering a range centered on v. The size of the window is implementation dependent. If the score of the root position is stable between iterations, then it is likely that the next iteration will return the a value within the aspiration window.

Taking the concept of aspiration windows and fail-soft Alpha-Beta to the extreme results in the notion of *minimal windows* [Pea80,Fis81], and *minimal window searching*. Assuming a strongly ordered game tree, it is likely that the first subtree searched either determines the value of the node or causes a cutoff. We need to search the first branch of each node with a full window because we want to know the exact minimax value, say v, of the subtree. However, we do not need to know the minimax value of the other subtrees. We can settle for merely proving that the other branches are no better than the first branch. Therefore, we search all subtrees other than the first one with the minimal window $(v,v + 1)$ which has *zero width*. The window is different but the fail-soft Alpha-Beta search algorithm does not change. Minimal window search is not used to determine the actual value of a subtree, but to determine bounds on its relative value.

Recall that with fail-soft Alpha-Beta, the value returned from a search, even if the exact minimax value lies outside of the initial window, provides bounding information. Suppose the full window of the node is (α,β) and the minimal window used is $(v,v + 1)$. If the search returns a value z, and $z \leq v$, the search is said to *fail low* and while the exact value of the subtree is not known, it is known to be no better than v. We determined this as cheaply as possible since a minimal window was used. If $z \geq \beta$, then a normal Alpha-Beta cutoff has occurred and the node does not need to be expanded any further. Otherwise, if $v < z < \beta$, the search is said to *fail high*. It needs to be re-searched with a full window to find its true minimax value. A narrower window of (z, β) can also be used because the fail-soft Alpha-Beta returned z, which is a lower bound on the actual minimax value of the subtree.

The main disadvantage of minimal window searching is the danger of having to re-search a fail high subtree. The main advantage of minimal window searching is its ability to determine the value of a node with potentially less effort. Only the best move subtree is searched with a full window and the sibling subtrees are proven inferior with a relatively inexpensive minimal window search. Overall, the search savings due to minimal window searches of strongly ordered game trees outweigh the costs of re-searches [Mar83].

2.5 Structure of the Minimal Alpha-Beta Tree

The minimal game tree searched by the fail-soft Alpha-Beta algorithm has a predictable and important structure. Cutoffs can only occur at certain nodes and can never occur at other nodes. In terms of Knuth and Moore's classification [KnM75], Type 1 and Type 3 nodes are ALL nodes. The Type 2 nodes are CUT nodes. The pseudo-state diagram in Figure 2.6 below shows the relationship between the different node types.

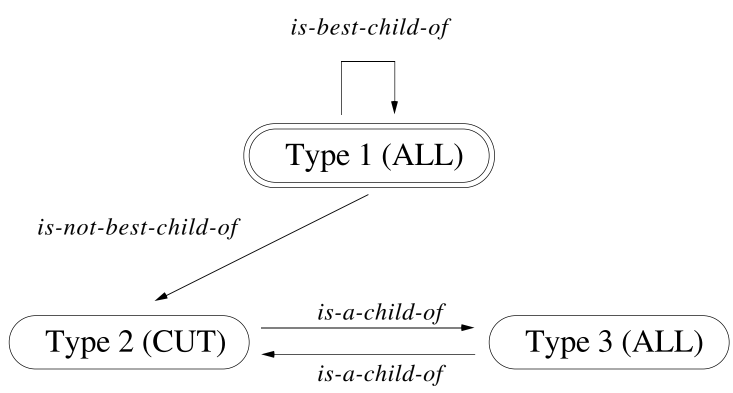

Figure 2.6 — State Diagram For Node Types

Cutoffs can occur at CUT nodes, but in worst case move ordering, all subtrees may actually be searched. With strongly ordered game trees, the worst case scenario is infrequent. Figure 2.7 shows the *ALL/CUT node structure* of the minimal Alpha-Beta game tree.

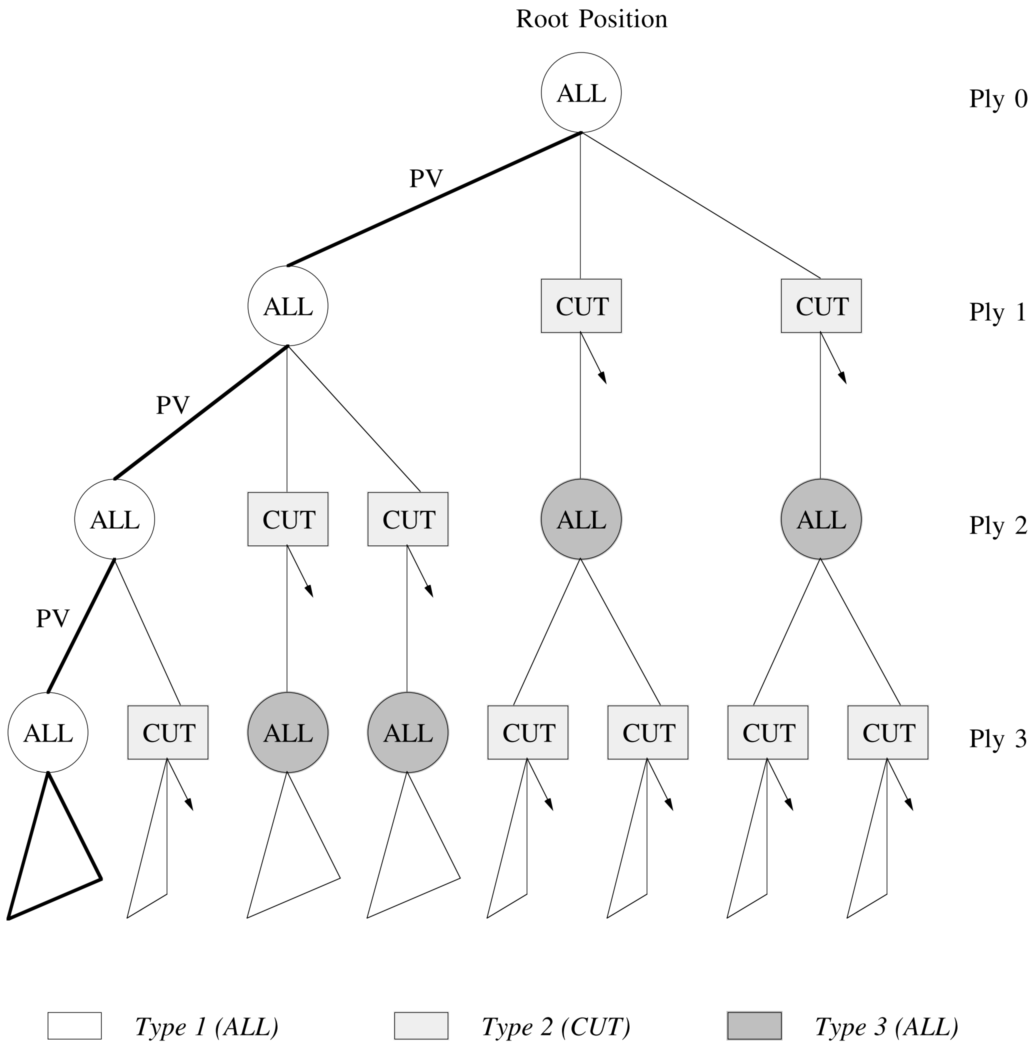

Figure 2.7 — ALL/CUT Node Structure of the Minimal Alpha-Beta Game Tree
(The principal variation (PV) moves are the best subtrees at Type 1 ALL nodes.)

The structure of the game tree in Figure 2.7 is true in the absence of aspiration window searches, which is the case for the fail-soft Alpha-Beta algorithm. When using minimal window searches,[2] there is one important change to the structure: at Type 3 (ALL) nodes, it may be possible to achieve a cutoff before all of the subtrees are searched. That is to say, Type 3 (ALL) nodes may in fact behave like CUT nodes if minimal windows are used. Intuitively, if the search window is of zero width, then any variation in the value of a subtree is enough to cause a cutoff. In fact, it is the possibility of generating a cutoff at a Type 3 (ALL) node that makes minimal window search more efficient than a full window search.

---

[2] Which should not be confused with the notion of the minimal Alpha-Beta tree.

2.6 Principal Variation Search

Recognizing the usefulness of minimal window search with the strongly ordered game trees seen in practice, a new variation of Alpha-Beta search, *Principal Variation Search* (PVS), was introduced [Fis80,Fis81,MaC82]. The principal variation determines the minimax value of a given node, therefore PVS searches the first subtree, as determined by the move ordering, with a full window. If a fail high situation occurs during a subsequent minimal window search, the original choice for the best move was in fact not best and the fail high move needs to be re-searched. In effect, PVS heuristically narrows the windows of the non-principal variation moves to minimal windows.

Figure 2.8 gives the pseudo-code for Principal Variation Search. The main difference between the pseudo-code for Alpha-Beta and for PVS is the special treatment of the first ordered move. In effect, PVS unrolls the `while`-loop and searches the principal variation with a full window. The other subtrees are searched with a minimal window based on the value of the principal variation. The shading in the pseudo-code highlights the key differences between Alpha-Beta and PVS. Also note that PVS recursively calls itself and never calls Alpha-Beta, which was a technique introduced to minimal window searching by Marsland [Mar83] and, independently, Reinefeld in his *Negascout* algorithm [Rei83].

```
FUNCTION
    PVS( p : position; alpha, beta, ply : integer ) : integer;

VAR
    index, value : integer;                 { Index and value of current move }
    move : moveType;                        { Current move being searched }

BEGIN { PVS }

{ Check if at leaf node }
    if ( ply = TreeDepth ) then             { At leaf node? }
        return( Evaluate( p ) );            { Yes.  Statically evaluate }

{ Generate legal moves and order them }
    Width[ ply ] := legalMoves( p, Moves[ ply ] );      { Store moves in Moves[] }
    if ( Width[ ply ] = 0 ) then            { Any legal moves in this position? }
        return( Evaluate( p ) );            { No.  In checkers, this is a loss }
    orderMoves( Moves[ ply ] );             { Yes. Do move ordering }

{ Set up global variables }
    Index[ ply ] := 2;                      { Continue with other subtrees, later }
    Alpha[ ply ] := alpha;
    Beta[ ply ] := beta;

{ Search first ordered move with full window. Unroll the loop by one }
    Score[ ply ] :=
        -PVS( p.Move[ ply ][ 1 ], -Beta[ ply ], -Alpha[ ply ], ply + 1 );
    Best[ ply ] := 1;

    if ( Score[ ply ] ≥ Beta[ ply ] ) then  { Cutoff after pv? }
        Index[ ply ] := ∞;                  { Yes.  Will never enter loop }
    Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );     { Improve window? }

{ Search each sibling subtree with minimal window }
    while ( Index[ ply ] ≤ Width[ ply ] ) do
      begin { Recurse }                     { Search each subtree }
        index := Index[ ply ];              { An unique move leads to the subtree }
        move := Moves[ ply ][ index ];
        Index[ ply ] := Index[ ply ] + 1;   { Advance index }

        value :=                            { Negamax value and minimal window }
            -PVS( p.move, -Alpha[ ply ] - 1, -Alpha[ ply ], ply + 1 );

        if ( value > Score[ ply ] ) then    { Score improved, but did it fail high? }
         begin { Update Best }
            if ( value  > Alpha[ ply ] and value < Beta[ ply ] ) then
                Score[ ply ] :=             { Yes.  Re-search with full window }
                    -PVS( p.move, -Beta[ ply ], -value, ply + 1 );
            else
                Score[ ply ] := value;      { No }
            Best[ ply ] := index;
          end;  { Update Best }

        if ( Score[ ply ] ≥ Beta[ ply ] ) then      { Cutoff? }
            Index[ ply ] := ∞;                      { Yes.  Force exit from loop }
        Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );   { Improve window? }
      end;  { Recurse }

    return( Score[ ply ] );                 { Done. Return best score }
END;  { PVS }
```

Figure 2.8 — Principal Variation Search (PVS)
(Shading indicates key changes from `AlphaBeta()`)

The efficiency of PVS depends on move ordering. With best ordered trees, the first subtree searched is always on the principal variation, therefore there are never any fail high situations. Searching the best move first also provides the actual value of the node, say v, so that the remaining moves can be searched with a minimal window $(v,v + 1)$ centered on the correct value. Those minimal window searches will never fail high. However, when the move ordering deviates from best ordered, the best move is not expanded first and so an erroneous minimal window of $(v',v' + 1)$ where $v' < v$ is used. The minimal window search of the actual best move will fail high and the resulting re-search will return the value v. It is important that the newly determined value v be used for the minimal windows of the remaining subtrees. There may be other moves with the same value v as the best move. If those maximal subtrees are searched with $(v',v' + 1)$ they will also fail high and be needlessly re-searched. Fortunately, with the strongly ordered game trees encountered in practice, the first move is often best.

2.7 Alpha-Beta as a Proof Procedure

An interesting way to think of the Alpha-Beta algorithm is to view it as a guess-and-check proof procedure. Intuitively, the algorithm "guesses" at the best move at each node in the game tree. Then, it searches that line of play further to "check" if the move is indeed best or it finds a counter-example or *refutation* to the optimality of the chosen move. In practice, the guess of the best move and the guess of the best refutation are both determined by the various move ordering mechanisms in use.

Proving that a move is best requires searching all other possible moves and showing that they are inferior to the best, or principal variation, move. No move on the principal variation can be eliminated from consideration. That is why the principal variation consists only of Type 1 (ALL) nodes.

Proving that a move is inferior only requires showing that there is at least one response by the other side for which all of the counter-responses lead to inferior positions. For example, if the player chooses a move that allows the opponent to win a checker and all of the player's responses cannot win back the checker, then the inferiority of the player's original move is established. Although there may be other refutations available to the opponent, only one is needed and the remaining subtrees are cut off. This is the intuition behind Alpha-Beta pruning. A Type 2 (CUT) node is reached by an inferior move from the parent node and all of the moves at the following Type 3 (ALL) node cannot overcome that previous inferior move because they lead to inferior positions themselves, as in the above example. This explains the strict alternating pattern between Type 2 and Type 3 nodes indicated by Figures 2.6 and 2.7. It also explains why the alternation between Type 2 and Type 3 nodes begins at the point of deviation from the principal variation.

If a non-principal variation move is actually superior, as in a fail-high situation, then the behaviour of the Type 2 and Type 3 nodes are reversed. Because the original hypothesis that the non-principal variation is inferior is false, the search cannot actually prove the inferiority of the move. Instead of the other side having at least one refutation, it has no refutation and all subtrees of the Type 2 node must be searched to discover this, contrary to its predicted behaviour as a CUT node. And for each move from the Type 2 node, there is at least one response that improves the position for the other side. Once that move is discovered, a cutoff may occur at the Type 3 node, which is contrary to its predicted behaviour as an ALL node.