3 CHAPTER

GAMES SOLVING (ADVERSIAL SEARCH)

3.0 INTRODUCTION

Adversial search technique is a mixture reasoning and creativity. Game playing is often called as an **adversial search**. Successful game playing is generally deemed to require intelligence. Game playing is a search problem with the following components—

(*a*) Initial state of the game.

(*b*) Operators defining legal moves.

(*c*) Successor function.

(*d*) Terminal test (defines end-of-game state).

(*e*) Goal test.

(*f*) Path cost (or utility or pay-off function).

Game playing has been one of the most publicized area of AI research. With the success of **Deep Blue** (1997) another landmark was reached i.e. a computer program that could defeat the best chess player in the world. Game playing happens to be a good domain to explore machine intelligence because of two main reasons:

(*a*) They provide a structured task in which it is very easy to measure success or failure.

(*b*) They did not require large amount of knowledge. They are solvable by straight forward search from starting state to final (winning) state.

3.1 GAME PLAYING

To solve our game problems, we draw a game tree (or graphs) to show all possible game states. The nodes are linked by moves (arcs). The branches are labeled with the players who can make the choice between them. In a game tree, alternative layers will be controlled by different players.

3.1.1 Game Trees

A game tree is defined as an instance of a tree in which the root node represents the state before any moves have been made, the nodes in the tree represent possible states of the games (or positions) and arcs in the tree represent moves. We represent the moves on alternate levels of the game tree so that all edges leading from root node to the first level represent possible moves for the first player and edges from the first level to the second represent moves for the first player and edges from the first level to the second represent moves for the second player and so on. The leaf nodes in this tree represent **final states** (where the game has been won, lost or drawn). In simple games, a **goal node** might represent a state in which the computer has won but for complex games (like chess, go etc) the concept of a goal state is rarely of any use.

One method of playing a game might be for the computer to use a tree search algorithm such as dfs or bfs, searching for a goal state i.e. the final state of the game where the computer has won). **But please note that this approach does not work because there is another intelligence involved in the game.** We will consider this to be a rationally informed opponent who plays to win. Whether this opponent is human or another computer does not matter.

Iterative methods apply here because search space is too large for interesting games to search for a solution. Therefore, search will be done before each moves in order to select the best move to be made.

Adversary methods apply here because an opponent who is trying to win, makes alternate moves, not controllable by you.

Evaluation function is used to evaluate the "**goodness**" of a configuration of the game. For a computer to use game-tree to make decisions about moves in a game of tic-tac-toe, it needs to use an evaluation function which enables it to decide whether a given position in the game is good or bad. If we use exhaustive search, then we only need a function that can recognize a win, a loss or a draw. Then the computer can treat "win" states as goal nodes and carry out search in normal way. **Also note that these evaluation functions are known as static evaluators because they are used to evaluate a game from just one static position. Since the size of the game tree is very large, so it is almost never possible to search the game tree completely.** In other words, the search will rarely reach a leaf node in the tree (at which the game is said to be either won, lost or drawn). This means that the software needs to be able to cutoff search and evaluate the position of the board at that node. Hence, an evaluation function is used to examine a particular position of the board. So, evaluation function must be efficient. This will not slow down the game play. To be effective, the evaluation function needs to give a higher score to a better position. For a chess game, this function will be very complex. These functions are usually weighted linear functions meaning that a number of different scores are determined for a given position and simply added together in a weighted fashion.

For e.g., For a **chess game**, a very **simple evaluation function might be—**

"Count the number of queens, the number of pawns, the number of bishops and so one and add them up using weights to indicate the relative value of those pieces"

That is,

$$\text{Score} = 9q + 5r + 3b + 3n + p$$

where q-number of queens.

r-number of rooks..

n-number of knights.

b-number of bishops.

p-number of pawns.

If the two computer programs were to compete with each other at a game (say checkers) and that they had

— Same processing capabilities and speeds.

— Used same algorithms for examining the search tree,

Then the game would be decided by the quality of the program's evaluation functions. In general, the evaluation functions for game playing programs do not need to be perfect but need to give a good way of comparing two positions to determine which is better. **But please understand that in complex games like chess, Nim, Othello, Checkers, Tic-Tac-Toe, Go etc. this is not an easy question as two grandmasters will sometimes differ on the evaluation of a position.** One way to develop an accurate evaluation function is to actually play games from each position and see who wins. If the play is perfect on both the sides then this will give a good indication of what the evaluation of the starting position should be.

For example, for Tic-Tac-Toe the evaluation function, f(n) is as follows—

$f(n)$ = [number of 3-lengths open for me] – [number of 3-lengths open for you]

Where a 3-length is a complete row, column or diagonal.

Most evaluation functions are specified as a weighted sum of "features":

(W1 *feature 1) + (W2 *feature 2) +...+ (Wn *feature n)

Like, in a chess game, some features evaluate piece placement on board and other features describe configurations of several pieces. The need today is to develop an evaluation function that is dynamic and is able to accurately evaluate positions it has never seen before.

Most of the games considered here are **zero-sum games. It means that if the overall score at the end of a game for each player can be 1 (a win), 0 (a draw) or – 1 (a loss), then the total score for both players for any game must always be 0. This means that if one player wins, the other must lose.** The only other alternative is that both players draw. That is why, we consider the search techniques that are discussed here to be adversarial methods because each player is not only trying to win but to cause the opponent to lose. In the algorithms like minimax and Alpha-Beta (discussed next), it is important that the computer can assume that the opponent is rational and adversarial. That is, **the computer needs to assume that the opponent will play to win.**

3.1.2 Minimax Algorithm

Minimax search procedure is a depth-first search procedure. When evaluating game trees, it is usual to assume that the computer is attempting to maximize some score that the opponent is trying to minimize. Normally, we would consider this score to be the result of the **evaluation function** for a given position, so we would usually have a **high positive score means a good position for the computer, a score of 0 means a neutral position and a high negative score means a good position for the opponent.**

Principle: The basic principle behind minimax is that a path through the tree is chosen by assuming that at its turn (a max node), the computer will choose the more that will give the highest eventual static evaluation and that at the human opponent's turn (a min node), he or she will choose the move that will give the lowest static evaluation.

The Minimax algorithm is used to choose good moves. It is assumed that a suitable static evaluation function is available, which is able to give an overall score to a given position. In applying Minimax the static evaluator will only be used on leaf nodes. The values of the leaf nodes will be filtered up through the tree, to pick out the best path that the computer can achieve. This is done by assuming that the opponent will play rationally and will always play the move that is best for him or her and thus worst for the computer. So, the computer's aim is to maximize the lowest possible score that can be achieved.

As an example, consider a simple game tree. Fig. 3.1 shows how minimax works on it.

Please note that the best result that max can achieve is a score of 6. If max chooses the left branch as its first choice then min will choose the right branch, which leaves max a choice of 1 or 3. In this case, max will choose a score of 3. If max starts by choosing the right branch, min will have a choice between a path that leads to a score of 7 or a path that leads to a score of 6. Therefore, it will choose the left branch, leaving max a choice between 2 and 6. Fig. 3.1 shows how Minimax can use dfs to traverse the game tree. The arrows start from the rest node at the top and go down to the bottom of the left branch. This leads to a max node, which will get a score of 5. The value 5 is, hence, passed up to the parent of this max node. Following the right path

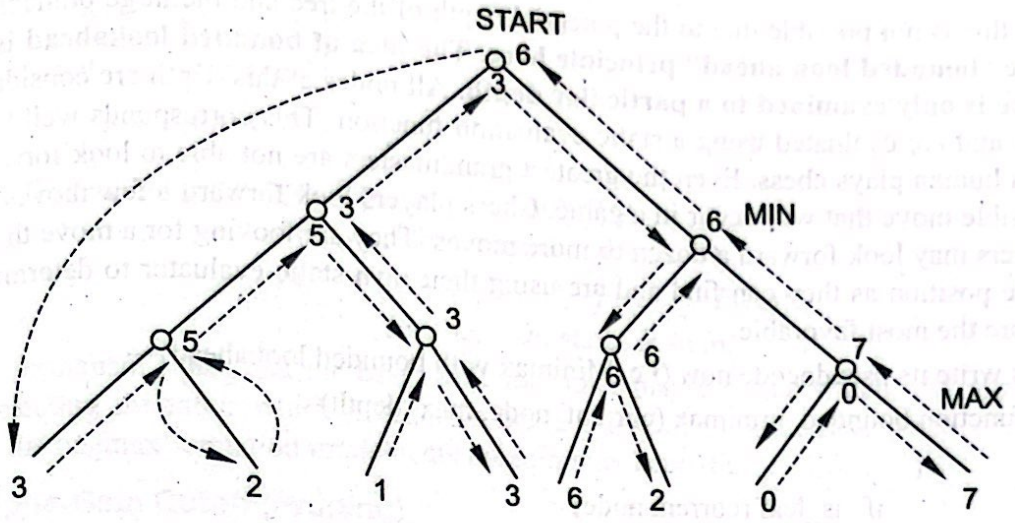


Fig. 3.1 A minimax example.

from this min node leads to another max node, this time getting a score of 3. **Also note that the arrows show the order in which the nodes are examined by the algorithm and the values that are passed through the tree.** This comes back up to the min node which now chooses the minimum of 3 and 5 and selects 3. Eventually, having traversed the whole tree, the best result for max comes back upto the root node (6).

Let us write its psendo-code now—

```
function minimax (current-node)
        {
          if  is-leaf (current-node)
              return static-evaluation (current-node);
          if  is-min-node (current-node)
              return min (minimax (children-of (current-node)));
          if  is-max-node (current-node)
              return max (minimax (children-of (current-node)));
        }
```

The else part is never required as every node must be a leaf node, a min node or a max node only.

We also observe that this is a **recursive function** because to evaluate the scores for the children of the current node, this algorithm must be applied recursively to those children until a leaf node is reached. **Also understand that Minimax can be done non-recursively, starting at the leaf nodes and going up the tree in a recursive breadth first search.**

Limitations of Minimax:

1. The effectiveness of Minimax is limited by the depth of the game tree which is itself limited by the time needed to construct and analyze it. The time increases exponentially with the depth of the tree.
2. A rapid tie between the assigned and actual value to static evaluation function fails to decide the expansion of game tree.

3.1.3 Horizon Effect Problem

Minimax algorithm is unsuitable for use in many games like chess where the game tree is very large. The problem is that in order to run Minimax, the entire game tree must be examined and for games

like chess this is not possible due to the potential depth of the tree and the large branching factor. So, we use **"bounded look ahead" principle here. The idea of bounded lookahead is that the search tree is only examined to a particular depth.** All nodes at this depth are considered to be leaf nodes and are evaluated using a static evaluation function. This corresponds well to the way in which a human plays chess. Even the greatest grandmasters are not able to look forward to see every possible move that will occur in a game. Chess players look forward a few moves and good chess players may look forward a dozen or more moves. They are looking for a move that leads to a favorable position as they can find and are using their own static evaluator to determine which positions are the most favorable.

Let us write its pseudocode now (i.e. Minimax with bounded lookahead)—

```
function bounded_minimax (current_node, max_depth)
        {
          if  is_leaf (current node)
              return static-evaluation (current-node);
          if  (depth-of (current-node) = = max-depth)
              return static-evaluation (current-node);
          if  (is-min-node (current-node))
              return min (minmax (children-of (current-node)));
          }
```

Actually, it is not reasonable to apply a fixed cut-off point for search. A bounded cut-off at a particular node can leave any one of the opponent in a problem. One way of avoiding this problem is to cut-off search at positions that are deemed to be **quiescent. Please note that a quiescent position is one where the next move is unlikely to cause a large change in the relative positions of the two players. Also note that a position where a piece of chess can be captured without a corresponding recapture is not quiescent.**

This bounded Minimax Search has another problem—**the horizon effect,** coined by Berliner in 1977.

Horizon effect: This problem involves an extremely long sequence of moves that clearly lead to a strong advantage for one player, but where the sequence of moves, although potentially obvious to a human player, takes more moves than is allowed by the bounded search. Hence, the significant end of the sequence has been pushed over the horizon. This problem occurs in a checkers playing program named as Chinook.

What is the solution?

There is **no universal solution** to the horizon problem but one method to minimize its effect is to always search a few ply deeper when a position is found that appears to be particularly good. The **singular-extension heuristic** is defined as follows:

"If a static evaluation of a move is much better than that of other moves being evaluated, continue searching."

It is possible for the nodes or positions to have same heuristic scores. This means that they lie on a **plateau** as in case of hill climbing. They tell nothing about the movement of the game. The better estimate of the score for each position is obtained beyond plateau. This is a **horizon effect.**

In 1975, Donald knuth and R.W. Moore introduced a **negmax procedure** which uses the same function (as used in minimax function) to evaluate each node in backing up from the successor nodes. Say, the evaluation function for the jth terminal node in a complete tree is as follows—

$$\left.\begin{array}{ll} f_j = -1 & \text{(for loss)} \\ f_j = 0 & \text{(for draw)} \\ f_j = +1 & \text{(for win)} \end{array}\right\}$$

Then, the negmax evaluation function $F(j)$ is given by

$F(j)$ = f_j for terminal nodes

$F(j)$ = MAX $\{-F(i1), -F(i2), \ldots -F(\text{in})\}$

for successor states $i1$ to in.

This formalism states that the best move for either player is that which maximizes F(j). By simply selecting the move which maximizes the negative of the successor position's evaluation function, the negmax formation implements the minimax heuristic.

3.1.4 Alpha-Beta Cutoff (Pruning)

Bounded lookahead makes our game tree smaller to examine. In some cases, it is extremely useful to be able to prune (cut) sections of the game tree. It is possible to remove sections of the game tree that are not worth examining, to make searching for a good move more efficient.

Principle: If a move is determined to be worse than another move that has already been examined, then further examining the possible consequences of that worse move is pointless.

Consider a partial game tree as shown in Fig. 3.2.

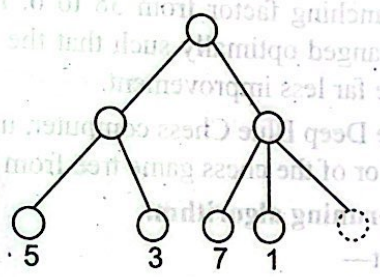


Fig. 3.2 A partial game tree.

This is a tree with five leaf nodes. The top arc represents a choice by the computer and so is a maximizing level. That is, the top node is a **max node**. After calculating the static evaluation function for the first four leaf nodes, it becomes unnecessary to evaluate the score for the fifth node (shown as a dotted circle). Let us see why?

In choosing the left-hand path from the root node, it is possible to achieve a score of 3 or 5. Because this level is a minimizing level, the opponent can be expected to choose the move that leads to a score of 3. So, by choosing the left-hand arc from the root node, the computer can achieve a score of 3. By choosing the right-hand arc, the computer can achieve a score of 7 or 1 or a mystery value. Because the opponent is aiming to minimize the score, he or she could choose the position with a score of 1, which is worse than the value the computer could achieve by choosing the left-hand path. So, the value of the right most left node doesn't matter. The computer must not choose the right-hand arc because it definitely leads to a score of at best 1, assuming the opponent does not irrationally choose the 7 option.

In this example, alpha-beta pruning removes only one leaf node from the tree. But in larger game trees, it can result in a valuable reductions in tree size. In 1993, Winston showed that it will not necessarily remove large positions of a game tree. But infact, in worst case, alpha-beta pruning will not prune any searches from the game tree but even in this case it will compute the same result as Minimax and will not perform any less efficiently.

Analysis of Alpha-Beta Pruning

Alpha-Beta Pruning method provides its best performance when the game tree is assigned such that the best choice at each level is the first one (i.e., the left most choice) to be examined by the algorithm using alpha-beta cut-off will examine a game tree to double the depth that a Minimax algorithm without alpha-beta pruning would examine in the same number of steps. How? If a game tree is arranged optimally, then the number of nodes that must be examined to find the best move using alpha-beta pruning can be derived as found—

$$S = \begin{cases} 2b^{d/2} - 1 & \text{if } d \text{ is even.} \\ b^{(d+1)/2} + b^{(d-1)/2} - 1 & \text{if } d \text{ is odd.} \end{cases}$$

where b–branching factor of game tree.

d–depth of game tree

S–number of nodes that must be examined.

$\therefore \quad S \approx 2b^{d/2}$

Without alpha-beta pruning, where all nodes must be examined:

$$S = b^d$$

$\therefore$ We find that using alpha-beta pruning reduces the effective branching factor from b to $\sqrt{b}$ meaning that in a fixed period of time, Minimax with alpha-beta pruning can look twice as far in the game tree as Minimax without pruning. This represents a significant improvement. For e.g., in chess it reduces the effective branching factor from 38 to 6. But it must be remembered that it assumes that the game tree is arranged optimally such that the best choice is always the leftmost choice. In reality, it might provide far less improvement.

For e.g., In implementing the Deep Blue Chess computer, use of alpha-beta method did in fact reduce the average branching factor of the chess game tree from 38 to around 6.

Implementation of alpha-beta pruning algorithm.

Let us write its algorithm first—

S1: The game tree is traversed in depth-first order. At each non-leaf node, a value is stored. For **max nodes,** this value is called alpha and for **min nodes** the value is **beta.**

S2: An **alpha value** is the **maximum** (best) **value** found so far in the max node's descendants.

S3: A **beta value** is the **minimum** (best) **value** found so far in the min node's descendants.

We write the function alpha-beta now.

```
function alpha-beta (current-node)
    {
      if  is-leaf (current-node)
          return static-evaluation (current-node)
      if  is-max-node (current-node) and
          alpha-value of (current-node)>=
          beta-value-of (min-ancestor-of (current-node)
          then cut-off-search-below (current-node);
      if  is-min-node (current-node) and
          beta-value-of (current-node) <=
          alpha-value-of (max-ancestor-of (current-node) then
          cut-off-search-below (current-node);
    }
```

Instead of searching back up the tree for ancestor values, we propagate values down the tree as follows:

(*a*) **For each max node**, the minimum beta value for all its **min node ancestors** is stored as **beta.**

(*b*) **For each min node,** the maximum alpha value for all its **max node ancestors** is stored as **alpha.**

(*c*) Hence, each **non-leaf node** will have a beta value and an alpha value stored.

(*d*) Initially, the root node is assigned an alpha value of negative infinity and a beta value of infinity.

The above alpha-beta function can be modified also wherein we use another variable, say **'children'** to represent all of the children of the current node. We add the following statement now—

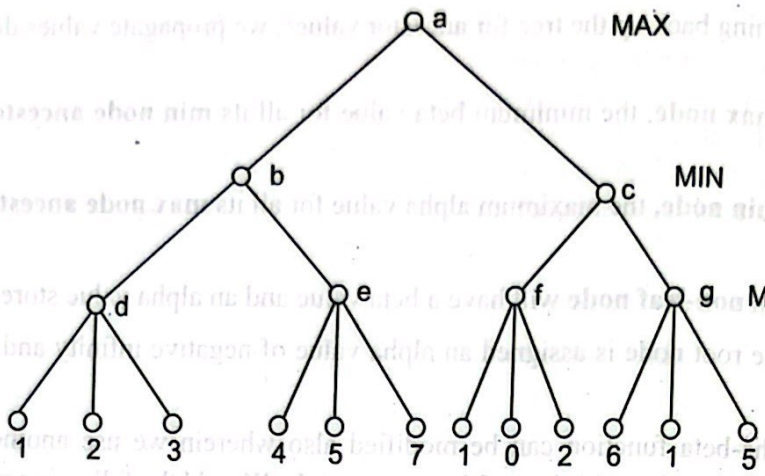
alpha = max (alpha, alpha-beta (children, alpha, beta))

If means that alpha is set to the greatest of the current value of alpha and the values of the current node's children are calculated by recursively calling alpha-beta ( ).

```
// Modified alpha-beta function using
// recursion.
function alpha-beta (current-node, alpha, beta)
{
    if is-root-node (current-node)
{
    alpha = – infinity
    beta = infinity
}
if is_leaf (current_node)
    return static_evaluation (current_node);
if is_max_node (current_node)
    {
    alpha = max(alpha, alpha-beta (children, alpha, beta));
    If alpha >= beta
      cut_off_search_below (current_node);
    }
if   is_min_node (current_node)
    {
    beta = min (beta, alpha_beta (children, alpha, beta));
    if beta <=alpha
      cut-off-search-below (current_node);
    }
}
```

We are in a position to solve an example now.

EXAMPLE. For the given game tree, apply alpha-beta pruning procedure—



SOLUTION. If it is my turn to move then the root is labeled as "**MAX**" node indicating it is my **turn** otherwise if is labeled as "**MIN**" node to indicate it is my **opponent's turn.** Here, we assume that the static evaluation function returns large values to indicate good situations for us. So, our goal is to **maximize** the value of the static evaluation function of the next board position. **Please note that acrs represent the possible legal moves for the player that the arcs emanate from.** Because moves alternate so each level of the tree has nodes that are all MAX or all MIN. **Also note that the nodes at level i are of the opposite kind from those at level (i + 1).**

With these assumptions, let us try to solve this problem now.

The non-terminal (or non-leaf) nodes in the tree are labeled from a to g. Terminal (or leaf nodes) nodes have scores assigned to them by static evaluation i.e.,

a is MAX node (i.e., my turn)

b and c are MIN nodes (i.e., opponent's turn)

d, e, f and g are MAX nodes (i.e., my turn)

Now, we follow the tree by depth-first search. The **first-step** is to follow the path a, b, d, and then to the three children of d i.e.,

Path 1 : $a \rightarrow b \rightarrow d \rightarrow (1 \rightarrow 2 \rightarrow 3)$

This gives an alpha value for 'd' as 3. This is passed up to 'b' which now has the beta value of 3 and an alpha value that has been passed down from 'a' of negative infinity.

Now, the first child of 'e' is examined and has score of 4. In this case, clearly, there is no need to examine the other children of 'e' because the minimizing choice at node 'b' will definitely do worse by choosing 'e' rather than 'd' so, cutoff is applied here and nodes with score of 5 and 7 are never examined. Let us show this in a tabular form now.

S. No.	Node	Alpha	Beta	Meaning
1.	a	$-\infty$	∞	Alpha starts at $-\infty$ Beta starts at ∞
2.	b	$-\infty$	∞	
3.	d	$-\infty$	∞	
4.	d	1	∞	
5.	d	2	∞	
6.	d	3	∞	Upto this point, we have examined the three children of d. We get an alpha value of 3, which is passed back upto node b.
7.	b	$-\infty$	3	At this MIN node, we get a score of 3 or better (lower). Now we need to examine the children of 'e' to see if we can geta lower score.

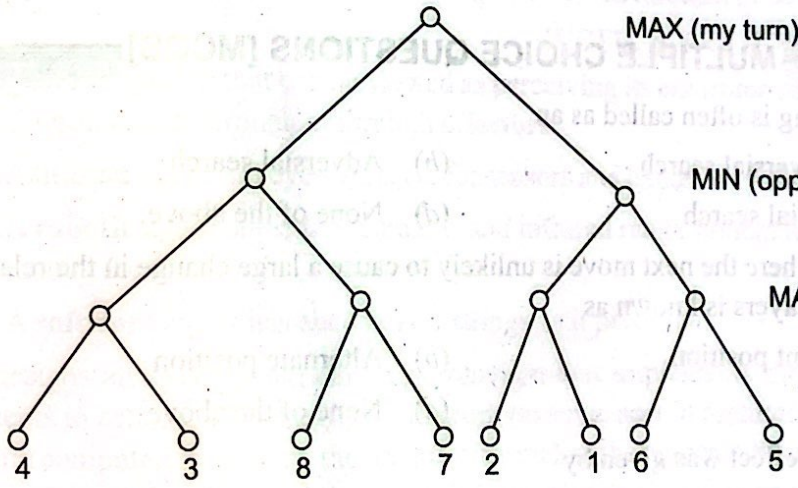
S. No.	Node	Alpha	Beta	Meaning
8.	e	$-\infty$	3	
9.	e	4	3	A score of 4 can be obtained from the first child of 'e'. MIN will do better if chosen rather than 'e' because if he chooses 'e' max can get atleast 4, which is worse for min than 3. Hence, we can ignore of ther children of 'e'. A CUT-OFF occurs.
10.	a	3	∞	The value of 3 has been passed back up to the root node. Hence, MAX now knows that he can score atleast 3. He now needs to see if he can do better.
11.	c	3	∞	
12.	f	3	∞	We now examine the three children of f and find that none of them is better than 3. So, we pass back a value of 3 to c.
13.	c	3	3	MAX has already found that by taking the left hand branch, he can achieve a score of 3. Now it seems that if he chooses the right-hand branch, MIN can choose 'f' which means he can only achieve a score of 2. So, cut-off can now occur because there is no need to examine it or its children.

Result: Thus, out of 12 leaf nodes in the tree, this algorithm needed to examine only 7 to conclude that the best move for MAX is 'b', so MIN will choose 'd' and MAX will choose the right hand node, ending with a static evaluation of 3.

Please note that at this stage, this is the right answer because if MAX chooses 'c' then MIN will clearly choose f, resulting in MAX being able to achieve a score of only 2.

∴ Optimized game tree (using our algorithm) is as shown in Fig.

Herein, the minimax algorithm with alpha—beta cut-off will need to examine only five of the eight leaf nodes.



3.1.2 Futility cut-off

The Alpha-Beta procedure discussed above can also be used to cutoff additional paths that provide only slight improvements over paths which have already been explored.

For e.g.: Consider a Minimax tree below—

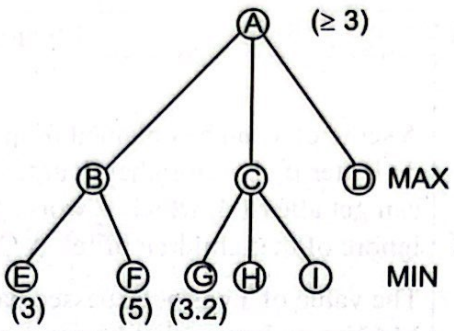


Fig. 3.5 Shows futility cutoff.

After examining node G, we can say that if the move is along C, the value of static evaluation function is 3.2 but the move towards B has a guaranteed score of 3. Because 3.2 is just slightly better than 3 (in the scale of 0-10), we should terminate exploration of C. Thus, we can devote more time in exploring other parts of the tree which could be more useful. **This limit of the static evaluation function is called as futility cutoff limit.** It helps in terminating the exploration of a **subtree** which offers little possibility for improvement over other known paths.

SUMMARY

Minimax Search procedure is a depth-first, depth limited search procedure. We start at the current position and use the plausible move generator to generate the set of possible successor positions. We can apply the static evaluation function to those positions and simply choose the best one. After doing so we can back that value upto the starting position to represent our evaluation of it. **Root node** represents the configuration of the board at which a decision must be made as to what is the best single move to make next. We assume that the static evaluation function returns large values to indicate good situations for us. So, our goal is to maximize the value of the static evaluation function of the next board position. This process of backing up values gives the optimal strategy that both players would follow given that they both have the information computed at the leaf nodes by the evaluation function.

MULTIPLE CHOICE QUESTIONS [MCQS]

1. Game playing is often called as an

(*a*) Non-adversial search (*b*) Adversial search

(*c*) Sequential search (*d*) None of the above.

2. A position where the next move is unlikely to cause a large change in the relative positions of the two players is known as

(*a*) Quiescent position (*b*) Alternate position

(*c*) Move (*d*) None of the above.

3. The horizon effect was given by

(*a*) Berliner (*b*) Boehm (*c*) Jacobson (*d*) None of the above.

4. Most evaluation functions are specified as a weighted sum of features given by the formula

(*a*) (W1*feat1) * (W2*feat2) *...* (Wn*featn)

(*b*) (W1+feat1) * (W2*feat2) *...* (Wn*featn)

(*c*) (W1*feat1) * (W2*feat2) *...* (Wn*featn)

(*d*) None of the above.

5. When we say that if one player wins, the other loose then it is a
   (*a*) Zero-sum game
   (*b*) Two-sum game
   (*c*) Four-sum game
   (*d*) Eight sum game.

ANSWERS

1. (*b*) 2. (*a*) 3. (*a*) 4. (*c*) 5. (*a*)

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. MIN/MAX strategy uses two rules. What are they?

[MDU, B.E. (CSE)-6th Sem. May 2009]

Ans. Two rules that are used in MIN/MAX strategy are as follows—

1. For maximizer, if static evaluation function value found at any node is less than the α-value, reject it.
2. For minimizer, if static evaluation function value found at any node is more than β-value, reject it.

Q. 2. What is plausible move generator?

Ans. These are certain moves that are unproductive (or dangerous) because they obviously lead away from engagement or to an immediate loss of a piece with no hope of trapping the after player. These moves are avoided by the substitution of a plausible move generator in place of the legal move generator. The plausible move generator is equivalent to a beam search in which only a limited number of promising nodes are expanded.

For e.g. In a chess game, the average branching from a node in chess could be cut from 30 to 10 by means of a plausible move generator.

Q. 3. What is an agent? How is it related to architecture and program? How agent should act? Name some agent types and their goals and environment descriptions.

[UPTU, B.Tech (CSE) 6th Sem. 2006-07]

Ans. An agent is anything that can be viewed as perceiving its environment through **sensors** and acting upon that environment through **effectors.**

For e.g. (*a*) A **human agent** has eyes, ears, etc for sensors and hands, legs, mouth, etc. for effectors.

(*b*) A **robotic agent** subsitutes cameras and infrared range finders for sensors and various motors for effectors.

(*c*) A **software agent** has encoded bit strings as it percepts.

AI designs an **agent program** i.e., a function that implements the agent mapping from percepts to actions. This program will work on some sort of architecture—which might be a plain computer. In general, the architecture makes the percepts from the sensors available to the program, runs the program and feeds the program's action choices to the effectors as they are generated. The relationship among agents, architectures and programs is—

$$\text{Aqent} = \text{Architecture} + \text{Program}$$

A rational agent is one that does the right thing. The right action is one that will cause the agent to be most successful. That leaves us with the problem of deciding 'how' and 'when' to evaluate the agent's success.

The term '**performance measure**' is used for how the criteria that determine how successful an agent is?

An **omniscient agent** knows the actual outcome of its actions and can act accordingly but omniscience is impossible in reality.

An **ideal rational agent** is one which, for each possible percept sequence, should do whatever action is expected to maximize its performance measures on the basis of the evidence provided by the percept sequence and whatever inbuilt knowledge the agent has.

For example Software robots (or softbots) is a software agent.

We list some agents, their percepts, actions, goals and their environment descriptions—

Agent Type	Percepts	Action	Goals	Environment
1. Refinery pressure	Temperature, pressure, readings	Open/close valves, adjust temperatures	Maximize purity	Refinery
2. Medical diagnosis system	Symptons, findings, patient's answers	Questions, tests, treatments	Healthy patients, minimize costs	Patient, hospital

Q. 4. Explain the following statement— "Exhaustive searching of game trees is not possible."

Ans. Exhaustive searching on game trees is not possible bcasue typically trees have very high braching factors and often will be very deep.

For E.g.1, a game tree of a chess game has an average branching factor of 38. A tree can have an infinite depth.

E.g.2 a tic-tac-toe game has maximum depth of 9 and a maximum braching factor of 9 meaning that it has nearly $9 \times 8 \times 7 \times \ldots \times 2 \times$ nodes in the tree i.e., more than 350,000 nodes to examine.

EXERCISE QUESTIONS

Q. 1. Explain why and how AI methods are suitable for game playing. Are there any limitations?

[RTM Nagpur University, BE (IT) 7th Sem., Summer 2003]

Q. 2. Explain Minimax search with reference to game playing.

[RTM Nagpur University, BE (IT) 7th Sem., Summer 2003, 2004, 2005, 2007, 2009]

Q. 3. Write short notes on - minimax search procedure. **[DTU (DCE)-ME (CS) 2005]**

Q. 4. Why game playing problems are considered as AI problems?

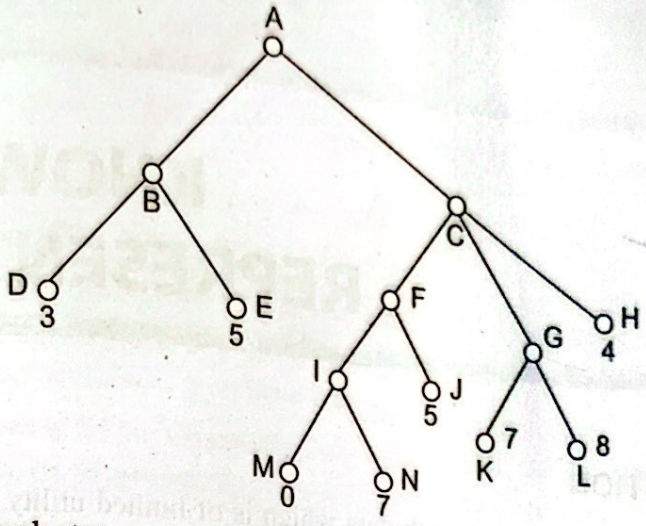
[Coachin University, B.Tech (CSE) 8th Sem., OCT 2000]

Q. 5. Explain the Mini-Max with α, β cut offs game playing strategy.

[MDU, B.E. (CSE) 6th Sem., 2008-2009 & Dec. 2009]

Q. 6. (*a*) What is a state space search? Give any example of a Game which happens to be a problem of state space search and justify your answer properly.

(*b*) Perform minimax on the following tree—



Explain each step.

[GGSIPU, B. Tech. (CSE) 8th Sem., May 2011]