MCA Part III

Paper- XXI

Topic: GAME PLAYING

Prepared by: Dr. Kiran Pandey

School of Computer science

Email-Id: kiranpandey.nou@gmail.com

INTRODUCTION

Mathematical game theory is a branch of economics, and it views any multi-agent environment as a game provided that the impact of each agent on the others is significant, regardless of whether the agents are cooperative or competitive. In AI "games" are of a specialized kind which are deterministic in nature, turn taking, two-player. There may be zero sum games of perfect information. We will also discuss topic like, min-max search procedure and alpha-beta cutoffs in this unit.

OVERVIEW OF GAME PLAYING

Game playing is the first tasks that was undertaken in AI. Game playing basically demonstrates several aspects of intelligence, particularly ability to plan at all the levels such as low level, tactical level or middle level and long-term strategic level and the ability to learn. Game playing is a good domain for machine intelligence as it provided a structured task and limited knowledge. Thus a game can be formally defined as a kind of search problem. It has the following components:

- The **initial state**, which includes the board position and identifies the players to move.

- A **successor function,** which returns a list of (move, state) pairs, each indicating a legal move and the resulting state.
- A **terminal test**, which determines when the game is over. States where the game has ended are called **terminal states**.
- A **utility function**, which gives a numeric value for the terminal states.

The initial state and the legal moves for each side define the **game tree**.

Various types of games

(i) **Single- player games:** "You against the problem" games .E.g. 8-puzzle, 15-puzzle.

(ii) **Two-player games**: Here, moves are constrained by the fact that one's opponent will exploit any opportunities which you make available. For example chess, checkers, Go etc.

(iii) **Multiplayer games**: this involves more than two players such as Bridge etc.

(iv) **Perfect information games :** At any given time during the game both players know everything there is to know about the opponent's position. Card games in which you don't know what cards your opponent holds are not perfect information games.

(v) **Zero-sum games:** In this game, if one side wins, then the other loses. e.g. "war- games" are a class of non-zero-sum games.

Game Trees

We can represent all possible games (of a given type) by a directed graph often called a **game tree**. The **nodes** of the graph represent the **states** of the game. The arcs of the graph represent possible **moves** by the **players** (+ and -).Consider the start of a game

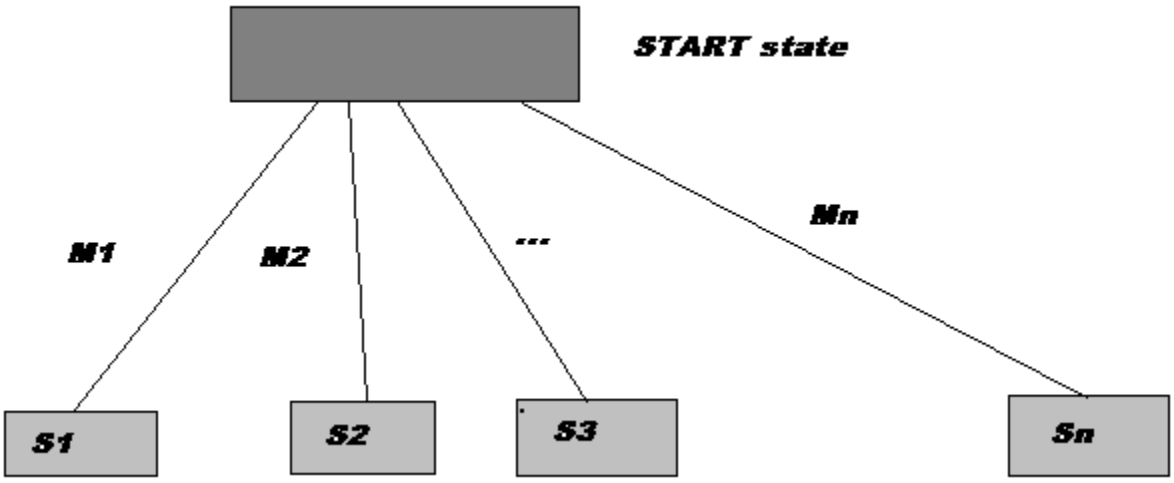

Figure 1: Game tree

The game starts in some "start" state, and there is a set of possible moves: m_1, m_2.. m_N. These give rise, respectively, to states: S_1, S_2.. S_N. By considering possible moves at any state Si (recursively), we develop a "game tree". The leaves of this tree ("tip nodes") represent the state of play so far. The rules of the game assign a value to every terminal position.

W -- Won
L -- Lost From the point of view of '+'
D -- Drawn

One way to guarantee a good game would be to analyses completely the game tree. However, for a moderate game tree containing about 10^{40} nodes, if we make an optimistic assumption that 3 nodes can be considered per nanosecond, then it would take 10^{21} centuries to explore the whole tree. Thus it will become difficult to decide which search technique we would use.

THE MIN-MAX SEARCH PROCEDURE

In normal search problem, the optimal solution would be a sequence of moves leading to a goal state- a terminal state that is a win. Given a game tree, the optimal strategy can be determined by examining the **minimax** value of each node, which we write as MINIMAX VALUE(n).

The 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. To decide one move, we need to explore the possibilities of winning by looking ahead to more than one step. This is called a **ply**. Thus in a two ply search, to decide the current move, game tree need to be explored for two levels farther.

MINIMAX algorithm

Given a minimax search tree, the optimal strategy can be determined by examining the minimax value of each node. Thus value indicates the utility(for MAX) being in a certain state, assuming both players play optimally. The minimax value of the terminal state is its own utility. Given a choice, MAX chooses to move to a state of maximum value and MIN chooses a state of minimum value. The minimax algorithm computes the minimax decision for the current state by recursively computing the minimax values of each successor state. The recursion proceeds to the leaves of the game tree, and then the minimax values are backed up through the tree to give the final value. The steps of the algorithms are given below:

Step 1: Generate the search tree from the current position till the depth **d** of the tree which is selected as **d**-ply search.

Step 2: Compute the static evaluation function that is, heuristics value of the leaf nodes at the depth **dz**.

Step 3: Propagate the values till the current position on the basis of the player:

(i) If it is the turn of MAX player, generate the successors of the current position, apply minimax to each of the successors, and return the maximum of the results.

(ii) If it is the turn of MIN player, generate the successors of the current position, apply minimax to each of the successors, and return the minimum of the results.

Let us consider the following example

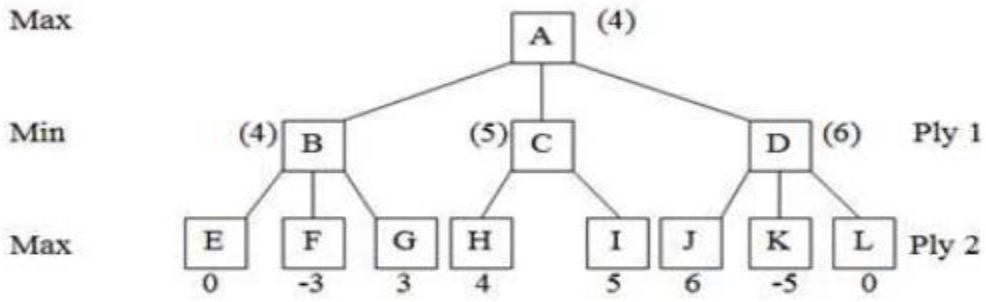

Figure 2: Tree showing two ply search

In this tree, node A represents current state of any game and nodes B, C and D represent three possible valid moves from state A. similarly E, F, G represents possible moves from B, H, I from C and J, K, L, from D. to decide which move to be taken from A, the different possibilities are explored to two next steps. 0, -3, 3, 4, 5, 6, -5, 0 represent the utility values of respective move. They indicate goodness of a move. The utility value is back propagated to ancestor node, according to situation whether it is max ply or min ply. As it is a two player game, the utility value is alternatively maximized and minimized. Here as the second player's move is maximizing, so maximum value of all children of one node will be back propagated to node. Thus, the nodes B, C, D, get the values 4, 5, 6 respectively. Again as ply 1 is minimizing, so the minimum value out of these i.e. 4 is propagated to A. then from A move will be taken to B.

ALPHA-BETA CUTOFFS

The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. It is not possible to remove the exponent but we can effectively cut it into half. **Alpha-Beta** pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which are not required to be searched because there already exists a better move available. It can be applied to any depth of the tree and it is often possible to prune entire subtree rather than just leaves. It is called Alpha-Beta pruning because it passes two extra parameters in the minimax function, namely **alpha** and **beta**. The parameters are defined below:

(i) **Alpha (α)** is the best value (highest value) that the **maximizer** currently can guarantee at that level or above.

(ii) **Beta (β)** is the best value (lowest value) that the **minimizer** currently can guarantee at that level or above.

Let us understand the concept using an example:

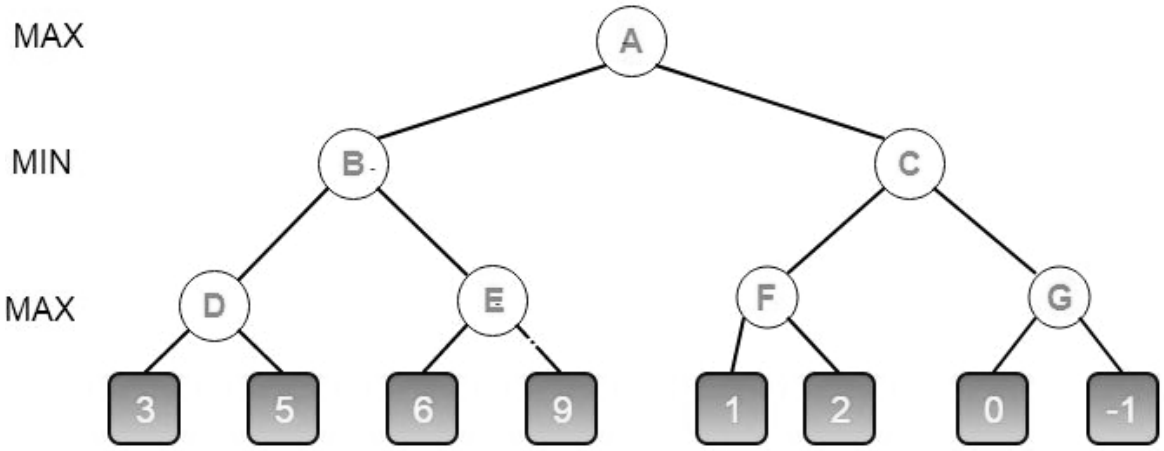

Figure 3: Alpha- Beta Cutoffs

- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first
- At **B** it the minimizer must choose min of **D** and **E** and hence calls **D** first.
- At **D**, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at **D** is max(-INF, 3) which is 3.

- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.

- **D** now looks at its right child which returns a value of 5.At **D**, alpha = max(3, 5) which is 5. Now the value of node **D** is 5.

- **D** returns a value of 5 to **B**. At **B**, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. **B** now calls **E** to see if he can get a lower value than 5.

- At **E** the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at **B** and that is what **B** passed down to **E**

- Now **E** looks at its left child which is 6. At **E**, alpha = max (-INF, 6) which is 6. Here the condition becomes true. Beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and **E** returns 6 to **B**
- Note how it did not matter what the value of **E**'s right child is. It could have been +INF or -INF, it still wouldn't matter, we never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of **B**. This way we dint have to look at that 9 and hence saved computation time.

- **E** returns a value of 6 to **B**. At **B**, beta = min(5, 6) which is 5.The value of node **B** is also 5.

So far this is how our game tree looks. The 9 is crossed out because it was never computed.

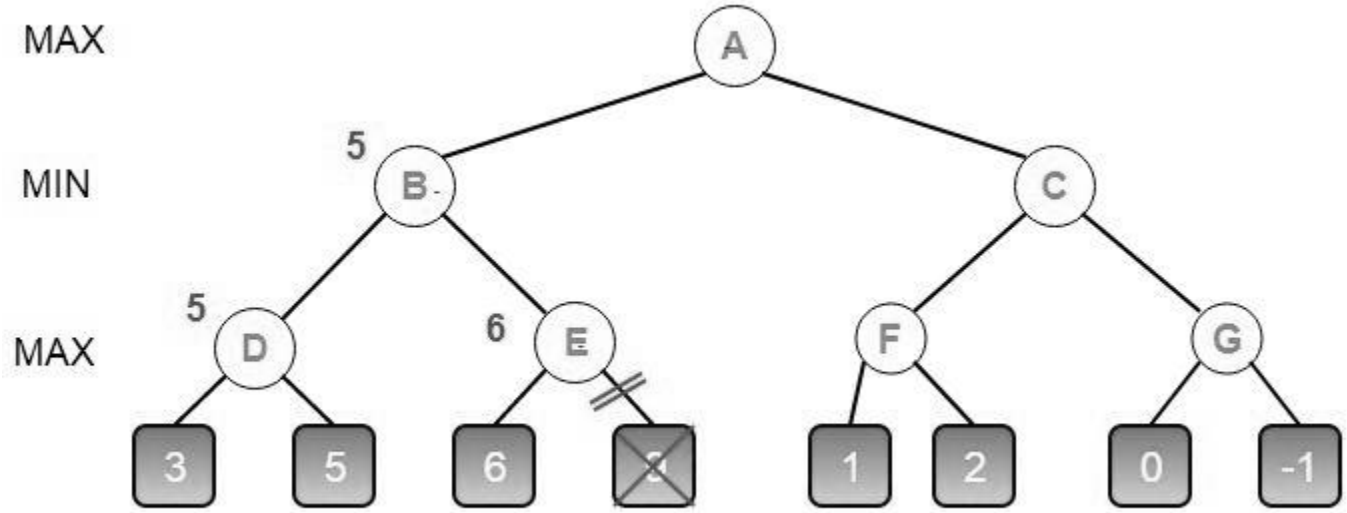

Figure 3: MIN-MAX graph

- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.

- At **C**, alpha = 5 and beta = +INF. **C** calls **F.**

- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.

- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5.

- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes false as beta = 2 and alpha = 5. So it breaks and it dose not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.

- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.

- Hence the optimal value that the maximizer can get is 5

This is how our final game tree looks like. As you can see **G** has been crossed out as it was never computed.

The advantage of alpha-beta pruning is that it does not affect final result and good move ordering improves the effectiveness of pruning. If there is perfect ordering, time complexity of minimax with alpha-beta pruning is $O(b^{m/2})$ that is doubles the depth of search which can be carried out for a given level of resources.

QUESTIONS

1. Explain the concept of Game playing.
2. Describe some games of one-player and two-players.
3. Discuss the theory MIN-MAX.
4. Explain the concepts of Alpha Beta cutoffs.