Game Playing in AI

1. INTRODUCTION

Game playing was one of the first tasks undertaken in Artificial Intelligence. Game theory has its history from 1950, almost from the days when computers became programmable. The very first game that is been tackled in AI is chess. Initiators in the field of game theory in AI were Konard Zuse (the inventor of the first programmable computer and the first programming language), Claude Shannon (the inventor of information theory), Norbert Wiener (the creator of modern control theory), and Alan Turing. Since then, there has been a steady progress in the standard of play, to the point that machines have defeated human champions (although not every time) in chess and backgammon, and are competitive in many other games.

1.1 Types of Game

1.1.1 Perfect Information Game: In which player knows all the possible moves of himself and opponent and their results.

E.g. Chess.

1.1.2 Imperfect Information Game: In which player does not know all the possible moves of the opponent.

E.g. Bridge since all the cards are not visible to player.

1.2 Definition

Game playing is a search problem defined by following components:

1.2.1 Initial state: This defines initial configuration of the game and identifies first payer to move.

1.2.2 **Successor function**: This identifies which are the possible states that can be achieved from the current state. This function returns a list of (move, state) pairs, each indicating a legal move and the resulting state.

1.2.3 **Goal test**: Which checks whether a given state is a goal state or not. States where the game ends are called as terminal states.

1.2.4 **Path cost / utility / payoff function**: Which gives a numeric value for the terminal states? In chess, the outcome is win, loss or draw, with values +1, -1, or 0. Some games have wider range of possible outcomes.

1.3 Characteristics of game playing

1.3.1 **Unpredictable Opponent**: Generally we cannot predict the behavior of the opponent. Thus we need to find a solution which is a strategy specifying a move for every possible opponent move or every possible state.

1.3.2 **Time Constraints**: Every game has a time constraints. Thus it may be infeasible to find the best move in this time.

2. HOW TO PLAY A GAME IN AI?

Typical structure of the game in the AI is:

- 2- person game
- Players alternate moves
- Zero-sum game: one player's loss is the other's gain
- Perfect information: both players have access to complete information about the state of the game. No information is hidden from either player.
- No chance (e. g. using dice) involved.

E.g. Tic- Tac- Toe, Checkers, Chess, Go, Nim, Othello

For dealing with such types of games, consider all the legal moves you can make from the current position. Compute the new position resulting from each move. Evaluate each resulting position and determine which is best for you. Make that move. Wait for your opponent to move and repeat the procedure. But for this procedure the main problem is how to evaluate the position? Evaluation function or static evaluator is used to evaluate the 'goodness' of a game position. The zero- sum assumption allows us to use a single evaluation function to describe the goodness of a position with respect to both players. Lets consider, f(n) is the evaluation function of the position 'n'. Then,

– f(n) >> 0: position n is good for me and bad for you
– f(n) << 0: position n is bad for me and good for you
– f(n) near 0: position n is a neutral position
e.g. evaluation function for Tic- Tac- Toe:
f(n) = [# of 3- lengths open for me] - [# of 3- lengths open for you]
where a 3- length is a complete row, column, or diagonal

3. Minimax

3.1 Game Trees

Games are represented in the form of trees wherein nodes represent all the possible states of a game and edges represent moves between them. Initial state of the game is represented by root and terminal states by leaves of the tree. In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state that is a win. Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree. Fig 1 shows part of the game tree for tic-tac-toe.

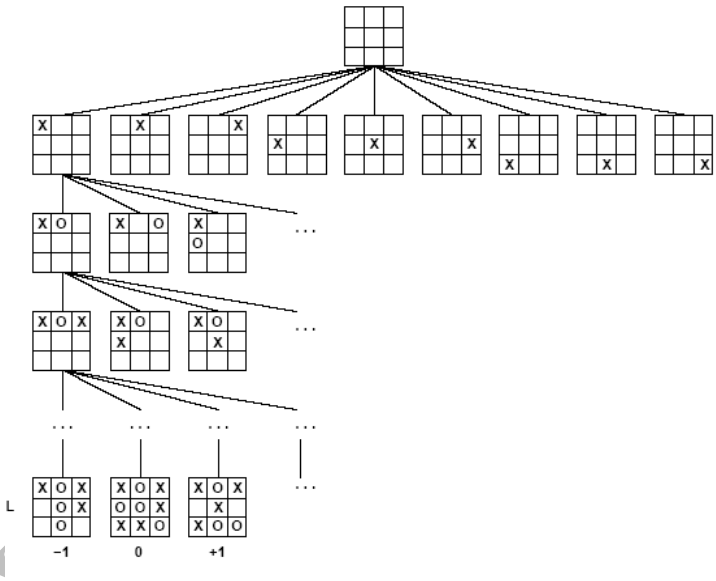

Fig. 1 Game tree for Tic-Tac-Toe

Ref: www.uni-koblenz.de/~beckert/ Lehre/Einfuehrung-KI-SS2003/folien06.pdf

Let us represent two players by ‘X’ and ‘O’. From the initial state, X has nine possible moves. Play alternates between X and O until we reach leaves. The number on each leaf node indicates the utility value of the terminal states from the point of view of X. High values are assumed to be good for X and bad for O.

3.2 Minimax Trees

Imagine a simple game that only has three moves, one move results in a win, another draw, and finally a loss. Therefore, we want to assess each node and figure what the outcome will be. Let us extend our game to one that allows you to make three moves per go, and only takes two goes to win. Therefore, we will want to look ahead to find out which move combination works for us. We can generate a game tree of all the possible moves.

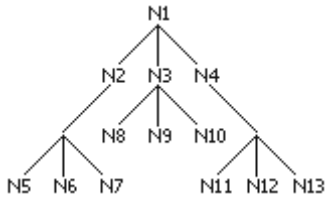

Fig 2. Game Tree

You can see how there are 9 final possible moves. Imagine that N11 is the winning situation; therefore our first move will have to be N4. How can we figure this out algorithmically? Let us assign values to a win, draw and loss. A win will be 1, a draw 0 and a loss -1. Say that N11 is the only winner, and the rest are drawing situations. So, what we'll want to do is evaluate the tree from the bottom-up propagating the *maximum* value for the nodes upwards. Therefore, for the N5-N7 group, 0 is the highest so this is applied to N2. N8-N10 also has 0 as the highest, which is taken on by N3. The N11-N13 group has 1 as the highest. The program knows to choose N4.

In our example, since the tree is only two layers deep this seems rather trivial. But imagine a tree 10 layers keep, this method would allow you to simply calculate which moves would lead to a winning situation. Most of you will already notice a large fault in this - trees this large are incredibly expensive in both memory and computational terms. A 10-layer tree that branches three times for each node would have 59,049 nodes. This is relatively simple - a Tic-tac-toe tree would have 362,880 (9!) nodes.

Therefore, we have to cut down the depth of our tree. This gives us a problem, though - if we limit the depth we are not guaranteed a winning scenario as one of the nodes. This is where the clever programming has to come in. You must create some sort of evaluation function that can assess how close to a winning situation the board is. Since the nodes are not so clear-cut (win, draw, loss) a more complicated numbering system has to be used. The system is completely dependent on the programmer and the board game in question.

Very few board games are one player, so how can we add this into our tree? When we are playing for ourselves, we are attempting to maximize our score, so our opponent will want to minimize our score. Here comes minimax into picture.

In a two-player game, the first player moves with MAX score and the second player move with MIN score. A minimax search is used to determine all possible continuations of the game up to a desired level. A score is originally assigned to the leaf known as utility value and indicated by UTILITY(n). Then by evaluating each possible set of moves, a score is assigned to the upper level by the minimax algorithm. The minimax algorithm performs a preorder traversal and computes the scores on the fly. The same would be obtained by a simple recursive algorithm. The rule is as follows:

```
Minimax_value(u)
{   //u is the node you want to score
        if u is a leaf return score of u;
        else
                if u is a min node
                        for all children of u: v1, .. vn ;
                                return min (Minimax_value(v1),..., Minimax_value(vn))
                else
                        for all children of u: v1, .. vn ;
                                return max (Minimax_value(v1),..., Minimax_value(vn))
```

}

Let us take a look at a game tree. Here is our game tree with evaluations assigned to the final nodes:

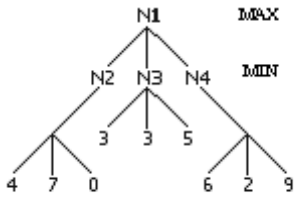

Fig. 3 Minimax tree with evaluation function applied to leaves

Now, assigned values are for boards representing our opponent's choice of boards. N1 stands for the current board, N2-N4 are our three possible moves and N5-N13 are the opponent's possible follow-up moves. Since our opponent will try to minimize our winning possibility, therefore will calculate the minimum value for each node and assign it to the parent. N2 will equal 0, N3 will equal 3, and N4 will equal 2. In making a choice for our best possible move, we look at the max of these values - which is 3 (N3).

3.3 Minimax Algorithm

Now let us put all this in the form of algorithm. Following is manimax algorithm, which takes current state as an input and returns a best possible operator to be applied to current state. Essentially this is same as what we have seen previously in recursive function. But this algorithm is written from the MAX player point of view.

Function MINIMAX-DECISION (state) returns **an** *operator*
For each *op* **in** OPERATORS[*game*] **do**
VALUE [*op*] = MINIMAX-VALUE (APPLY (*op*, *state*), *game*)
End
Return the *op* with the highest VALUE [*op*]

Function MINIMAX-VALUE (*state, game*) **returns** *a utility value*

If TERMINAL-TEST (*state*) **then**

Return UTILITY (*state*)

Else If MAX is to move in *state* **then**

Return the highest MINIMAX-VALUE of SUCCESSORS (*state*)

Else

Return the lowest MINIMAX-VALUE of SUCCESSORS (*state*)

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.

Let's see an example. In fig 4, a game with 2 plies is shown. One ply indicates one move of one player. MAX has three possible moves, which are followed by three possible moves each for MIN. Tree shows these moves. Leaf nodes are evaluated and utility values are assigned to them. These values are propagated upward to assign the utility values to the parents.

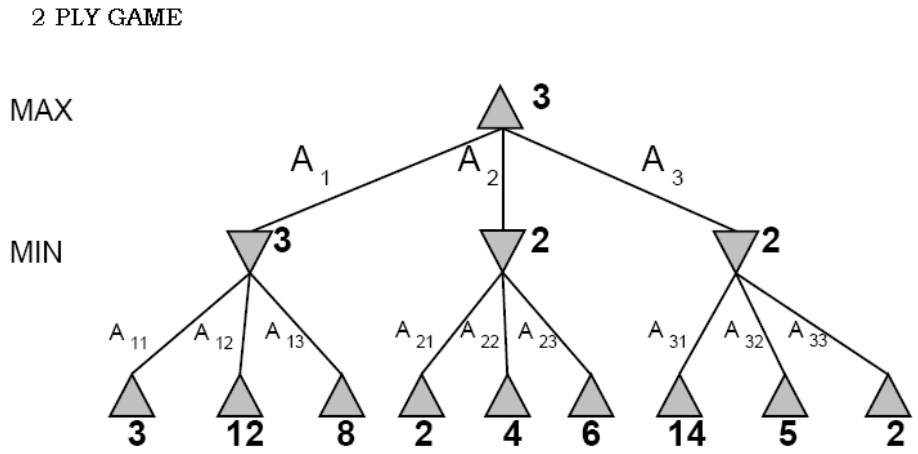

Fig 5. Minimax Game Tree

Ref: www.uni-koblenz.de/~beckert/ Lehre/Einfuehrung-KI-SS2003/folien06.pdf

Summarizing, one can view in its entire form, the score values at each of the levels of the tree at any given point during the game. By viewing this tree, a player may be able to foresee which moves are more advantageous and beneficial for them. The root of the tree represents the position of the current player, thus, depending on the number of levels that is to be searched, all odd levels represent the first player while the even levels represent the second player.

3.4 Characteristics of minimax algorithm

3.4.1 Completeness: Minimax is complete if the tree is finite.
E.g Chess has a very large but finite tree. Thus minimax is complete in case of chess.

3.4.2 Optimality: If observed carefully, algorithm is optimal against an optimal player only. In fig.4, A1 is the optimal move if the opponent is also optimal. If opponent is not optimal then MAX can get more utility by selecting A3 if in turn MIN selects A31 giving utility of 14. But if opponent is optimal, he will select move A33 giving utility of 2 to MAX that is not optimal.

3.4.3 Time Complexity: Algorithm performs a complete depth-first search exploration, time complexity is $O(b^m)$, where b is branching factor and m is depth of the tree.

3.4.4 Space Complexity: Space complexity is O(bm).

For chess, b is approximately equal to 35 and m approximately equal to 100. Thus it is infeasible to find exact solution within given time limit. Thus one standard approach is applied called as depth limit search, in which search is limited to some depth from the current node. Nodes at that depth are assumed to be leaves and their utility values are estimated. Estimation function estimates desirability of the state. This is different for each

game. There are different methods of estimation for a game. This is most interesting part of game theory.

4. ALPHA-BETA PRUNING

The problem with the minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately we can't eliminate the exponent, but we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. The particular technique we will examine is called alpha-beta pruning. When applied to the standard minimax tree, it returns the same move as minimax would, but prunes away branches that can't possibly influence the final decision.

Alpha and Beta are the variables defined as:
At each MAX node n, alpha(n) = maximum value found so far
At each MIN node n, beta(n) = minimum value found so far

The alpha values start at -infinity and only increase, while beta values start at +infinity and only decrease. Let us see an example. In fig. 5, three possible moves of MIN corresponding to given move of MAX are evaluated. They have utility values of 3, 12 and 8. Thus MIN will select minimum of them i.e. 3. Thus from this move of MAX, it can get utility value of 3. Here value of alpha becomes 3. Thus it is assured that utility value of MAX can no be less than 3.

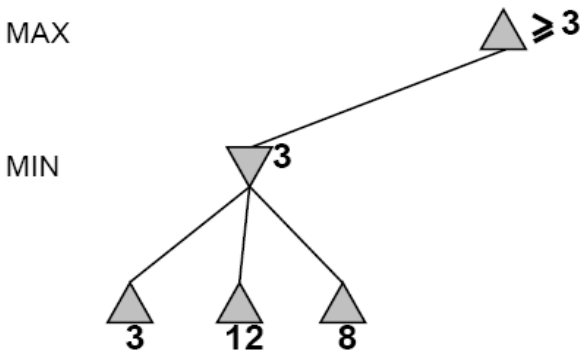

Fig 5. Alpha Pruning
Ref: www.uni-koblenz.de/~beckert/ Lehre/Einfuehrung-KI-SS2003/folien06.pdf

Now consider another move of MAX (fig 6). That will lead to 3 possible moves of MIN. One of these moves has utility value of 2. Now this becomes beta value of MIN node. Now whatever may be the utility values of other two children, utility value of MIN node cannot be grater than 2. Thus this MIN can propagate upward at the max value of 2. This 2 is less than current alpha of MAX node i.e. 3. Because MAX is assured with the utility value of at least 3, it will not consider any MIN node returning utility value less than 3. And thus it prunes the search there for this MIN node. Thus without considering other 2 children of MIN node we can proceed further. This is called as alpha pruning i.e. pruning based on alpha value and applied at the MIN node. Beta pruning is same as alpha pruning with the difference that it is applied at MAX node

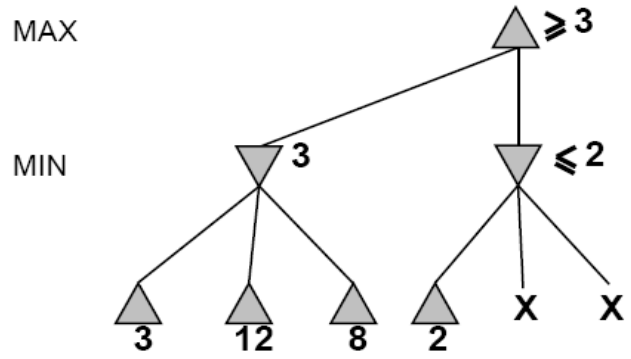

Fig. 6 Alpha Pruning

Ref: www.uni-koblenz.de/~beckert/ Lehre/Einfuehrung-KI-SS2003/folien06.pdf

Thus, we can define the procedure for alpha cutoff and beta cutoff as follows:

- Beta cutoff: Cut off the search below MAX node n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(i) for some MIN node ancestor i of n.
- Alpha cutoff: Stop searching below MIN node n if beta(n) <=alpha(i) for some MAX node ancestor i of n.

α - β pruning reduces the search space without affecting final result. Order in which successors are scanned affects performance. With a good ordering we can improve the performance but in the worst case it may result in no improvement in performance. Time complexity with best case comes to be $O(b^{m/2})$. Thus with the same time constraints, we can double the depth of search.

This method of alpha beta pruning can be applied in the games with chance. E.g chance card game where chance introduced by card shuffling, or games involving dice rolling.

5. APPLICATIONS

Game theory has vast applications in different fields. Some of the important are mentioned below.

5.1 Entertainment: Game theory is used to define different strategies of different games.

5.2 Economics: Each factor in the market, such as seasonal preferences, buyer choice, changes in supply and material costs, and other such market factors can be used to describe strategies to maximize the outcome and thus the profit.

5.3 Military: Game theory can be useful in Military also. Military strategists have turned to game theory to play "war games". Usually, such games are not zero-sum games, for loses to one side are not won by the other.

5.4 Political science: The properties of n-person non-zero-sum games can be used to study different aspects of political science and social science. Matters such as distribution of power, interactions between nations, the distribution of classes and their effects of government, and many other matters can be easily investigated by breaking the problem down into smaller games, each of whose outcomes affect the final result of a larger game.

6. CONCLUSION

Game theory remained the most interesting part of AI from the birth of AI. Game theory is very vast and interesting topic. Game theory mainly deals with working in the constrained areas to get the desired results. They illustrate several important points about Artificial Intelligence like perfection can not be attained but we can approximate to it.

References

[1] Artificial Intelligence: A Modern Approach' (Second Edition) by Stuart Russell and Peter Norvig, Prentice Hall Pub.

[2] http://www.cs.umbc.edu/471/notes/pdf/games.pdf

[3] http://l3d.cs.colorado.edu/courses/AI-96/sept23glecture.pdf

[4] Theodore L. Turocy, Texas A&M University, Bernhard von Stengel, London School of Economics "Game Theory" CDAM Research Report Oct. 2001

[5] http://www.uni-koblenz.de/~beckert/Lehre/Einfuehrung-KI-SS2003/folien06.pdf

[6] http://ai-depot.com/LogicGames/MiniMax.html