CPS331 Lecture: Search in Games

last revised September 17, 2018

Objectives:

1. To introduce mini-max search
2. To introduce the use of static evaluation functions
3. To introduce alpha-beta pruning

Materials:

1. Slide show of Tic-Tac-Toe game tree (ply 6), showing development of min-max values
2. Projectable of Alpha beta algorithm
3. Slide show of alpha-beta example
4. Game tree with alpha-beta exercise to do in class

I. Introduction

A. In the search methods we have discussed so far, we have assumed that we are in total control of the decision points along the way. This is not necessarily the case if we are working in an adversary environment. The classic illustration of this is two-player games, in which every other move is totally out of our control.

B. Two player games constitute an interesting and important class of search problems.

1. The ability to play games such as chess is often regarded as a mark of high intelligence, and efforts to produce programs that could do so began early in the history of AI.

2. Game playing is a search problem, in the sense that whenever it is the computer's turn to move, it must choose the best move from a set of legal moves for the current board position. Typically, this is done by exploring the consequences of each move through several moves into the future - e.g. “if I do this, then my opponent can do ..., to which I respond by doing ...”.

3. That, is, at any given time, the computer is confronted by a tree of possible moves - its choices, possible opponents responses to each, possible countermoves by the computer ...

   PROJECT: Tic-tac-toe example - just first slide. Note terminology "6th ply"

C. In addition to raising the problem of our controlling only half the moves, games present a second problem: for any interesting game, the search tree is much too large to be exhaustively searched.

   Ex: Even an uninteresting game like tic-tac-toe has a search tree with about 9! = 362,880 nodes at the outset (i.e. when choosing an initial move). For chess, the estimates range well over 10^100.

   (A game would hardly be fun if one could easily discover a path from the start state to a guaranteed win!)

D. To address these two issues, we introduce two new ideas:

   1. A search technique known as min-maxing, which accounts for the back and forth nature of play between opponents. We will discuss this in a moment.

   2. A heuristic technique known as static evaluation functions.

E. We will also consider various approaches to pruning the search tree in the face of combinatorial explosion. The ability to do this well is one of the key things that distinguishes great game programs in games like chess from just good ones.

II. Min-Max Search

A. For games, we use a search procedure called mini-maxing, which can be viewed as a variant of an and/or tree.

1. Levels of the tree alternate as to whose turn it is to move. At level 1, 3, 5 etc. it is our turn; at levels 2, 4, 6 etc. it is our opponent's turn.

2. When it is our turn to move, we can choose any move that leads to our goal; thus we are happy with the node if any move leads to victory and the nodes corresponding to our moves are OR nodes - i.e. we control which branch to take, so a node is good if branch 1 is good or branch 2 is good or ... An or node is good for me if ANY of its branches is good for me.

3. When it is our opponent's turn to move, we have to be able to cope with anything he might do - that is, we want to put him/her into a situation where any choice still leaves it possible for us to achieve our goal. Thus, these are AND nodes - i.e. the node is good only if branch 1 is good and branch 2 is good ... An and node is good for me iff ALL of its branches are good for me.

   PROJECT TIC-TAC TOE EXAMPLE AGAIN: the top node is an or node; the next level are and nodes, etc. The arcs on the and nodes indicate that we must find an option further down the tree that works no matter which of the possibilities our opponent might choose at this level.

B. The basic idea is to work backward from the leaves of the tree, toward the root, assigning a value to each node.

1. Nodes representing a final state in the game are assigned values as follows:

   ∞ = win for us

$-\infty$ = loss for us (win for opponent)
0 = draw

(SECOND AND THIRD SLIDES - note that in tic-tac-toe at the 8th ply there is only one move possible, so we can back up values immediately)

2. Once all the children of a non-terminal node are labelled, it can be labelled as follows:

   a) If it is an or node, we MAXIMIZE - i.e. we choose the maximum value from among any of the children.

      Rationale: We are in control, so we can choose the best path.

      FOURTH SLIDE

   b) If it is an and node, we MINIMIZE - i.e. we choose the minimum value from among the children.

      Rationale: Our opponent is in control, and will presumably choose he path which is best for him and hence worst for us.

      FIFTH SLIDE

3. Ultimately, when we get to the top level, we will choose from among the moves available to us the one having the highest value.

   SIXTH SLIDE

C. Although we think of min-maxing as being done bottom up, a computational algorithm typically does it top-down, using a recursive, depth first like function. We will look at this shortly.

III. Static Evaluation Functions

A. In our initial discussion of min-maxing, we have assumed that it would be possible to generate the game tree all the way down to the terminal nodes. In general, of course, this is not the case. (In fact, a game for which we can develop the entire game tree is not likely to hold our interest for long!)

B. Thus, at some level in the search, we have to assign values to nodes that are not terminal nodes for the game as a whole. We can do this using a static evaluation function.

1. A static evaluation function is a function that looks at a given board position and assigns it a score - without doing further search.

2. If the board configuration does happen to represent a terminal state, then the static evaluation function can assign an appropriate value for win, lose, or draw.

3. Otherwise, the function returns a value based on an estimate of how likely it is that configuration will ultimately lead to a win for us or our opponent. A position that is "good" for us will be given a value close to the "win" value, while one that is good for opponent will be given a value that is close to the "lose" value.

C. The development of such a function is a difficult task, and requires a good grasp of what is important in the game. What we want is a function that gives a large positive score if the situation is good for us, and a negative score if the situation is good for our opponent.

Example: For checkers, one possibility (though not the best) is the difference between the number of my checkers remaining and the number of my opponent's - perhaps counting kings as 2.

Example: Beginning chess players often make use of a heuristic that values pieces as Queen 9, Rook 5, Bishop 3, Knight 3, Pawn 1 and then compares total scores for each player

1. When a static evaluation function is used, generally the overall strength of the program as a player depends on the quality of this function.

2. One interesting class of applications of machine learning is actually learning a static evaluation function based on playing games. (We will talk about several examples of this later in the course.)

D. We combine the static evaluation function with a limit on the depth of a search. When we get to the prescribed depth limit, we evaluate the board using the static evaluation function.

   1. A simple variant of game tree search uses a fixed maximum depth at all times.

   2. More sophisticated variants may allow some "interesting" branches of the tree to be searched more deeply than others by using some game-specific heuristic.

E. This leads, then, to the following general approach:

   1. To decide what move to make at any given time, explore all the alternatives from the current configuration and choose the one with the highest score.

   2. To explore a node

      a) If it is beyond the maximum depth, evaluate it using the static evaluation function.

      b) Otherwise, if it is a maximizer, explore each of its children and then choose the greatest value

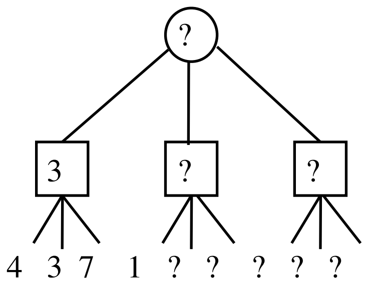
      c) Otherwise (it must be a minimizer), explore each of its children and then choose the smallest value

IV. **Pruning the Tree**

A. One problem with game tree search is that combinatorial explosion quickly sets in, limiting the depth of the search.

B. This effect can be postponed, but not eliminated, by a technique called alpha-beta pruning. At best, alpha-beta pruning can cut the exponent in the exponential growth in half; but in the worst case it may yield no improvement at all. The key idea is that at certain points in a min-max search, we can conclude that pursuing certain branches of the tree will yield no information of any value to us.

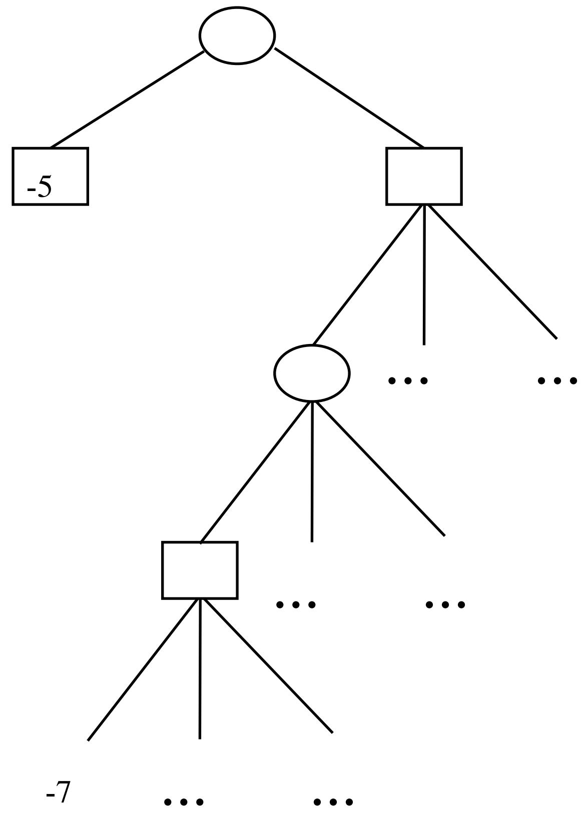
1. Example: Consider the following partially-completed min-max search



Circle = maximizer
Square = minimizer
Number at leaves = static value

Evaluating the remaining two children of the middle minimizer cannot possibly affect the outcome of the overall search. Since we know that the minimizer has one child of value 1, we can be sure that it will not return a value greater than 1 (though it could return something less). However, since the left subtree has value 3, the maximizer at the top will always prefer this 3 to a value of 1 or less, so knowing the exact value of the middle minimizer subtree is irrelevant.

2. Example: Consider the following (where irrelevant subtrees are omitted



At this point, the remaining branches of the minimizer can again be cut off. Why?

ASK CLASS

a) The bottom minimizer is guaranteed to return a value of -7 or less.

b) There are now two possibilities. Either its parent maximizer will find a better value down some other branch, or it won't

   (1) If it will, then the choice made by the bottom minimizer will be ignored, in which case refining it further will not gain us anything.

(2) If it won't then it will have to choose the value returned by the minimizer, which will be -7 or less. But that, in turn, means that its parent minimizer is guaranteed to be able to choose -7 or less, in which case the maximizer at the top of the tree will ignore this value in favor of the -5 it has already found.

Either way, the choice made by the minimizer at the bottom is guaranteed to be ignored, and so is irrelevant. Hence further exploration of the minimizer is not worthwhile.

This is known as DEEP CUTOFF.

C. We can implement this sort of pruning algorithmically as follows:

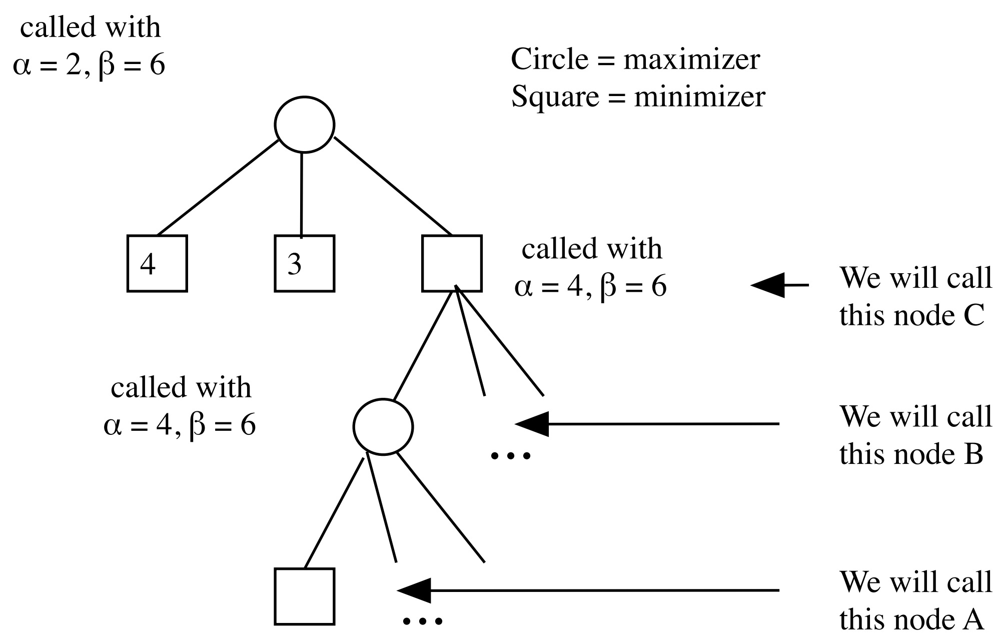
1. When we are exploring a node, we make use of two additional parameters conventionally called alpha and beta. Alpha represents the <u>smallest</u> value we are interested in (because we already know we can do at least as well down some other path.) Beta represents the <u>largest</u> value we are interested in (because we know our opponent will take a different path if we can do better than this.)

2. At the root of the game tree, alpha is set to $-\infty$ and beta to $+\infty$.

3. At a minimizer node, as soon as we find some subtree that has a value less than or equal to alpha, we can stop exploring that node, and simply report this value.

4. At a maximizer node, as soon as we find some subtree that has a value greater or equal to than beta, we can stop exploring that node and simply report this value.

   Ex: Consider the following partial search tree (where only the final result of searching the first two branches is shown, and alpha and beta values come from higher up the tree.)



If node A reports a value greater than or equal to 6, then B can also report this value immediately, and need not consider its other alternatives. (As a maximizer, it cannot be forced to report a value less than what any of its children returned, and the value of beta tells us that we are not interested in anything greater than 6.)

If all the children of the B report a value less than or equal to 4, causing B to report some value less than or equal to 4, then C can also report this value immediately, and need not consider alternatives. (As a minimizer, C cannot be forced to report a value greater than what B reports, and the value of alpha tells us that we are not interested in anything less than 4.
If the node we are working on reports a value of 4 or less, then its parent minimizer can report it and stop searching its siblings. (If he parent minimizer reports a value less than 4, the top level maximizer will prefer the 4 it has already found.) This leads, then, to the following general approach:

To decide what move to make at any given time, explore all the alternatives from the current configuration and choose the one with the highest score.

5. Summary: The alpha-beta algorithm

   PROJECT

6. EXAMPLE: SLIDE SHOW showing development of Alpha/Beta values (Based on Winston figure 4-18, Page 119)

D. There are a number of other heuristics that can be applied to help decide how much of the search tree to actually consider, given search time constraints - e.g.

1. Quiescense

   It is better to stop the search in a portion of the tree where computed values are changing rather slowly, rather than in the middle of rapid change. (I.e. if in a region of rapid change, push search deeper in this portion of the tree.)

2. The Killer Heuristic

   Consider first the move that has the highest likelihood of being ultimately chosen (perhaps using a special heuristic to decide which one this is.) When using alpha-beta pruning, this is the one most likely to lie outside the range defined by alpha and beta, in which case the rest of the subtree can be cut off.

E. There is also a danger with minimaxing called the Horizon Effect

   The danger that a serious problem can lie just beyond the point where we stopped searching and did a static evaluation.

F. Exercise to do in class

   HANDOUT with game tree to label using minimax + alpha-beta

V. **What Has AI Work on Games Accomplished?**

A. In the case of some games, computer programs have been produced that can consistently do well against human players.

1. In some cases, this is because the game has been solved - i.e. the game tree has been totally explored for all possibilities.

    a) A simple example: tic-tac-toe. (Actually, even a good human player can play in such a way as to at least draw, regardless of who starts the game.)

    b) Go-Moku. In this case, the player who goes first can always win.

    c) Checkers. The following description of a checkers-playing program was published in *Science* in July, 2007:

    The game of checkers has roughly 500 billion billion possible positions (5 x 1020). The task of solving the game, determining the final result in a game with no mistakes made by either player, is daunting. Since 1989, almost continuously, dozens of computers have been working on solving checkers, applying state-of-the-art artificial intelligence techniques to the proving process. This paper announces that checkers is now solved: perfect play by both sides leads to a draw. This is the most challenging popular game to be solved to date, roughly one million times more complex than Connect Four. Artificial intelligence technology has been used to generate strong heuristic-based game-playing programs, such as DEEP BLUE for chess. Solving a game takes this to the next level, by replacing the heuristics with perfection.

2. In other cases, there are computer programs which can consistently play well against humans, though not perfectly. Perhaps the best known example is Chess.

a) In 1997, Deep Blue, a chess-playing program developed by IBM (and using special hardware) defeated the then world-champion of chess, Gary Kasparov, by a score of 3.5 to 2.5. (Deep Blue won 2 games, Kasparov won 1, and 3 were draws).

b) Since then, various other chess-playing programs have done extremely well against humans. (If you want to be humiliated, try playing even something like Gnu-chess, a free chess-playing program!)

B. There are, however, other games - such as bridge - where computer programs cannot yet beat good human players.