CS61B Summer 2006
Instructor: Erin Korber
Lecture 22, 2 Aug.

Game Trees

1 Why Game Trees?

- Consider the problem of finding the best move in a two-player game.
- One idea: assign a value to each possible move and pick the highest. (e.g. making the move that gets you the most pieces in Othello)
- But this is often not the best move - it might set up a good move for the opponent.
- So for each possible move, we should consider the opponent's possible moves, assume that they choose the best move for them, and use that to determine the value of our move.
- But to determine the opponent's best move, we need to consider our possible responses to *their* moves, etc.
- How to organize all of these calculations? A *game tree*. Each node is a game state (e.g. board configuation), and each edge is a move. The game tree shows us the structure of the sequence of recursive calls that compute the possible ways that the game can continue after each move.

2 Minimax

- Once we have the game tree, how do we use it?
- The leaves of the tree are completed game states.
- We can assign a value to each final (game-over) position. For example:
  - In a simple win-loss game like tic-tac-toe, we could assign +1 to a win for us, 0 for a tie, and -1 for a win for the opponent.
  - In Othello, supposing that our gamepiece color is white, we could assign (# white pieces on board - # black pieces on board) as the value of a completed game.

- We can use the values of a node's children to determine its value.
- Based on how we assigned values to the final game states, suppose that I am trying to *maximize* the value of the finished game, and my opponent is trying to *minimize* the value.
- So at levels of the tree where it is my turn, assign each node a value of (max(its childrens' values)). At levels where it is the opponent's turn, assign each node a value of (min(its childrens' values)).
- When playing the game, I always choose the next move that leads to the node with the maximum value; my opponent always chooses to go to the minimum value.
- This is the *minimax* algorithm.

Example game tree (diagram from JRS):

So once we have generated the whole game tree, we can use the minimax algorithm to calculate how to play perfectly at each step. There's a problem with this though - the full game tree for most games is so huge, it's impossible to compute in any reasonable time (or even unreasonable time!). So we need some ways to avoid computing the whole tree. We'll look at two - one that keeps us from looking at every branch at a certain level (reducing the with of the tree), and one that allows us to "cut off" the search after a certain level (reducing the depth).

3 Alpha-beta pruning

Alpha-beta pruning is a method of cutting off ("pruning") branches of the game tree based on the idea that if you know you will never get to a certain position (because your opponent will always make a better move down a different branch of the tree), it's pointless to search for an optimal move down that branch.

How do we capture this idea in a way that we can translate into code? We use two variables, α and β to track the best score so far at a maximizing (our turn) node and a minimizing (opponent's turn) node. So α will *increase* as we look at more children (as we find better moves), and β will *decrease*.

Pseudocode for alpha-beta:

- Start with α = Integer.MIN_VALUE, β = Integer.MAX_VALUE
- At a leaf node: return the value of that (final) node.
- At a minimizing level (opponent's move):
  - For each child, until $\alpha \geq \beta$:
    - Set $\beta = \min(\beta, \text{alpha-beta}(\alpha, \beta))$
  - Return β.
- At a maximizing level (our move):
  - For each child, until $\alpha \geq \beta$:
    - Set $\alpha = \max(\alpha, \text{alpha-beta}(\alpha, \beta))$
  - Return α.

Example game tree with alpha-beta pruning (diagram from JRS):

4 Estimator functions

For many games (and essentially all interesting ones), alpha-beta pruning is not enough - we still have no hope of searching the entire game tree. In order to make the search tractable, we need to stop at some specified depth - this amounts to only looking a fixed number of moves into the future of the game, instead of all the way to the end of the game. So now, our leaf nodes will not necessarily be only completed game states - we may cut off a branch before it is complete. We need some way to assign a value to these states - we can do this using "estimator functions" or "evaluation functions".

These functions are a way of deciding how "good" a particular (not-completed) game state is. What this function will be depends entirely on the particular game and the cleverness of the programmer! For example, one very simple evaluation function for the game of Othello would be the same one that we use at the end-game state; (# white pieces on board - # black pieces on board). A smarter evaluation function would use the fact that edge pieces are more valuable than center pieces, and corner pieces are best of all.

Writing good evaluation functions that are also quick to compute is perhaps the trickiest and most "black magic"-esque part of designing game-playing AIs.