Playing Games

Henry Z. Lo

June 23, 2014

1 Games

We consider writing AI to play games with the following properties:

- Two players.
- Determinism: no chance is involved; game state based purely on decisions made by players.
- No hidden information: at any given time, both players have the same amount of knowledge.
- Finite states: a limited number of possible game states.

By game state, we mean state of the board, which contains all the necessary information about the current game (along with whose turn it is). Games which satisfy these properties include checkers, chess, tic-tac-toe, and nim.

1.1 Game Trees

Games satisfying these criteria can be represented completely using a game tree. The tree is built as such:

- The root represents the current game state.
- Every other node represents a possible future game state.
- Each node has as its children all game states which can be reached in one move.

Thus, leaves of the tree signify ways in which the game can end. Paths from root to leaf represent how the game can evolve. As an example, consider the partial game tree for tic-tac-toe in Figure 1.

It is important to know what player is making the current move. In the games mentioned here, each level corresponds to a different player. However, note that the game tree can accommodate players taking multiple turns in a row, as long as we know who is in control of the board at what node.

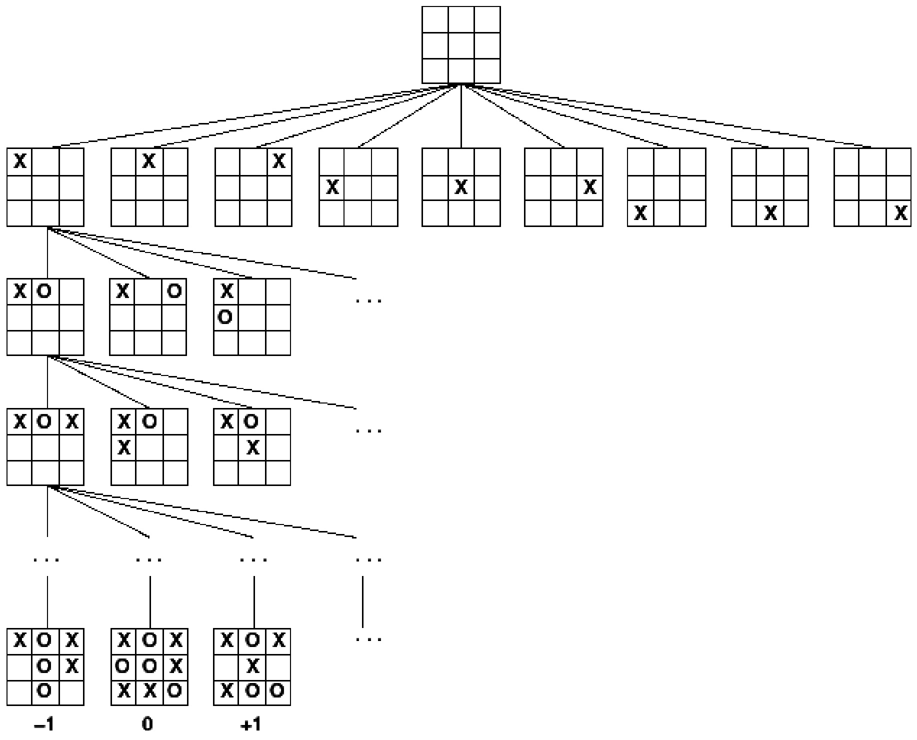

Figure 1: Incomplete game tree for tic-tac-toe, in which x goes first.

2 Game AI

Given the entire game tree, it is possible to find the optimal sequence of moves, or paths, which solve the game. How can we score the paths?

Consider if we were at the state s, which connects to several leaves. We can score winning move as a +1, a losing move as a -1, and a draw as a 0.

What is the score of s? It may not be so important if we were already at s, but it is relevant if getting to s is an option.

2.1 Minimax

The score depends on the player. We can assume that both us and our opponents will always take the best move available to them. If the move from s is ours, then we are in a pretty good position (we can get to 1). If the move from s is our opponents, then we will lose (opponent can get to -1).

The minimax algorithm uses this notion to recursively label nodes in a tree based on its children. Labeling goes from leaf to root. Once we get to the root, we can see which paths will definitely lead us to a win. The minimax algorithm takes this path. See Figure 2.

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := -
        for each child of node
            val := minimax(child, depth - 1, FALSE)
            bestValue := max(bestValue, val)
        return bestValue
    else
        bestValue := +
        for each child of node
            val := minimax(child, depth - 1, TRUE)
            bestValue := min(bestValue, val)
        return bestValue

(* Initial call for maximizing player *)
minimax(origin, depth, TRUE)
```

3 Performance

In practice, game trees can get very wide and deep, and evaluating them can take very long. There are some methods to counteract this.

One method is to use heuristics to give a rough estimation of the value of a node. This heuristic will be used instead of recursive computation at some

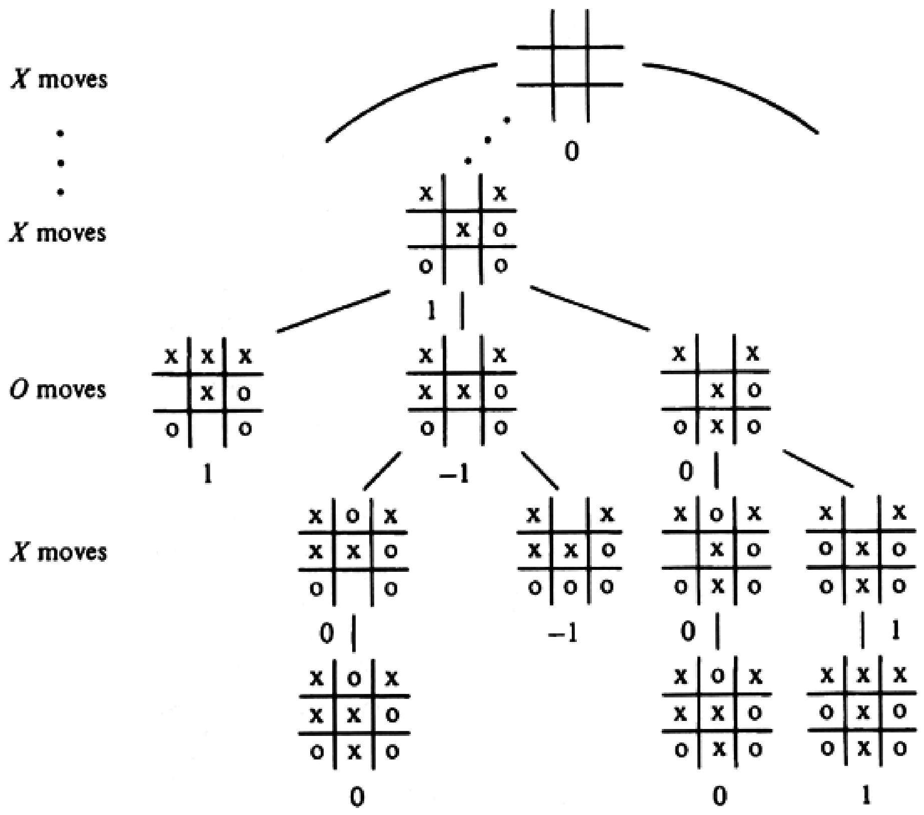

Figure 2: State scores +1 if it leads to a win for X, and -1 if it leads to a win for O.

depth, so that we don't have to traverse down a very deep tree. An example of a heuristic is the ratio of pieces in a checkers game.

Another method is to use memoization. In games with overlapping subproblems (of which there are many), it may be useful to cache values of pre-computed trees. However, this may require a lot of memory to be used, especially in complex games.

4 Alpha-Beta pruning

It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision

```
function alphabeta(node, depth, , , maximizingPlayer)
        if depth = 0 or node is a terminal node
            return the heuristic value of node
        if maximizingPlayer
            for each child of node
                 := max(, alphabeta(child, depth - 1, , , FALSE))
                if
                    break (*  cut-off *)
            return
        else
            for each child of node
                 := min(, alphabeta(child, depth - 1, , , TRUE))
                if
                    break (*  cut-off *)
            return
```