Topics: Minimax, Graphs

1 Minimax

Many times when we are writing a machine player for a game, we would like a way of determining the best possible move for the machine. As a start, we ignore how long it would take to compute such a move. First we will come up with an algorithm to do so and then optmize it.

One slow way to determine the best possible move is to enumerate all the possible moves. But just enumerating the next move may not result in the game ending, so we'll have to recursively generate all possible moves for each turn until the game is over.

We are most interested in turn-based games. Of course, we can easily decide which move to pick for the machine during its turn, but we also need to decide what move the opponent will pick during his turn. To be safe, we assume the opponent is brilliant and will pick the best possible move for himself.

After reaching the end of the game, there are three possible situations: machine win, human win, and draw. Let's assign the values 1, -1, and 0, respectively, to these situations. Then a human player will try to minimize the game score, and the machine player will try to maximize it.

We can trivially score the final boards in our enumeration, but what about intermediate boards between the current move and the final move? If each player is perfect, the at any particular move, they will choose a move that will result in a board with the best possible score for them. So we might as well assign the board at that move the best value out of any of the boards that can result from making the next move. This is done recursively by starting at the final boards and propogating their values upwards. Figure 1 is an example using tic-tac-toe.

The *minimax* algorithm carries out this simulation, but it only recurses on one subtree at a time. So it will recursively score the left subtree, then the middle one, then the right one, and return the best of them.

1.1 Evaluation Functions

The minimax game tree grows exponentially with depth, so the algorithm runs in $O(2^d)$ time. Even in tic-tac-toe, running minimax to completion is very slow. In practice, we stop evaluating after a certain depth. However, not all boards at that depth will be gameover boards, so we have to come up with some way of assigning a value to it. We can use continuous instead of discrete values, and then when we think we are winning, we can assign the board a value between 0 and 1, and when we are losing, we can assign it a value between -1 and 0. The minimax algorithm will still work fine.

1.2 Simple Pruning

Since the minimax algorithm only examines one subtree at a time, once we find a subtree for which we attain the best possible score, we no longer have to check the other subtrees.

1.3 Alpha-Beta Pruning

We can do even better than simple pruning, if we keep track of the best score that each player can achieve so far. We call α the best score the computer can accomplish, and β the best score the human can accomplish. Then if α and β ever converge, we know that continuing to search the current subtree is useless. We always start α and β at -2 and 2 so that some move is always chosen.

It is not immediately clear why we can stop searching when $\alpha \geq \beta$. We may gain some intuition by looking at an example. Figure 5 is a game tree for an arbitrary game. The leftmost leaf board is known to have value 0.3, and the next leaf board value 0.7. When examining the leftmost leaf board, we set $\beta = 0.3$

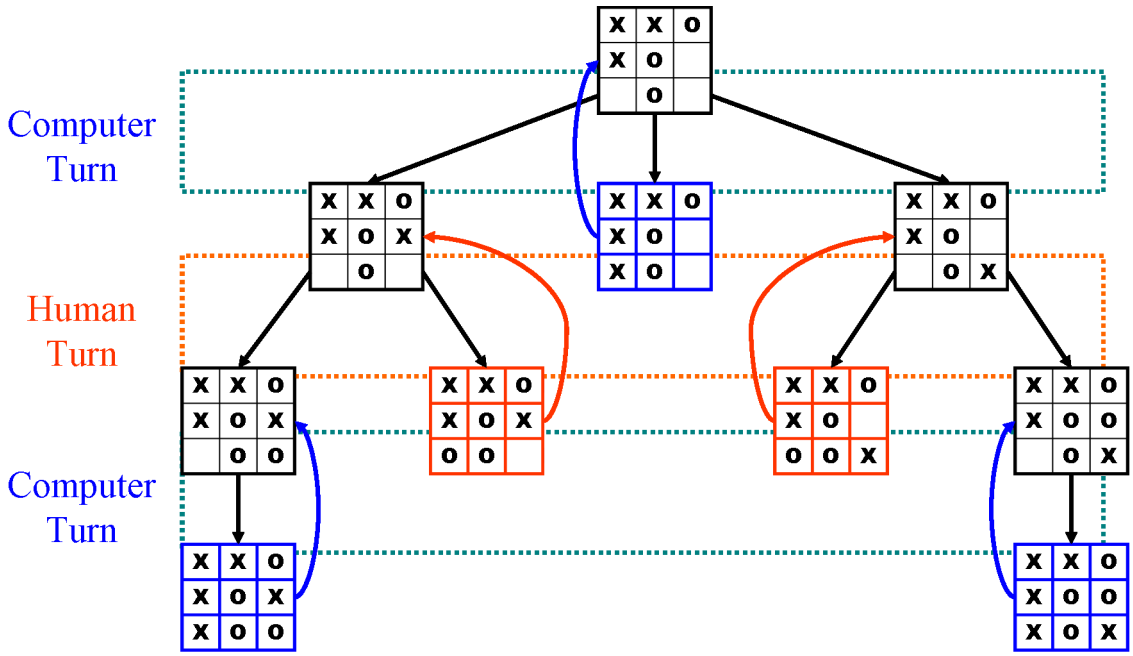


Figure 1: The minimax algorithm applied to a tic-tac-toe game. The computer player is X and the human O. Blue boards indicate a win for the computer, red for the human. Reverse edges indicate the move chosen at each point.

since it is the human player's move, and the new value of β is less than the start value of -2. Then we move on to the next leaf board and set $\alpha = 0.7$. Since α and β have converged, we don't have to examine any of that leaf board's siblings. This is because its parent board must have a value of at least 0.7, because the computer will only choose a sibling of the second leaf board if its score is greater than 0.7. But since the human player knows the parent board has value at least 0.7, it will immediately choose the parent board's sibling, the first leaf board, since its value at 0.3 is better for the human. Thus the first move we examined has value 0.3 regardless of whatever values any other leaf boards in that subtree may have.

Even with pruning, the minimax algorithm is still too slow to run to completion. So the search must still be cut off at some predetermined depth.

2 Graphs

Recall that trees are extensions of linked lists such that each node may have multiple children. However, trees are still prohibited from having cycles or nodes with multiple parents. If we remove these restrictions, the result is a *graph*.

A graph is defined as a set of *vertices* (nodes) and *edges* that connect vertices. Edges may be *directed* or *undirected*, and may have *weights*. A graph with only undirected edges is an undirected graph, and a graph with directed edges is a directed graph. An undirected graph is *connected* if there exists a path from each node to every other node. A directed graph is *strongly connected* if it satisfies this condition. Directed graphs also have *strongly connected components*, subsets of the graph that are strongly connected. A graph is *cyclic* if a cycle exists in the graph or *acyclic* if none do. Vertices have *in-degree*, the number of edges coming into a vertex, and *out-degree*, the number of edges originating from a vertex. Thus a tree is just a directed acyclic graph (DAG) in which each node has in-degree of 1, except the root which has in-degree of 0.

```
Minimax(player)
Let Move be an object corresponding to a move, ScoredMove be an
object corresponding to a Move and its score.
Then:
    ScoredMove bestSoFar = new ScoredMove(); // default
    ScoredMove result;
    // If the game is over, return a fake move and the score
    if the state is a draw then:
      return new ScoredMove(null, 0);
    else if the state is a win for the computer then:
      return new ScoredMove(null, 1);
    else if the state is a win for the human then:
      return new ScoredMove(null, -1);
    fi;
    // We set scores initially out of range so as to ensure we will
    // get a move
    if player is computer then:
      bestSoFar.score = -2;
    else:
      bestSoFar.score = 2;
    fi;
    for each move m do:
      perform m;
      result = Minimax(next player);
      undo m;
      if it is computer's turn and the result is better than the bestSoFar then:
        bestSoFar.move = m; // new best move
        bestSoFar.score = result.score
      else if it is the human's turn and the result is worse than the bestSoFar then:
        bestSoFar.move = m;
        bestSoFar.score = result;
      fi;
    od;
    return bestSoFar;
```

Figure 2: The minimax algorithm.

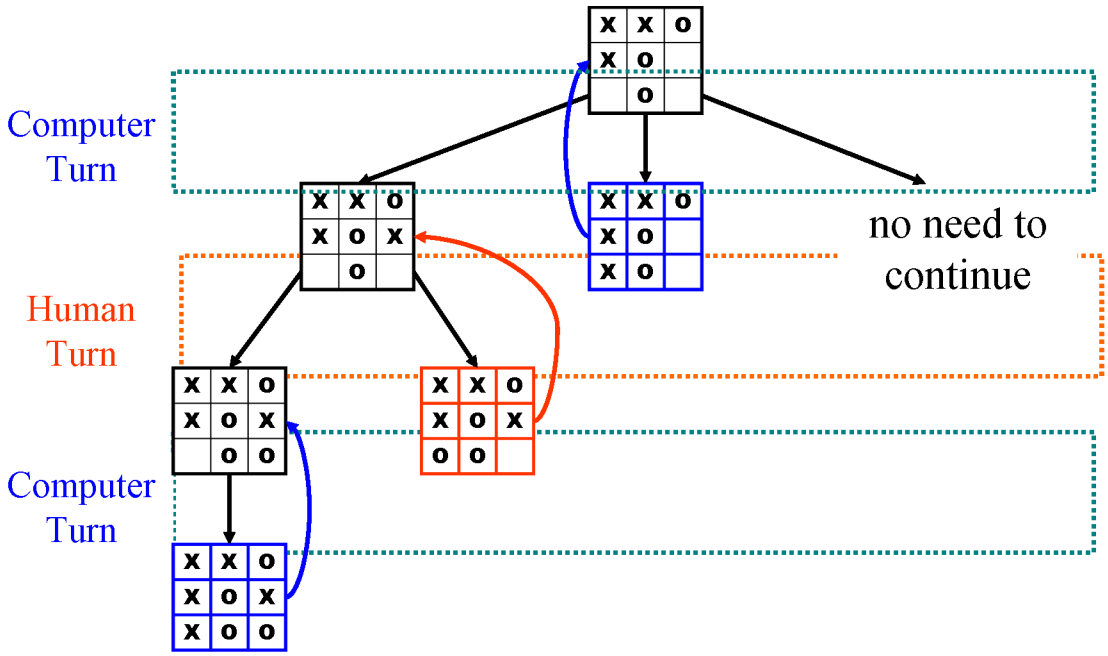


Figure 3: Simple pruning. When a winning move is found, the search is immediately terminated.

There are two major representations for graphs, *adjacency matrices* and *adjacency lists*. An adjacency matrix has a column for each vertex, and a row for each. Where there is an edge between vertices, the adjacency matrix has a 1 in the corresponding position, or the edge weight if the graph is weighted. The row corresponds to the start of the edge and the column the end. An undirected graph is actually equivalent to a directed graph in which each edge has a corresponding reverse edge, so its adjacency matrix represention is as such. When a graph is *sparse*, or only has few edges, the adjacency matrix representation wastes a lot of space since it requires quadratic space. A *dense* graph has nearly n^2 edges, so an adjacency matrix is efficient in that case.

The adjacency list representation requires actual objects corresponding to each edge. Then each vertex stores both incoming to and outgoing edges from itself. Such a representation requires space linear in the number of vertices and edges.

```
ABMinimax(player, alpha, beta)
Let Move be an object corresponding to a move, ScoredMove be an
object corresponding to a Move and its score.
Then:
    ScoredMove bestSoFar = new ScoredMove(); // default
    ScoredMove result;
    // If the game is over, return a fake move and the score
    if the state is a draw then:
      return new ScoredMove(null, 0);
    else if the state is a win for the computer then:
      return new ScoredMove(null, 1);
    else if the state is a win for the human then:
      return new ScoredMove(null, -1);
    fi;
    // We set scores initially out of range so as to ensure we will
    // get a move
    if player is computer then:
      bestSoFar.score = alpha;
    else:
      bestSoFar.score = beta;
    fi;
    for each move m do:
      perform m;
      result = ABMinimax(next player, alpha, beta);
      undo m;
      if it is computer's turn and the result is better than the bestSoFar then:
        bestSoFar.move = m; // new best move
        alpha = bestSoFar.score = result.score
      else if it is the human's turn and the result is worse than the bestSoFar then:
        bestSoFar.move = m;
        beta = bestSoFar.score = result;
      fi;
      if alpha >= beta then:
        return bestSoFar;
      fi;
    od;
    return bestSoFar;
```

Figure 4: Minimax with alpha-beta pruning.

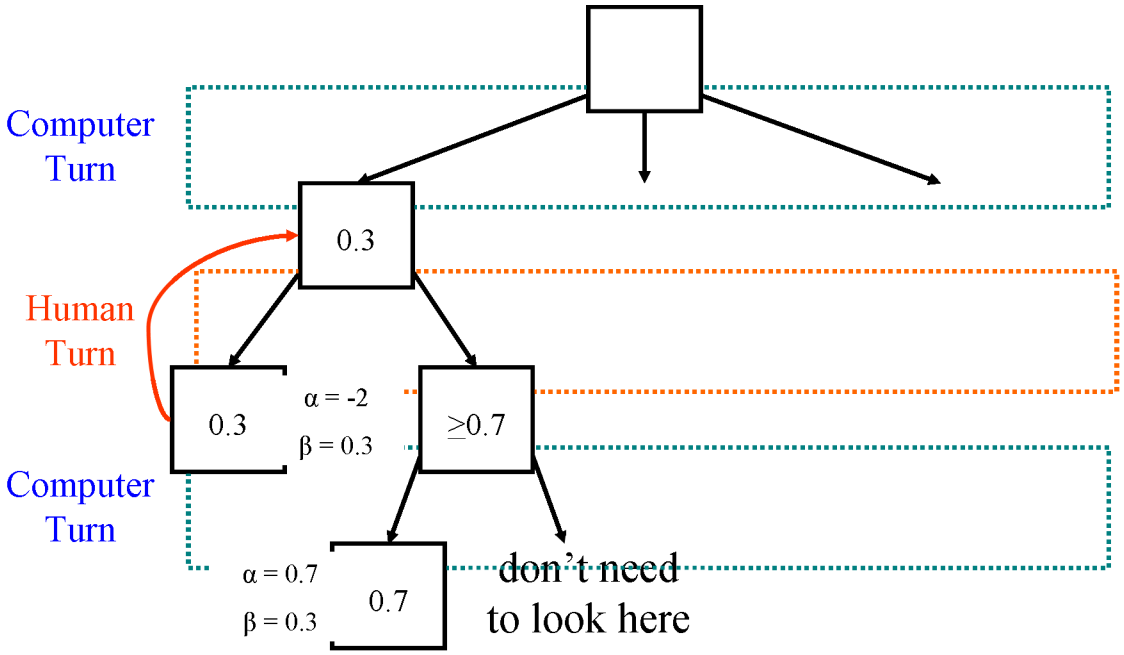
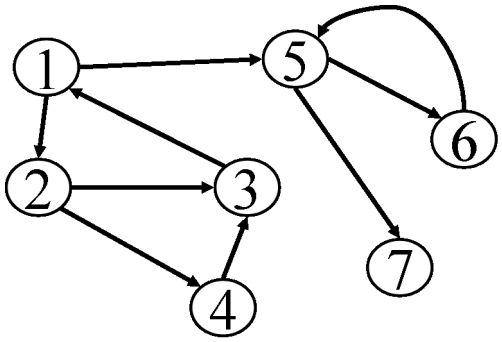


Figure 5: Alpha-beta pruning. When alpha and beta converge, it is useless to continue searching the current subtree.



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Node	In	Out
1	31	12, 15

Node	In	Out
2	12	23, 24

Node	In	Out
3	23, 43	31

Node	In	Out
4	24	43

Node	In	Out
5	15, 65	56, 57

Node	In	Out
6	56	65

Node	In	Out
7	57	

Figure 6: A directed graph and its adjacency matrix and adjacency list representations.