Why study games and game playing?

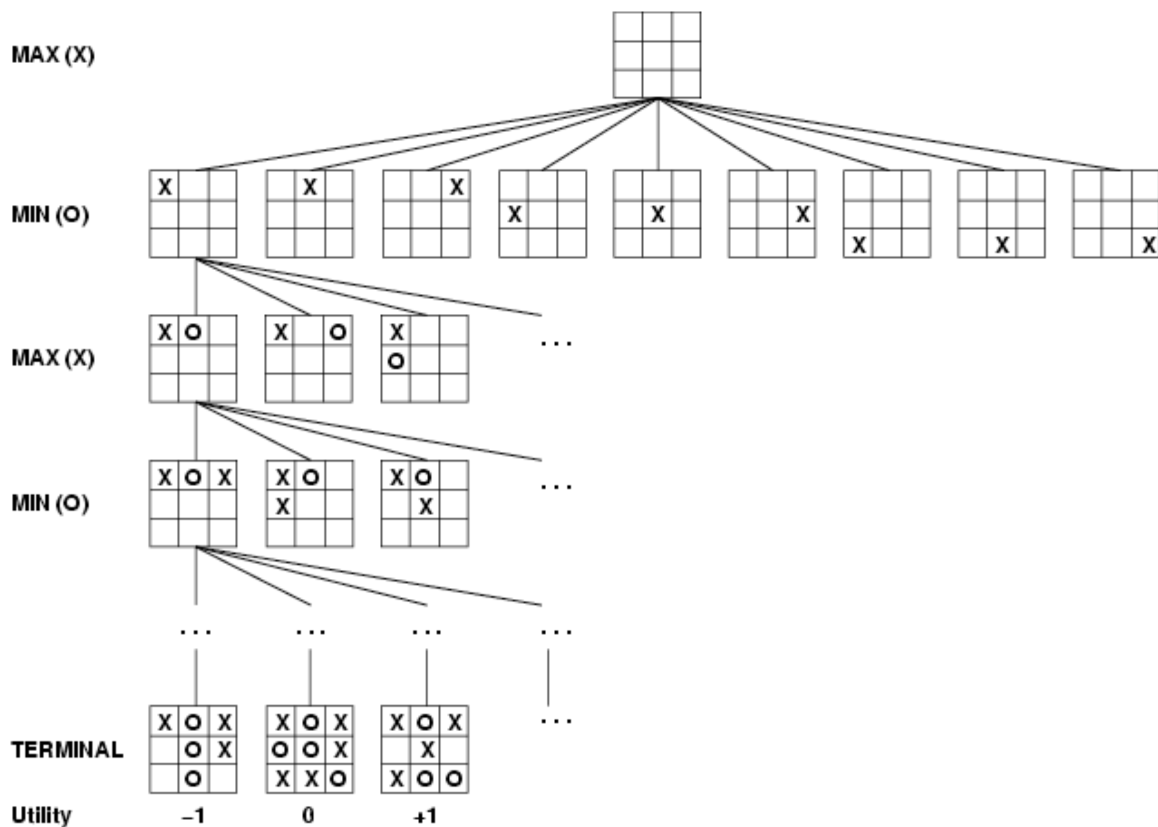
Well-defined example of a problem-solving application.

Easy to measure success and failure.

While the amount of knowledge necessary to perform the problem solving might be large, it's also generally easy to specify. (Rules of a game are generally easier to itemize than rules of the world.)

Complexity of uncertainty (adversary's actions, roll of dice, etc.).

Game playing as a search problem



Note that each level in the game tree (i.e., each half move) is called a **ply**.

Formulating game playing as search

Initial State is the current board/position

Operators define the set of legal moves from any position

Terminal Test determines when the game is over

Utility Function give a numeric outcome for the game

[Note the relationship between the Terminal Test/Utility Function and the Goal Test we discussed earlier.]

The tricky part, of course, is that you can't control an entire path to the conclusion of the game – you can only control your own moves.

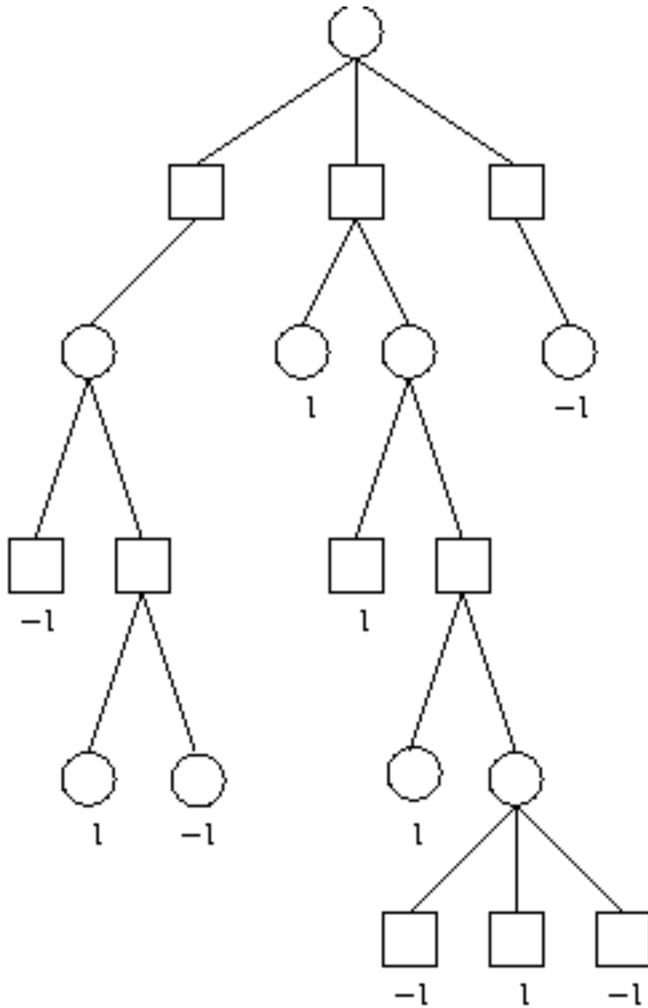
Simple Minimax

Basic ideas behind the algorithm:

Generate the complete game tree, applying the utility function to each terminal state.

Select a move that is best for you, given that your opponent will select moves that are worst for you.

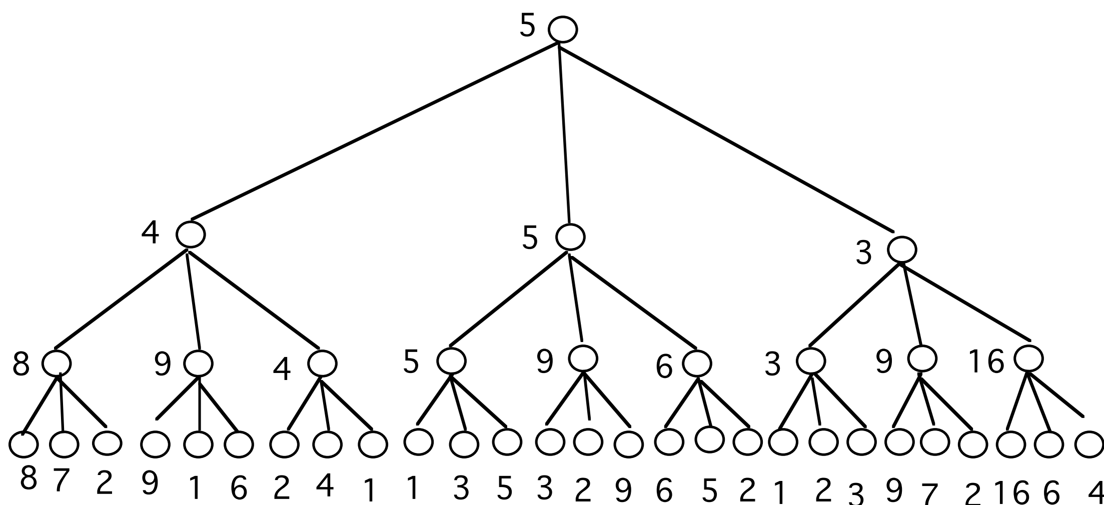
Example 1. Assume that the following game tree is complete.



Simplified Minimax Algorithm

1. Expand the entire game tree.
2. Evaluate the terminal nodes as wins for the minimizer or maximizer.
3. Select an unlabeled node, n, all of whose children have been assigned values. If there is no such node, we're done – return the value assigned to the root.
4. If n is a minimizer move, assign it a value that is the minimum of the values of its children. If n is a maximizer move, assign it a value that is the maximum of the values of its children. Return to step 3.

Example 2. Again, assume the following game tree is complete. The opening player is the maximizer. This time the object of the game is to maximize one's score. The terminal nodes are evaluated to determine the utility for the maximizer. The values of the internal nodes are determined by applying the simple minimax algorithm.



The Need for Imperfect Decisions

Problem: Simple minimax assumes that the program has time to search to the terminal nodes.

Solution: Cut off the search earlier and apply a heuristic evaluation function to the states on the frontier.

Static Evaluation Functions

Minimax depends on the translation of game state quality into a single, summarizing number. This is difficult and potentially quite expensive.

Some ideas for constructing evaluation functions:

- Add up the values of the pieces each player has.
- Isolated pawns are bad.
- How well protected is your kind? (other vital pieces?)
- How much maneuverability do you have?
- Do you control the center of the board?

Minimax

State:

Current game configuration
Player to move
Most recent move made [To help focus on whether an immediate block is needed]
Current ply = 0

Method **minimax-decision** takes a state as a parameter:

```
for each successor in state.successors()
    value = successor.minimaxValue();
return successor with the best value;
```

Method **minimax-value** also takes a state as a parameter:

```
if (state.terminalTest())
    return state.utility();
else if (state.maxPly())
    return state.heuristicEvaluation();
else if (state.maxToMove())
    return max(minimaxValue() applied to each of state.successors())
else // min is about to move
    return min(minimaxValue() applied to each of state.successors())
```

Note that we go about a minimax search **depth first**!

Example 3.