Lab 7: Game Playing and Adversarial Search

AI701: Artificial Intelligence

October 2021

Abstract

Tic-Tac-Toe is a *zero-sum game*, meaning in the end, the gains and losses among the players add up to zero. In Tic-Tac-Toe, one player wins, another loses, so the sum of "utility" for each player at the end is zero.

The minimax algorithm does exactly that: it finds the move that minimizes the maximum utility the opponent can obtain. Minimax search is very costly, since far too many states are searched. Alpha-beta pruning helps to reduce the searching states.

In this lab, we will implement minimax algorithm and alpha-beta pruning on a Tic-Tac-Toe game. You are required to implement the following functions without using any pre-built libraries. The following exercises require implementation in code skeleton provided in "tic-tac-toe.ipynb".

1 Part A: Understanding Minimax Search

In this section, we go through the minimax search algorithm. The components of a search problem (such as a tic-toc-toe game) included the following:

1. **Initial:** Some description of the agent's starting situation.
2. **Players:** A list of players, we'll look at 2-player games in this lab, such as HUMAN and COMPUTER.
3. **Possible actions:** The set of actions available to the agent, such as the function *empty_cells()* returns the valid actions. The possible actions depend on the state.
4. **Terminal tests:** A function or functions that test for final states (win / lose / draw states), such as the functions *game_over()* and *wins()*.
5. **Transition model:** Some ways of figuring out what an action does, such as function *transition_board()* update the board.
6. **Utility function:** A function returns a utility which measures the state of player, such as the functions *min_utility()* and *max_utility()*. Generally speaking, winning states have the highest utility '1', losing states have the lowest utility '-1', drawing states have zero utility '0'.

Minimax Search algorithm considers all the possible moves made by players. Among all possible moves, we want to find the move that leads to the best utility (max; that is, we are the maximum, the opponent is the minimum), under the assumption that the opponent also performs best (min; that is, we are the minimum, the opponent is the maximum). In other words, we seem to find the best result in the worst case.

Here is an example search tree for tic-tac-toe. We basically have a depth-first search but also propogate information about utilities (max or min) upwards.

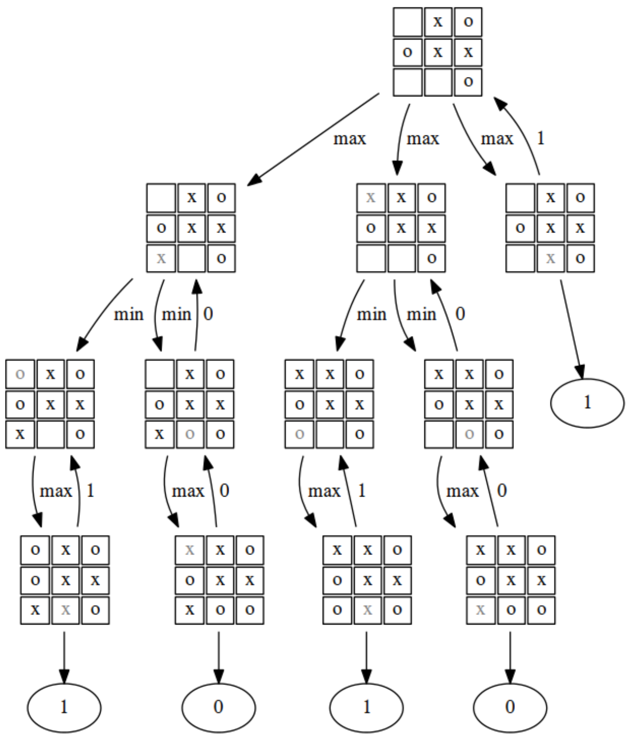

Figure 1: Minimax search tree for a tic-tac-toe game

Similarly, in the function *minimax()*, we consider all of our possible moves.

1. If it's not a final move, we have to look further to calculate its utility, *e.g.*, see the 1st layer in Figure 1. We first choose one move from which to "look further", *e.g.*, see the 2nd layer, and then consider all the possible moves the opponent might make, *e.g.*, see the 3th layer. If the opponent wins, the utility of the opponent's move is -1. Of all the possible moves the opponent might make, we find the worst (minimum) utility.

2. If a move is a final move (winning/losing/drawing), *e.g.*, see the 4th layer in Figure 1, then we calculate its utility directly (1 for a win, -1 for a loss, 0 for a draw).

2 Part B: Implementing Alpha-beta Pruning

In this section, you need to update computer's decision-making module with alpha-beta pruning algorithm.

Minimax search is very expensive since it considers all the possible moves. We can use a very simple and somewhat obvious optimization to search only a small number of states, and at the same time arrive at exactly the same answer as minimax.

The key insight is that when we are performing the "max" step, if we find that a certain move gives a maximum utility X by searching its entire subtree, we can avoid any searches of subtrees whose minimum value is less than X.