Artificial intelligence

Othello

David Tran
John Tengvall

Artificial intelligence - DA272A
Computer science
2016-09-28

Assignment

The purpose of the assignment was to get a deeper understanding of the alpha-beta pruning search algorithm.
The game we are using this algorithm on is Othello, where our mission was to design a computer player against the human player who will play the game. The computer will use the algorithm to calculate which move it will do next.

Implemented Rules of Othello:

The rules is based on a fairly simple othello game with 4x4 board. It starts with four pieces in the middle so player and computer have 2 pieces each to start with. Because it is 4x4 board and not the usual 8x8 board there is not a limit where you can lay your move. Player always start and can choose from 12 different positions. If the player wants to reset a game when it’s finished or if he made or if he wants to forfeit he can press the reset button. This will initialize the starting board.

Game classes

Othello class:

Gui

The user interface is build with panels inside the othello class. When the game starts it creates 16 JPanels inside a gridlayout. These panels will represent the pieces that turn with white color respectively black if either computer or player made a move. For each of the 16 panels there is a mouselistener who listens where the player clicked. On the right of the gameboard there is two images with the color of human (black) and computer (white). Next to the images there is two text labels so the player can keep track of the score.

render()

The render function updates the board with every move. When player or computer have made a move it will call the “render function” where it will go in a nested for-loop. It will check every piece and what number it possesses. If the piece has number zero it means that it is empty and will paint the piece green and if the piece either has the number one respectively negative one it will paint it black or white. After each turn the scores are updated.

playerMadeMove():

The first thing the function does is to take the row and column where the player made its move. This will show up in the console.
It will then call the checkAllDirections() function and check every direction. If it finds the opponent's piece right next to its piece it will check if it has its own in the same direction. If so, it will turn the other piece/pieces. At last it will call the function render which are updating the board and then change currentPlayer so it’s the computers turn.

comMove():

This is where the computer's move is taken care of. On every computer move, a new instance of ComMove is made. The object then calls the method alphaBetaSearch() which will calculate the best move and return an index number between 0 - 15. To get the row it divides the index number with 4 and to get the column it takes the reminder of modulo 4. When it get the place it will do the same check as player did with checkAllDirections(). After that it will render the board and change to human player.

checkAllDirections():

This method checks if the placed piece will flip any pieces and returns the a new state with an updated board.

State class

The class represents the current state of the game. It contains a two dimensional integer array, which is represented by values. 0 means empty, +1 is computer piece, and -1 is a human piece. The class has different necessary get and set methods.

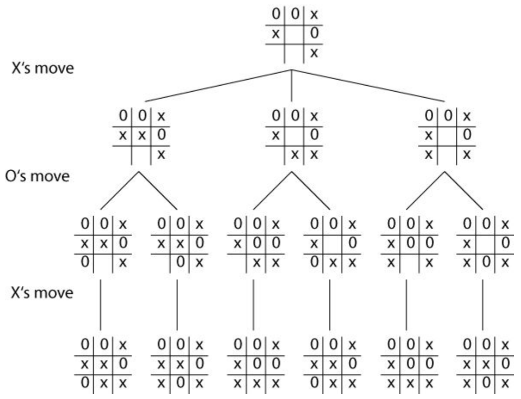
ComMove class

This class is used to calculate the best possible move for the agent to make by using alpha-beta search algorithm.
When the Othello game calls for the function it parses its current state. The function returns an index from 0-15 which represents the square of where the agent should place its piece.

Search algorithm

Minimax

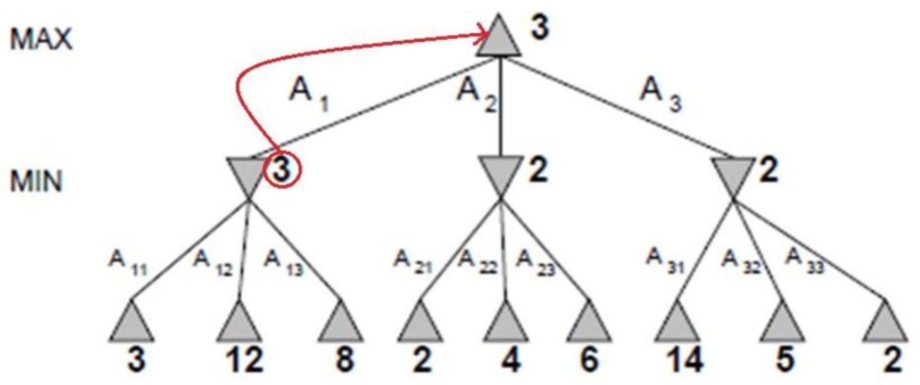
When trying to apply a best move for the agent (computer player) in a deterministic game, a common way is to use Minimax. Deterministic is when the next state of the environment is completely determined by the current state and the action executed by the agent [1].
The minimax is a game tree, where from the root, all possible game states are shown.
Here is an example of a part from a game tree of the game three in a row

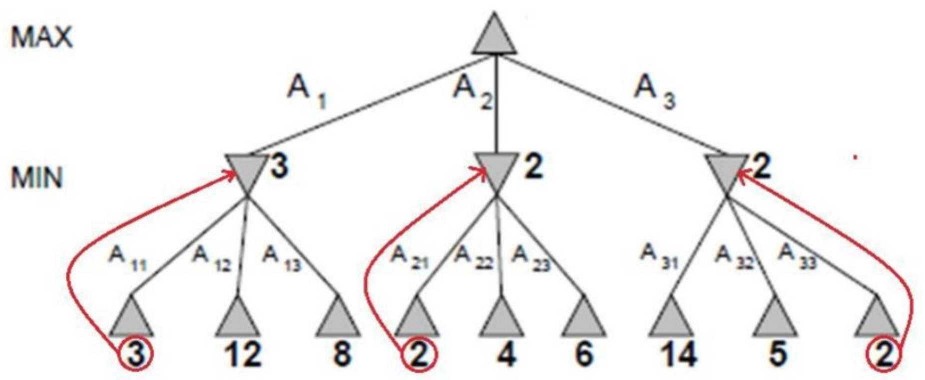


[2] Image of a gametree

Each level of the tree is the possible next states of each player. The root on the picture shows that there are three empty slots, and that node has three children, meaning that there are three possible new states. Each node has a utility function. In this case, it can be the number of possible winning rows for the agent minus winning rows for the player. In othello it can be number of white pieces minus black pieces.

Meaning of the tree is that the root will node will choose the path to the child with highest value. The root is called a max node. The child will be a Min node, meaning it will choose the path to its child with the lowest value. Example shown below



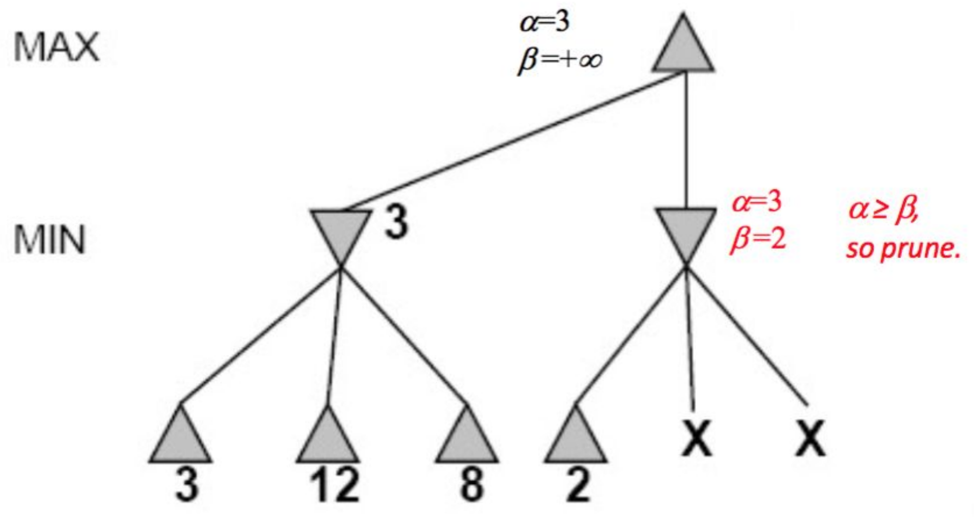


[1] Image of MinMax algorithm

Alpha-beta pruning

The problem with Minimax is that the whole tree has to be constructed. This is time and power consuming. Alpha-beta pruning is based on the same principle as Minimax, where each level is a Max node or Min node.
But instead of looking at every node in the tree, it prunes when no more evaluation is needed in that bransch. On every level alpha and beta value is parsed from the parent, where alpha initially is -infinity and beta is +infinity.

These variables are updated during search. Alpha is updated at Max nodes and beta is updated at Min nodes. Example shown below



[1] example of Alpha beta search with pruning

Alpha-beta pruning search uses depth first search meaning it will traverse as far left as possible and then gradiently take the child to the right. Also it is a recursive method, which means that utility values are found “at the bottom”. In the example, the left Min node updated its beta value to 3 and brought it back to the root node, which updated its alpha value to 3. The alpha value is parsed to the roots second child, and because the child found a 2 in its first child, pruning can be made, because the root will always take the Max of its children and no matter what the second child finds in the pruned nodes, the beta value can not change to be higher than 3.

The search algorithm in our game

In our assignment we used the alpha-beta pruning search algorithm. We followed the pseudo code we were given to structure our agents move.
The utility value of the nodes is the current score of the board. Agent's pieces (illustrated with white pieces and +1 in code) minus human pieces (black, -1).
As cut off criterion we are using a maximum depth search. During building of the game we have been using 8 as depth level. If a leaf node is reached before the depth limit, there will also be a cutoff.
Because every node only has a utility value, and we want to return an index to where the agent should place its piece, we are using a variable. This variable, rowColIndex, is only allowed to be updated in the root node, because that state contains of all empty squares which are possible for the agent to place a piece on. Root node is also a Max node, and that means its alpha value is being updated, and when that

occurs, we also update the rowColIndex. We know the path of the tree for when the alpha is updated, so the row and column value to the roots chosen child is the row and column for the rowColIndex.

What would you have done differently if you had done it all over again?

We would probably make the game play dynamical. Now we are playing Othello on a 4x4 board, without other option. And we are allowed to place a piece on any empty place, which is not allowed in real Othello. We would make that as a restriction in game.

What experiences have you acquired?

David:
It was very fun to make the AI side. I have always wondered how a computer player can have various difficulties, and in this particular alpha-beta pruning algorithm it is done by changing the depth of the search limit.

John:
We have made some search algorithms but his was very helpful and funny to make because it is applicable on so many levels. If I have to say one negative thing is that it is hard to debug the algorithm when it goes down to millions of nodes and you don't know where the problem is.

Reference:
[1] Font.J (2016) 'Games and searching' available at: http://mah.itslearning.com/ (Accessed: September 14)
[2] Chris Thornton, Gametree , viewed 28 september 2016,http://users.sussex.ac.uk/~christ/crs/kr-ist/lec05a.html