Minimax Isolation Report

Dylan Chung

CS 4200.01 Spring 2020

Dominick Atanasio

May 1, 2020

Program Description:

The following program implements the Isolation Game in which two players play against one another in 8x8 board with two pieces; “X” for the computer, and “O” for the player. Each player may move like a Queen moves in chess, diagonal, updown, and left right as long as the path is not obstructed “#” or the opposite player. Once a spot has been occupied, it’s updated with a “#”.

The goal of the game is to exhaust the opponent’s moves before your moves are exhausted. The following program implements both the game with an A.I. that controls the player “O” using the minimax algorithm with alpha beta pruning and iterative deepening.

My Approach:

I implemented the UI first along with the board and game logic so that I could playtest all the functions of movement through player vs. player. Possible movement across the board was determined by checking each diagonal, up, right, left, and down index the current player could come across without obstruction for each character. Legality of a move was determined in a similar manner. In retrospect, I realize I could have used the list of possible movements to determine whether the selected move was legal or not rather than recalculating it. It would have been more efficient, however checking if a movement in one direction is legal or not again is not very expensive in the first place as at most you would be checking 7 indexes. Regardless, once I was sure that all the movement functions and game logic was working as intended: including proper UI/game log display and ensuring a player could not perform illegal moves, I implemented the computer AI.

The bulk of the implementation comes down to the minimax algorithm with alpha beta pruning algorithm. For this I decided to group boards in a node class with access to a list of childrens. Following the minimax pseudocode, I was able to get a “v” utility value of the node with the best fitness at the bottom most depth. As I passed the “v” value up through recursion, I saved it to the nodes. This meant that when I returned “v” through minimax, I could traverse the root node’s children node to find the first board that had a “v’ value matching what minimax returned in the first depth, and in theory, that first depth node should lay on the path of the best bottom most node found.

Strategies:

The heuristic used to evaluate the fitness of a node is not very complex. I simply researched game strategies online like suggested in class and came across the heuristic function of:

```
// Heuristic Defensive
// Number of available moves - opponent moves
public int getFitnessVal() {
    computerPossibleMoves = new ArrayList<int[]>();
    playerPossibleMoves = new ArrayList<int[]>();
    numCompMoves = getAvailableMoves("C"); // COMPUTER
    numHumanMoves = getAvailableMoves("O"); // HUMAN
    //System.out.println("X has " + numPlayerMoves + " moves possible at this stage...");
    //System.out.println("O has " + numEnemyMoves + " moves possibles at this stage...");


    double weight = (numCompMoves + numHumanMoves)/64;

    // Go on offensive during second half
    if(weight <= 0.5)
        fitness = numCompMoves - (numHumanMoves*2);
    // Then play defenseively second half
    if(weight > 0.5)
        fitness = (numCompMoves*2) - numHumanMoves;


    return fitness;
}

}
```

It's really simple, however I also read that it was possible to get it to switch strategies mid gameplay. I debated whether I should switch strategies when the AI had significantly less moves than the human player left, and go on defense by using the defensive heuristic. In the end, I decided that when half the board was used up, the AI would switch to defense. It would be reasonable to assume that much of the board has been occupied by then and the better idea would be to focus on not getting trapped rather than attacking.

Problems and Difficulties:

Naming Inconsistencies

The program was progressing swimmingly during the game logic, UI, and player vs player implementation phase. The only issue that arose here was forgetting my naming scheme. I had labelled the enemy with symbol "X" and player with "O", but at times I would forget I had done that and call them "C" and "P" for player and computer. Although minor, the inconsistent naming became a nuisance at one point as I could not make logical sense of what was what

anymore and when to use which symbol. I didn't fix this, and just soldiered through, but I would do it differently next time.

Attempting to Implement Minimax with AB Pruning on a misunderstanding

I watched a few YouTube video lectures on the subject and thought I understood it enough and went to implement it. Because the lectures showed them working off a full tree to some depth, I thought I was to build an entire tree then run Minimax on it. So I created a tree structure only to realize that the lecturer only created an entire tree to demonstrate visually how the process worked and that I should be generating the tree as I run minimax. However even when I finally understood it and had it running, I could not understand how getting the fitness of the bottom most node that represented the best path would help. That's why I thought to pair the bottom most node in an object with the utility value and just return that. This would give me the best board at the lowest depth and allow me to just iterate up it until the next parent is not the root node, effectively giving me the next move.

In the end, I realized since I was recursively crawling up the tree anyways, I could just assign the utility value to the nodes as I went, and search the children of the root for which one was assigned the utility value minimax gave.

Time Limit in Iterative Deepening

I set the loop to stop when the elapsed time exceeds the time limit and in Max_Value() and Min_Value(). Each of these method has something to check for time, but generally, even if time limit is exceeded, there is no way for Max_Value() and Min_Value() to stop it's recursive function. I made it pass up a random value as obviously I wasn't using that depth anymore so it didn't matter what Minimax generated at that depth and no more calculations had to be run, but it still takes a few seconds to finish up all the recursive calls it made. This ate into time, but also child generation was taking up too much time. You could enter a loop at 4.99 seconds and you'd have to deal with it generating a tree from depth 1 all the way to whatever depth it was again. This ate up the most time.

To solve this, I found that I shouldn't be re-generating successors during each loop if they had been made for that node before during previous iterations. Instead I said that if the child node below it is not null or it exists, then just set that as the successor list and carry on. If it doesn't exist, then generate it. Now a 5 second time limit will realistically stop at like 6-7 seconds. I timed this with a stopwatch so it could be my reaction, but it's close enough.