COMP 4106 Project Report - Othello

Jordan King (101043160)

Description:

Othello is a game that is played on an 8x8 game board of square tiles. The middle 4 tiles each start occupied by a circular chip, two white, two black, where two similar color chips are not placed side by side, but rather diagonal. The game is played by placing a chip (of your respective color) adjacent to any chip on the board. If there is another chip of the same color in the vertical column, horizontal row, or diagonal row of the new chip that was just placed down, then any chip of the opponent (different color) found in between your two chips will be "captured" and turned to your chip color. The goal of the game is to own as many chips as possible by the end of the game.

Motivation:

The motivation behind creating an AI version of Othello is honestly just to see how it would run. It'd be pretty interesting to see how Othello can be played with an AI going against an AI of similar strength and of weaker strength, as well as seeing the consequences of what happens on a bigger grid.

Since Othello is typically played on an 8x8 grid, if it is possible to run this on a 30x30 grid, how would things change? Would we be able to create a multiplayer game out of this as well? Three players may be tough and give an unfair advantage to a certain player, but 4 players should work just fine. The only advantage may come from the order in which the players must make their moves.

Also, how would the shape of the board look in a bigger grid? Clearly on an 8x8 grid there isn't many situations where there would be any open pockets on the board that gets surrounded and closed in by the chips, but how often would this happen in a bigger grid, would it be played often, how effective would that area be, etc. It's a pretty interesting idea to see how things may be affected by a bigger grid or multiple players.

Maybe you can even have a few mini games of Othello going on at once, or rather, multiple starting points. For example, you could have four regular Othello boards side by side with their regular starting points (so 4 total). Each player can still only place 1 chip per turn, but they have 4 locations in which they can affect change. At some point these 4 positions will merge into 1, and I wonder how the strategy would be for a situation like that based off some simple heuristic functions.

AI Techniques:

Ai techniques that could be used for Othello are simple two player search algorithms such as minimax, or minimax alpha-beta. This should be a definite addition to this problem as it is a basic way to solve this two-player game through searching using heuristics. It allows for the classic Othello to be played, and it allows me to test my thoughts mentioned above about playing Othello on a bigger grid.

Ai techniques that could be used for Othello are simple two player search algorithms such as the Max-N algorithm which is like a greedy version of minimax for N players. It's a nice algorithm that allows depends on the heuristics given to it. It also works for N players, which allows me to attempt to implement the idea of having both a 2 player and 4 player version of Othello to see what the results of the game would be with additional players. It should also allow me to change the gameboard size as I please, depending on what situation I would like to test.

Another interesting search that I could attempt to implement if I have the time would be the Monte-Carlo search. This would be a nice algorithm to implement as it is done through the use of randomness and probability, not through heuristics. I would be curious to see how a search such as this one works with Othello, and then how it would scale to 4 players or to bigger board sizes. It would also be interesting to see which algorithm (randomness and statistics, or heuristics) ends up winning the most games.

Design choices:

There were a few decisions that were made for the design choices of Othello.

Firstly, I decided to use a basic console UI instead of a graphical UI. The graphical UI would be favorable when creating a bigger board size, as with a graphical UI you would be able to scale the size of the gameboard to fit full-screened on a monitor. But, considering that running a 100x100 grid isn't necessary to show differences in how Othello is played on a bigger, I opted not to create a graphical UI. Instead, I went to a max size of 40x40 on a console screen, which still has the ability to shrink its text size if I really needed to create a bigger grid. I also opted to use ALT keys to print out the grid as it is more visually appealing.

For the actual game design, I decided to make a simple actions list that consisted of 8 tuples, each containing instructions of how to go from one location to the 8 adjacent squares. This is used to create valid moves.

The state space is saved as a 2D array holding the location of every square. It would be impossible to represent the game otherwise in a relatively straightforward manner as the game board gets increasingly more convoluted as the game goes on. The other option would be, for each row, to list what element is in the first position, followed by the amount of similar elements directly after it until a new element or game piece is reached, and then do the same thing for that game piece, etc. until we reach the end of the row. The issue with this is that after the half way mark of the game, the array could be holding N entries already (N being the number of squares on the grid), and by the end of the game, worst case, the array could hold around 2N entries. This didn't seem ideal in the long run, so I stuck with the simpler format.

Max-N was implemented essentially as described on the slides with a few minor changes. Instead of holding values and tuples, everything is done with a node class that also contains the current state, the previous state, the new coordinates of where the player wants to play their piece, which player is playing that piece, and a few other variables.

The Monte-Carlos algorithm was to be implemented in a similar fashion to what is in the slides for UCT algorithm, while also referencing an article from *Towards Data Science*. The state would still be the same, with additional node variables, some of which being, for example, the probability of winning with this move.

There is a grid generator that was created for this game. It essentially creates a grid of any size with the following restrictions:

1. For a 2-player game, the starting point will always be in the 4 middle tiles of the grid, with each players chip or game piece being diagonal to each other, not directly beside each other. I refer to this as a "pocket". The pocket can't grow in size, regardless of the size of the gameboard, and always stays as 4 pieces. The minimum gameboard size is 8x8, with no maximum size.

2. For either 2 or 4 players, the game board is created with 4 pockets. Imagine taking the gameboard from the first restriction point above, creating 4 identical versions of it (essentially creating 4 quadrants), and putting those 4 identical versions together to make a square. Now there are 4 possible starting locations (4 "pockets"). If 2 players are selected, the pockets are all the same. If 4 players are selected, then 2 diagonal pockets will have game pieces relating to the first two players (as seen before), and the other 2 diagonal pockets will mimic the same pattern using the 3rd and 4th players game pieces. There is a minimum grid size of 16x16, with no maximum size.

Keep in mind that the first restriction is done to allow the play of classic Othello with 2 players on an 8x8 grid, but also allows the grid to scale as desired, since this is a feature that I was interested in seeing the results of. The second restriction allows generically allows me to answer the question of how the game changes when using multiple starting points, as well as allowing more than 2 players to play if desired, and allows the grid to increase in size as desired. Also note that there are no restrictions on any grid that says it has to be a square.

Results:

The resulting application of everything mentioned above went well for everything that was truly attempted. The UI looked clean with the ALT keys, and functioned as planned. The action list of 8 tuples worked well in terms of validating moves and aiding in the creation of new, valid moves. The state representation of a 2D grid worked as planned, and in a very simplistic manner. It fit in well with the console-based UI as well as the algorithms. Man-N was implemented without many issues. The biggest issue being how to handle variables that are supposed to be consistent throughout the entire algorithm and whether or not they should be held and duplicated in the nodes, or created as global variables that can change depending on the game. The grid creator was done relatively easily, even though it was slightly more hard-coded than desired. It still allowed all the required flexibility that was described above though.

The Monte-Carlo algorithm was the only things that wasn't implemented. This was not due to any difficulties in implementing the algorithm, but really more due to personal health issues that held me back from even attempting to implement it. What was implemented in it's place is the randomness of

the Monte-Carlo algorithm, where you could play a random game of Othello up to a certain amount of turns, or until the game is over. In a more lame way of creating the Monte-Carlo algorithm, I could have just run this a randomness a predefined number of times and gotten probabilities based off the current scores of the game, and then made a decision based off of that, but again I wasn't really in a position where I could do that.

When is comes to answering the questions and desires posed in the motivation and AI techniques section, everything was answered except for seeing how a heuristic based player would play against a probability based player (Max-N vs Monto-Carlo algorithm in the same game). Max-N works well for N players (2 or 4 in this case, since its crazy enough with 4 players, but I've tested it with other values and it works all the same) and gives a nice view of the different strategies occurring with different gameboards. With 1 pocket, the focus is on trying to one-up the opponents in terms of score, and then when a player gets in a position where they are alone and it doesn't make sense for another opponent to play around that piece, they swap their focus to trying to own a line directly across from that lone piece. With 4 pockets, there seems to be less of a desire to quickly meet the other pockets, but rather branch away from them, and away form the middle to start. When moves seem useless on the big pocket, they AI will move to a smaller pocket and play there. Sometimes due to the nature of the heuristic, a player will dominate one pocket and end up with a stalemate there until the bigger pocket can join up with it.

It's also worth mentioning that at some point the pockets become irrelevant due to the size of the game. when the grid is of size 32x32, for example, there will be a total of 1024 squares on the board. The effect of a 2x2 pocket in later stages of the is so minimal that it usually gets taken over within about 5 moves. However, with something smaller such as a 16x16 grid, it allows for a quick expansion in the direction away from the big group of game pieces.

Enhancements:

Some possible enhancements would be adding multiple heuristics that focus on different plans of attack to see how the overall outcome on the board looks, or even changing the algorithm used. Right now I'm using a greedy algorithm and relatively greedy heuristic to play the game. Having the options of using less greedy heuristics that focus on capturing bigger sections at a time, or using algorithms that try to minimize the winners score may give an interesting outcome on the game board.

Another enhancement would obviously be implementing the Monte-Carlo algorithm as expected, and not just a random algorithm to play the game with random moves. I could help to try to implement it with the UCT value as well.

References:

Prof. Oommen Lecture Slides

Sharma, Sagar. "Monte Carlo Tree Search." *Medium*, Towards Data Science, 19 Apr. 2019, https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa

Appendix:

To run this code, you will need python 3.x installed on your computer. Navigate to the “Project” file using the command line, and run the code with the following line: “python3 project.py”.

Follow the instruction on the screen to select how the grid will be set up (one “central” pocket or 4 “pockets”, 2 players or 4 players). Let the program run, and then analyze the results.