Oscar Rojas

Kiera Crist

Othello Phase I

Professor Kautz

CSC-242

The Othello class derives its functionality from the State and Move class. This class is provides the input and output to the GUI interface and determines whether the program should play as black or white. The State class describes the state of a particular board configuration. Specifically this class contains a 2D array that represents the board, a list of legal moves, and a list of children that are generated from the list of legal moves. The 2D array uses the integers -1, 0, and 1, to represent white discs, empty spaces, and black discs, respectively. This class also implements the Alpha-Beta search algorithm which uses the ArrayList of children to access nodes until a prune occurs. The Move class serves to represent a location on the board, it's used to create an instance of a move and stores the new board configuration.

The data structures used in the Othello program comprise of ArrayLists and an implicit implementation of a stack . ArrayLists were used because of their ability to expand as needed as well as their ability to store a predefined set of objects. We used ArrayLists to store new State objects and also to maintain a list of legal moves within a particular State. We had two options in designing the Alpha-Beta search algorithm; one option was to use a stack that would hold State objects and could explicitly generate children in depth-first order. The second option was to write a recursive algorithm that would implicitly use a stack, through recursive method calls, to generate children as long as a prune was not possible;we chose to implement the latter.

The design of the Othello program directly mirrors the structure of a directed graph. The ArrayList that contains State objects is an analogue to an adjacency list of a vertex. It is a directed

graph because you can never return to a previous state since you cannot remove discs from the current board configuration. This design allows an easy implementation of Alpha-Beta search because it is so similar to Dijkstra's algorithm in the sense that it also uses adjacency lists. Another design decision was to make a location on the board an object so that we could store the resulting board configuration after applying a move to a particular state. This decision was deemed appropriate because it provides an easy way to match the opponents move to the current board configuration without having to decipher what effect a particular move has on the board. The most prominent reason to write the Othello program in Java is that our design is an object oriented design. Another reason for using Java is that it flushes the output stream automatically. In addition, Java guarantees portability between machines through the JVM.

Testing the Othello program's functionality was done incrementally. Prior to testing the program with the provided GUI interface, we first tested the program against other Othello programs found on the internet, entering the opponents move manually. Next we wrote the bash script and began testing against the KautzPlayer program. This approach only permits two different tests because the outcome of the game will always be the same. Similarly, testing our program against itself only yielded two possible outcomes. To get past this bottleneck we further tested our program by changing the depth limit on the Alpha-Beta search with the intent to generate different moves.

We also tested our Alpha-Beta search algorithm for correctness. This was achieved by writing a MiniMax search algorithm and comparing the results to those obtained from Alpha-Beta. Using the fact that MiniMax and Alpha-Beta search both return the same move we can prove that the Alpha-Beta search algorithm prunes correctly and returns the optimal move. Moreover, when testing the Alpha-Beta algorithm we used print statements to observe the values of alpha, beta, and value of each State object. .

Arguably the most important method of the State class is the utility function. This function is used to assign a value to a terminal State. We formulated our utility function from an analysis on

Othello heuristics published by Vaishanavi Sannidhanam and Muthukaruppan Annamalai from the University of Washington. The authors conducted numerous experiments on the implementation of a dynamic utility function and a utility function that uses a board with static weights on each position. If the weights are set correctly the static board can be used to favor states with stable discs, such as corners, and avoid states with lots of unstable discs. They found that unless the board is implemented dynamically, such that it is updated as the game progresses it performs worse than a utility function that is implemented dynamically based on several features of the board.

The utility function is a mathematical model of a particular State object. It accounts for the relative difference of the mobility – number of legal moves - of MIN and MAX and the stability – how likely a disc is to be flipped - of the discs on the board. In addition, it accounts for the disc parity, relative difference of discs positioned on corners, and squares adjacent to corners. The relative difference of the mobility is a representation of how good a board configuration is for MIN or MAX. This is an important feature of the game because the utility function will maximize moves for the program and minimize moves for the opponent. Similarly, the stability of a disc is a good performance measure because the program will favor states that contain stable discs and reject states that contain discs that are prone to being flanked or flipped. By extension, it will lead to states with more stable discs. A more complex stability function would classify discs on being stable, semi-stable, and unstable. However, the more complex the stability function is the more time that is spent to calculate each move.

Squares that are adjacent to corners carry a very big negative weight for MAX and a very big positive weight for MIN because these locations are the last place one would want to place a disc. Given these heuristics, one can argue that the best strategy to winning Othello is to capture a corner and squares around this particular corner. In summary, one can derive that the number of discs for a given player is irrelevant to determining the utility of a terminal state since such a value only matters in the final state of the game.

In the second phase of the Othello project we ran several trials to construct a mapping from time limits to depth limits. To further convince ourselves of the validity of our mapping from time to depth limit we tested these depth limits on two different machines. We also tried running several applications simultaneously to see if the mapping was still reliable.

To find a solution to this problem we had to choose between an iterative deepening search and a direct mapping from time limits to depth limits. Our first approach was to use an iterative deepening solution; we tried to implement a class that would trigger an event when the time limit was reached. We faced several issues with the GUI and could not confidently determine whether the problems were in our implementation or the GUI's. Instead we implemented functionality for a time limit through a brute force approach. We constructed a mapping from time limits to depth limits. The principal trade off between the two solutions is that an iterative deepening solution will work for any time limit while a static function will only work for predetermined depth limits. In addition, a static function solution also sacrifices search time depending on the time limit in order to guarantee functionality. For instance, for a time limit of 4 seconds our program uses a depth limit of 8 and sacrifices about 1.5 seconds because a depth limit of 9 takes more than 4 seconds.

To implement our solution we made minor modifications to the Othello and State class. In the Othello class there is a switch statement that maps the TIMELIMIT1 to a depth limit. A small modification was made to the State class so that it passes the depth limit along during the duration of the game. We decided to use a switch statement instead of if-else statements because switch statements are usually implemented similar to a hash tables by the compiler and thus increases performance. We use a default of depth limit 4 if the program fails to map the TIMELIMIT1 to a corresponding depth limit.

Link to *An Analysis of Heuristics in Othello*:

http://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf