**Project II : Checkers AI**

**Alec Farfan**

**CSC – 17A &**

**CSC – 7**

**06/09/14**

**Introduction**

When I first started programming back in CSC 5 last fall, I had no particular interest within computer science. I had just changed my major from business because, despite finding interest in it, I could not get excited enough to study it extensively outside of the classroom. So in short, when I came to computer science I was simply looking for something to do in school period. After a couple months of programming I realized I liked it and it was what I would like to do in school and then for a living. Now after programming for a little while longer something I’ve come to take an interest in within computer science is the field of artificial intelligence. The main thing that drew me to computer science in the first place was that I just didn’t understand how it was even possible that we could make computers do some of the things they do. So naturally from that I mostly became interested in making computers do cool things, and now I believe artificial intelligence does the coolest stuff so that’s what I want to do. That’s in part what led me to this project, a classic one in AI.

My previous AI project connect four in CSC 5 was a really cool first project and I was happy with it, but it was exactly that a really cool *first* project. I know we’re always aloud to reuse a project from a previous class but I wanted to make some progress. In general it’s a good idea to do a new project because someday as programmers we’ll have to learn how to produce things regularly, also if I want to be able to work in AI particularly, I’ll have to program stupidly complex things, so I have to keep on solving harder and harder problems. Checkers being much more technically sophisticated then connect four makes for the requirement of more sophisticated programming techniques. The techniques I employed to approach this AI are some of the classics in AI. I followed the basic Minimax algorithm which requires generating the game tree and giving scores to the utility of each state. Finally for each legal move available from one position, the one with the highest scored sequence of moves (branch in the tree) is chosen as the computers choice for that turn. Read on to see the details and enjoy

**Game Description**

Checkers, sometimes referred to as Draughts, is a two person strategy board game where players compete against each other by moving along a diagonal axis, capturing each other’s game pieces by jumping over them. The object of the game is to capture all the opponents’ game pieces or block them from being able to make any valid moves. The game is most often played on an 8 x 8 grid with alternating white and black squares painted across each row. The game checkers is what is called a solved game, which means that any particular game's outcome can be predicted from any position at any time, granted both players were to play from that point on with optimum strategy. Strong Checkers AIs have existed for decades now, Checkers AIs like Chinook can play better than the top ranked human players. This particular AI recognized as the checkers world champion taught itself its own evaluation function by playing itself thousands of times learning along the way.

**Quick Checklist**

**CSC – 17A:**

|  |  |
| --- | --- |
| Chapter 9: Pointers | File Name: Queue.h, GameTree.h  Location: All throughout, dynamic array in  GameTree class definition |
| Chapter 10: String manipulation | File Name: Board.cpp  Location: getChoice function |
| Chapter 11: Structures | File Name: Queue.h  Location: class definition |
| Chapter 12: File I/O | File Name: main.cpp  Location: rules function |
| Chapter 13: Intro to Classes | File Name: Any file but main  Location: The folder I turned the assignment in |
| Chapter 14: More classes (operator overloading) | File Name: State.cpp  Location: operator= function |
| Chapter 15: Polymorphism, Inheritance, and Pure Virtual Functions | File Names: GamePiece.h/cpp,RegBlack.h/cpp,  RegWhite.h/cpp,King.h/cpp  Board.h,Board.cpp  State.h,State.cpp,  Location: Abstract base class--------------------->  --------------->GamePiece->RegBlack (inherits to)  GamePiece->RegWhite (inherits to)  GamePiece->King (inherits to)  Polymorphism in Board and State  get choice function. |
| Chapter 16: Templates | File Name: Queue.h  Location: Whole file is a template |
| Line Count: | 2000+ |

**CSC - 7**

|  |  |
| --- | --- |
| Chapter 5: Recursion | File Name: GameTree.cpp  Location: insert and minimax value functions |
| Chapter 9: Counting and Probability | File Name: GameTree.cpp, State.cpp  Location: GameTree constructor calling State  constructor to create probability tree |
| Chapter 10: Trees | File Name: GameTree.cpp, State.cpp  Location: GameTree constructor calling State  constructor to create probability tree |
| Artificial Intelligence | File Name: GameTree.h/cpp, State. h/cpp  Location: Entirety of the two classes are devoted  to the computer agent |

**:**

**Class Structure**

This program contains eight different classes. The Board class represents a regular checkerboard and is the base class of the State class which represents a game state. The GamePiece abstract base class contains both virtual and regular functions which are inherited to the RegBlack, RegWhite, and King classes which all represent their respective checker pieces. The GameTree class is a self-designed and implemented tree data structure specifically for generating the game tree. The queue class is a very simple implementation of a queue data structure.

* **Board**
* **GamePiece**
* **RegBlack**
* **RegWhite**
* **King**
* **State**
* **GameTree**
* **Queue**

**Board:** The Board class represents the physical checkerboard that a game of checkers is usually played on. To mimic a checkerboard's black and white squares, plus + and minus - characters are printed to the board. The classes representing black checkers, white checkers, and king checkers are friendswith this class. The board is changed by these objects by the charactersheld within them, an object doesn't actually need to be placed on the board. The Board class is responsible for all the tasks involving searching the board as well as making changes to it.

**GamePiece:** The GamePiece class is an abstract base class representing a general game piece to be used in a game of checkers. The class contains five virtual functions for selecting and validating choices of: game squares for initial selecting of a piece, and moving the piece to the new square. Three classes are derived from GamePiece; RegBlack, RegWhite, and King. Each of these three classes require their own unique pointer arithmetic in order to give them the proper functionality with respect to their place in the game. Two non-virtual methods are included also for checking for a double jump and swapping characters.

**RegBlack:** The RegBlack class represents the regular black checker which has not been kinged. Since black checkers start at the top of the board where the index values start (element 0 ), regular black checkers can only move in plus + directions. The ascii character 149 which appears as a black bullet is used to represent a black checker piece. The RegBlack class contains all the necessary functions to evaluate moves across the board for black checkers. As with all of the checker piece objects, only one needs to be instantiate for the game, which acts as a key to graining access to the board.

**RegWhite:** The RegWhite class represents the regular white checker which has not been kinged. Since white checkers start at the bottom of the board where the index values approach their max regular black checkers can only move in minus - directions. An uppercase 'O' character is used to represent a white checker piece. The RegWhite class contains all the necessary functions to evaluate moves across the board for white checkers. As with all of the checker piece objects, only one needs to be instantiated for the game, which acts as a key to gaining access to the game board.

**King:** The King class represents pieces of the checkerboard which have been 'kinged'

Player one (White) has kings that appear as the character 'M' and player two (Black) has kings that appear as the character 'W'. Kings can move backwards and forward on the board, therefore the pointer arithmetic is the same for both the white and black kings. The King class has its own function for validating it's move, but shares the valPick move with whatever regular checker class is calling it. Aside from that one function, the King class contains all of the functions responsible for moving the king around the board.

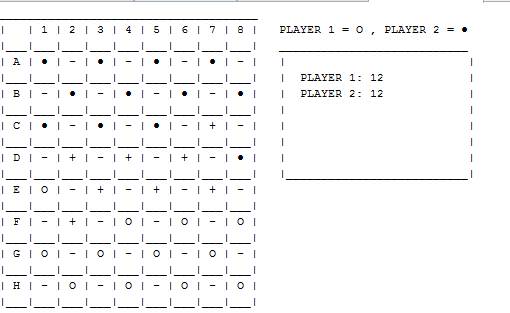
**State:** The State class represents the state of the game at a given position. This classes primary purpose is for use in the AI's decision. The GameTree class creates a tree by filling it up with State objects representing all the possible legal moves for each player starting from a given position, up to a certain depth limit. The state class inherits from the Board class and uses its copy constructor in the derived constructor call and then makes the appropriate changes for each legal move. Then the utility function is applied to give the instance being declared a numeric score to be used in the minimax decision.

**GameTree:** The GameTree class is a tree data structure used to hold all the possible moves from a given position up to a given depth. The tree is filled with State objects which contain the current state of the game board as well as numeric scores based on the utility of that position. This class applies the AI's minimax algorithm by searching the tree for the best move by finding the sequence of moves which leads to the highest utility. The minimax algorithm assumes the player opposing the computer is going to play their optimal move. The depth limit of the search is set at six, effectively looking into the future three moves each for both players.

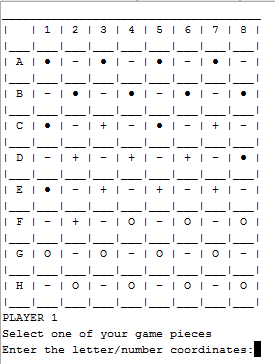
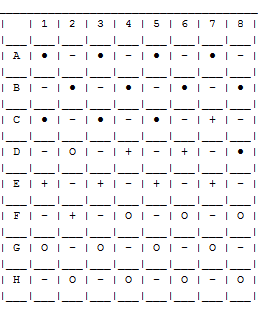
**Queue:** The Queue class is an implementation of a queue data structure. A queue stores and retrieves its elements in a first in first out manner (FIFO). This implementation is dynamic, so the size of the data structure may fluctuate during the program if necessary.

**AI Algorithm**

The algorithm used by the AI agent in this program is called the minimax algorithm. Minimax is a technique that also appears in other fields such as economics. The idea is to maximize your gains on the assumption that your adversaries are going to do their best to minimize your gains. The way this works in terms of computer logic for the game of checkers all starts with the game tree. The game tree starts with an initial board state and then creates all possible game states that can be legally derived from that position according to the rules of the game. As this process occurs an evaluation or utility function is applied to each game state that is produced in the tree, giving it an objective numeric score. After the tree is generated up to a given depth and the scores have been applied, the agent looks at the available moves and finds the branch with the highest total sum from each node representing a legal move. Then the one of these nodes which has the branch with the highest sum is used as the next move. This technique essentially lets the AI look into the future for a couple of turns. This general technique is the basis of all checkers and chess bots (In AI terms, things become different with machine learning from what I hear). The program really resembles intelligence in some basic ways. For example the tree data structure plays the part like of a neuron in the brain, it gives the program the structural basis needed to organize, search, and retrieve information. The utility function applying the score is something like an artificial consciousness, because despite having near perfect information of the game, the agent is only as good as it’s able to make sense of the information. Pictured below are some examples of how the agent in this AI reasons.



In the this frame you can see the Ai agent (black checker) took the best avaiable move by selecting the square in the outside collumn least suceptible to being jumped by and opposing players checke piece



Here we can see the Ai properly responds to a chance to jump over an opponent’s piece (positioned in d2). Once again the AI choses the safer of two possible jumps and takes the outside collumn in the coordinate e1 instead of a column closer to the middle e3.

**Conclusion**

The area in which my agent is still very much lacking is the utility function. I spent most of my time working on getting the tree generated correctly (very frustrating) and so I was only able to program some basic functionality into the utility function. Right now I would say that it’s about as intelligent as an average person you just recently taught the rules of checkers to. The heuristic kind of stuff like in the utility function is actually the thing I enjoy about artificial intelligence the most. The critical thinking required to figure out not only “how to” build this but “how should” this be built. I really enjoy artificial intelligence so this is a program I’ll continue to improve in the coming future. More important than this project though is that I’m starting to gather information and form a plan for actually perusing artificial intelligence. From what I’ve seen from comments from students and professionals on websites like Quora, to actually work in artificial intelligence you need a graduate degree. I used to think it was naïve of people to say they wanted to get a masters or phd at this point, but after studying a technical subject like computer science I realized if you even want that option you need to start thinking about that now because those skill sets don’t just build themselves in a matter of a couple months. The one thing this discrete mathematics class made me realize is that my skills in mathematics are severely lacking (for what I want to do) and as I am now I have no chance. Despite this it’s my current weakness in math that’s providing me the biggest advantage in succeeding because now; I know what I want to do, I know what I need to do, and I have the time to do it. It’ll be at least a year and a half before I apply for transfer, so that gives me a good time to focus on building the skill sets I need for artificial intelligence. Obviously I put in some hard work to get here, but it wasn’t consistent work. Had I worked as hard as I did to learn the prerequistes for starting this project and the project itself the entire semester, I feel like I would be capable of some pretty cool things right now. So now I’m focusing on putting in hard consistent work to get to my goal. I’m going to continue my studies of discrete mathematics and put a large emphasize on math in general, as well as starting to focus on ACM competition questions. Anyways I hope you enjoyed and look forward to the next one!