Lines of Action C++

Solo or Team?: Solo
Personal or School?: School
Approximate date: February 2024

Rules of the Game:

Lines of action is a board game whereby each player takes turns moving a piece of their color either horizontally, or vertically, or diagonally. You can move your piece the number of squares equal to the number of pieces in that row/column/diagonal. The object of the game is to connect all of your pieces in a single group, and block your opponent from doing so.

Description:

For my Organization of Programming Languages course, I was asked to create a terminal based version of Lines of Action in C++. There were strict coding standards and requirements, namely that the game utilize many advanced C++ topics (polymorphism, inheritance, hierarchical design, data encapsulation, etc.). I was required to create a fully functioning game, with home menu options, previous game saving and loading, an intelligent computer opponent, and an assist mode that would tell users all available moves and the best move in their position.

Below is the feature log for the game, detailing the data structures and algorithms involved in the game's creation, as well as my general progression in the coding process.

Data Structures

- **Board Class:** Manages the game board, including initializing the board, updating the board with player moves, checking the validity of moves, counting pieces, displaying the board, and providing functions for game state evaluation.
    - char board[BOARD_SIZE][BOARD_SIZE];
        - Description: This is the array of board elements that will be used to represent the game board. It is an 8x8 array of characters (W, B, .)

- **Player Class:** Represents a player in the game, responsible for storing player-specific attributes such as color and number of wins, translating user input coordinates, and determining if the player is the winner based on the game board's state. This class is the parent class for the Computer and Human classes, and its attributes (color and wins), functions, and virtual function playerAction() exhibit polymorphic behavior.
    - pair<int, int> startPos = make_pair(-1, -1);
        - Description: This pair of integers represents a position on the game board. The first integer indicates the row index, and the second integer indicates the column index. It's initially set to (-1, -1) to signify that no valid starting position has been found yet.
    - vector<vector<bool>> visited(board.getBoardSize(), vector<bool>(board.getBoardSize(), false));
        - This 2D vector of booleans keeps track of visited positions on the game board. Each element in the vector corresponds to a position on the board, and true indicates that the position has been visited, while false indicates it hasn't. It's initialized with false values for all positions. It is used in the code to ensure that spaces aren’t visited multiple times in the DFS loop for calculating connected pieces.
    - stack<pair<int, int>> stack;

        - This stack holds positions on the game board that need to be visited during the DFS algorithm. Each position is represented by a pair of integers: the row index and the column index. The stack holds these positions, and the most recently found connected position is placed on the top of the stack, ensuring that the search goes as far down the chain of pieces as possible before backtracking.

- **Computer Class:** The Computer class manages the behavior of the computer player in the game, including generating possible moves, evaluating and selecting the best move based on certain criteria, executing moves on the game board, providing move recommendations, and translating move indices into human-readable notation. It is inherited from the Player Class.
    - vector<vector<int>> possibleMoves;
        - Description: This 2D vector stores all the possible legal moves that the computer player can make on the game board. Each inner vector represents a single move and contains four integers: the starting row index, starting column index, ending row index, and ending column index of the move. The outer vector holds multiple inner vectors, each representing a different possible move.

- **Human Class:** Represents a human player in the game, responsible for making moves on the game board by interacting with the user interface. It inherits attributes and functionalities from the Player class such as player action execution.

- **Round Class:** Manages the state of a round in the game, including whether the round is over or not, with methods to check and set the game over state.

- **Tournament Class**: Manages the flow of a tournament between two players (Human and Computer), including starting the tournament, playing rounds, and announcing the winner.

Feature & Time log

- Designed the base layout for classes, edited the task JSON of the VS Code project to compile multiple files, and the project compiles on VS Code
- Created stdafx.h precompiled header to hold include files

Time: 2 hours Date: 2/2/24

- Defined board as 2D char array, defined Board::InitializeBoard(), displayBoard(), and updateBoard() functions

Time: 1 hour Date: 2/3/24

- Defined Player class and created child class Human for inheritance
- Defined executeMove() function in Human class

Time: 2 hours Date: 2/5/24

- Added inputTranslator() function so moves can be converted from strings to numeric values to pass to board functions
- Added functionality to game where moves can be inputted and board updates accordingly

Time: 2 hours Date: 2/6/24

- Defined the utility functions for checking if a move is valid, and move validation itself
- Fixed out of bounds error with diagonal move code by using getPieces function instead of direct board array access (was reading garbage values as pieces, instead of as empty squares)

Time: 2 hours Date: 2/9/24

- Outlined Computer and Player classes by declaring functions to be defined later

Time: 0.5 hours Date: 2/11/24

- Added isWinner() function in Player class using DFS algorithm to determine winning player

- Edited main game loop to allow back and forth move inputs and corresponding displays
- Moves displayed were completely validated from the starting position (all legal moves from new game position)
- Game ends if all player's pieces are connected

Time: 2 hours Date: 2/13/24

- Implemented loading to and from files within board class
- Can successfully load board from serialization files

Time: 1 hour Date: 2/15/24

- Game options implemented in human class to allow for multiple actions
- Edited inheritance from player class from executeMove() to playerAction() to better manage game flow

Time: 0.5 hours Date: 2/16/24

- Fully implemented Rounds class, holds gameOver status
- Edited main function to use Tournament objects for game flow, tournament class manages game loop
- Tournament class announces winner, and asks user to play again
- Fixed error where tournament was giving incorrect rounds won, initialized wins to 0 in player class

Time: 2 hour Date: 2/17/24

- Main function handles coin toss
- implemented menu in main function to determine user action (new game, import game, exit)
- validates user input on main menu and coin toss
- Fixed error where Human and Player objects weren't declared by including their declaration and use in each block of the if-else statement

Time: 2 hours Date: 2/17/24

- Implemented functions on computer class: generateMoves(), selectBestMove(), evaluateMoves()
- Bug with selectBestMove(), only looks for connectivity, so will only stay on edges of the board and connect with own pieces
- Created makeRandomMove() function for testing back and forth between human and computer
- Edited main function to be able to handle Computer and Human moves
- Computer toggles between best move and random move

Time: 2 hours Date: 2/17/24

- Fixed bug where computer only moves the on side of the board with calculateDistanceToCenterScore() function
- Computer now makes moves based on the formula 2*connectedPieces + distanceFromCenterScore, where a high center score means closer to the center

- Computer now makes optimal moves in select situations (i.e. Serialization Case 3)
Time: 2 hours Date: 2/17/24
- Implemented help function for computer class to give a list of every possible move for the human player, and select the best possible one
- Implemented reverseTranslateMove() function so that computer can output human-readable
- Computer now gives the player winning move in select situations (i.e. Serialization Case 4)
Time: 2 hours Date: 2/17/24
- Created two game loops to handle imported file game vs new game
Time: 1 hour Date: 2/18/24
- Checked over every instance of user input to ensure input validation was secure. Improved input validation in imported game loop by including cin.clear() and cin.ignore
- Added validation to save game option in Human Class to verify no spaces, includes .txt extension
Time: 2 hours Date: 2/18/24
- Documentation on all functions in: Board, round, and Tournament classes
Time: 2 hours Date: 2/18/24
- Documentation on all functions in: main(), Player, Computer, and Human classes
Time: 2 hours Date: 2/18/24
Total Time:: 30 hours