Othello- Project Report

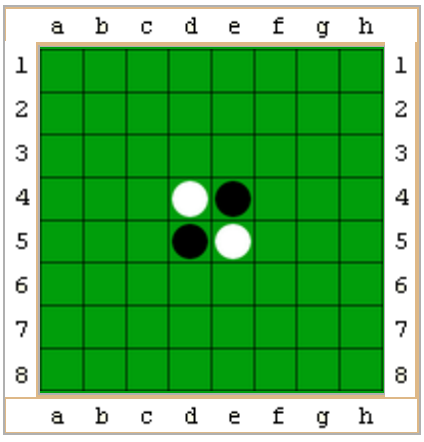
-**Jatin Sadhwani**
-Anurag Obulampalli

Introduction and Game Description

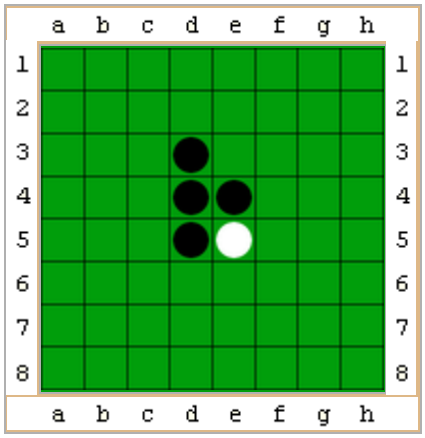
We decided to develop the Othello game for Project 1. It is strategy board game for two players played on 8x8 checkered board. Players place the discs with their assigned color on the board when their turn comes. There are two colors back and white. A player wins when the count of his colored discs is maximum on the board and there are no further legal moves possible for the opponent. The board will have two black and two white disks placed in the center of the of the board.

Rules of the game:

The initial board looks like the one below. Where black color corresponds to a let's say Player 1 and white corresponds to Player 2. Black makes the first move always. A legal move is the one where there exists at least one straight, vertical or horizontal line between two same colored discs and opponents disc in between. So, the legal moves for Player 1 would be d3, c4, e6 or f5. Once Player 1 makes a move the discs of the Player 2 between the newly placed disc and any anchoring disc flips to opposite color meaning they flip to black there by it belongs to Player 1.



Initial State



After Player 1 places a disk at d3

Player Winning Rule:

Players take turns alternatively, and if a player does not have a legal move possible he passes the turn to the opponent. If there are no legal moves possible for both the players the game ends and the player with maximum discs on board wins.

UI Design

Index Page -
The index page will display the title of the game, and a text box for the user to enter the name of the game and join that game. When the user enters the game name and hits join button, they will join that game and will be redirected to the corresponding page.

The index page will also show a list of live games that are currently being played by different users. A game will be displayed in the live games section only when it has at least on active player in the game. Players can join the game they want to from the live games list if their game appears in that list.

Game Page –
The game page displays the game board with the initial tile setup. The game asks the user to enter their name to join. Once the user enters their name, they will join the game as player 1. This process is repeated is for player 2. The game can start only when both the players join the game. When both players join the game, all consequent players will join the game as spectators and won't be able to play it. Once a player joins (Player 1 and 2), the option to enter their name will disappear.

A score card is also displayed on the right of the game board showing the name and score of both the players and an alert message showing who's turn it is to play. The score card is updated when each player takes their turn. Each participating player is also given an option to quit the game. When a player chooses that option, the other player is automatically declared as winner. When a player wins the game, an alert window message is generated on the browser of all the joined players indicating who won.

UI to Server Protocol

The UI to Server Protocol uses the following concepts to send data from UI to server and vice-versa: -

Channels
"Channels are based on a simple idea - sending and receiving messages. Senders broadcast messages about topics. Receivers subscribe to topics so that they can get those messages. Senders and receivers can switch roles on the same topic at any time."

In our design, each player joined the games channel by entering the name of the game as the topic. Each player then subscribes to that game. All the updates for that game are broadcasted to each player. The state of the game is pushed from the UI to the server using this channel. Once the operation has been performed on the server side, the new state is then received on the UI side this through the channel. This change is also broadcasted to all the subscribers (players) of that topic (game name) on the channel. When the state of a game is pushed to the server to perform a certain operation, a

particular message is associated with that push so that server can identify what updates are to be performed on the state. For example, when a player makes a move, the state is pushed with a message “MadeAMove”. When the server receives the state with this message, it knows that it must change the state of tiles and score according to the game rules.

Sockets

“Phoenix.socket is used as a module for establishing and maintaining the socket state via the Phoenix.socket struct. Once connected to a socket, incoming and outgoing events are routed to channels. The incoming client data is routed to channels via transports. It is the responsibility of the socket to tie transports and channels together.”

First, we established a connection using a socket. Then, once a connection is created, it joins different channels (games in our case) and all the topics of that channel. All the game state is assigned to the socket and passed through the socket.

To reflect the change of state to all the users we have used the concept of broadcast. That is, once the state of the game is changed (like if a winner is declared or a new tile is clicked) it is reflected to all the players by using the broadcast feature of channels.
For example, when a player clicks on a tile and the state of the game changes, we broadcast this message by passing the message “PlayerMadeAMove” along with the broadcast so that UI can identify the changes in this particular broadcast and reflect the changes accordingly. Another example for this is when the second player joins after the first player, the broadcast is sent with a message “PlayerJoined” and the second player is updated for all the players.

Data Structures on the Server

The data structures on the server end are used to store the state of the game. The state of the game is defined as the information about the game at any given point. That state remains the same for all the users. The following data structures are used in the state of the server:-

Tiles – Stores the state of all the tiles in the game. Each tile contains two attributes, index representing the number of the tile, and value representing that if that tile has been visited or not, and if it is visited, it is visited by which player.

Is_player – Represents which player is currently making a move. It is a Boolean which returns true if player 1 is making a move and returns false if player 2 is making a move.

Player1 – Represents the name entered by player 1. Initially it is set to nil. When the first player enters their name, it is set to that name.

Player2 – Represents the name entered by player 2. Initially it is set to nil. When the second player enters their name, it is set to that name.

Player_count – Returns the total number of players joined in that game at any given moment. Initially it is set to 0. It is incremented each time a new player joins the game.

P1_score – Represents the total number of tiles marked by player 1. In our choice of design, it returns the number of black tiles at any given moment. It is updated each time a player makes a move.

P2_score – Represents the total number of tiles marked by player 2. In our choice of design, it returns the number of white tiles at any given moment. It is updated each time a player makes a move.

Winner – Returns the winner of that game. Initially set to 0. When a player wins, it is set to that player. That is, if player 1 wins, it is set to 1, if player 2 wins it is set to 2.

Pos1 – Represents if player 1 has any possible moves left. Initially it is set to true as initially both the players have legal moves. When there comes a scenario, when player 1 does not have any legal moves left, it is set to false. The chance is then shifted to player 2.

Pos2 – Represents if player 2 has any possible moves left. Initially it is set to true as initially both the players have legal moves. When there comes a scenario, when player 2 does not have any legal moves left, it is set to false. The chance is then shifted to player 1.

Alert_message – It returns any alert messages during the course of the game. For example, if any player joins it returns that a new player has joined the game.

Apart from the state data structures, server also has some additional data structures:-

Id – represents the id of the player. It is passed when a tile clicks a tile to check if that player is authenticated to click it. That is, if the player is a participant or a spectator.

AdjacentTiles – represents all the adjacent tiles of a given tile. That is the tile immediately next to it horizontally, vertically, and diagonally.

Enemies – Represents all the adjacent tiles which need to be flipped to the opposite color when a particular tile is clicked.

These data structures are used during function calls.

Implementation of Game Rules

We have implemented the game rules as follows: -

a. Each player will see an 8 x 8 game board similar to the one mentioned above in the initial state

b. Initially, when two players decide to play a game, one of the player will create a game with a game name.

c. The second player will then join that game by entering that game name. The game cannot start until we have both the players. As soon as the second player joins, the game will start.

d. Once a player joins the game, **they cannot refresh the game** as the data for each player will be reset to null.

e. They can play against each other by clicking on the tile or a square where they would like to place their disc.

f. Each player can only click the tile which has at least one adjacent tile clicked by the other player. For example, if a player wishes to flip a certain tile to black, it should have at least one tile flipped to white.

g. Once a player clicks a tile, it changes all adjacent tiles of other color to their own color till they come across a tile with their own color. For example, according to our implementation, if player 1 clicks on a tile (Player 1 is black) and that tile has adjacent white tiles, all the white tiles in that line are changed to black until the next black tile. If that line has no black tile, no action is taken.

h. With each tile click, the score is updated for both the players. The score returns the number of black and white tiles on the board.

i. When a player does not have a legal move left, the turn is then passed to the opponent on the next click. For example, if player 1 does not have a legal move left and they click any tile, the turn is passed to player 2 with a message saying that they have no legal moves left.

j. If both the players do not have legal moves left, the player with more number of tiles at that moment is declared the winner.

k. All the players who join after the first two players are joined as spectators. That is, they can observe the game but they do not have any power to change the state of the game.

l. Each participating player also has an option of quitting the game at any moment. When a player quits a game, the other player is automatically declared as the winner and that game comes to an end. When the game ends, all the players are redirected to the Welcome page.

m. At the end of the game, when all the tiles are clicked, the player with more number of tiles of their color, is declared as the winner.

Challenges and Solutions

The major challenges faced by our group during this project were:-

a. One of the major challenges that we faced was implementing the logic to determine the legal moves that a player can perform. As we have implemented the logic of transferring the chance to the other player if the current player does not have any legal move left.

   Solution
   We had to dry run with random possible values to solve this problem. That is, we took in factor a number of scenarios in which there might be no legal moves left and how we can deal with such scenarios. We then implemented a method of keeping a check of all the legal moves a player has. If a player has no legal moves left, the check returns false. When that happens, the turn is passed to the opponent. For the turn to be passed to the opponent, the current player has to click on a tile. The turn will not be passed on until the current player makes a move.

b. The second challenge we faced was to reflect the changes in the state to all the players currently joined in the game.

   Solution
   We used the feature of broadcast offered by phoenix channels. Once a change is made, the new state of the game is broadcasted to all the players in a game using a broadcast message. This broadcast message is used to identify the change and the final state after the change in the react file. Once this broadcast message is received, the state changes are then reflected according to the new state.

c. Next challenge we faced was to implement multiplayer logic to our game.

   Solution
   We maintained a separate player id and player name for each player that joined the game. Each time a player joined the game, the player id is set to the total number of players + 1. This id is sent to the server every time a user clicks a tile.

The server then takes an action depending on the id of the tile (whether the player is a participant or a spectator).

References and attribution:

1) https://en.wikipedia.org/wiki/Reversi
2) http://www.othelloonline.org/
3) https://hexdocs.pm/phoenix/channels.html
4) https://hexdocs.pm/phoenix/Phoenix.Socket.html
5) https://stackoverflow.com/questions/9358882/how-to-make-a-circle-around-content-using-css
6)https://i0.wp.com/clipset.20minutos.es/wpcontent/uploads/2016/03/alphago.jpg?w=1600&ssl=1
7) https://piazza.com/class/jbri1u23q9yx6?cid=539
8) https://www.w3schools.com/howto/howto_css_modals.asp
9) https://hexdocs.pm/phoenix/Phoenix.Channel.html#broadcast/3