Othello C++

Angie Tatiana Solano Rodriguez

Universidad de Antioquia

Faculta de Ingeniería

Ingeniería de Telecomunicaciones

Medellín, Antioquia, Colombia

2023

Tabla de contenido

Contextualización del problema

Othello es un juego de tablero que requiere estrategia, en este problema se desarrollara en un tablero de 8x8 con fichas de color blanco y negro, cada una representando un jugador.

El objetivo principal es tener el mayor número de fichas de tú color al finalizar la partida, aparte de ello, el juego se basa en encierros tipo sándwich donde una ficha se posiciona de tal manera que atrapa las fichas del oponente.

Las reglas del juego son sencillas, el juego se hace por turnos, el jugador debe poder hacer el movimiento según las reglas, es decir espacios donde arme una línea ya sea vertical, horizontal o diagonal con otra de si misma y que dentro halla fichas del equipo contrario.

Análisis

Consideraciones para el desarrollo del programa

Modelado de las clases

Se definirán las clases para la representación del juego como tablero, jugadores y las fichas. Estas son como las clases esenciales a la hora del juego.

Representación del tablero

Podemos representar este como un arreglo bidimensional, de igual manera en el desarrollo del código veremos si es la forma más eficiente.

Visualización del tablero

También debemos tener en cuenta que la visualización del tablero debe ser organizada y sencilla para que sea adaptable al momento de mostrarlo en la consola.

Reglas del juego

Se deben implementar de manera que sean eficientes y aparte tener en cuenta el encierro tipo sándwich, ya que es la única forma de hacer una jugada, también considerar los posibles errores al desarrollar la lógica.

Almacenamiento de las partidas

Tener en cuenta que el código debe guardar los registros de cada partida, aparte tener la opción de visualizar el historial, esta información se debe guardar en un archivo, por lo cual utilizaremos un archivo plano con extensión .txt

Estrategia de solución

Teniendo en cuenta el análisis, la estrategia de solución se haría de la siguiente manera:

1. ***Diseño de clases***

Se deben de definir las clases para representar los diferentes partes del juego como son: el tablero, el jugador, entre otras. Aparte de ello identificar los atributos y métodos de cada clase.

2. ***Estructura de datos para Tablero***

Se debe de buscar la mejor manera de representar el tablero, teniendo que se debe de mostrar en consola y debe permitir un acceso fácil a cada posición.

3. ***Lógica del juego***

Debemos de conocer correctamente las reglas y entenderlas para de esta manera desarrollar la lógica de manera correcta, para este tendremos en cuenta la verificación de los movimientos según las reglas y la captura de las fichas del oponente.

4. ***Interfaz o mostrar el juego en consola***

Se debe de analizar, planear y diseñar un algoritmo que permita mostrar de manera adecuada el tablero, permitiendo actualizar cada posición según la interacción de los jugadores.

5. ***Manejo de excepciones***

Al desarrollar el juego se debe validando el código para visualizar situaciones en las que es necesario el manejo de excepciones como en los errores de entrada de datos por parte del usuario al seleccionar la posición en la que quiere poner su ficha.

6. ***Almacenamiento de datos***

Se debe de crear una funcionalidad que permita capturar los datos al finalizar cada juego y guardar este en un archivo plano de texto.

7. ***Historial de partidas***

Debemos tener en cuenta que se debe desarrollar un apartado que permita visualizar el historial de partidas.

8. ***Pruebas y depuración***

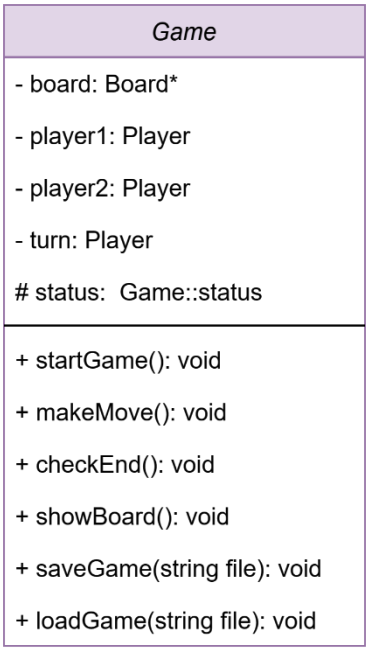
Al finalizar el desarrollo del código se debe de hacer realizar pruebas que nos permita visualizar errores que debamos visualizar, también visualizar el comportamiento del juego y aparte de ello validar que las reglas del juego se cumplan.

9. Documentación

Tener en cuenta que debemos de ir desarrollando el juego de manera interna, de esta manera sabremos en cualquier momento que significa el método o el bloque de código.

Diseño

Clase Game



Especificador de acceso y justificaciones

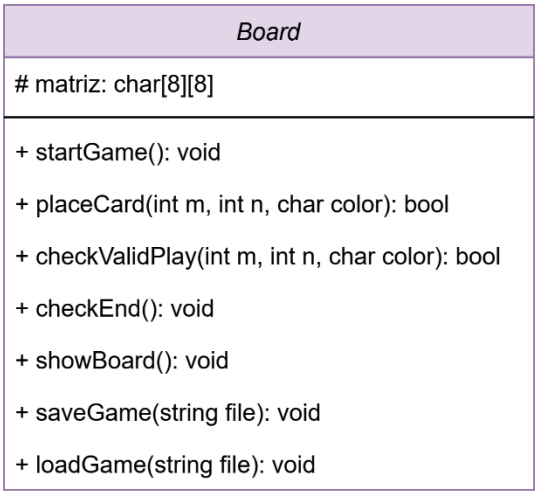
- Privado: los atributos board, player1, player2 y turn son de acceso privado, es decir solo se pueden acceder desde los métodos de la clase Game, esto se decide con el fin de evitar la modificación de los datos en cualquier parte del juego, ya que guarda información importante.
- Protegido: el atributo status es protegido, permitiendo solo el acceso a este desde la clase Game y las clases que hereden esta clase, de esta manera se tiene un mayor acceso a este atributo el cual guardar la información del estado del juego.

- Publico: los métodos son públicos con el fin de tener un acceso fácil a esto, permitiendo que otras clases puedan usar estos, ejemplo: los usuarios pueden usar el método startGame().

Lógica

- El algoritmo del método startGame():
    1. Inicializar el tablero del juego
    2. Inicializar los jugadores
    3. Establecer el turno del primer jugador
- El algoritmo del método makeMove():
    1. Obtiene los datos del jugador actual
    2. Obtiene la jugada del jugador
    3. Coloca la ficha en la posición indicada en el tablero
    4. Actualizar estado del juego

Clase Board



Especificador de acceso y justificaciones

- Protegido: el atributo matriz es protegido para poder controlar y proteger los datos que almacena, este atributo guardar la información respecto al tablero del juego, siendo importante que solo sea modificado desde la clase Board y clases que hereden esta y no en cualquier parte del juego.
- Públicos: todos los métodos de la clase son públicos con el fin de facilitar el acceso a este en las diferentes partes del juego, ejemplo: un usuario puede usar el método placeCard() que permite colocar una ficha en el tablero.

Lógica

- El algoritmo del método placeCard():
    1. Validar si la posición indicada por el usuario está dentro del tablero
    2. Validar si la posición seleccionada este vacía
    3. Validar si el movimiento del usuario encierra fichas del oponente.
    4. Si es válido todo lo anterior, se pone la ficha y se actualiza el estado del juego.
- El algoritmo del método checkEnd():
    1. Validar si el tablero está lleno
    2. Comprobar si un jugador tiene más fichas que otro.
    3. Retornar true si el juego ha terminado
    4. Retornar false si el juego no ha finalizado

Clase Player

Especificador de acceso y justificaciones

- Privados: los atributos name y color son privados con el fin de proteger y evitar la modificación de estos en partes del juego no permitidas.
- Protegido: el atributo score es protegido con el fin de permitirle a las clases que hereden la clase Player puedan acceder a este atributo, ya que este guarda la puntuación del jugador en cada momento del juego.
- Públicos: los métodos de esta clase son públicos con el fin de permitir el acceso fácil a estos, ya que se puede usar el método getNombre() en alguna parte del código para obtener el nombre del jugador, por lo cual al ser publico permite hacer esto de manera ágil.

Lógica

- El algoritmo checkAvailableMoves():
    1. Obtiene la matriz de fichas del tablero
    2. Itera sobre esta matriz
    3. Si la posición está vacía, devuelve true
    4. Devolver false

Algoritmos implementados

Enlace video: https://youtu.be/q8_VYZzKBF0

Enlace repositorio: https://github.com/AntekR/ParcialII.git

Experiencias de desarrollo

Errores

Inicio

Al iniciar, es difícil saber porque punto iniciar, tenia en mente una matriz para el tablero tipo 8x8, pero era necesario que esta no tuviera datos fijos, por lo tanto, al querer definirla en la clase, me indicaba error porque los datos no se habían inicializado entre otras cosas, por lo tanto, busque ayuda y me apoye en diversas paginas y halle una, la cual me dio una solución a esta y aparte entender mejor como seria.

Ref: https://mauricioavilesc.blogspot.com/2015/08/matriz-dinamica-en-c.html

Creación de métodos en momentos de desarrollo

En momentos se crearon algunos métodos en clases para mejorar el código o para hacer ciertas validaciones, pero al final de todo era innecesario ya que podía ser más completo o resultaría mejor haciéndolo en otra clase o desde otro sentido, un ejemplo de ello es el siguiente código.

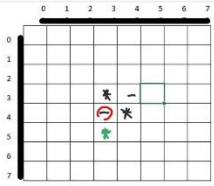
```
bool Player::checkAvailableMoves(Board* boardMatrix, int rowSelection, int columnSelection, char colorPlayer){
    bool available = false;
    char opponent = (colorPlayer == '-') ? '+' : '-';

    // Valida que la posicion seleccionada por el usuario no este fuera del tablero
    if(0<rowSelection-1<boardMatrix->getRow() && 0<columnSelection-1<boardMatrix->getRow()){
        // Valida si hay encierro de manera vertical hacia arriba
        // Pendiente validar la comparacion a 0.
        if(rowSelection-1>0 && boardMatrix->getBoard()[rowSelection-2][columnSelection-1]==opponent){
            available = true;
        }
        // Valida si hay encierro de manera vertical hacia abajo
        if(rowSelection-1<boardMatrix->getRow() && boardMatrix->getBoard()[rowSelection+2][columnSelection-1]==opponent){
            available = true;
        }
        //Valida si hay encierro de manera horizontal hacia la derecha
        if(columnSelection-1>=0 && boardMatrix->getBoard()[rowSelection-1][columnSelection]==opponent){
            available = true;
        }
        //Valida si hay encierro de manera horizontal hacia la izquierda
        if(columnSelection-1<boardMatrix->getRow() && boardMatrix->getBoard()[rowSelection-1][columnSelection-2]==opponent){
            available = true;
        }
    }else{
        available = false;
    }

    return available;
}
```

En la imagen anterior se ve el código del método checkAvailableMoves(), el cual permitía saber si el usuario tenia movimientos disponibles o posiciones en las cuales podía hacer un encierro, este método pertenecía a la clase ***Player***, pero se eliminó y se implementó en la clase ***board,*** ya que había una mayor facilidad en el acceso a la matriz que guardaba los datos del tablero del juego.

Errores en el desarrollo de la validación de encierros tipo sándwich.

Esta parte del código fue la más difícil en lograr que funcionara correctamente, según las pruebas realizadas, ya que muchas veces el encierro funcionaba intercambiando las fichas del oponente, pero omitía poner la ficha de si mismo.

Por lo tanto, en muchas ocasiones se hicieron procesos escritos, mirando o pensando si los condicionales del código estaban correctas, se hizo lo siguiente como validación.

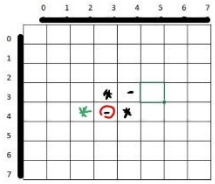


RowBoard = 8.

Vertical hacia arriba

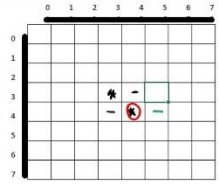
User = 5,3 enemigo = (Ux-1, Uy) = (5-1,3) = (4,3)

• (RowSelection > 0 && RowSelection ≤ RowBoard-1 && boardMatrix[rowSelection-1][ColumnSelection] == opponent) {



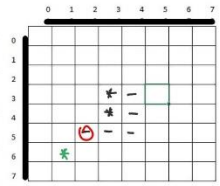
User = 4,2 enemigo: 4,3 → se suma 1 a la column del user.

(ColumnSelect ≥ 0 && columnSelection < RowBoard-1 && boardMatrix[rowSelection][ColumnSelection+1] == opponent



user = 4,5 enemigo: 4,4 → se le resta 1 a la columna.

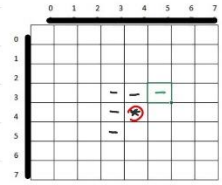
(ColumnSelect ≥ 0 && columnSelection < RowBoard-1 && boardMatrix[rowSelection][ColumnSelection-1] == opponent)



Diagonal superior derecha.

user = 6,1 enemigo = 5,2

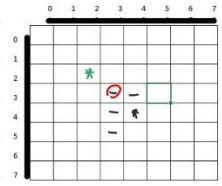
rowSelection ≤ RowBoard-1 && columnSelection < RowBoard-1 && boardMatrix[rowselection-1][columnSelection+1] == op.



Diagonal inferior izquierda

user = 3,5 enemigos = 4,4.

rowSelect < RowBoard-1 && columnSelection > 0 && BoardMatrix[rowSelect+1][columSelect-1] == opponent)



Diagonal inferior derecha

user = 2,2 enemigo = 3,3

rowSelect < RowBoard-1 && columnSelect < RowBoard-1 && board Matrix[row+1][column+1]

Luego de todo esto, se solucionaron la mayoría de los errores en los encierros excepto el diagonal inferior izquierda.

Error de programación en la diagonal inferior izquierda.

Al correr el código o juego, se hace depuración o pruebas del juego, se visualiza que al poner una ficha para hacer un encierro de manera diagonal hacia la parte inferior izquierda el código no

funciona correctamente, ya que este debe de cambiar las fichas del oponente por las del jugador y aparte de esto poner la ficha de sí mismo en la posición ingresada por el jugador, y solo se visualiza el cambio de las fichas de oponente, pero se salta la parte de poner su propia ficha.

```
if(rowSelection>=0 && rowSelection-1<rowBoard-1 && columnSelection-1>=0 &&
boardMatrix[rowSelection][columnSelection-2]==opponent){
       int i= rowSelection;
       int j= columnSelection-2;

       while (i<rowBoard-1 && j>=0 && boardMatrix[i][j]==opponent) {
                i++;
                j--;
      }
      if(boardMatrix[i][j]==colorPlayer){
              for (int x = rowSelection, y = columnSelection-2; x<i &&
                  y>j;  x++,y--) {
                    boardMatrix[x][y]=colorPlayer;
              }
      }
}
```

Al finalizar las revisiones se llegó al siguiente código:

```
if(rowSelection<rowBoard-1 && columnSelection<=rowBoard-1 &&
boardMatrix[rowSelection+1][columnSelection-1]==opponent){
          int i= rowSelection+1; //4+1=5
          int j= columnSelection-1; //6-1=5

while (i<rowBoard && j<=rowBoard-1) {
    if(boardMatrix[i][j]==colorPlayer){
        for (int x = rowSelection, y = columnSelection; x<i && y>j; x++,y--) {
                        boardMatrix[x][y]=colorPlayer;
        }
        break;
    }
    i++;
    j--;
    }
}
```

Con este, se llegó a la solución y el funcionamiento correcto.

Consideraciones y resultados de aprendizajes.

Este proyecto me permitió poner en conocimiento y en practica los diversos temas estudiados, aparte de ello entenderlo de una mejor manera, ya que no hay nada mejor como practicar o desarrollar para entender y aprender.

Si tuvo demasiados conflictos iniciando con el proceso de solución debido a que es difícil saber o dar con el punto de inicio, aparte de ello es importante el manejo de las emociones, ya que al querer solucionar o dar con el desarrollo sin ningún problema, da un alto nivel de estrés y entendí que no es necesario, ya que todo fluye a medida que vas desarrollando la solución.

Y también observe que tenia algunos conocimientos muy bajos, por lo tanto, es un punto que reforzar para mejorar.