Timbiriche

Alumno: Marc Holste Legajo: 56306

Diseño:

En este trabajo, diseñe una interfaz grafica simple, un controlador, jugadores y un tablero. Empieza el programa inicializando la interfaz grafica, el tablero y los jugadores, y una vez finalizado, comienza el juego.

El Controlador:

El controlador es el que controla los turnos de cada jugador y le permite hacer movimientos a los jugadores, ya sean AI o Usuarios. La implementación usada para esto, fue hacer una interfaz Player, que no distingue entre Usuario y AI, y que tenga el método "move" para mover en el tablero. Esto permite que sea muy sencillo hacer un movimiento, ya sea del Usuario o del AI. A su vez, también es escalable, porque se puede agregar más jugadores al juego sin mucha modificación de código.

El Tablero:

El tablero consiste de una matriz de líneas dibujables horizontales y verticales, un Stack de movimientos "Undo" para poder revertir las jugadas previas y un Stack de cuadrados para guardar la cantidad de cuadrados completados por los jugadores y para poder deshacer los cuadrados en caso de undo.

La Interfaz Grafica:

La interfaz grafica es una matriz de puntos con líneas entre ellos, un header que marca el turno del jugador a mover y 3 botones.

- **Undo:** Deshace el último movimiento hecho por el jugador. Se puede hacer undo indefinidamente. En caso de movimiento AI, deshace todas las líneas del último movimiento y en caso del Usuario, deshace solo la última línea seleccionada
- **DOT:** Genera un archivo "Timbiriche.dot" con el árbol explorado por el ultimo movimiento del AI
- **Standings:** Abre una ventana que muestra la cantidad de cuadrados completados por cada jugador hasta ese momento

Generación de jugadas:

El controlador le pide al jugador que le toque, un movimiento que se representa a través de una lista de puntos (líneas). El AI usa el algoritmo minimax y el Usuario selecciona por la interfaz grafica. El tablero recibe el movimiento y pinta las líneas correspondientes, que se ven reflejadas en la interfaz grafica, porque las líneas del tablero tienen una relación uno a uno con la interfaz grafica. El controlador le pregunta al tablero si dicho jugador completo un cuadrado. En caso de completar un cuadrado, el controlador no cambia de turno para que siga jugando el mismo jugador, de lo contrario, cambia el turno para que juegue el siguiente.

Generación del Archivo DOT:

El archivo dot se genera con Writer que escribe en el archivo creado “Timbiriche.dot”. Para la traducción del árbol a formato dot, hice una cola de nodos, donde el nodo crea los atributos de sus hijos, los agrega a la cola y después une todos los nodos hijos consigo mismo. Ver ejemplo de cómo queda el dot en la figura 1. Si fue podado o es terminal, lo pinta de color gris, si fue elegido por el algoritmo lo pinta de color naranja, max es representado con un cuadrado y min es representado con un circulo.

```
digraph Timbiriche {

0 [shape=box, height=0.18, fontsize=12, label="START 0",style=filled,
color=orange];

1 [height=0.18, fontsize=12, label="[(0,0)] 0", style=filled,
color=peachpuff4];

2 [height=0.18, fontsize=12, label="[(1,0)] 0", style=filled,
color=peachpuff4];

3 [height=0.18, fontsize=12, label="[(1,1)] 0", style=filled,
color=peachpuff4];

4 [height=0.18, fontsize=12, label="[(1,2)] 0", style=filled,
color=peachpuff4];

5 [height=0.18, fontsize=12, label="[(2,1)] 0", style=filled,
color=peachpuff4];

6 [height=0.18, fontsize=12, label="[(3,0)] 0", style=filled,
color=peachpuff4];

7 [height=0.18, fontsize=12, label="[(3,1)] 0", style=filled,
color=peachpuff4];


8 [height=0.18, fontsize=12, label="[(4,0)] 0", style=filled,
color=peachpuff4];

9 [height=0.18, fontsize=12, label="[(4,1)] 0",style=filled, color=orange];

0 -> {1,2,3,4,5,6,7,8,9};
```

Figura 1

Algoritmo Minimax:

Cuando el modo es “depth”, el Arbol lo creo en forma DFS hasta profundidad k. Esto la hago en forma recursiva, donde cada hijo crea sus árboles hijos hasta llegar a la profundidad k y para reducir la complejidad espacial, en vez de guardar el tablero en cada estado, solo guardo una lista de las líneas dibujadas para llegar a ese estado desde su padre. Es decir, un estado tiene algunos movimientos posibles, el movimiento elegido para llegar al próximo estado, se guarda en su hijo.

Cuando el modo es “time”, el Arbol lo creo en forma BFS hasta que el tiempo se acabe. La creación es por medio de una cola, donde un nodo crea a sus hijos y los mete al final de la cola. El problema de hacer una cola, es que se pierde la referencia del tablero para cada nodo. Mi idea para no guardar el tablero en cada estado, es reconstruir el tablero en cada llamada de un nodo, donde dibujo todas las líneas previas requeridas para llegar a ese estado, crea sus hijos y después desdibujo todas las líneas. Decidí complejidad espacial sobre temporal, porque la creación del árbol ya requiere de mucho espacio, si se le agrega un tablero a cada nodo, el tamaño requerido sería demasiado grande para un simple juego de Timbiriche. Esto sin embargo baja la performance del AI.

El Arbol es doblemente encadenado, porque al final del algoritmo, se recibe la mejor opción, pero esta no pertenece a los movimientos del estado inicial, por lo que retrocedo hasta llegar a uno de los estados posibles.

La heurística utilizada para el algoritmo es la cantidad de cuadrados que tiene max menos la cantidad de cuadrados que tiene min, con los movimientos hecho hasta ese estado.

Pseudo-codigo del algoritmo sin Poda:

```
function notPrune(Arbol) is
    if Arbol is terminal then
        return Arbol
    if Arbol is max then
        v:= - maxAmountOfSquares
        for each kid of Arbol do
            aux:= notPrune(kid)
            if aux value is greater than v then
                v:= aux value
                found:= aux
            if aux value is equal to v then
                if Random then
                    found:= aux
        return found
    if Arbol is min then
        v:= maxAmountOfSquares
        for each kid of Arbol do
            aux:= notPrune(kid)
            if aux value is smaller than v then
                v:= aux value
                found:= aux
            if aux value is equal to v then
                if Random then
                    found:= aux
        return found
```

Pseudo-codigo del algoritmo con Poda:

```
function Prune(Arbol,alpha,beta) is
    if Arbol is terminal then
        return Arbol
    if Arbol is max then
        max:= -maxAmountOfSquares
        for each kid of Arbol do
            aux:= Prune(kid,alpha,beta)
            if aux value is greater than max then
                max:= aux value
                found:= aux
            alpha:= max(alpha,max)
            if beta is smaller or equal to alpha then
                return found
        return found
    if Arbol is min then
        min:= maxAmountOfSquares
        for each kid of Arbol do
            aux:= Prune(kid,alpha,beta)
        if aux value is smaller than min then
            min:= aux value
            found:= aux
        beta:= min(beta,min)
        if beta is smaller or equal to alpha then
            return found
    return found
```

Observaciones:

Tuve otras ideas de cómo diseñar el tablero, por ejemplo hacer una matriz de cuadrados, sin embargo esta idea no prospero, porque era muy complicado representar una interfaz grafica, ya que los cuadrados comparten líneas con otros cuadrados. Al final elegí la matriz de líneas porque la relación con la interfaz grafica era uno a uno, lo cual lo hizo más sencillo.

En el algoritmo minimax, hay una optimización al algoritmo utilizado pero por falta de conocimiento y práctica no la logre hacer funcionar. Esta es hacer la búsqueda y creación del árbol de movimientos al mismo tiempo, donde la poda no crea los nodos podados, reduciendo complejidad espacial y recorriendo el árbol solo una vez, reduciendo el tiempo necesario para el movimiento AI.

La restricción del tiempo fue un problema que a primera vista no parecía muy complejo, sin embargo se dificulto sobre la marcha. Tal vez la implementación que hice no es la mejor, se podría haber usado otras formas como el IDS, sin embargo esta, no la entendí del todo bien y estaba restringido por cómo cree el árbol en profundidad.

La heurística elegida me pareció simple y funcional. Si hay ideas más sofisticadas, no se me ocurrieron y tampoco me pareció que necesitaba una mejor.

Conclusion:

En comparación de los algoritmos, el minimax toma mejor decisiones cuando se analiza en profundidad, en vez de cuando es restringido por el tiempo. Esto se debe a la generación del árbol, que performa mejor en DFS que en BFS.