OTHELLO

Descripción de estructuras de datos y algoritmos

Clase Pair

- ***Nombre de la clase:*** Pair
- ***Breve descripción de la clase:*** Utilizada para gestionar las coordenadas de la matriz (posiciones en el tablero).
- ***Descripción de los atributos:***

  - First: Coordenada x.

  - Second: Coordenada y.
- ***Descripción de funcionalidades principales:***

  Sum(Pair p)/Sub(Pair p): Permite sumar/restar una coordenada con otra.

Enum Casilla

- ***Nombre:*** Casilla.
- ***Breve descripción de la clase:*** La matriz del tablero está formada por estos. Simulan el estado de una casilla (ocupada por blanco, negro, o sin ocupar).
- ***Descripción funcionalidades principales:***

  this.contrary(): devuelve el color contrario al de la instancia: Negro -> Blanco y viceversa.

Clase Node

- ***Nombre de la clase:*** Node
- ***Breve descripción de la clase:*** Clase usada para almacenar la información necesaria para el algoritmo de la IA.
- ***Descripción de los atributos:***

-Pair Coord: La coordenada de la jugada que se va a evaluar

-Casilla Player: Color de la jugada que se evalúa.

-Score: Puntuación asociada a la jugada mediante una heurística.

Clase Tree

- ***Nombre de la clase:*** Tree<T>
- ***Breve descripción de la clase:*** Clase genérica que permite almacenar objetos de clase T en su interior en formato de árbol N-ario.
- ***Descripción de los atributos:***
  - root: Objeto de clase T almacenado. Representa el valor del nodo.
  - leafs: Lista de Tree's. Representa los nodos hijos (adyacentes a this).
  - parent: Tree que representa el nodo padre del mismo (si es la raíz, éste será null).
  - locate: Mapa de hash que permite el acceso rápido a los nodos.

- ***Descripción de funcionalidades principales:***

  - Collection<Tree> getSubTrees(): devuelve el atributo “leafs” para permitir navegar por la estructura.

  - static <T> Collection<T> getSuccessors(T of, Collection<Tree<T>> in):

    Busca y devuelve los objetos de clase T en los hijos del nodo.

Comprobar legalidad de jugadas

Para comprobar que las jugadas se atienen a las normas de la partida, hemos de comprobar que, al colocar una ficha, ésta “encierre” al menos una ficha enemiga. Con encierre me refiero a que dicha ficha enemiga quede entre dos negras en alguna dirección recta.

Para comprobar esto recorremos el tablero desde la posición en la que se quiere realizar la jugada, y se comprueba que haya alguna adyacente del color contrario.

Si la condición contraria se cumple, hay que comprobar que la siguiente posición después de salir de la serie de fichas del color contrario sea del mismo color que la jugada.

Representación:

- - - J O O O X
- - - - - - - -

J sería una jugada correcta si J es de “color X”.

- - - J O O O -
- - - - - - - -

En cambio, J no sería adecuada aquí, pues aunque encuentra una ficha del color contrario, al final de esta hay una posición vacía.

```
protected boolean is_legal(Pair p, Casilla c) {
        if (matrix[p.first()][p.second()] != Casilla.VACIA) {
            return false;
        }
        for (Pair dir : directions) {
            Pair start = p.sum(dir);
            if (inBounds(start) && matrix[start.first()][start.second()] == c.contrary()) {
                while (inBounds(start)) {
                    if (matrix[start.first()][start.second()] == c) {
                                        return true;
                    }
                    if (matrix[start.first()][start.second()] == Casilla.VACIA) {
                        break;
                    }
                    start = start.sum(dir);
                }
            }
        }
        return false;
    }
```

Buscar jugadas legales

Para buscar jugadas legales para un color, el algoritmo consiste en el siguiente concepto:

La base de la legalidad de una jugada reside en que la ficha que se quiera poner sea adyacente a una del color contrario, así que sólo hay que buscar las posiciones adyacentes al color contrario a la jugada que estén vacías.
Una vez se tienen las posiciones vacías hay que descartar las ilegales.

```
public ArrayList<Pair> getLegalMoves(Casilla c) {
        ArrayList<Pair> positions;
        if (c == Casilla.BLANCA) {
            positions = (ArrayList<Pair>) negras.clone();
        } else {
            positions = (ArrayList<Pair>) blancas.clone();
        }
        ArrayList<Pair> Legals = new ArrayList<Pair>(0);
        for (int i = 0; i < positions.size(); ++i) {
            Get_empty_nearby(positions.get(i), Legals);
        }
        int s = Legals.size();
        if (!Legals.isEmpty()) {
            for (int j = s-1; j >= 0; --j) {
                if (!is_legal(Legals.get(j), c)) {
                    Legals.remove(j);
                }
            }
        }
        return Legals;
    }
```

Realizar una jugada

Para poder realizar una jugada (asumiendo que ésta es legal), hay que colocar la ficha en la posición deseada y cambiar el color de todas las fichas enemigas que queden “encerradas”.
Esto consiste en una idea parecida a la comprobación de la legalidad de una jugada, pero en este caso no hay que salir una vez se encuentra una dirección que cumpla la premisa. Lo que hacemos consiste en: una vez compruebas que una dirección es válida, damos marcha atrás y cambiar los colores de las fichas enemigas en el camino. Ésto último se repite por cada dirección.
Hay que tener en cuenta que los cambios de color no se pueden realizar hasta que se han comprobado todas las direcciones, pues si editas simultáneamente puede causar problemas.

```
public ArrayList<Pair> commitPlay(Pair p, Casilla c) {
        ArrayList<Pair> swaps = new ArrayList<>(0);

        if (inBounds(p) && is_legal(p, c)) {
            matrix[p.first()][p.second()] = c;
            addtoColor(p, c);
            swaps.add(p);
            swapEnemy(p, c, swaps);
        }
        return swaps;
    }
```

```
protected void swapEnemy(Pair p, Casilla c, ArrayList<Pair> swaps) {
      if (matrix[p.first()][p.second()] == c) {
         for (Pair dir : directions) {
            Pair start = p.sum(dir);
            if (inBounds(start) && matrix[start.first()][start.second()] == c.contrary()) {
               while(inBounds(start)){
                  if (matrix[start.first()][start.second()] == c) {
                     start = start.sub(dir);
                     while(!start.convertString().equals(p.convertString())) {
                        removefromColor(start ,c.contrary());
                        addtoColor(start, c);
                        swaps.add(start);
                        start = start.sub(dir);
                     }
                     break;
                  }
                  if (matrix[start.first()][start.second()] == Casilla.VACIA) {
                     break;
                  }
                   start = start.sum(dir);
               }
            }
         }
             for(int i = 1; i < swaps.size(); ++i) {
             swapTile(swaps.get(i));
         }
      }
   }
```

Escoger jugada IA

Para que la IA pueda escoger un movimiento necesitamos que pueda hacer una evaluación sobre todos los que tiene disponibles. Para ello, intentamos que haga una “predicción” sobre cada jugada, y con esto escoja la que más beneficio traiga a futuro.

El modo en que hacemos esto consiste en, por cada jugada disponible, la simula (movimiento A) e intenta predecir el movimiento del enemigo(movimiento B). Sobre el estado del tablero donde A y B ya han ocurrido, intenta hacer una jugada más, y de estas jugadas se selecciona la jugada más óptima.
Para esto usamos el algoritmo Minimax para minimizar pérdidas. El algoritmo consiste en generar el peor caso posible para sí mismo, y de esta situación escoger el movimiento que más beneficio le traiga.

```
 Tree createTree(Tablero tablero, IA ia) {
        Tree jugadas = new Tree(new Node());
        createTree_rec(jugadas, ia.getDepth(), tablero, ia, ia.getColor());
        return jugadas;
  }
private static void createTree_rec(Tree tree, int depth, Tablero tablero, IA player, Casilla color) {
        if (depth >= 0) {
            ArrayList<Pair> legalMoves = tablero.getLegalMoves(color);
            for (Pair p : legalMoves) {
                Tablero tDeepCopy = tablero.DeepCopy();
                ArrayList<Pair> pa = tDeepCopy.commitPlay(p, color);

                Tree tr;
                int score = player.obtenerScore(pa);
                tr = tree.addLeaf(new Node(p, color, score));
                createTree_rec(tr, depth - 1, tDeepCopy, player, color.contrary());

            }
        }
    }
```

```
public Pair escogerMovimiento(Tree<Node> t) {
        Node maxEval = new Node(NEGATIVE_INFINITY);

        for (Tree<Node> child : t.getSubTrees()) {
                Node eval = alpha_beta(child, this.depth, NEGATIVE_INFINITY, POSITIVE_INFINITY, true);
                if(maxEval.getScore() < eval.getScore()) {
                    maxEval.setScore(eval.getScore());
                    maxEval.setCord(child.getRoot().getCord());
                }
            }

        Node movimiento = maxEval;
        return movimiento.getCord();
    }
}
```

```
private Node alpha_beta(Tree<Node> t, int depth, int alpha, int beta, boolean maximizingPlayer) {
        if (depth == 0 || t.getSubTrees().isEmpty()) {
            return t.getRoot();
        }

        if (maximizingPlayer) {
            Node maxEval = new Node(NEGATIVE_INFINITY);
            for (Tree<Node> child : t.getSubTrees()) {
                //System.out.println(depth + " " + this.depth);
                Node eval = alpha_beta(child, depth - 1, alpha, beta, false);

                    int scr = ((maxEval.getScore() > eval.getScore()) ? maxEval.getScore() : eval.getScore());
                    //System.out.println(depth + " " + scr);
                    maxEval.setScore(scr);


                alpha = ((alpha > eval.getScore()) ? alpha : eval.getScore());
                if (beta <= alpha) {
                    break;
                }
            }
            return maxEval;
```

```
} else {
        Node minEval = new Node(POSITIVE_INFINITY);
        for (Tree<Node> child : t.getSubTrees()) {

            //System.out.println(depth + " " + this.depth);
            Node eval = alpha_beta(child, depth - 1, alpha, beta, true);

                int scr = ((minEval.getScore() < eval.getScore()) ? minEval.getScore() : eval.getScore());
                //System.out.println(depth + " " + scr);
                minEval.setScore(scr);


            beta = ((beta < eval.getScore()) ? beta : eval.getScore());
            if (beta <= alpha) {
                break;
            }
        }
        return minEval;
    }
 }
```