Estructuras de Datos y Algoritmos

Informe TPE

Integrantes

- Karpovich, Lucia Legajo 58131
- Tarradellas del Campo, Manuel Legajo 58091
- Rolandelli, Alejandro Legajo 56644

2° Cuatrimestre- 2018

Estructura de datos

Reversi

Es la estructura principal del backEnd que maneja el juego en general. Fue esta clase la que se mantuvo casi constante a través del desarrollo del proyecto ya que se adapta a las reglas del juego, es decir, desde un inicio se sabía que funciones debía tener para que el *Controller* del frontEnd pudiera replicar el juego en pantalla.

Contiene el tablero, los jugadores, es el encargado de decidir a quién le toca jugar, crear los jugadores dependiendo de la entrada del usuario, y de llamar al algoritmo *Minimax.* Es la encargada de recibir los parámetros del *main* y armar dos tipos de jugadores (AI y Humano) y decirle al AI que estrategia de búsqueda va a tener. También se encuentra el método que maneja el "undo", que reemplaza la instancia del tablero que se encuentra en la clase.

Board

En un comienzo se decidió que un tablero sería una matriz ya que es lo más cercano a lo que en realidad representa y una estructura fácil de acceder, recorrer y actualizar. Esta fue una decisión que se replanteó varias veces dado que se arman muchos tableros a lo largo del juego y se perdía eficiencia con cada recorrido. Pero se conservó la estructura original dado que todas las estructuras que cumplían con los requisitos para funcionar como tabla, requerían algún tipo de recorrido similar y acababan teniendo una eficiencia análoga.

Esta estructura tiene dos tipos de constructores: uno que será llamado únicamente al comienzo de un juego y que inicializa el tablera de la manera tradicional que lo hace el juego *Reversi*.

El segundo constructor es un método de instancia, el cual devuelve un nuevo tablero que será un clon del tablero con el que se llamó al método, solo que contiene un movimiento más. Esto fue resultado de la idea de que cada tablero está directamente relacionado con el tablero anterior de la manera que se describió recién, y consecuentemente, esta fue la manera lógica de generar el siguiente tablero.

Cada tablero tiene una serie de métodos y variables que permiten evaluarlo y modificarlo de distintas maneras (agregando un disco, ver si una posición es válida, devolver una lista con las posiciones válidas para un jugador, etc). También contiene el jugador activo de la tabla, quien debe hacer el próximo movimiento, para evitar movimientos ilegales. Por otro lado tiene los puntajes de los jugadores, dado que estos dependen de la cantidad de fichas que tiene cada uno, y esto varía según el tablero

Se evaluó repetidas veces la creación de alguna variable que tuviera las posiciones libres pero se descartó ya que la actualización de dicha variable obligaba a utilizar los mismos recorridos y métodos que estábamos tratando de evitar y/o eficientizar.

La única modificación que se realizó finalmente se aplicó solo para los tableros involucrados en el algoritmo Minimax (ver Algoritmo Minimax y Heurística)

Disc

Un disco es la representación de una ficha del juego. Cada una tiene un color (haciendo referencia a un jugador), y una clase *Move* que representa una posición en el tablero.

Player

Esta clase contiene un color que identifica a cada jugador e información tal como si es un jugador humano y su cantidad de fichas. Inicialmente esta clase no contaba con la cantidad de fichas ya que estas ya se encontraban en cada tablero y asignárselas a cada jugador con cada cambio de tablero parecía innecesario, pero nos pareció que se adaptaba más al paradigma de objetos si cada jugador contaba con este dato.

Cada jugador, además, cuenta con una instancia de su enemigo. En el inicio no se había diseñado de esta forma ya que como objeto el jugador no tiene por qué saber quién es otro jugador, pero a medida que más y más métodos y estructuras (sobre todo los *Nodos* de *Minimax*) nos planteaban la necesidad de tener acceso a el otro jugador para plantear el siguiente turno, se decidió hacer esta implementación.

Move

En la versión original de las clases no existía esta clase, cada disco contaba con las dos variables que actualmente conforman a la clase *Move* (una posición *x* y una posición *y*). Pero pronto nos dimos cuenta de que varios métodos requerían la misma información, y se decidió agregar esta clase como una forma de simplificar el código.

Node

La clase *Node* es una estructura auxiliar que se creó para la correcta implementación del algoritmo *Minimax*. Contiene la información necesaria para proveer al algoritmo (un tablero y su jugador activo entre otros) pero además, posee la información necesaria para poder crear el archivo .dot de la decisión de *Minimax*. Dado que la creación de dicho archivo se hace de manera recursiva desde la clase *Minimax*, cada nodo debe saber "escribirse" en el archivo .dot.

Controller

Esta clase es la que hace el manejo del frontEnd. Tiene acceso a las clases del backEnd a través de una instancia de juego (*Reversi*) que contiene.

Algoritmo Minimax y Heurística

Minimax es un algoritmo de búsqueda que, partiendo del tablero actual, permuta sobre todas las posibles opciones inmediatas que tiene comparando los resultados con el fin de devolver la mejor jugada posible basándose en posibles jugadas a futuro. Las jugadas son evaluadas a través de una función propia de la clase *Node* que calcula la heurística.

Originalmente se planteó la clase con métodos no static pero esto significaba instanciarla por cada decisión que le tocara a AI. Fue entonces cuando se planteó que los métodos fueran propios de *Reversi,* que es quien los usaría. Se descartó esta opción por un tema de prolijidad y separación de código, planteándose finalmente la estructura actual (clase con métodos static).

El algoritmo puede ser invocado en dos modos distintos: profundidad y tiempo. En el primer modo opera con una recursiva sobre el nivel de profundidad del árbol, el nodo hoja (en el que se calcula la heurística) será aquel en la profundidad indicada. Para el segundo modo se decidió implementar un

algoritmo que actúe con profundidad incremental dependiendo del tiempo, es decir, invoca al algoritmo por profundidad primero con profundidad 1 y aumentando esta variable si es que el tiempo no se terminó.

Para la heurística, se intentó replicar un pensamiento bastante similar al de un jugador humano, mientras se hacía lo más genérico posible. Esto incluyo calcular 3 cosas: paridad de discos, rincones conquistados y espacios x conquistados.

La paridad de discos representa la diferencia entre la cantidad de discos que posee el AI menos la cantidad de su contrincante sobre la totalidad de los discos.

Como los rincones del tableo suelen ser una posición de ventaja se realizó un proceso similar al anterior para devolver un número positivo cuando el AI tiene más rincones que su oponente.

De la misma forma se calcula la tercera componente de la heurística. Los espacios x son espacios que le darían al otro jugador acceso a las esquinas, por lo cual, si AI tiene más que el humano, esta función devolverá un peso negativo.

En un principio la heurística iba a ser un método de *Minimax* dado que es este algoritmo el que lo usa, pero se decidió que la heurística era algo propio de cada nodo, y es por es que en la estructura actual es allí donde se encuentra.

Problemas

La creación del archivo .dot, el guardado de archivos, la compilación y ejecución con *Maven* y la obtención de parámetros agregaron complejidad a la generación del trabajo dado que implicaban una investigación por separado.

Otro gran problema fue el caso donde un jugador se queda sin jugadas pero el juego no ha terminado aún. Estos casos complicaron significativamente el algoritmo *Minimax* y todo el sistema en general ya que resulto en muchas nuevas verificaciones y cambios de turno que no suelen ser los usuales.

El mayor problema encontrado fue el de lograr que *Minimax* se resuelva de manera eficiente. Por cada decisión, este algoritmo crea un árbol con una gran cantidad de nodos, en donde, en cada uno, se genera una nueva instancia de *Board.* Esto genera una significante pérdida de recursos. Se plantearon modificaciones y nuevas estructuras repetidas veces pero sin grandes resultados. Se decidió que el árbol de *Minimax* tendría en cada nodo la misma instancia de *Board* para evitar estar creando siempre nuevas, esto fue posible mediante backtracking y algunas variables nuevas (*lastAdded* y *flipped*) y métodos nuevo (*flip* y *unflip*). Este cambio solo se aplicó para el algoritmo de AI dado que *Reversi* si necesita que todas las *Boards* usadas a través del juego sean instancias distintas para poder resolver el funcionamiento del botón "Undo".

Adicionalmente, se mejoró la eficiencia del algoritmo creando los nodos únicamente a medida que se van recorriendo, de esta manera y contando con el *garbage collector* se pudo hacer un uso más eficiente de la memoria. Pero este último cambio, si bien lograba el resultado deseado con respecto a la eficiencia del algoritmo, fue desechado. Esta decisión fue tomada dado que el código dificultaba en grandes medidas la creación del archivo .dot. Es por esto que la versión final del algoritmo resulto ser la que aplicaba el segundo cambio (crea todo el árbol pero cada nodo tiene la misma instancia de *board* y trabaja con backtracking).