TPE

Reversi

24 de octubre de 2011

Autores:

Conrado Mader Blanco	Legajo: *51270*
Federico Ramundo	Legajo: *51596*
Tomás Mehdi	Legajo: *51014*

Índice

1. Introducción

En este trabajo se desarrollo una aplicación en Java para jugar al juego Reversi con la computadora como oponente. Para la inteligencia artificial de esta última se eligió el algoritmo Minimax.

2. Reglas del juego

Reversi es un juego de 2 jugadores que ubican fichas blancas y negras por turnos (un color para cada jugador) en un tablero de 8 filas por 8 columnas.

En cada turno, el jugador debe ubicar una ficha de su color en una celda vacía, adyacente a alguna celda del otro jugador (en sentido horizontal, vertical o diagonal), y de manera tal que una o mas fichas del oponente quede "encerrada" por fichas del jugador que está jugando. Al realizar la jugada, estas fichas del oponente modifican su color al del jugador que acaba de jugar. Si no pueden realizarse movimientos, el jugador debe pasar, cediéndole el turno al oponente.

Se considera el juego por terminado cuando se cumple alguna de las siguientes situaciones:

- No quedan lugares vacíos en el tablero.
- Ningún jugador puede ubicar fichas.

En ambos casos gana el jugador con mas fichas presentes de su color.

3. Diseño y Estructuras de datos

Para implementar el árbol minimax se utilizó la clase MiniMaxTree, que utiliza para representar el árbol la clase abstracta Node y sus subclases MiniNode y MaxNode.

Cada nodo guarda el estado del tablero así como también la posición en la que se colocó una ficha para llegar a dicho estado desde el anterior.

Para el tablero se implemento la clase Board que contiene una matriz de fichas de 8x8 para representar las celdas del tablero. La clase implementa un método que permite poner una ficha, y que como resultado devuelve un nuevo tablero sin modificarse a si mismo. Se llegó a esta decisión dado que para aplicar el arlogritmo minimax se evaluan muchos tableros distintos antes de efectivamente aplicarle un cambio al tablero actual. Este algoritmo de implemento de manera recursiva, y controla que el movimiento sea valido en al

menos una de las ocho posiciones aledañas (representadas por un enumerativo) a la posición donde se quiere poner una ficha.

Para representar las fichas se creo un enumerativo con tres valores: EMPTY, PLAYER_1 y PLAYER_2. Estas determinan que hay en cada posición del tablero.

La clase Reversi, que representa al juego en si, contiene una instancia de Board y guarda información acerca de qué jugador es el turno de jugar y si algún jugador puede o no ejecutar movimientos, entre otras cosas.

4. Algoritmo Minimax

El algoritmo elige la mejor jugada asumiendo que el adversario elegirá la que más lo perjudica. Para llevar esto a cabo se modela un árbol, en el cual, cada vértice corresponde a un estado del juego. Los nodos son diferenciados: los nodos *Max* corresponden a la computadora y los nodos *Mini* al jugador.

1. Si el nodo es MAX, su valor heurístico corresponde al máximo del valor heurístico de sus hijos.
2. Si el nodo es $MINI$, su valor heurístico corresponde al mínimo de sus hijos.
3. Si el nodo es *hoja*, su valor heurístico se calcula aplicando una función heurística al estado. Esta función es clave para la eficiencia y efectividad del algoritmo.

El algoritmo construye el árbol mediante DFS (Depth First Search, crece a lo largo), acotado a una profundidad máxima. Esto se lleva a cabo mediante un algoritmo recursivo en los nodos.

4.1. Limitado por nivel

Limitar el algoritmo por nivel es fácil de implementar debido al recorrido DFS y la recursión. Basta que un nodo sepa su nivel en el árbol y que termine la recursión al llegar al nivel máximo.

4.2. Limitado por tiempo

La limitación por tiempo presenta dificultades ya que si se limita el recorrido DFS por tiempo, y no por un nivel máximo como en el caso anterior, se obtendría una rama (un camino simple) que en muy pocos casos corresponderá con la mejor jugada. Si no se

tuviese en cuenta la poda alfa-beta se podría utilizar un recorrido BFS (Breadth First Search) para generar el árbol y luego recorrerlo con DFS para obtener la mejor solución. Al existir la opción de utilizar poda alfa-beta, esto no es posible, dado que no se sabe si un nodo sera podado hasta calcular los valores de los hermanos izquierdos de dicho nodo en el árbol. Si este crece de manera BFS, es posible que se calcule un nodo, y que más adelante resulte que un ancestro de ese nodo deba haber sido podado, por lo que nunca debería haber sido calculado.

Al existir dicha complicación se decidió resolver el problema de la siguiente forma:

1. Se toma como solución la del árbol de nivel 1.
2. Mientras quede tiempo se elabora un árbol con un nivel más.
3. Si el árbol se completa, se guarda la solución la devuelta por el mismo y se vuelve al paso 2.

Aunque a simple vista esta solución puede parecer ineficiente, puede calcularse que es de $O(N)$ siendo N la cantidad de nodos del ultimo árbol calculado.

El tiempo se controla cada N iteraciones de la recursión (siendo N un numero arbitrario), para evitar comprobar en cada paso y dotar al algoritmo de mejor eficiencia.

Dado que no se poseen conocimientos de Threads, el tiempo que tarda el algoritmo no es exacto, y cuando este llega a su límite, se lanza una excepción que es capturada por la función wrapper de la recursiva, que retorna como resultado la respuesta calculada por el árbol anterior (el de un nivel menor al ultimo).

5. Poda Alfa-Beta

Esta poda permite cortar ramas del árbol para evitar realizar cálculos innecesarios y mejorar la eficiencia. Se realiza en los siguientes casos:

1. El nodo actual es MAX y el valor heurístico mínimo hasta el momento del padre es P. Si el valor heurístico C de un hijo cumple $C \geq P$ entonces no es necesario analizar los demás hijos.
   Explicación: el nodo actual tendrá un valor heurístico mayor o igual que C, por ende el padre no lo elegirá ya que P es menor.

2. El nodo actual es MINI y el valor heurístico máximo hasta el momento del padre es P. Si el valor heurístico C de un hijo cumple C<=P entonces no es necesario analizar los demas hijos.
   Explicación: el nodo actual tendrá un valor heuristico menor o igual que C, por ende el padre no lo eligirá ya que P es mayor.

6. Valores Heurísticos

El valor heurístico de cada tablero fue dado de la siguiente manera:

cada posición en la matriz que representa el tablero tiene un valor, que representa qué tan beneficioso resulta tener una ficha ahí. Por ejemplo, la esquina tiene el máximo valor, dado que si se tiene una ficha en esta posición, es imposible que sea robada por el otro jugador, por lo que resulta una posición conveniente.

La matriz de valores es la siguiente:

99	-8	8	6	6	8	-8	99
-8	-24	-4	-3	-3	-4	-24	-8
8	-4	7	4	4	7	-4	8
6	-3	4	0	0	4	-3	6
6	-3	4	0	0	4	-3	6
8	-4	7	4	4	7	-4	8
-8	-24	-4	-3	-3	-4	-24	-8
99	-8	8	6	6	8	-8	99

Como es lógico, la matriz es simétrica. Obsérvese que los valores mínimos son los que corresponden a las posiciones que están una antes del borde del tablero, ya que si se coloca una ficha ahí, es muy probable que el otro jugador ponga una en el borde, que son las fichas mas dificiles de robar.

A partir de esta matriz se obtiene el valor heurístico del tablero de la siguiente forma:

- Si en dicha posición no hay ninguna ficha, no se hace nada.
- Si en la posición correspondiente hay una ficha de el jugador, se suma el valor correspondiente a la matriz de valores.
- Si en la posición correspondiente hay una ficha del oponente, se resta el valor correspondiente a la matriz de valores.

El cálculo del valor heurístico de un tablero es de $O(N)$, es un algoritmo rápido, lo cual es deseable, dado que cuanto menos tarde

en calcularse más tiempo tendrá el algoritmo Minimax a realizar otros cálculos y llegar a una mejor solución en caso de ser acotado por tiempo.

6.1. Heurística descartada

Se pensó otra heurística, que experimentalmente demostro ser mejor que la actualmente implementada. Dicha heurística devolvía un valor correspondiente a n tablero sumando los siguientes valores:

- La cantidad de fichas del jugador.
- La cantidad de posiciones en donde se pueden poner fichas multiplicada por tres.
- Si una esquina esta ocupada por el jugador, vale 50.
- Si una esquina no esta ocupada, entonces sus tres celdas aledañas valen -10.
- Las fichas de los bordes restantes (ni las esquinas ni sus aledañas) si están ocupadas por el jugador valen 5.
- Si el juego esta terminado y el jugador es el ganador, devuelve 10000, si el oponente gana, devuelve -10000.

Esta heurística fue descartada dado que es mucho mas compleja: recorre el tablero muchas veces, para calcular posibles movimientos, contar fichas, etc. por lo que resultaba inconveniente si se juega acotado por tiempo.

7. Parsing

En el trabajo fue necesario realizar dos tipos de parsing distintos. Ambos fueron realizados de forma imperativa, dado que la naturaleza de este hace que sea más fácil realizarlo de esta manera.

7.1. Parsing de tablero

En el trabajo fue necesario poder abrir un archivo que determinaba lo configuración de un tablero, y mediante el algoritmo minimax se debia imprimir en consola el movimiento óptimo a realizar a continuación, por el jugador explicitado por linea de comandos. Esto se llevaba a cabo realizando un parsing de un archivo que tenia las siguientes características:

- Una matriz de caracteres del tamaño del tablero.
- Si la posición en esa matriz se encuentra un 1, entonces en esa posición en el tablero hay una ficha del jugador 1 (el humano en nuestro caso).
- Si se encuentra un 2, entonces en esa posición hay una ficha de el jugador 2 (la computadora).
- Si se encuentra un espacio en blanco, entonces no hay fichas en esa posición.
- Si se encuentra cualquier otra cosa, se lanza una excepción (que es capturada y en consecuencia se imprime en pantalla un mensaje de error).

Dado que el enunciado no especifica que hacer con los caracteres que se encuentran más alla de la matriz de 8x8 en el archivo, nos encontramos con una disyuntiva de dos opciones, igualmente válidas:

- Ignorar lo que este a fuera de la matriz (de este modo podría comentarse el archivo).
- Tomar el archivo como invalido y lanzar una excepción.

Finalmente nos decantamos por la segunda opción dado que consideramos más claro que en este archivo haya solo información del tablero.

7.2. Parsing de linea de comandos

Fue necesario poder levantar el archivo jar por consola, especificando algunos argumentos. Estos utlimos fueron leidos y controlados separados en tres etapas:

7.2.1. Parsing de modo de juego

Esta primera etapa determinaba si se deseaba jugar de forma visual, o si por el contrario se queria levantar un archivo e imprimir la siguiente mejor jugada. Si se daba este ultimo caso, se leia ademas el archivo del tablero de donde se determinaria la jugada, y de quien es el turno de realizar la jugada.

7.2.2. Parsing de la dificultad de juego

En esta segunda parte se determinaba si el algoritmo Minimax estaría acotado por nivel o por tiempo, y en cada caso, cual es su cota maxima.

7.2.3. Parsing de argumentos opcionales

Finalmente se determinaba si se deseaba ejecutar el algoritmo Minimax con poda, y si se deseaba generar un archivo dot. Si se daba este último caso, se controlaba que no se haya escogido jugar en modo visual, dado que esta configuración es incorrecta.

Dado que la cantidad de argumentos es muy variable (si se elige modo visual deben haber al menos 3 argumentos, mientras que si se escoge leer un archivo, el minimo de argumentos es 6) los métodos reciben y devuelven un índice que indica cual es argumento siguiente a leer en el vector de strings con el cual se representa los argumentos de consola.

NOTA: Dado que resultaba más fácil de implementar, y mas rápido, (además de que era válido hacerlo) no aceptamos que los comandos lleguen en distinto orden al dado por el enunciado.

8. Archivos DOT

Se agregó a la clase MiniMaxTree la posibilidad de generar un archivo en formato DOT. Este formato permite representar digrafos (en este caso un árbol dirigido) y mediante alguna herramienta de visualización obtener una imagen gráfica. Estos archivos son utiles para visualizar los pasos del algoritmo minimax.

Caracteristicas:

- Los nodos están etiquetados con la última posición en donde se colocó una ficha, y su valor heurístico.
- Los nodos MAX tienen forma rectangular.
- Los nodos MIN tienen forma ovalada.
- El hijo elegido por un nodo es rojo.
- Los hijos podados mediante alfa-beta (si se activa) son azules, y no tienen valor heurístico.
- Los nodos no elegidos son blancos.

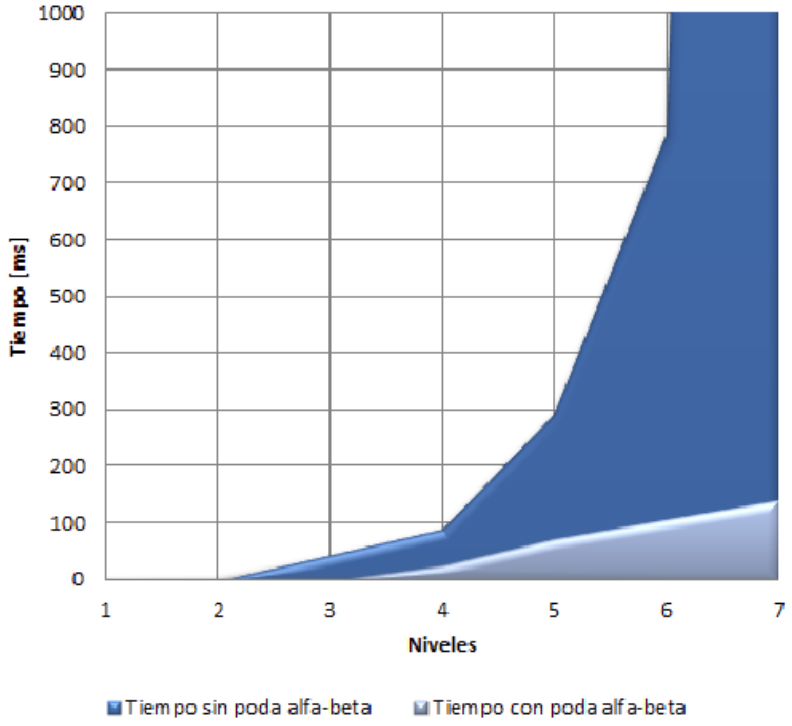
Consideramos necesario que la generación del DOT se realice después de elegir la jugada (aplicar el algoritmo) ya que de no ser así disminuiría la eficiencia del algoritmo; además, en el caso de limitar el árbol por tiempo, no deben generarse el DOT de cada árbol generado sino solamente el que corresponde al árbol final.

9. Comparación de tiempos

En la siguiente tabla se observan los tiempos promedio para el algoritmo minimax con distintos parámetros en un nuevo tablero.

Nivel	Tiempo sin poda alfa-beta [ms]	Tiempo con poda alfa-beta [ms]
1	4	3
2	5	5
3	45	7
4	92	27
5	295	75
6	792	106
7	7540	141

Gráfico de los tiempos en función del nivel



Análisis de resultados:

Con los tiempos promedios obtenidos se puede observar el crecimiento no lineal del tiempo en función del nivel. Esto se debe a que el algoritmo minimax depende linealmente de la cantidad de nodos, pero la cantidad de nodos no crece linealmente al aumentar el nivel.

No es posible saber de antemano cuantos hijos tendrá cada nodo del árbol, dado que la cantidad de posiciones en donde es posible colocar una ficha es variable.

Hay un trade-off entre el tiempo que se tarda en aplicar el algoritmo con un nivel dado y la calidad del resultado. Este compromiso es introducido por la complejidad de la función heurística. En nuestro caso, la complejidad de la función es lineal con respecto a la cantidad de posiciones del tablero.

9.1. Conclusión

Si bien en el peor caso al podar se recorren todos los hijos, experimentalmente se observa que la rapidez al podar es notoriamente mayor frente a recorrer el árbol completo, especialmente en niveles altos como el 7 (véase gráfico).

La poda resulta especialmente útil al limitar por tiempo: en un mismo tiempo se puede llegar a un nivel mucho mayor. En cambio, al limitar por nivel el resultado es el mismo a pesar de la poda (aunque el tiempo que tarda en obtener el resultado es menor).