INSTITUTO TECNOLÓGICO DE BUENOS AIRES

Azulejos

Estructura de datos y algoritmos
Informe de desarrollo

Alejandro Bezdjian - 52108
Francisco Depascuali - 53080
Eugenia Sakuda - 53191

27 de Mayo del 2014

Tabla de contenido

1. Introducción 2
2. Tipo de problema 3
   a. Algoritmo Minimax 3
   b. Funcionamiento de minimax 3
   c. Poda Alfa-Beta 4
3. Estructuras Utilizadas 5
4. Algoritmos 6
   a. Implementación de Minimax 6
   b. Cálculo de jugadas posibles 7
   c. Validación del movimiento 8
   d. Árbol dot 8
5. Otras clases útiles 9
   a. Chronometer 9
   b. PrimeNumbers 9
6. Conclusiones 10
7. Referencias 11
8. Anexo 12
   Tabla de tiempos 12

1. Introducción

El trabajo que a continuación se describe es una implementación del juego "Azulejos", en el cual se le permite a un jugador jugar contra la computadora en modo visual mediante una interfaz gráfica o permite calcular la mejor jugada posible de un determinado tablero pasado por parámetro (en modo consola). El objeto de este informe es exponer los algoritmos creados para resolver el juego (y su orden de complejidad). Para ejecutar el programa, se debe respetar la siguiente sintaxis en la línea de comandos (los paréntesis indican opciones de las cuales se debe elegir una, y los corchetes indican parámetros optativos):

java -jar tpe.jar -file archivo (-maxtime n | -depth n) (-visual | -console) [-prune] [-tree]

2. Tipo de problema

El juego “Azulejos” entra en la categoría de problema de juego de dos jugadores por turnos en el cual ambos jugadores conocen:

- El tablero en todo momento.
- Las acciones que realiza cada uno y las que realiza el oponente.
- Las consecuencias de esas acciones en el tablero y el juego.

a. Algoritmo Minimax

"Un juego es una situación conflictiva en la que uno debe tomar una decisión sabiendo que los demás también toman decisiones, y que el resultado del conflicto se determina, de algún modo, a partir de todas las decisiones realizadas.", "Siempre existe una forma racional de actuar en juegos de dos participantes, si los intereses que los gobiernan son completamente opuestos." - John von Neumann

Otros juegos y problemas conocidos que cumplen con estas mismas situaciones son: Damas, Ajedrez, Go, Tres en raya, entre otros.

Todos estos juegos y problemas que cumplan con las condiciones anteriormente mencionadas, pueden resolverse mediante el algoritmo Minimax (uno de los más utilizados), un algoritmo recursivo que calcula la mejor opción para un jugador suponiendo que el oponente elegirá la opción que más desfavorece al primero.

A priori, minimax parece ser una excelente alternativa para la solución del problema en cuestión, pero tiene algunas desventajas. Debido a que utiliza un árbol en el cual los hijos de un nodo son todos los estados del juego posibles a partir del mismo, el orden espacial de este árbol es O(b*m) siendo b los posibles movimientos en cada paso y m los movimientos hasta el fin del juego. Esto quiere decir que el árbol para ciertos juegos donde el tablero es relativamente grande y la cantidad de movimientos posibles en cada momento también es grande, el tamaño del árbol que se genera es muy difícil (sino imposible) de procesar en un tiempo “normal” que un jugador espera o estima que su oponente realice un movimiento.

Para la solución de este gran problema temporal que se encuentra, existen algunas opciones posibles:

1. Limitar la profundidad del árbol generado.
2. Dar un tiempo límite de cálculo a la máquina.
3. Utilizar la poda alfa-beta a ese árbol para evitar cálculos innecesarios.
4. Combinar las opciones anteriores.

En el desarrollo de este trabajo práctico se utilizó minimax (con las mejoras mencionadas) para implementar la inteligencia artificial de la computadora tanto para el juego hombre vs. computadora como para la búsqueda de la mejor solución posible para un tablero dado.

b. Funcionamiento de minimax

Este algoritmo separa a dos jugadores a los cuales llama **Min** y **Max** (de allí su nombre), estos representan cada nodo en el árbol. Cada uno de los jugadores, tal como lo indica su nombre, elegirá de las jugadas posibles a realizar la que más lo favorece, es decir

la que menos le haga perder y la que más le haga ganar respectivamente. Para poder decidir cual de las opciones posibles elegir, es necesario que cada nodo esté cuantificado utilizando alguna función matemática que de un valor heurístico a cada estado del juego de la forma más precisa posible, una vez calculado estos valores los nodos Min elegirán el menor valor posible y los nodos Max los mayores valores, lógicamente. "Elegir" un valor significa que el valor heurístico de ese nodo ahora será aquel valor seleccionado, de esta forma la función matemática solo se aplica a las hojas del árbol, mientras que los nodos intermedios simplemente toman un valor de sus hijos como propio. Debido a que cada juego es diferente, esta función heurística también lo es. En este caso particular se utilizó la función: puntaje jugador 2 - puntaje jugador 1. Como la resta no es una operación conmutativa se puede observar que el jugador Min claramente elige la jugada que más favorece al jugador 1 y Max elige la jugada que más favorece al jugador 2. De haber cambiado el orden de la resta en la función elegida, también se hubiese invertido lo que buscan los jugadores Min y Max.

Con la opción de diseño elegida en mente, en modo visual la computadora debe ser un nodo Max (ya que es el jugador 2) y en modo consola la computadora actúa como nodo Min (ya que en este modo se busca la mejor jugada para el jugador 1).

c. Poda Alfa-Beta

La estrategia de poda del algoritmo Minimax es llamada poda alfa-beta, debido a que se utilizan dos valores umbral alfa y beta para acotar la búsqueda de cada jugador.

- El valor **alfa** representa la cota inferior del valor que puede asignarse a un nodo maximizador.
- El valor **beta** representa la cota superior del valor que puede asignarse a un nodo minimizador.

Debido a que minimax es un algoritmo recursivo que recorre el árbol de manera DFS, cada nodo obtiene el valor heurístico de uno de sus hijos antes de aplicar la función recursiva al resto. Es aquí donde se decide si se debe realizar la poda o no. Al principio el nodo en cuestión toma el valor de su primer hijo calculado y lo compara con el valor de alfa o de beta según corresponda. Un nodo minimizador realizará la poda si el valor es menor que alfa y un nodo max lo hará si el valor es mayor que beta.

3. Estructuras Utilizadas

Los estados del juego fueron representados mediante la clase abstracta `GameState`. La misma contiene una instancia de la clase `Board` (que representa el tablero), los puntajes de los jugadores y el valor heurístico del estado a utilizarse en el algoritmo minimax.

En la descripción de la inteligencia artificial vimos la necesidad de contar con dos tipos de estado de juego, uno para el minimizador y otro para el maximizador. Estos fueron representados mediante dos clases concretas que heredan de `GameState`, estas son `MinState` y `MaxState`. En ellas se implementa el comportamiento específico de cada nodo para el algoritmo, es decir métodos que cambien los valores de alfa, beta y que digan cuando se puede podar una rama y cuando no. También Incluye otros métodos que suman el puntaje a los jugadores según corresponda y un método que permite la interacción con el jugador en modo visual para realizar un movimiento.

La clase `Board` mencionada contiene una matriz de tipo `Cell` y una instancia de la clase `GameData`, las mismas serán explicadas más adelante. También esta clase contiene las coordenadas X e Y que generaron el tablero en cuestión (si es que no es el inicial) necesario para el modo consola. Tanto la clase `Board` como `GameState` contienen los métodos `hashCode()` y `equals()`, que son utilizados para evitar estados y tableros repetidos en el cálculo de las jugadas posibles. En esta clase está implementada la gravedad del tablero.

La clase `Cell` guarda un entero que representa el color (0 es vacío) y un vector de celdas adyacentes para facilitar el recorrido del tablero al intentar explotar una celda.

Finalmente en la clase `GameData` se tienen los datos generales a todos los tableros y estados (filas, columnas, si es modo visual o consola, si se permite la poda, etc…), así como también métodos que validan todos los datos ingresados antes de comenzar el juego. Esta clase es la encargada de verificar que los parámetros se hayan ingresado correctamente (en cantidad y escritos de forma correcta), que los valores de dichos parámetros sean correctos (por ejemplo que la profundidad máxima del árbol no sea menor a 0 o un número real), hay un chequeo del archivo, en el cual se busca que el número de filas, columnas y puntajes sean números naturales, así como que los datos en las filas y columnas sean números que representen colores y no haya basura.

4. Algoritmos

En esta sección se detallan los algoritmos considerados fundamentales para el desarrollo de este trabajo, así como también se mencionaran los cambios que sufrieron durante el transcurso del desarrollo.

No se encuentra en esta sección una descripción exacta de los pasos que sigue cada uno de los algoritmos, ya que incluir descripciones de ese tipo sería demasiado extenso y aumentaría la complejidad de entendimiento de los mismos, por eso se optó por la utilización de pseudocódigo. La implementación real de los mismos se encuentra en el código fuente.

a. Implementación de Minimax

Para evitar repetir código se implementaron las mejoras del algoritmo original en el mismo algoritmo mediante cláusulas `if()`.

```
Procedimiento minimax(current, depth, chronometer, alpha, beta)
Si no es por tiempo y no se puede ir más profundo
      calcular el valor heurístico de current;
      return current;
Fin Si
childs: {} <- calcular posibles estados;
Si set esta vacio
      calcular el valor heuristico de current;
      return current;
Fin Si
bestChild;
Para cada child en childs Hacer
      Si queda tiempo
            minimax(child, depth -1, chronometer, alpha, beta);
            Si child es mejor que bestChild
                  bestChild <- child;
            Fin Si
            Si la poda esta activada
                  alpha <- calcular nuevo alpha;
                  beta <- calcular nuevo beta;
                  Si se puede podar
                        return bestChild;
                  Fin Si
            Fin Si
      return bestChild;
      Fin Si
Fin Para
Fin Procedimiento
```

Cabe mencionar que en un principio se había realizado una implementación aparte para el modo limitado por una cantidad de tiempo. El mismo utilizaba una versión modificada de recorrido BFS en lugar de un DFS. La modificación hacía que se separaran los *n* hijos de la raíz como *n* sub-árboles distintos y se tomaban valores heurísticos aproximados, los cuales (de haber tiempo) calculaban un nivel más para mejorar dicha aproximación. Esta versión lograba jugar para tableros grandes y/o poco tiempo límite de manera más inteligente que la versión DFS. Las desventajas con aquella versión eran que dado un tiempo suficiente para recorrer el árbol completo de jugadas, esta versión BFS resulta más ineficiente que la DFS (debido a que cada nodo perteneciente al nuevo nivel debe comunicar al padre la información nueva y éste actualizar su valor y hacer lo mismo con su padre, repitiendo esto hasta llegar a la raíz) y la mayor desventaja era que no se podía aplicar la poda. Entonces, luego de consultar con miembros de la cátedra, se nos recomendó no tener dos algoritmos separados (uno para el modo por tiempo y sin poda y otro para el resto de posibilidades) sino utilizar el algoritmo DFS.

b. Cálculo de jugadas posibles

Este método está separado en dos partes. La primera se realiza en la clase `Board` la cual calcula a partir de un tablero cuales son los tableros posibles aplicando un movimiento válido posible. La segunda parte esta en la clase `GameState`, la cual a partir de los tableros posibles crea los estados (`MinState` o `MaxState` según corresponda) con los puntajes obtenidos por la jugada. En esta sección se mostrará solo el cálculo de tableros por ser la parte más compleja.

```
Procedimiento calculatePossibleBoards(state, set)
Para i <- 1 hasta filas con paso 1 Hacer
      Para j <- 1 hasta columnas con paso 1 Hacer
            Si se puede realizar un movimiento en (i,j)
                  agregar nuevo estado a set;
            Fin Si
      Fin Para
Fin Procedimiento
```

Es importante mencionar que en un principio se intentaba realizar un movimiento para cada celda del tablero, pero como cada jugada posible como mínimo explota 2 azulejos, esto hacía que en varias celdas se calcularán los mismos tableros (aunque al ser un set no se guardan repetidos, se perdía tiempo calculando lo mismo). Para solucionar esto, se creó una matriz de booleanos que se pasaba por parámetro al método de explosión y marcaba en la misma las posiciones ya exploradas, de esta forma primero se preguntaba en esa matriz si era necesario intentar un movimiento o no. Esta fue una gran optimización en tiempo ya que logró reducir el tiempo de cálculo en un tercio aproximadamente.

Con respecto a por que se pasa por parámetro un set de tipo GameState a un método de la clase Board (que desde un punto de vista de OOP sería incorrecto), en un principio se tuvo en cuenta el diseño orientado a objetos y el método `calculatePossibleStates()` llamaba al método `calculatePossibleBoards()` que

no recibía parámetros y simplemente devolvía un Set de Board con los tableros posibles, para luego recorrer el set y crear otro set de posibles estados. Esto si bien es más prolijo en cuanto a OOP, significaba una gran pérdida de tiempo debido a que es un método que se llama muchas veces por cada árbol y se agregaba n*m recorridos sobre sets que resultaban innecesarios.

c. Validación del movimiento

Este es un algoritmo simple, el cual solo chequea que alrededor de una determinada celda haya por lo menos alguna otra celda con el mismo color. Lo que resulta interesante destacar de este método es que se intentó hacer que reciba la matriz de booleanos mencionada en algoritmos anteriores para que en caso de encontrar una celda vacía se le aplique un método que hacía que cada celda adyacente que también estuviera vacía marque en la matriz que no habia que intentar realizar un movimiento allí. Esto que en un principio parecía una mejora de eficiencia, se descubrió más tarde que la complejidad agregada con este cambio hacia que el tiempo de cálculo se viera incrementado.

d. Árbol dot

Este algoritmo fue fusionado junto con el minimax para evitar repetir código u almacenar demasiada información dentro de una estructura auxiliar, dado que necesita almacenar en el archivo la información de los nodos con sus respectivos cambios de estados (“selected” o “prune”). Se agregaron en la clase GameState los métodos necesarios para poder ir construyendo en cada iteración del minimax el archivo “.dot”, como funciones para imprimir que supieran adaptarse a los diferentes estados para armar la estructura del nodo adecuado.
Inicialmente se conmenzó guardando las aristas en el un Set, bajo la creencia de que era indispensable que estuviesen al final del archivo para que funcionase el graficador. Además, al agregarle la poda, se estaba recorriendo y, aún peor, imprimiendo nodos extras. Por este motivo se buscó una implementación que resultara más eficiente: se eliminó la colección, imprimiendo las aristas mezclados con los nodos a medida que iban apareciendo puesto que se observó que el graficador podría reproducir de todas formas el archivo. A sí mismo, se evitó el repetir varios nodos, con la utilización de un flag para marcar que se había realizado una poda evitando hacer las llamadas recursivas, pero continuando con la iteración de posibles estados para la correspondiente impresión. Finalmente, con estas modificaciones se lograron bajar algunas milésimas en el funcionamiento del algoritmo en el caso de tener activada la poda. Esta diferencia se incrementa con el aumento del tamaño de los tableros. Respecto al tiempo sin el dot, apenas se ve una diferencia favorable en tableros medianos.

5. Otras clases útiles

Esta sección describe las clases extras utilizadas, las cuales se encuentran en el paquete utilities.

a. Chronometer

Esta clase es la utilizada en el algoritmo por tiempo. En su constructor se especifica el tiempo límite que se desea, en caso de ser 0 se comporta como un cronómetro normal que mide el tiempo transcurrido entre dos eventos.

Los métodos principales de esta clase son:

- `start()`: inicia la contabilización del tiempo.
- `stop()`: para el cronometro.
- `isOver()`: retorna true en caso de que el tiempo transcurrido desde start() sea mayor que el tiempo límite.
- `getFinalSeconds()`: devuelve el tiempo transcurrido, en segundos, desde que se ejecutó start() hasta que se ejecutó stop().

b. PrimeNumbers

Esta clase contiene los primeros 120 números primos. La función de la misma en el trabajo fue la de devolver diferentes números primos para las funciones de hashing de los tableros y estados haciendo que el código en esas clases sea más claro y simple.

El método principal de esta clase es:

- `get(n)`: devuelve el n-ésimo número primo. Los números primos están guardados en un vector y para elegir la posición del mismo se aplica la función modulo a n, de esta manera la única restricción de n es que sea un número entero.

6. Conclusiones

La principal conclusión que se obtiene de la realización de este trabajo es que Minimax es un algoritmo que resulta fácil de entender y programar y brinda excelentes resultados temporales si se aplica la poda alfa-beta.

La utilización de este algoritmo con límite de tiempo, debido a que el recorrido del árbol de jugadas es DFS, hace que si la relación "tiempo límite / tamaño de tablero y colores distintos" es muy pequeña, solo se recorren pocos estados posibles a partir de la raíz, haciendo que la computadora realice jugadas que no son muy inteligentes desde el punto de vista del juego. Vimos que un recorrido BFS lograba mejores jugadas manteniendo la misma relación pero se perdía la gran ventaja de podar el árbol, la cual permite aumentar la performance en ciertos tableros en más de 10 veces. Entonces como conclusión podemos rescatar que la mejor manera de utilizar este algoritmo es limitando la profundidad del árbol a calcular y utilizar la poda. La profundidad del árbol a elegir en un juego que se resuelva con Minimax dependerá del tamaño del tablero y los movimientos válidos para cada juego, por esto la única forma de elegir un valor de profundidad que respete los estándares de tiempo y calidad para los usuarios que jugarán será mediante pruebas, ya que cualquier cambio en estos parámetros puede significar la diferencia entre segundos, minutos y hasta horas en el cálculo de una jugada.

7. Referencias

- Material didáctico provisto por la cátedra.
- Wikipedia: http://es.wikipedia.org/wiki/Minimax
- Wikipedia: http://es.wikipedia.org/wiki/Poda_alfa-beta
- Introduccion a la inteligencia artificial: http://dmi.uib.es/~abasolo/intart/2-juegos.html

8. Anexo

Tabla de tiempos

A continuación se presentan las tablas que se construyeron para la verificación de tiempos y respuestas para distintas situaciones de juego. Cada tabla indica, además del tamaño del tablero usado para el test, el archivo correspondiente (los mismo se encuentran en la carpeta "file" del repositorio).

Tablero de 5x5 (XsGame.txt)

Parámetros		Prune		Sin Prune		Vers ant (sin poda)
Depth	Maxtime(seg)	Tiempo(seg)	Jugada	Tiempo(seg)	Jugada	Tiempo(seg)
2	-	0,019	(0,2)	0,018	(0,2)	0,026
4	-	0,112	(3,3)	0,136	(3,3)	0,139
6	-	0,194	(0,2)	0,216	(0,2)	0,23
-	10	0,230	(0,2)	0,284	(0,2)	0,304
-	20	0,230	(0,2)	0,284	(0,2)	0,304
-	30	0,230	(0,2)	0,284	(0,2)	0,304

Tablero de 10x10 (SmallGame.txt)

Parámetros		Prune		Sin Prune		Vers ant (sin poda)
Depth	Maxtime(seg)	Tiempo(seg)	Jugada	Tiempo(seg)	Jugada	Tiempo(seg)
2	-	0,168	(5,7)	0,153	(5,7)	0,175
4	-	1,615	(5,7)	4,816	(5,7)	4,601
6	-	57,114	(5,7)	889,952	(5,7)	915.900
-	10	10,001	(5,2)	10,000	(5,2)	10,000
-	20	20,001	(5,2)	20,001	(5,2)	20,001
-	30	30,0	(5,2)	30,002	(5,2)	30,002

Tablero de 15x15 (MediumGame.txt)

Parámetros		Prune		Sin Prune		Vers ant (sin poda)
Depth	Maxtime(seg)	Tiempo(seg)	Jugada	Tiempo(seg)	Jugada	Tiempo(seg)
2	-	0,368	(5,8)	0,360	(5,8)	0,360
4	-	26,691	(5,8)	265,305	(5,8)	258,289
6	-	-		-		-
-	10	10,001	(14,5)	10,006	(14,5)	10,006
-	20	20,004	(14,5)	20,003	(14,5)	20,003
-	30	30,002	(14,5)	30,01	(14,5)	30,01

Tablero de 20x20 (BigGame.txt)

Parámetros		Prune		Sin Prune		Vers ant (sin poda)
Depth	Maxtime(seg)	Tiempo(seg)	Jugada	Tiempo(seg)	Jugada	Tiempo(seg)
2	-	1,036	(4,15)	1,025	(4,15)	1,139
4	-	-		-		
6	-	-		-		
-	10	10,018	(15,7)	10,009	(15,7)	10,009
-	20	20,002	(15,7)	20,017	(15,7)	20,017
-	30	30,008	(15,7)	30,01	(15,7)	30,01

Tablero de 50x50 (HugeGame.txt)

Parámetros		Prune		Sin Prune		Vers ant (sin poda)
Depth	Maxtime	Tiempo(seg)	Jugada	Tiempo(seg)	Jugada	Tiempo(seg)
2	-	254,352	(0,9)	-		
4	-	-		-		
6	-	-		-		
-	10	11,104	(46,15)	10,011	(46,15)	10,011
-	20	20,762	(46,15)	20,163	(46,15)	20,163
-	30	31,501	(46,15)	32,056	(46,15)	32,056

* Valores obtenidos de una pc de 4GB de Ram con arquitectura de 64 bits.
** En la estimación de los valores se debe contemplar un mínimo error ya que al correr en distintos momentos el programa no tarda el mismo tiempo.
***Los campos de tiempos completados con - indican que no hubo prueba para ese algoritmo, debido a que el tiempo que lleva podría ser demasiado grande.