Trabajo Práctico Especial

Estructuras de Datos y Algoritmos

Grupo 1 | ITBA | Primer Cuatrimestre 2017

Alonso, Juan Manuel (56080)
Balfour, Oliver (55177)
Nielavitzky, Jonathan (54820)
Paganini, Nicolás A. (54321)
Vázquez, Agustín (55354)

Introducción

El Go es un juego de tablero estratégico para dos jugadores. Ha obtenido notoria popularidad recientemente debido a que Alpha Go, un programa informático desarrollado por Google, derrotó al jugador profesional y múltiple campeón Ke Jie. El Go es un desafío complejo en el mundo de la programación ya que el número de posibles partida, según muestran estimaciones numéricas, excede en gran proporción el número de átomos en el universo observable.

Se nos pidió desarrollar una aplicación en Java que pueda jugar al Go, cumpliendo las reglas generales del juego, y que además implementa como estrategia el árbol de decisión minimax.

Algoritmos y Estructuras principales utilizadas

MVC y CommandParser

Para el desarrollo del juego hicimos uso del patrón de diseño Modelo-Vista-Controlador visto en POO, que brinda una separación favorable entre datos y lógica e interfaces y gestión de eventos. La clase *CommandParser* se encarga de la sintaxis completa de la línea de comandos.

Tablero

Para emular el tablero de Go se creó una clase Board que contiene una matriz de *int*, la cual tiene 1 o 2 dependiendo de la ficha que el jugador colocó en una posición, siendo 1 para las negras y 2 para las blancas, o 0 en el caso de que no haya ninguna ficha (espacio en blanco).

Iterator: Vecinos Minimax

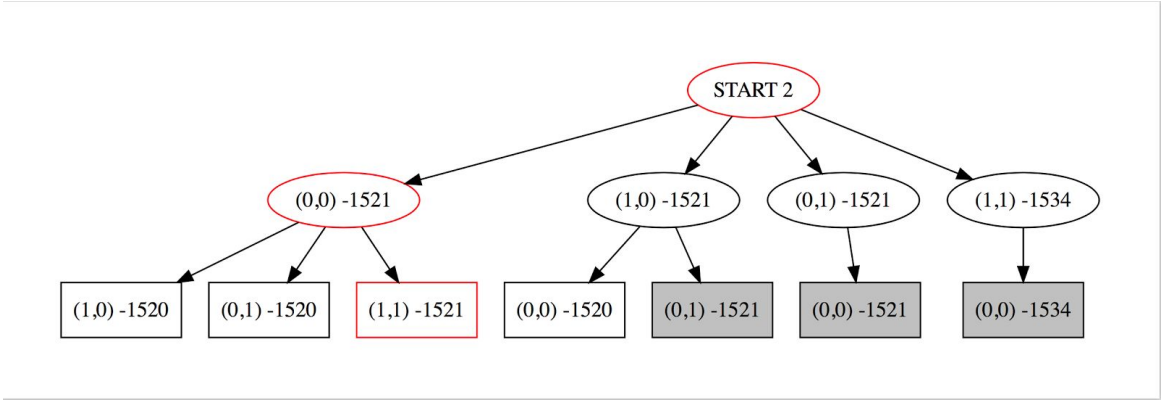
El iterador que utiliza el minimax para ver todos los vecinos del Board actual funciona de la siguiente manera: en vez de crear todas las matrices, donde cada una representa un cambio al tablero, se devuelve un iterador que al llamar a su función *next*() crea el tablero siguiente en el momento. Esto ahorra mucha complejidad espacial ya que nunca tiene más tableros en memoria que la profundidad deseada.

Hashing: Zobrist

Nuestro iterator, que toma tableros en su estado actual, devuelve los próximos tableros para ser analizados por nuestra heurística. Para poder guardar todos ellos y no tener repetidos decidimos utilizar el HashSet provisto por las librerías de Java. Para evitar tableros equivalentes (traspuestos) implementamos el hashing ideado por **Albert Lindsey Zobrist**. Este cuenta con una tabla de números elegidos al azar para cada posición del tablero, por cada posibilidad en una posición: negro, blanco o vacío. Cuando se necesita calcular el *hashcode* de un tablero, basta con utilizar la operación XOR con un valor inicial (en nuestro caso, el cero) por cada uno de los lugares modificados del tablero. Esta técnica es utilizada también a la hora de introducir un cambio y actualizar el valor del *hashcode*. Esta técnica no solo garantiza que las matrices traspuestas mapean al mismo *hashcode*, sino que, dependiendo de qué tan grande sean los números utilizados por la tabla de números elegidos al azar, las posibilidades de que se produzcan colisiones de *hash* son disminuidas drásticamente.

Árbol en formato dot

Debido a que el árbol en formato dot requiere que esté todo declarado antes de construirlo, decidimos utilizar una estructura auxilia, *DotNode,* que es una implementación de un árbol común: cada nodo tiene una lista de hijos e información sobre si es máximo, mínimo, podado, su heurística, su posición en el tablero y nivel de minimax. Esto nos da la posibilidad de utilizar una simple función recursiva para crear el archivo entero



Árbol generado con: -file test.txt -player 2 -depth 2 -prune -tree

Algoritmos del *BoardManager*

Implementamos ciertos algoritmos de relevancia dentro de *BoardManager*, la clase que se encarga de operar sobre el *Board*, que se basan en un algoritmo central: el *floodfill.* Este algoritmo de relleno por difusión determina el área formada por elementos contiguos en una matriz multidimensional, utilizado en la herramienta bote de pintura de programas de dibujo para determinar qué partes de un mapa de bits se van a rellenar de un color, y en juegos como el Buscaminas, Puyo Puyo, Lumines y Magical Drop para determinar qué piezas pueden retirarse o seleccionarse. (*Wikipedia).* Para adaptar el mismo a nuestros objetivos implementamos la clase *Node* dentro de *BoardManager* que representa una posición y un color según el estado de la primera. Además, desarrollamos un método que extrapola la matriz del *Board* como una matriz equivalente con *Nodes,* con el objetivo principal de lograr pintar áreas congruentes. Utilizamos *floodfill* principalmente para verificar si una jugada captura prisioneros, si una jugada es suicidio y para calcular territorio total capturado al finalizar el juego.

Minimax con límite de profundidad y de tiempo. Poda alfa-beta. Clase MMTree.

En base a lo visto en clase sobre *minimax,* avanzamos sobre los requisitos pedidos: límites de profundidad, tiempo y habilitación de poda alfa-beta. En cuestión a lo primero, no hubo modificaciones respecto a la implementación neta del minimax, pues consta de la profundidad como condición de corte, salvo la inclusión de un *path* de tableros visitados para no entrar en ciclos y el pasaje de *hashcode* para el uso del iterador de tableros. En base al límite de tiempo, se encapsula lo anterior con un método de la librería Guava de Google que limita la ejecución del método por tiempo. De no terminar en el tiempo estipulado, devuelve el mejor tablero obtenido hasta el momento. Por último, la poda alfa-beta fue implementada como fue visto en clase, con parámetros *alpha* y *beta* y la condición de corte correspondiente. En los resultados sobre el final del informe se puede apreciar su optimización. Todo lo explicitado anteriormente se encuentra como comportamiento de la clase *MMTree.*

Heuristicas

Siendo que el Go es un juego muy complejo en su funcionamiento y en la forma de computar quién tiene ventaja en cada tablero, tuvimos que dividir la heurística en varias partes:

- Libertades: heurística que calcula cuántos grados de libertad tiene cada ficha.
- Ojos: calcula cuántos "ojos". Un ojo es una figura ventajosa en Go.
- Prisioneros: calcula la cantidad de prisioneros recién adquiridos.
- Triángulos malos: calcula la cantidad de triángulos. Un triángulo es una figura perjudicial en Go.
- Knight Moves: calcula la cantidad de *Knight moves*. Un *Knight move* es una figura ventajosa en Go.
- Tiger's Mouth: calcula la cantidad de *Tiger's Mouth*. Un *Tiger's Mouth* es una figura ventajosa en Go.
- Bamboo Joint: calcula la cantidad de *Bamboo Joints*. Un *Bamboo Joint* es una figura ventajosa en Go.
- Dumpling: calcula la cantidad de *Dumplings*. Un *Dumpling* es una figura ventajosa en Go.

Cada una de estas heurísticas son sumadas de la siguiente manera:

$$HeurísticaTotal \mathrel{+}= V * (HeurísticaX(Blanco) * PP - HeurísticaX(Negro) * PN$$

Siendo V un 1 si es ventajosa la heurística o -1 si es perjudicial.

PP es el peso que multiplica la heurística a favor de la computadora.

PN es el peso que multiplica la heurística en contra de la computadora.

En muchos casos PP y PN son iguales ya que es tan ventajoso tener esa configuración como es perjudicial que el oponente la tenga.

Problemas encontrados durante el desarrollo y decisiones tomadas

Uno de los primeros problemas que surgió fue el de guardar los distintos tableros en nuestro HashSet. Debido al hecho de que el hashcode de matrices es distinto para tableros que serían considerados iguales para el ojo humano, nos vimos obligados a encontrar una forma de hacer que éstos fuesen mapeados al mismo tablero. Luego de investigar, dimos con el concepto de Zobrist Hashing. La idea del inventor *Albert Lindsey Zobrist* fue de utilizar hashing de manera que estos tableros traspuestos dieran el mismo hashcode. Al implementar su algoritmo luego de varios intentos fallidos, logramos resolver el problema de tableros repetidos.

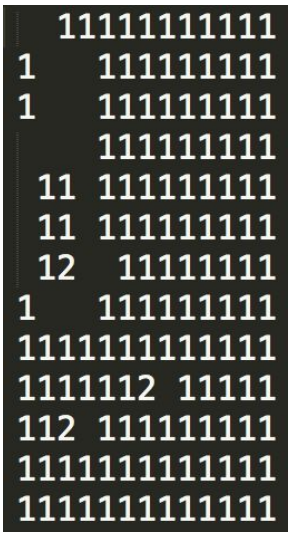
También nos encontramos con problemas al momento de contabilizar el puntaje de los jugadores debido al permanente cambio del tablero. Es así que investigando logramos sacar provecho del algoritmo *floodfill* anteriormente explicitado.

A la hora de tomar mediciones sobre los tiempos nos encontramos con dificultades para medir casos con profundidades de complejidad temporal alta, ya que decidimos terminar la medición por la longitud que está estaba acumulando (más de 20 minutos). Esto podría ser corregido el futuro con computadoras de mayor procesamiento, o mayor paciencia del equipo.

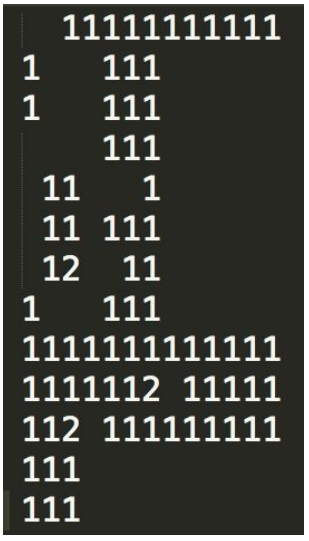
Por último, pero no de menor relevancia, fue compleja la decisión de cómo definir una heurística que influencie en el nivel del juego. Al principio contábamos con heurísticas simples o triviales, pero a lo largo del desarrollo del juego y de sucesivas pruebas e investigaciones, conseguimos una heurística general basada en figuras lo suficientemente inteligente para agregar competitividad al juego.

Tablas de comparación de tiempos y resultados

He aquí nuestra tabla de comparaciones. Los valores fueron tomados utilizando los modos *-depth* entre 1 y 5. Los siguientes valores fueron promediados con alrededor de 20 iteraciones. Los datos están mostrados en escala logarítmica.
NOTA: la toma de datos del tablero 2 fue realizada hasta una *depth* de 3 inclusive, dado que el tiempo requerido fue demasiado en el momento que medimos.



Tablero 1.



Tablero 2.

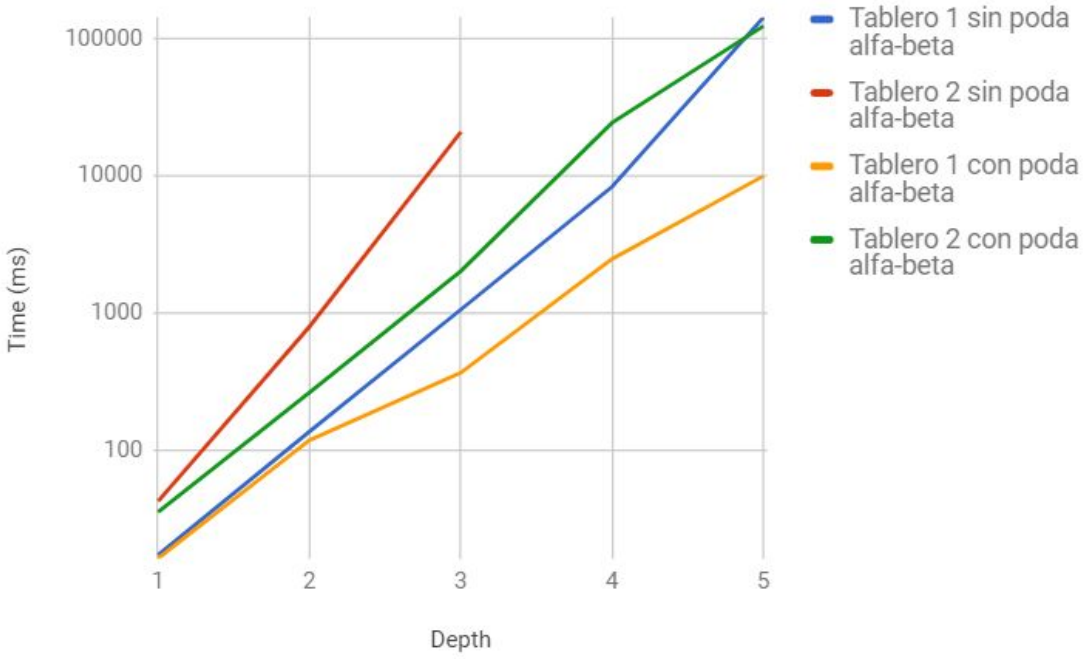


Tabla comparativa de rendimiento.

	maxDepth - no prune (ms)		maxDepth - with prune (ms)	
	Tablero 1 sin poda alfa-beta	Tablero 2 sin poda alfa-beta	Tablero 1 con poda alfa-beta	Tablero 2 con poda alfa-beta
1	17	42	16	35
2	135	790	117	260
3	1050	20813	363	2007
4	8303		2466	24321
5	142875		9927	123429

Tabla de resultados con diferentes configuraciones.

Conclusiones

Sobre los resultados obtenidos por las diferentes configuraciones, observamos que en ambos tableros aplicar la poda es beneficioso en cuanto al tiempo que demora en responder. Se puede observar que no se registraron mediciones para el tablero 2 sin poda, pues decidimos que era demasiado esperar una respuesta en un tiempo prudente (excede los 20 minutos).

Una de las problemáticas más determinantes fue comprender a fondo el juego, incluyendo qué era bueno y qué era malo, entendiendo la complejidad considerable del Go. Notemos que aunque las reglas del Go son simples, su estrategia es extremadamente compleja porque involucra equilibrar requisitos que pueden ser contradictorios; ubicar piedras cercanas ayuda en su conectividad pero tenerlas distantes mejora su influencia a lo largo del tablero.