ALGORITMO MINIMAX

Estructura de Datos y Algoritmos

Informe de Desarrollo

GRUPO

53852 Vera, J. Sebastián

López, Noelia B.

ÍNDICE

Tabla de contenido

1. Introducción

El trabajo práctico especial consiste en implementar el juego *La Fuga del Rey*, con todo lo que eso significa, haciendo particular énfasis en la creación de una buena inteligencia del oponente, la 'computadora' que juega el papel del enemigo. Para dicho propósito se propuso la implementación de un algoritmo del tipo *minimax*.

El objetivo del presente informe es detallar el proceso de desarrollo e implementación del juego, incluyendo el algoritmo. Se presenta, además, información sobre la evolución del mismo. Es importante remarcar que, para obtener mejores resultados, se probaron distintas heurísticas y se compararon los resultados obtenidos entre ellas, a fin de obtener un mejor resultado; teniendo en cuenta el tiempo que tarda la computadora en realizar el movimiento, y la eficacia del mismo.

A su vez, se presentan los puntos flojos del juego, cuyas explicaciones –muchas veces- escapan al entendimiento; como también los puntos favorables del mismo, y las razones por las cuales se optó elegir la efectividad de los últimos ante los detalles desfavorables de los primeros.

2. Estructuras

Se presentan, a continuación, las clases que jugaron el papel de estructuras elegidas para la implementación del juego.

2.1. Box

Para implementar los casilleros del tablero en los que se guardarán la pieza (de tipo Piece), el valor que dicha celda contiene (a ser explicado más adelante por la implementación de las heurísticas) y el valor entero de la fila y la columna correspondiente.

Dicho en otras palabras, la implementación de la clase Box, representaría el lugar "físico" de un casillero en el tablero.

De acá es de donde se obtiene la información necesaria para situar los castillos y el trono, ya que estos no son representados por ninguna pieza en particular.

2.2. Piece

La clase Piece, representa la pieza, propiamente dicha, que se encuentra en un casillero. Para ello se utilizó una variable de instancia de tipo *char* la cual guarda un caracter que representa el tipo de pieza o ficha que se le atribuye al *guardia, rey*, o al *enemigo* (G, K, N respectivamente).

En el caso de que un casillero se encuentre vacío, que no exista una ficha que ocupe ese lugar, el casillero se llena una pieza representada por el caracter del 0.

De esta manera, para saber si un casillero está o no vacío, solo basta preguntarle si su pieza es representada con el caracter 0 o no (método *isEmpty* dentro de la clase *Box*).

A su vez, cada pieza tiene un jugador representado con la clase Player.

2.3. Player

La única función de la clase *player* es aproximar la representación de un jugador. Para ellos, cada jugador tiene un entero que varía entre 1 o 2 significando a que jugador le corresponde la pieza, si al usuario o a la computadora, respectivamente.

Es de esta manera que se obtiene una mayor aproximación a la programación orientada a objetos, para poder aprovechar, en posibles mejoras futuras, mayores características presentes en cada una de las implementaciones. A su vez, era idea original, colocar mayor cantidad de atributos dentro de cada jugador (como por ejemplo la cantidad de piezas que dispone, o la cantidad de movimientos que realizó), que podrían ser utilizados en una futura versión.

2.4. Board

Para representar el tablero se utilizó una clase Board que contiene una matriz de Box con el fin de representar "físicamente" el tablero en si, como composición de casillos –

composición de Box-. Cada tablero cuenta también con un valor heurístico y las dimensiones del mismo.

De esta manera, cada movimiento (representado a fines puramente prácticos por la clase *Move*), validación del mismo, u obtención de los movimientos posibles dado un casillero; se ejecutan desde el tablero. Lo cual tiene sentido, ya que sin un tablero no tendría que ser posible realizar un movimiento, y una pieza por sí sola no puede realizar un movimiento que la participe dentro de un juego.

Es importante remarcar que el valor heurístico de un tablero es asignado solo cuando se solicitan los movimientos posibles de un casillero (previa validación de si en dicho casillero se encuentra una ficha), ya que cuando un movimiento es realizado manualmente, poco importa el valor del tablero al que se concluye.

Visto y considerado que son muy pocos los casos en los que se encuentran dos movimientos posibles distintos que poseen un mismo valor heurístico gracias a la simetría del tablero, se optó por que el algoritmo *minimax* no verificara simetría de tableros, ya que se llevaría igual o mayor tiempo y memoria rotar tableros que analizar los pocos casos de más que hubieran sido evitados por realizar dicho procedimiento.

En esta clase se encuentran, además de los métodos que permiten el correcto funcionamiento del juego, métodos que verifican si corresponde que una pieza sea 'comida' por otra luego de un movimiento (se reconoce que existen casos muy particulares y puntuales en los que la pieza del Rey debería ser comida, pero no sucede, cuya explicación escapa al entendimiento de quienes suscriben), y métodos que realizan copias auxiliares tanto del tablero actual como del tablero luego de un posible movimiento, para poder ser analizado en particular, con el objetivo de colocar el valor heurístico correspondiente.

2.5. Game

Para representar el estado del juego se utilizó la clase Game que contiene un tablero del tipo *Board*, y guarda un registro la información necesaria para el correcto funcionamiento y desarrollo del juego, como ser el turno actual y los parámetros recibidos, a ser utilizados al llamar a *minimax*.

A su vez, dicha clase es la responsable de los métodos que permiten el inicio de la interfaz gráfica (si así fuera solicitado), el guardado de una partica en curso, la continuación de una partida ya comenzada recibida por parámetro mediante un archivo de texto.

Como para obtener los posibles movimientos, es necesario llamar a la función *Move* (que dentro de esta clase, ejecutara la función *Move* dentro del board correspondiente al estado de la partida en dicho momento en particular), que inevitablemente realizará un movimiento, devolviendo el tablero – del tipo *Board* – resultante; esta clase cuenta con los métodos *copy* y *duplicate* que realizan una nueva instancia como copia del estado del juego actual, para poder realizar todos los posibles movimientos sobre un estado de juego ficticio, sin afectar el estado de juego actual, y así poder mantener el tablero.

2.6. Move

Se creó una clase Move con el fin de representar un movimiento en el tablero, esta clase contiene las posiciones de origen y destino (guardada como enteros que representan la fila y la columna origen como también la fila y columna destino), así como también el valor heurístico del movimiento (que representa el valor heurístico al del tablero al que se puede llegar con ese movimiento), el cual se asigna durante la obtención de movimientos posibles desde una posición válida.

2.7. pcBehave

La clase *pcBehave* dentro del package *minimax* contiene el método estático *minimax* que llama al método recursivo *mm2* que representa en sí, el método minimax.

Debido a que el hecho de realizar un método para cada caso distinto de minimax (con o sin poda, restringiendo en tiempo o en profundidad), se optó por realizar una pregunta más en cada paso recursivo, lo que convertiría la complejidad de N a 2N sin modificar el orden el algoritmo, para consultar si corresponde o no tener en cuenta una poda.

Para diferenciar entre los casos en los que se restringe la profundidad de aquellos en los cuales se restringe el tiempo, se optó por pasarle como parámetro un depth fijo (elegido el 2, por análisis costo temporal / funcionalidad) a menos que se indique lo contrario, en cuyo caso se pasaría como parámetro un número muy alto de cantidad de milisegundos durante los cuales el algoritmo tuviera la posibilidad de ser ejecutado.

3. Algoritmo

3.1. Explicación general

Para las jugadas de la computadora, el programa utiliza el algoritmo minimax presente en la clase pcBehave.

Como visto en clase, el algoritmo del tipo *minimax* se comporta y ejecuta de la siguiente manera.

El mismo es un algoritmo que se utiliza en situaciones en donde hay dos participantes, cada uno por turno debe realizar una acción o movimiento, y cada uno tiene un objetivo, estos son tales que mientras uno se acerca a su objetivo el otro se aleja, además los participantes tienen información del entorno, conocen sus acciones, las de su contrincante y los efectos de las mismas en el juego.

El algoritmo analiza todas las posibles combinaciones de movimientos (cadena de sucesivos movimientos) y elige aquel movimiento que lo lleve más próximo a su objetivo, contemplando que su oponente elegirá aquellos movimientos que lo acerquen a su objetivo. Al participante que va a ejecutar el movimiento se lo denomina MAX, mientras que a su oponente se lo denomina MIN (por la forma de elegir los mejores movimientos).

Dado que la cantidad de combinaciones de movimientos es muy grande, comúnmente, mediante un valor heurístico se ponderan los estados y no se analizan todas las combinaciones sino que se pone algún tipo de límite (denominadas podas) y se utiliza ese valor heurístico para decidir el movimiento.

3.2. Límites

Como ya se mencionó anteriormente, debido a que la cantidad de combinaciones posibles es muy grande se requiere establecer un límite de corte para la profundidad del algoritmo.

Se exigió que hubiese tanto un límite por tiempo, como un límite por profundidad, los cuáles se explican de manera más detallada a continuación.

3.2.1. Límite por profundidad

El límite por profundidad consiste en llamar al minimax pasándole por parámetro cuantos nodos más en profundidad puede entrar. Una vez que el nivel de profundidad alcanza 0, el paso recursivo retorna.

3.2.2. Límite por tiempo

El límite por tiempo consiste en pasar como parámetro al algoritmo minimax el tiempo límite que posee para realizar una jugada. A ese tiempo se le suma el tiempo actual del sistema, una vez que el tiempo del sistema supera dicho límite, el algoritmo se

Por cuestiones de eficiencia, el algoritmo no irá más de dos nodos en profundidad. Para lograr una aproximación al tipo de recorrido BFS de acuerdo al tiempo que se le permite ser ejecutado.

3.3. Poda

Como el árbol que genera el minimax es muy grande, es muy útil hacer podas sobre el mismo para mejorar su velocidad y reducir ampliamente la cantidad de estados que serán evaluados.

3.3.1. Poda alfa-beta

Esta poda utiliza los valores heurísticos ya calculados y la propiedad de ser MAX o MIN para evitar analizar estados en los cuales ya sabemos de antemano que no serán tenidos en consideración. Para ello, los nodos reciben el "mejor" valor de sus hermanos procesados y podan si ya saben que su valor no "superará" este mismo (las definiciones de mejor y superará dependerán si es MAX o MIN).

3.4. Heurística

Si bien el código del algoritmo *minimax* no depende en forma directa de las reglas del juego, ya que puede ser implementado en varios escenarios similares, se requiere de un valor heurístico para la elección de la jugada a realizar, así como también es de vital importancia para el uso de la poda alfa-beta.

Si bien sólo una de las heurísticas es la utilizada para evaluar cada tablero (valor que será utilizado por el algoritmo) a lo largo del trabajo, fueron surgiendo distintas maneras de asignar valores a los posibles tableros existentes.

3.4.1. Heurística 1

La primer heurística que surgió, fue la de colocarle a cada casillero como valor un proporcional a la cantidad de casilleros que los distancian en línea recta hacia alguno de los bordes. De esta manera cada tablero tendría como valor heurístico el valor del casillero al que se movió la última ficha.

El orden de asignación de este valor fue muy bajo, ya que al crear un casillero, este se creaba con su correspondiente valor, y para trasladar el valor de una casilla a un tablero, sólo hacía falta observar el valor del casillero destino en un movimiento.

Si bien, la heurística no resultó ser la más efectiva, ya que el algoritmo únicamente buscaría mantener al rey en el centro del tablero, no era una idea del todo errada, ya que tenía como objetivo evitar que el jugador ganara, independientemente de si podría hacerlo perder.

3.4.2. Heurística 2

Para la mejora de la primera heurística se pensó en variables externas a la cantidad de casilleros que separan una posición de alguna de las esquinas, y pasó a considerarse, además, si en algún movimiento sería posible posicionarse en torno al rey, bloquear alguna salida, eliminar algún guardia, o incluso eliminar al mismo rey.

Para poder llevar a cabo dicha heurística, mejora de la primera, se optó por sumarle (o restarle, dependiendo de quién sea turno jugar) al valor del último casillero allegado, la cantidad de enemigos que se lograba eliminar, si alguno de aquellos era el rey, o cuantos enemigos alrededor se lograba obtener (lo que sería, en un futuro una cantidad de posibles eliminados).

3.5. Árbol de llamadas

Dado que ya habiamos implementado un programa parecido para graficar arboles en una de las guias de la materia. Utilizamos parte de ese codigo, o mejor dicho la logica para desarrollar el arbol de llamadas de este juego.

Basicamente lo que hacemos es graficar los nodos cada vez que se generan en el algoritmo minimax.

El encargado de adjudicarle el color y la forma es minimax, ya que este algoritmo es el que tiene el absoluto control sobre el arbol que se va generando, nuestro graficador solo se encarga de graficar sin modificar ninguna variable.

4. Evaluaciones

Tabla 1

Profundidad	**2**		**3**	
Poda	No	Si	No	Si
Dimensión	Promedio de tiempo en ms			
7	59,8	21,8	8660	354,5
9	1429,8	37,8	7039,7	1885,3
11	7689	250	6241,9	2940,8
17	5810,8	47,4	7316,3	3013,3

En la tabla 1 se puede observar la diferencia de tiempos que hubo en el algoritmo *minimax* de acuerdo a la profundidad aceptada y a la existencia o no de poda. Es

importante remarcar que para obtener los resultados se analizaron los tiempos de 10 jugadas de 5 juegos.

A cada ejecución del algoritmo se le pidió que cortara luego de haber estado ejecutándose durante 4 segundos, es por ello que, y teniendo la tabla como muestra, si el usuario no especificara una profundidad, utilizaríamos profundidad 2, ya que es más probable mantenerse dentro de los parámetros especificados de esta manera.

También podemos observar que las podas no solo permiten analizar el nivel completo dentro del periodo tiempo especificado – lo que significaría la obtención de un mejor movimiento – sino que además, logra mejorar el tiempo de ejecución considerablemente.

Tabla2

-Con Poda- Profundidad	**4**	**5**	**6**	**7**
Dimensión		Promedio de tiempo en ms		
7	1943,8	1987,3	2538,4	2603,3

Como se puede observar en la tabla 2, la utilización de podas permite que el algoritmo vaya en mayor profundidad, incluso sin superar el tiempo máximo establecido.

5. Conclusión

Dado a que era un juego que ya se habia desarrolado en una materia correlativa de Estructuras de Datos y Algoritmos nos fue dificil para nosotros aprender el juego. Era un juego totalmente nuevo y nos costo incluso aprender a jugarlo.

Los primeros días fueron dedicados a aprender el la estrategia del juego, para luego centrarnos a diseñarlo.

Nos encontramos con varios problemas durante el desarrollo, el primero y el principal es la implementacion del algoritmo minimax. Lo desarrollamos al menos 2 veces y siempre nos devolvia el mismo tablero, o el primer movimiento que encontraba. Despues de dos dias de arduo trabajo, nos dimos cuenta de nuestro error y lo corregimos.

Otro obstaculo que nos encontramos fue la parte grafica del trabajo. Al principio no sabiamos bien como diseñarlo, si botones individuales, con un tabla o con JPanels individuales. Probamos varias opciones, al principio botones individuales. Todo parecía andar bien porque las pruebas las hacíamos siempre en base al mismo tablero, que era chico, pero al probar con unos mas grandes los botones se desacomodaban y el tablero no servía. Ahí nos dimos cuenta que era momento de cambiar la implementación y optamos por una matriz de Jpanels, que ya de al iniciarse el juego le designábamos su tamaño.

El desarrollo de este TP nos brindo un panorama mas amplio de programación. Dado que la mayoría de las veces nosotros programamos para jugar y/o interactuar con la consola, y tener que diseñar el desarrollo de este juego nos pareció quizás un poco difícil. Lo mismo podría decirse del algoritmo minimax, nos aporto mucho conocimientos y cosas que antes no sabíamos, como el saber como están diseñados los juegos y el comportamiento de la computadora.