INSTITUTO TECNOLÓGICO DE BUENOS AIRES

ALGORITMO MINIMAX

Estructura de Datos y Algoritmos

Informe de Desarrollo

GRUPO 1

53202 Cavo, María Victoria

53214 Fraga, Matías Agustín

54377 Vazquez, Diego

ÍNDICE

1. Introducción

Se propuso como trabajo práctico especial la implementación del algoritmo ***minimax*** para el juego ***La Fuga del Rey.***

Se presenta a continuación un informe detallado con todo el proceso de desarrollo e implementación de dicho algoritmo, como fue evolucionando y mejorando su funcionamiento, así como también su relación con las heurísticas que fueron surgiendo iteración tras iteración. Además se evaluó si dicho algoritmo cumple los requisitos necesarios para la implementación del juego, cuales son sus puntos favorables y en qué circunstancias no presenta soluciones claras.

2. Estructuras

A continuación se presentan las estructuras elegidas para la implementación del trabajo práctico, tanto para el algoritmo ***minimax*** como para la representación del juego, contrastando las decisiones elegidas con otras opciones de implementación que fueron discutidas a lo largo del desarrollo del trabajo.

2.1. Piece

Para representar los diferentes tipos de pieza del juego (*Throne*, *Fighter*, *King*, *Tower* y *Empty*) se decidió utilizar una clase abstracta *Piece* que exige la implementación de algunos métodos comunes a toda pieza (*canGetKilled*, *canBeStepBy* y *canBeJumpBy*). La ventaja que acarrea esta decisión es que se puede abstraer del tipo de pieza que se está manejando ya que cada pieza responde de manera propia, además como las piezas no tienen información referente al estado del juego o del tablero, pueden ser *singleton* lo cual ahorra mucha memoria (ya que no es necesario mantener muchas instancias de cada pieza), sobre todo dentro del algoritmo ***minimax***, dichas ventajas hacen buen uso del paradigma objetos. Se evaluó previamente la posibilidad de no tener una pieza *Empty* para representar una casilla vacía y en vez de ello utilizar un *null,* pero se descartó esa posibilidad debido a que al ser *singleton,* solo se guardaría una referencia a la pieza, además de ser una práctica más imperativa. La principal desventaja de esta decisión (tener una clase abstracta *Piece*) es que los comportamientos de las piezas dependen, en algunos casos (principalmente el rey), de la pieza con la que interactúan, y es necesario ver que tipo de pieza es. Lo cual rompe con el paradigma de objetos.

Dadas las reglas del juego, se decidió tomar una convención respecto a los propietarios de las piezas (*owner*), usando la representación binaria de un número entero para decidir si dos piezas son contrincantes o no. Así, se tomó la decisión que las piezas con *owner* (en binario) que terminen en 1 serán aliados del rey y los que su anteúltimo dígito sea 1 serán aliados de los enemigos del rey. Y para diferenciar los guardias y el rey del trono se decidió usar un valor de owner distinto para el trono pero que mantenga la convención tomada. De esta manera, para ver si dos piezas son enemigas basta utilizar el operador binario *and* (&) entre ellos y ver si el resultado es 0.

Por lo tanto los valores de owner tomados son los siguientes:

- Torres: 0 — 00000000b
- Guardias y Rey: 1 — 00000001b
- Enemigos del rey: 2 — 00000010b
- Casilla vacia: 3 — 00000011b
- Trono: 5 — 00000101b

2.2. Board

Para representar el tablero se utilizó una clase *Board* que contiene una matriz de *Pieces* con el fin de representar cada casilla del tablero. Además esta clase posee información como la cantidad de piezas enemigas, guardias y la posición del rey para disminuir la complejidad de los algoritmos heurísticos.

Con la finalidad de aplicar la poda por tableros iguales al ser rotados o ser simétricos entre sí, se incorporaron métodos que, dada una posición devuelvan la posición que resulta de aplicar la simetría o la rotación respectiva. También fue necesario un método (*symmetries*) que devuelve qué tipo de simetrías presenta el tablero , dicho método devuelve un número que al ser representado en binario indica en cada posición con un 0 o 1 si posee una simetría específica o no.

La máscara de bits que representa lo mencionado anteriormente es la siguiente:

- Rotación de 90 grados hacia la derecha: 00000001b
- Rotación de 180 grados: 00000010b
- Simetría respecto a "*x*": 00000100b
- Simetría respecto a "*y*": 00001000b
- Simetría respecto a la diagonal principal: 00010000b
- Simetría respecto a la diagonal secundaria: 00100000b

Por otra parte, en esta clase se encuentran los métodos *value1*, *value2* y *value3* que retornan el valor heurístico del tablero, es decir, representan las distintas heurísticas elegidas, que más adelante serán detalladas.

Esta clase cuenta con un método que devuelve una copia de la instancia para poder utilizarla en el ***minimax***.

2.3. Minimax

Se cuenta con una clase *Minimax* la cual contiene las diferentes implementaciones del algoritmo ***minimax*** utilizadas. Los métodos más importantes son los métodos wrappers, *minimaxByTime* y *minimaxByDepth*, que llaman al algoritmo minimax. De este último existen dos implementaciones, el método llamado *minimax2* que implementa la poda por rotaciones y simetrías, y *minimax* que no implementa dicha poda. Más adelante se evaluará cuál de estos se cree que es mejor. Se tienen estas dos implementaciones del algoritmo para simplificar la lectura del código y a pesar de tener código repetido se prefirió tener un código claro.

Cabe aclarar que también se discutió sobre la cantidad de métodos que tendría esta clase, si hacer un método para el uso de la poda alfa-beta, otro para crear el árbol de llamadas o bien tener un único método que contemple todos los casos. Se llegó a la conclusión de que si se separa en los distintos casos habría código repetido ya que en esencia el algoritmo utilizado es el mismo. Y por lo tanto se decidió hacer un único método que contemple todos los casos.

2.4. 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, así como también el valor heurístico del movimiento (que representa el mejor valor heurístico al que se puede llegar con ese movimiento en un tablero específico) el cual se asigna en el algoritmo ***minimax***. Dado que el valor se genera a partir de los valores heurísticos del tablero sin conocimiento del movimiento fue necesario habilitar la creación de un *Move* que tenga valor pero no tenga posiciones de origen ni destino (se discutió esta decisión ya que existía la posibilidad de hacer dos métodos ***minimax***, uno que devuelva el movimiento y otro que devuelve los valores heurísticos, pero esta solución repetía el código casi en su totalidad y por lo tanto fue descartada). Otra situación a tener en consideración es que el método minimax devuelve *null* cuando corta por tiempo, por lo que era necesario devolver un Move que sea distinto de *null* para indicar que no hay posibles movimientos, por estas razones se creó una variable isValid que indica si el movimiento es válido (si se crea con origen y destino) o no (si solo tiene valor). Si bien la solución tiene cierto carácter imperativo, no siempre es posible modelar algoritmos manteniendo el paradigma de objetos ya que apegarse mucho al mismo puede resultar perjudicial para el proyecto.

2.5. Game

Para representar el estado del juego se utilizó la clase *Game* que contiene una instancia de un tablero (*Board*), y guarda un registro de quién es el turno. La misma almacena, además, los parámetros con los cuales se ejecutará el algoritmo ***minimax*** (*depth*, *maxtime* y *prune*). Posee métodos como *start* que abre una interfaz gráfica para jugar una partida contra la computadora, *update* que es un método que complementa a *start* para que se realicen los movimientos de la computadora y se verifique si la partida ha concluído, *getBestNextMove* que devuelve cuál es el mejor movimiento posible. Tanto para jugar la partida como para devolver el mejor movimiento, se utiliza el algoritmo ***minimax*** con los parámetros recibidos.

Esta clase cuenta con un método *copy* que devuelve una copia de la instancia para poder utilizarla en el ***minimax***.

3. Algoritmo

En el siguiente apartado se explicarán en detalle las implementaciones de las diferentes versiones del algoritmo ***minimax.***

3.1. Explicación general

Para las jugadas de la computadora o para devolver la siguiente mejor jugada, el programa utiliza el algoritmo ***minimax***.

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

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). A continuación se puede observar en la ***figura 1*** un pseudocódigo del algoritmo.

```
minimax(estado){
        if(estado.estaTerminado()){
                if(estado.gane())
                        return gane;
                if(estado.empate())
                        return empate;
                if(estado.perdi())
                        return perdi;
        }
        hijos=estado.calcularHijos();
        for(hijo:hijos){
                jugada=minimax(hijo);
                if(jugada>mejorJugada)
                        mejorJugada=jugada;

        }
        return mejorJugada;
}
```

Figura 1: Pseudocódigo de minimax.

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 y se utiliza ese valor heurístico para decidir el movimiento, como se observa en el ejemplo de la ***figura 2***.

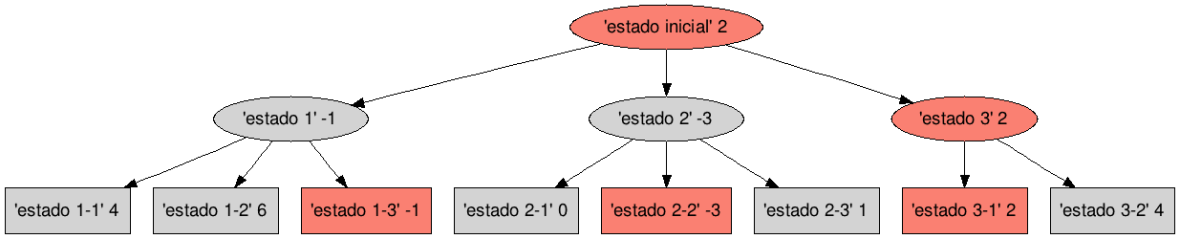

Figura 2: Ejemplo de árbol de llamadas de minimax con límite de profundidad 2, en rojo se encuentran los estados elegidos por cada participante (1er nivel elige un jugador (max) y 2do nivel elige el otro (min)). Además los nodos en forma rectangular son aquellos donde se calculó el valor heurístico (número que acompaña al nombre del estado), y los de forma elíptica toman el valor heurístico del "mejor" de sus hijos (la definición de mejor depende del nivel donde se esté parado).

En la implementación del método ***minimax,*** se realiza un recorrido sobre todo el tablero y para cada posición donde haya una pieza del jugador que tiene que mover se llama a un método de Board llamado *getPossibleMovesFrom* que recibe la posición de dicha pieza y devuelve una lista con sus posibles movimientos, de esta forma se calculan todos los posibles movimientos para un estado. Es para considerar que mientras se recorre el tablero también se procesan los estados, de esta forma se reduce el uso memoria ya se que solo se guardan los posibles movimientos de la pieza que se esté procesando.

En adición, con la finalidad de no repetir código, cuando se reciben los valores de los hijos (del estado que se está procesando), se los multiplica por -1. De esta forma se puede usar el mismo código siempre, ya sea MAX o MIN. Por ejemplo, si se está procesando MIN, es lo mismo tomar el mínimo de los valores de los hijos que tomar el máximo de los opuestos de los mismos y devolver el opuesto de este. Así mismo cuando se procesa MAX, el valor que se recibe de MIN es el verdadero valor, y no hay problemas.

Se utilizaron los valores *Integer.MIN_VALUE+1 y Integer.MAX_VALUE* para representar los valores de los tableros en los que se perdió (-inf) o se ganó (+inf) respectivamente.

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.

En el pseudocódigo de la ***figura 3*** explica de manera resumida la metodología de imponer un límite.

```
minimax(estado,limite){
        if(limite.alcanzado() || estado.estaTerminado()){
                return estado.valorHeuristico();
        }
        hijos=estado.calcularHijos();
        for(hijo:hijos){
                jugada=minimax(hijo,nuevoLimite);
                if(jugada>mejorJugada)
                        mejorJugada=jugada;
        }
        return mejorJugada;
}
```

Figura 3: Pseudocódigo del minimax acotado por un límite.

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 la profundidad que debe alcanzar y la profundidad actual. Una vez que el nivel que debe alcanzar y el actual coinciden, se alcanza el límite fijado y se devuelve el valor heurístico.

Al implementarlo se utilizó un método *wrapper* llamado *minimaxByDepth* por prolijidad y simpleza del código, el cual llama al método de *minimax* que por su implementación recibe como parámetro el tiempo límite para realizar una jugada (ya que no se quiso repetir código y tener dos métodos distinto para límite por profundidad y límite por tiempo), con el valor de

Long.MAX_VALUE en este tipo de límite. De esta forma, se puede utilizar el código sin modificaciones. Se pensó también que se podría pasar por como valor de tiempo límite el valor *null* en vez de Long.MAX_VALUE pero esto llevaría a tener que hacer checkeo internos en el minimax y por lo tanto no se eligió esta opción.

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 interrumpe y se devuelve la mejor jugada encontrada hasta el momento.

Para implementar la búsqueda de dicha jugada, se utilizó un método *wrapper* llamado *minimaxByTime* quien realizará sucesivas llamadas al método ***minimax***, e irá aumentando en una unidad la profundidad a la que llegará (siempre comenzando desde la profundidad mínima).

Si bien la implementación repite estados y fue motivo de debate, se llegó a la conclusión de implementarla ya que otro tipo de recorrido (BFS) resultaría muy pesado de mantener en memoria e impediría la posibilidad de utilizar podas y como los primeros niveles a evaluar por el algoritmo cuestan muy poco tiempo (como se verá en los próximos apartados), se consideró aceptable el mecanismo empleado.

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).

Por ejemplo, si se está procesando un nodo que decide por el máximo y se recibe el valor 5, si se encuentra un hijo con valor 5 o más ya se puede podar, ya que el valor del nodo que se esta procesando será mayor o igual a 5, entonces

su padre (que elige por el mínimo) elegirá a su hermano. Así mismo, si el nodo que se está procesando decide por el mínimo y recibe el valor 5, podará al encontrar un valor menor o igual a 5. A continuación, en la ***figura 4*** se explica de manera breve el funcionamiento del algoritmo con poda.

En la implementación de la poda, como se toma el valor opuesto al cambiar de nivel, basta con pasar como poda el valor negativo de la mejor jugada actual y de esta forma se puede utilizar el mismo código tanto para MAX como para MIN. Es importante aclarar que para activar esta poda se debe pasar como parámetro la poda inicial (como no se tiene un valor para la poda inicial, se toma el máximo valor entero, *Integer.MAX_VALUE*, de forma que no se pode ningún nodo por esta cota) y si se quiere deshabilitar basta pasar *null* como parámetro.

```
minimax(estado,limite,poda){
        if(limite.alcanzado() || estado.estaTerminado()){
                return estado.valorHeuristico();
        }
        hijos=estado.calcularHijos();
        podaActual=+inf;
        for(hijo:hijos){
                jugada=minimax(hijo,nuevoLimite,podaActual);
                if(jugada>mejorJugada){
                        mejorJugada=jugada;
                        podaActual=mejorJugada.valor();
                        if(mejorJugada>=poda)
                                return mejorJugada;
                }

        }
        return mejorJugada;
}
```

Figura 4: Pseudocódigo del minimax contemplando la poda alfa-beta.

3.3.2. Poda por simetrías

Esta poda tiene como fin, evitar analizar tableros iguales. Considerando que dos tablero son iguales cuando al rotarse quedan iguales, o bien cuando uno es simétrico del otro respecto a algún eje o diagonal.

Se puede observar en la ***figura 5***, el pseudocódigo que explica este tipo de poda.

En la implementación de esta poda, con el objetivo de no introducir excesivas consultas, se utilizó el método *symmtries,* de la clase *Board*, que devuelve qué tipo de simetrías ó rotaciones presenta el tablero, y si posee algún tipo de ellas entonces se guardarán en un *Set* los movimientos realizados, y

mediante los metodos: *rotated90, rotated180, rotated270, xSymmetric, ySimmetric, firstDiagSymmetric y secondDiagSymmetric,* se obtienen los movimientos simétricos que serán comparados con los que ya se encuentran en el Set.

```
minimax(estado,limite){
        if(limite.alcanzado() || estado.estaTerminado()){
                return estado.valorHeuristico();
        }
        hijos=estado.calcularHijos();
        for(hijo:hijos){
                if(!hijosProcesados.contiene(hijo)){ //No contiene ningun estado igual
                        jugada=minimax(hijo,nuevoLimite,podaActual);
                        if(jugada>mejorJugada){
                                mejorJugada=jugada;
                        }
                }
                hijosProcesados.agregar(hijo);
        }
        return mejorJugada;
}
```

Figura 5: Pseudocódigo del minimax contemplando la poda por simetrías.

De esta forma, si dos tableros son simétricos de alguna de las formas ya explicadas, y se realizan dos movimientos con la misma simetría los tableros resultantes serán iguales, por lo tanto no hace falta volver a analizar ambos.

La desventaja de esta poda es el hecho que incluye comparaciones extras, y en tableros no simétricos puede ser menos veloz que el ***minimax*** sin esta poda. En el apartado de evaluaciones se evidenciará la ventaja de usar esta poda en tableros simétricos, también se mostrará el tiempo extra que tarda en tableros no simétricos, y que en promedio ***minimax*** y ***minimax2*** tardan aproximadamente lo mismo. Dado que la implementación que hace esta poda utiliza recursos extras al usar un Set y guardarse los movimientos hechos, se decidió no utilizar esta poda en la implementación. Sin embargo, resulta interesante mostrar la mejora que se logra en tableros simétricos.

Esta poda está implementada en el método *minimax2,* y recibe los mismos parámetros que ***minimax***.

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 por el algoritmo, a lo largo del trabajo, fueron surgiendo varias posibles heurísticas las cuales se fueron mejorando, y en algunos casos se han reemplazado por otras. A continuación se muestran las 3 heurísticas elegidas que más adelante serán comparadas entre sí a fines de obtener valores cuantitativos que representen la calidad de las mismas y la cantidad de recursos que necesitan para ser implementadas.

3.4.1. Heurística 1

La primer heurística propuesta (método *value1* de la clase *Board*), y lógicamente la más simple, fue considerar únicamente la cantidad de piezas que poseía cada jugador. Inicialmente todas las piezas tenían el mismo valor y simplemente se calculaba la resta entre las piezas del jugador y las del enemigo, al avanzar en el desarrollo del trabajo se notó que las piezas guardianes tenían más valor que las piezas enemigas, por lo que se las ponderó con un valor mayor.

El orden de calcular esta heurística es O(*1)* ya que la cantidad de piezas que posee cada jugador es precalculada, por lo que solamente se efectúa una resta entre valores precalculados.

3.4.2. Heurística 2

La segunda heurística propuesta (método *value2* de la clase *Board*), heurística utilizada por el algoritmo ***minimax*** en su implementación es una heurística más elaborada que la anterior. Surgió luego de considerar que la primer heurística no ponderaba de manera acertada el valor de algunos movimientos. En la misma aparece un factor clave en el desarrollo del juego que es la posición del rey en el tablero ya que es dicha pieza la que condiciona los movimientos sobre todo el tablero. Por lo tanto, al valor de la heurística anterior se le añadió un factor que pondera la posición del rey en el tablero y como existen varios tamaños de tableros disponibles, el mismo factor se ve determinado también por el tamaño del tablero.

El orden de calcular esta heurística es O(*1)* ya que, como fue mencionado anteriormente, la cantidad de piezas que posee cada jugador es precalculada, así como también la posición en la que se encuentra el rey y el tamaño del tablero. Cabe destacar que inicialmente el orden de la heurística era $O(N^2)$ siendo *N* el tamaño del tablero ya que se recorría el tablero en busca del rey, pero se optó por almacenar la posición del rey dentro de la clase *Board* e ir

actualizando dicha posición cada vez que el rey realizaba un movimiento para mejorar la calidad de la heurística.

3.4.3. Heurística 3

La tercer heurística propuesta (método *value3* de la clase *Board*) surgió como alternativa frente a la heurística anterior. Si bien consideramos que la heurística 2 ya satisfacía las necesidades del trabajo, continuamos la búsqueda de otras alternativas, como la que será presentada a continuación. Esta heurística recorre todo el tablero para analizar la densidad de enemigos en 4 zonas del tablero, zonas que quedarán delimitadas tomando como centro de un eje de coordenadas al rey, es decir, las zonas superior izquierda, superior derecha, inferior izquierda e inferior derecha. Este factor también será añadido al valor de la heurística anterior y fue introducido pensando en resguardar/atacar (según sea el turno de MAX o de MIN) las 4 torres que hay en el tablero ya que se toma como factor la mínima densidad de enemigos entre los 4 sectores, es decir, que se busca mantener resguardados/dominados (según sea el turno de MAX o de MIN) todos los sectores del tablero debido a que una pieza puede ir de un borde al otro en tan sólo una jugada.

El orden de calcular esta heurística es $O(N^2)$ ya que, es necesario recorrer todo el tablero para determinar cuántas piezas enemigas hay en cada sector. En este caso, a diferencia de las heurísticas anteriores, no es tan trivial mantener precalculados los valores necesarios ya que cada vez que el rey cambia de posición hay que recalcular todos los valores. Observaremos más adelante en el informe comparaciones de performance y tiempo entre las diferentes heurísticas presentadas en este apartado.

3.5. Árbol de llamadas

Para la creación del árbol de llamadas en formato .dot, se creó una clase *Node* la cual posee tres métodos de clase, *start, rename* y *close*, los cuales sirven para abrir el archivo en el cual se va a crear el árbol, renombrar archivos (para poder crear archivos auxiliares), y cerrar el archivo actual, respectivamente. Además, posee una variable de clase para asignarle un *String* (name) único a cada instancia creada. Las instancias tienen métodos para setear etiquetas, formas, colores, como también para unir dos nodos.

Por la forma en que se implementó el ***minimax*** (multiplicando por -1 al pasar de nivel), el valor que reciben de sus hijos los nodos que deciden por el mínimo es el opuesto al correcto, por esta razón se creó un método *setLabelMove* que recibe un Move, y recibe un *boolean* (*invert*) para indicar si se debe tomar el valor opuesto al que indica el move, o bien, si es el correcto. Para esto, en ***minimax*** se debe llamar a *setLabelMove* con el valor *invert* en *true* cuando se esté procesando un nodo MIN, y en *false* cuando se esté procesando un nodo MAX. Por esta razón, fue necesario tener el valor de la profundidad donde se encontraba (y no cuántos niveles de profundidad faltaban analizar), ya que era sencillo saber si se encontraba en un MAX (nivel par) o en un MIN (nivel impar).

Esta decisión fue discutida, e hizo replantearse la estructura de clases y métodos utilizada. Y como otras soluciones se contemplaron, por ejemplo, el hecho de crear dos métodos (minimaxMin y minimaxMax) cada uno con la logística propia de tomar el máximo o mínimo de sus hijos, además de la lógica correspondiente para aplicar la poda alfa-beta. Sin embargo, se concluyó que estos dos métodos tendrían en su mayoría código *repetido* (mismo código pero intercambiando los "<" por ">" y viceversa), por lo que se optó por no utilizar esta solución.

Por otro lado, al momento de añadir al árbol los nodos que fueron podados, se consideró que podría ocurrir que no todos los posibles movimientos estén generados al momento de podar (ya que mientras se recorre el tablero se generan y procesan los movimientos posibles). Y por lo tanto, en pos de la eficiencia del algoritmo (para no generar posibles movimientos, sabiendo que no se van a analizar y únicamente se crearán para que aparezcan en el árbol), se decidió crear un único *Node* con una etiqueta que dice "*Poda*" el cual representa a los nodos de ese nivel que serían podados.

La implementación de *minimax* recibe por parámetro una referencia del nodo del árbol de llamadas que referencia al movimiento que se realizó para llegar a ese estado (me). Antes de hacer la llamada al método *minimax* se debe abrir el archivo donde se va a guardar el árbol y crear el Node "start", y luego del *minimax* se debe cerrar este archivo. En la primer llamada se debe enviar la referencia al Node "start" o bien *null* si no se quiere realizar el árbol de llamadas. Dentro del algoritmo se crean los Nodes de los hijos, y al momento de hacer la llamada recursiva se pasa como parámetro la referencia a ellos.

4. Evaluaciones

Se consideró apropiado evaluar los siguientes parámetros:

- Heurísticas de juego.
- Poda por simetrías.

Para ello se elaboraron sesenta tableros de cuatro tamaños distintos (7, 11, 15 y 19) para que las evaluaciones sean menos azarosas.

Se prosiguió a evaluar el tiempo requerido para alcanzar cierto nivel y el nivel alcanzado al imponer una cota de tiempo. Además de considerar que sea eficiente a la hora de ofrecer una respuesta, se tiene en cuenta la calidad de ésta. Por eso se consideró evaluar también la inteligencia del juego, dichas evaluaciones anteriormente mencionadas serán detalladas a continuación.

4.1. Evaluación por límite de profundidad

Se decidió que el tablero de dimensión 7x7 debería completar el nivel de profundidad 4, los de dimensiones 11x11 y 15x15 el nivel de profundidad 3 y por último el de 19x19 debería de completar sólo 2 niveles.

Estos niveles de profundidad se decidieron en base a algunas pruebas previas para testear los tiempos aproximados y se llegó a la conclusión de que aumentar más la profundidad tomaba demasiado tiempo considerando la cantidad de pruebas que se deseaban hacer.

Por otro lado se decidió testear sólo las *heurísticas 2* y *3* debido a que el orden de la *heurística 1* es igual al de la *heurística 2*, por lo que los resultados serán aproximadamente iguales y no proveerán ninguna información útil.

Una vez finalizadas las pruebas, cuyos resultados se observan en la ***tabla 1***, se llegaron a las siguientes conclusiones:

Comparando los valores de columna a columna se observa que la poda por simetrías en algunos grupos de tableros contribuyó a disminuir el tiempo mientras que otros lo incrementó. Se creé que esto fue debido a la elección de tableros donde algunos presentaban más simetrías que otros. Aún así la diferencia que conlleva utilizar la poda por simetría no es demasiado significativa como para determinar qué implementación del algoritmo es superior. Asimismo se observa como la poda alfa-beta disminuye considerablemente los tiempos, por lo cual se considera que es un excelente complemento para el minimax.

Evaluación por límite de profundidad						
Dimensión tableros	Nivel de profundidad	Heurísticas	Tiempos promedio [ms]			
			Sin podas	Poda por simetrías	Poda alfa-beta	Ambas podas
7	4	2	1302	1351	103	91
		3	1670	1748	129	134
11	3	2	1165	1176	81	85
		3	1707	1698	175	176
15	3	2	17897	17523	744	737
		3	26158	25569	1222	1185
19	2	2	270	256	22	22
		3	400	256	35	32

Tabla 1: Resultados de la evaluación por límite de profundidad

En comparación, si se observan las diferencias de los valores entre filas se puede apreciar que la *heurística 2* es mucho más veloz que la tercera, esto es así debido a que la *heurística 2* es de orden 1 y la heurística 3 es de orden $O(N^2)$ siendo *N* la dimensión del tablero.

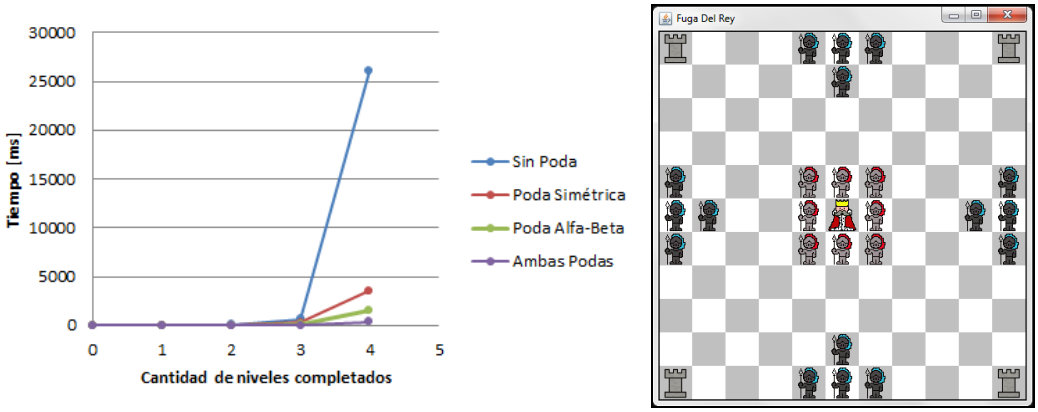

Figura 6: Evaluación de la cantidad de niveles completados en función del tiempo de un tablero de 11x11 simétrico.

Aparte se realizó otra evaluación, cuyos resultados se reflejan en las ***figuras 6*** y ***7***, para mostrar la diferencia de tiempo entre las distintas podas en dos tableros puntuales, uno simétrico y otro no simétrico.

Para cada tablero se cronometró el tiempo que tardaba en completar de uno a cuatro niveles, observando que a medida que la cantidad a completar aumentaba la diferencia de tiempo registrado entre métodos también. Así se evidenció los beneficios que aporta la poda simétrica a un tablero simétrico y como para uno no simétrico

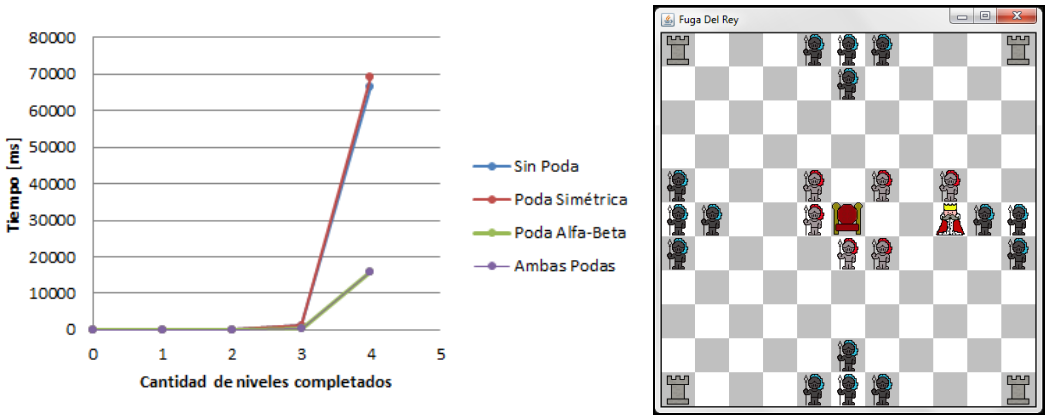

Figura 7: Evaluación de la cantidad de niveles completados en función del tiempo de un tablero de 11x11 no simétrico.

4.2. Evaluación por límite de tiempo

Para las siguientes evaluaciones, en base a los tiempos registrados en el experimento anterior, se consideró que darle 10 segundos al algoritmo para devolver la respuesta era suficiente para evaluar hasta qué nivel es capaz de completar y lograr apreciar alguna diferencia entre algoritmos y heurísticas.

Evaluación por límite de tiempo					
Dimensión tableros	Heurísticas	Niveles completados promedio			
		Sin podas	Poda por simetrías	Poda alfa-beta	Ambas podas
7	Simples	4.00	4.07	5.40	5.40
	Compleja	4.00	4.07	5.33	5.40
11	Simples	3.00	3.07	4.13	4.00
	Compleja	3.00	3.07	3.93	3.93
15	Simples	2.07	2.20	3.33	3.33
	Compleja	2.00	2.13	3.20	3.20
19	Simples	2.00	2.07	3.00	3.07
	Compleja	2.00	2.07	2.73	2.80

Tabla 2: Resultados de la evaluación por límite de tiempo

Por los mismos motivos que se omitió realizar las pruebas para la *heurística 1* en las evaluaciones por límite de profundidad, se omitió en ésta.

Los resultados, que fueron plasmados en la ***tabla 2*** muestran casi lo mismo que los resultados de la evaluación por tiempo pero en un menor grado. Esto se debe a que el tiempo para completar un nivel crece exponencialmente, como se puede ver en las ***figuras 6*** y ***7***. Por lo que al imponer un tiempo de 10 segundos de tope no se va a poder apreciar una gran diferencia a menos que la diferencia entre métodos sea muy importante (como comparar ***minimax*** sin poda contra ***minimax*** con poda alfa-beta).

Conclusión

A lo largo del desarrollo del trabajo pudimos observar diversas ventajas y desventajas que presentó la implementación del algoritmo ***minimax*** para el juego *La Fuga del Rey*. Varias de ellas ya han sido mencionadas en este informe, pero cabe destacar que más allá de las complicaciones que ocasiona el uso de un algoritmo dentro del paradigma de objetos, a nivel de performance consideramos que el algoritmo ***minimax*** es una buena solución para resolver juegos con condiciones similares a las actuales. Si bien no es una solución óptima en algunos aspectos, permite separar de manera clara el algoritmo propiamente y las heurísticas del juego, permitiendo así que sea adaptable a varios escenarios. Por otro lado, creemos que allí también se encuentra la mayor falencia del algoritmo ya que la definición del mismo restringe sus posibilidades. No sólo eso, sino que también a las mismas funciones heurísticas, ya que si bien es importante y absolutamente necesario tener conocimientos específicos de las reglas de juego, tanto atacantes como defensores deben implementar la misma función heurística, lo cual en este caso no resulta tan conveniente.

Es cierto que a los fines prácticos de la materia no es el objetivo la implementación de una heurística ideal, pero es importante destacar las limitaciones del algoritmo, así como también han sido destacadas sus virtudes.

Concluimos que más allá de algunas desventajas presentadas a la hora de modelar el algoritmo ***minimax*** para *La Fuga del Rey*, el algoritmo es una buena solución para el problema a resolver, y como se vió a lo largo de las iteraciones, detalladas en el informe, existe la posibilidad de incorporar podas, simetrías, limitaciones tanto por tiempo como por profundidad, generar un árbol de llamadas en el algoritmo lo cual enriquece aún más al ***minimax***.