ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA (COMPUTACIÓN)

TUTOR DE ALGORITMOS DE JUEGOS

ADVERSARIAL SEARCH TUTOR

Realizado por
Cristina Gómez Gómez
Tutorizado por
José Antonio Montenegro Montes
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Septiembre 2016

Fecha defensa:
El Secretario del Tribunal

Resumen: La asignatura Sistemas Inteligentes, impartida en los tres grados de Ingeniería Informática en la Universidad de Málaga, es un primer contacto del alumno de informática con la Inteligencia Artificial (IA).

La asignatura tiene como objetivo mostrar al alumno una serie de técnicas de IA, explicando detalladamente el funcionamiento de los algoritmos más representativos de cada técnica. Para conseguir tal fin se realiza una doble tarea, inicialmente los algoritmos son explicados en clase y se realizan ejercicios de una duración apropiada para poder ser realizados a mano y en el laboratorio ejecutamos los algoritmos sobre problemas más complejos para ver la aplicación del algoritmo a problemas más extensos.

Hasta el momento los profesores de la asignatura han desarrollado una serie de prácticas sobre algunos algoritmos de la asignatura. La idea general es actualizar las prácticas con problemas actuales.

El objetivo de este proyecto es la generación de una práctica, en concreto del tema de Juegos, que permita de forma detallada al alumno observar y realizar el seguimiento de los algoritmos Minimax y Expectimax. En concreto el juego escogido es el 2048.

Palabras: Inteligencia Artificial, Juegos, 2048, Heurísticas, Algoritmos, Minimax, Alfa-Beta, Expectimax

Abstract: The course “Intelligent Systems”, taught in the three degrees of Computer Science at the University of Málaga, is student's first contact with the Artificial Intelligence (AI).

The course aims to show students a number of AI techniques, explaining in detail the most representative algorithms of each technique. In order to achieve this purpose a double task is carried out, initially algorithms are explained in class and some exercises with an appropiate duration are made by hand and later, in laboratory, we run these algorithms to solve more complex problems in order to see the application of the algorithm to more extensive problems.

Until now, the course professors have developed a number of practices about some algorithms. The general idea is to update practices with current problems.

The aim of this project is to develop a practice that allows the student to observe in detail and track the Minimax and Expectimax algorithms. Specifically, the chosen game is 2048.

Keywords: Artificial Intelligence, Games, 2048, Heuristics, Algorithms, Minimax, Alpha-Beta, Expectimax

ÍNDICE

Contenido

Introducción 9

1.1 Estado del arte 9

1.2 Motivación y objetivos 10

1.3 Tecnologías a utilizar 10

1.4 Contenido y estructura de la memoria 11

Inteligencia artificial para Juegos 12

2.1 Definición formal de un juego 12

2.2 Algoritmos 13

2.2.1 Algoritmo minimax 13

2.2.2 Algoritmo Alfa-beta 16

2.2.3 Algoritmos expectiminimax y expectimax 19

Inteligencia artificial aplicada al 2048 23

3.1 Juego 2048 23

3.2 Definición formal del 2048 24

3.3 Funciones de evaluación: heurísticas 26

3.3.1 Heurísticas para 2048 26

3.4 Aplicación de algoritmos al juego......33

Diseño de la aplicación 36

Resultados 42

Conclusiones 53

Referencias bibliográficas......55

Anexos técnicos......56

Anexo A. Implementación......56

Implementación del juego 56

Implementación de los algoritmos......71

Implementación gráfica 72

Anexo B. Manual de usuario......74

1 Introducción

1.1 Estado del arte

Los juegos siempre han sido un ámbito de interés en el campo de la inteligencia artificial. Muchos de los pioneros de la informática ya intentaron trabajar con el problema del ajedrez. Por ejemplo, Alan M. Turing, considerado padre de la informática, desarrolló el primer programa de ordenador de la historia para jugar al ajedrez, a finales de los años 40. Otros también abordaron el problema del ajedrez, como Konrad Zuse, Noibert Wiener y Claude Shannon, conocidos por sus importantes aportaciones en la informática.

¿Qué tenía de especial para los primeros informáticos el ajedrez? ¿Por qué sigue siendo de tanto interés la aplicación de la inteligencia artificial a juegos? Los juegos son interesantes porque son demasiado difíciles para resolverlos. Por ejemplo, actualmente la complejidad de árbol de juego del ajedrez se calcula en torno a 10^{123}. Esto, además, nos hace darnos cuenta que la eficiencia debe ser importante, pues el árbol de juego suele ser de dimensiones realmente considerables.

A partir de su nacimiento por aquel entonces, la inteligencia artificial ha conseguido avanzar considerablemente, de manera que las máquinas ya consiguen ganar a los mejores jugadores del mundo en algunos juegos, como el ajedrez o el backgammon. Uno de los retos en la Inteligencia Artificial durante las últimas décadas ha sido el Go debido a su gran complejidad. Hasta este mismo año había sido imposible que una máquina venciera al mejor jugador de este juego, sin embargo, el pasado mes de marzo el programa AlphaGo, desarrollado por Google, ganó al campeón coreano del juego de mesa Go, Lee Sedol.

Esto demuestra que éste es un campo que continúa en desarrollo, alcanzándose logros hasta el día de hoy.

La inteligencia artificial ha ido ganando terreno en los juegos de tablero. En 1979, el programa BKG de Hans Berliner vence al campeón del mundo en Backgammon. Años más tarde, en 1994, cayeron las damas a manos de Chinook. Poco después, en 1997, el ajedrez gracias a Deep Blue, la supercomputadora que venció al campeón del mundo de ajedrez por aquel entonces, Gary Kaspárov.

Otros de los juegos típicos de la Inteligencia Artificial es el Tres en raya pues, debido a su simplicidad, es útil usarlo de manera de ejemplo o introducción, para entender los algoritmos y otros conceptos usados.

Pese a ser los juegos anteriores los más conocidos, existen infinidad de juegos más en los que se ha aplicado la Inteligencia Artificial, por ejemplo, el Scrabble, Bridge o Jeopardy, entre otros.

1.2 Motivación y objetivos

Tal y como es comentado en la sección anterior, hoy en día los juegos siguen siendo un reto para la inteligencia artificial. Por ello, en la asignatura Sistemas Inteligentes, impartida en los tres grados de Ingeniería Informática en la Universidad de Málaga, parte importante del temario es dedicada a las técnicas usadas en la Inteligencia Artificial aplicada a Juegos. Para mostrar dichas técnicas y algoritmos, se realiza una doble tarea, inicialmente los algoritmos son explicados en clase y realizados ejercicios de una duración apropiada para poder ser realizados a mano y en el laboratorio ejecutamos los algoritmos sobre problemas más complejos para ver la aplicación del algoritmo a problemas más extensos.

Para ello, hasta el momento los profesores de la asignatura han desarrollado una serie de prácticas sobre algunos algoritmos de la asignatura.

Por tanto, la idea general es actualizar las prácticas con problemas actuales.

El objetivo de este proyecto es la generación de una práctica, en concreto del tema de Juegos, que permita de forma detallada al alumno observar y realizar el seguimiento de los algoritmos Minimax y Expectimax. En concreto el juego escogido es el 2048.

1.3 Tecnologías a utilizar

1. Como sistema operativo se ha usado Windows.

2. El entorno de desarrollo de la aplicación es Eclipse.

3. Aplicaciones ofimáticas: Microsoft Office para la redacción de la memoria y la creación de gráficos.

1.4 Contenido y estructura de la memoria

La memoria está dividida en 6 capítulos y los anexos.

El presente capítulo es, en sí, una introducción a los demás. Por ello incluye el estado del arte, y la motivación y objetivos del trabajo, así como la estructura de la memoria descrita en esta misma sección.

El segundo capítulo define los conceptos básicos de la inteligencia artificial aplicada a los juegos. En él se verá cómo definir un juego de manera formal así cómo los algoritmos típicos utilizados en el ámbito.

El capítulo tercero define el juego 2048, primer de manera "informal" para entender su funcionamiento y posteriormente en función de los conceptos formales tratados en el segundo capítulo de manera que se puedan aplicar los algoritmos.

En el siguiente capítulo, el cuarto, veremos el diseño de la aplicación desarrollada mediante un esquema UML.

El quinto capítulo analiza los resultados obtenidos mediante tablas y gráficas. Los datos obtenidos nos permiten realizar comparaciones entre los diferentes algoritmos y heurísticas propuestas.

Finalizamos el proyecto con una sección dedicada a las conclusiones obtenidas sobre el trabajo.

El anexo se divide en dos secciones. En la primera aparece el código y descripción de la implementación. La segunda parte contiene el manual de usuario de la aplicación.

2 Inteligencia artificial para Juegos

2.1 Definición formal de un juego

Muchos problemas pueden ser resueltos mediante técnicas de inteligencia artificial. Para ello tenemos que modelar los distintos estados posibles del problema, de manera que el propósito será alcanzar un estado objetivo partiendo de un estado inicial aplicando unas reglas entre estados, también llamadas movimientos en el caso de los juegos.

Por tanto, para definir formalmente un juego debemos definir:

- Conjunto de estados: distintas situaciones del problema. Puede denominarse también espacio de estados. Debemos precisar una representación de los estados, pues una buena representación del juego es importante. Su tamaño mide la complejidad del problema.

  Normalmente existirá un estado inicial, y uno o varios estados objetivo.

- Movimientos: son reglas que modifican los estados, es decir, permiten pasar de un estado a otro. No siempre se podrán aplicar todos los movimientos, sólo se podrán aplicar los movimientos cuyas precondiciones se cumplen. Por tanto, serán de la forma *precondición → acción*, donde la *acción* indica como modificar el estado actual para generar un nuevo estado y la *precondición* impone restricciones sobre la aplicabilidad de la regla según el estado actual.

Necesitaremos también definir un test terminal y una función utilidad:

- El test terminal determina cuándo termina el juego. A los estados donde el juego ha terminado se les llama estados terminales. Habitualmente un estado será terminal si es un estado objetivo o si no existen reglas aplicables.

- Una función utilidad que da un valor numérico a los estados terminales.

En el presente documento me centro únicamente en las técnicas que se aplican en los juegos con adversario. Sin embargo, existen juegos de un sólo jugador, como el puzzle 8 por ejemplo, en los cuales se utilizan otras técnicas de búsqueda, siendo uno de los principales algoritmos utilizados el de A*.

Por tanto, nos centramos en los juegos con adversario, en los cuales dos jugadores se enfrentan.

El estado inicial y los movimientos legales para cada jugador definen el árbol de juego. Habitualmente, se consideran que los jugadores son MAX y MIN de manera que inicialmente es el turno MAX, y hacen sus movimientos alternativamente. Por ello, cada nivel del árbol se corresponderá a uno de los jugadores.

Un ejemplo, en el juego de las tres en raya, sería el siguiente:

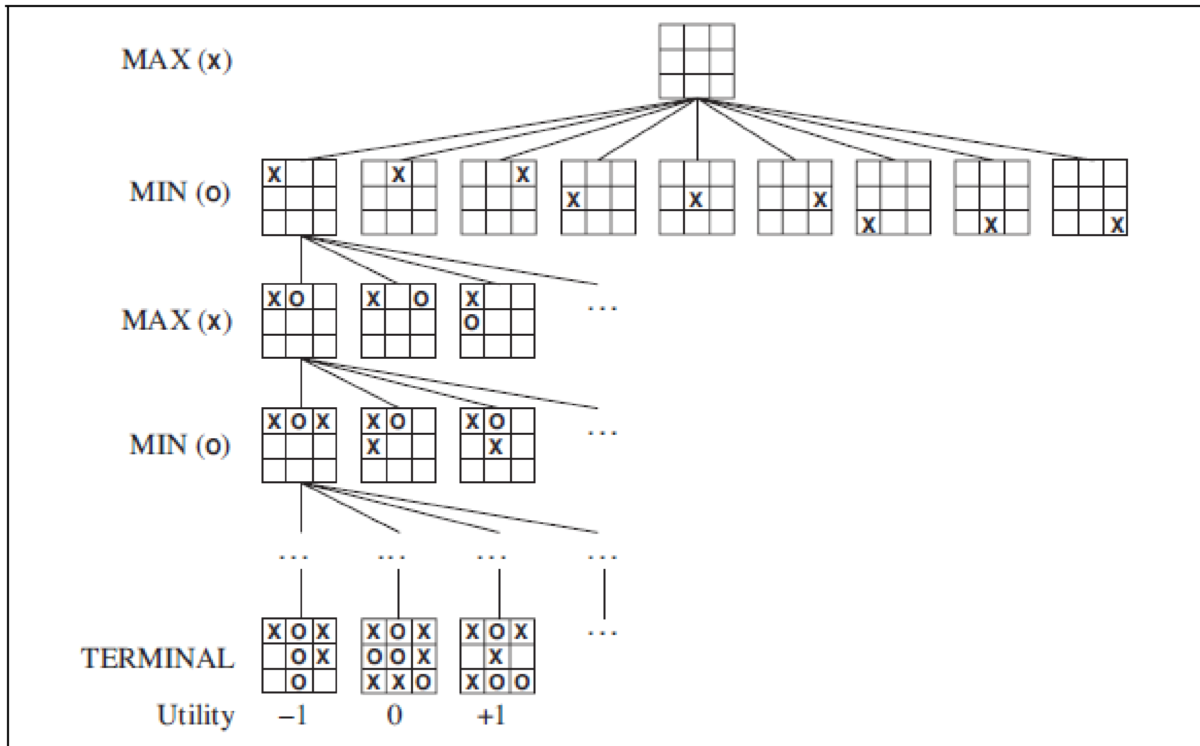

Ilustración 1: Ejemplo de árbol de juego del juego Tres en raya

Incluso un juego simple como las tres en raya es demasiado complejo para dibujar el árbol de juegos entero.

2.2 Algoritmos

Necesitamos algoritmos que nos permitan decidir qué movimientos aplicar en cada situación.

2.2.1 Algoritmo minimax

El algoritmo minimax es el principal algoritmo de búsqueda entre adversarios. Es el algoritmo más importante ya que, además, los demás suelen ser variaciones u optimizaciones suyas.

En teoría de juegos, minimax es un método de decisión para minimizar la pérdida máxima esperada en juegos con adversario.

En palabras más coloquiales, podríamos decir que este algoritmo se encarga de elegir el mejor movimiento para el jugador suponiendo que su rival escogerá el peor movimiento para el jugador.

2.2.1.1 Desarrollo del algoritmo

Pasos del algoritmo minimax:

1. Se genera el árbol de juego a partir del estado inicial.
2. Se da valor a los nodos terminales (estados terminales) mediante la función de utilidad definida para el juego.
3. Se da valor a los nodos superiores a partir de los inferiores, seleccionando como valor de un nodo el máximo de los valores de sus hijos si éste es de tipo MAX, o el mínimo de los valores de sus hijos si el nodo es de tipo MIN.
4. Elegir la acción en función de los valores asociados a los nodos.

2.2.1.2 Ejemplo

En la siguiente ilustración podemos ver un ejemplo del funcionamiento del Minimax.

1. Inicialmente se genera el árbol de juego.
2. Mediante una función de utilidad se les da un valor a los nodos terminales, los verdes en el caso del ejemplo, la cual tiene un rango de 1 a 9.
3. Una vez evaluados los nodos terminales podremos ascender a valores superiores otorgándoles un valor. Los nodos del nivel 5 ya disponen de un valor, pues son terminales.
4. Por ello pasamos a calcular los valores de los nodos de nivel 4. Como son de tipo MIN tendrán el menor valor de sus hijos, por ello se seleccionan los valores 5 (entre 5 y 8), 2 (entre 2 y 9) y 1 (entre 9 y 1).
5. Una vez calculados los valores de los nodos de nivel 4, pasaríamos a calcular los valores de nivel 3. Al ser de tipo MAX tendrán el mayor valor de sus hijos, por lo que se selecciona el 5 (entre 5 y 2) y el 9 (entre 1 y 9).
6. Ahora tocaría calcular los valores de nivel 2, de tipo MIN, por lo que se escogen los menores de nuevo: 5 (es el menor de 7,5 y 8), 3 (menor de 3 y 4) y 1 (menor de 1,9 y 2).
7. Finalmente, el nodo raíz es MAX, por lo que escogerá el mayor que es 5. Esto nos permitirá tomar la decisión de cuál es el movimiento adecuado.

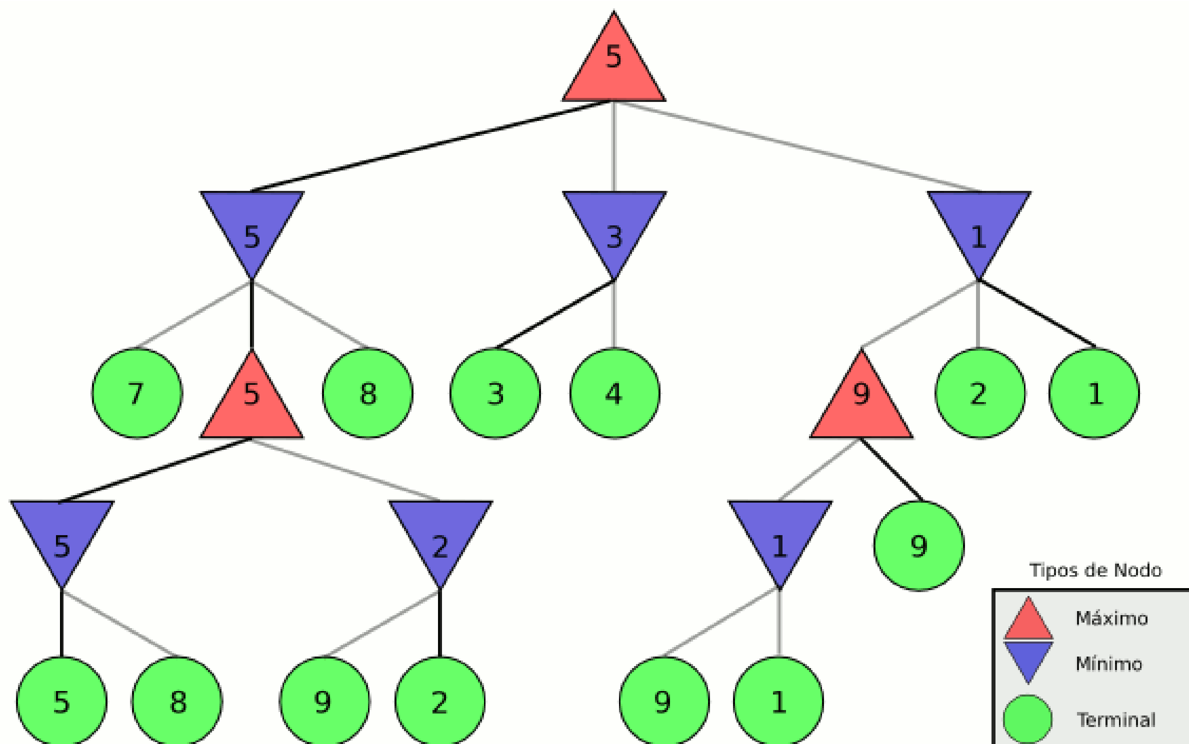

Ilustración 2: Ejemplo del algoritmo Minimax

2.2.1.3 Optimizaciones del algoritmo minimax

El algoritmo minimax genera el árbol completo de juego. Por tanto, si la profundidad del árbol es m y b representa las jugadas permitidas en cada punto entonces la complejidad en tiempo del algoritmo minimax es $O(b^m)$. La complejidad en espacio es O(bm).

En la práctica el método minimax es impracticable excepto en juegos cuyo árbol no tenga un tamaño demasiado grande, ya que para hacer la búsqueda en su totalidad necesitamos grandes cantidades de memoria y tiempo.

Una optimización muy intuitiva es limitar la profundidad de la búsqueda en el árbol determinando el valor de los nodos hoja no terminales a través de una función de evaluación heurística. Este método fue propuesto por Claude Shannon en 1950 en su trabajo "Programming a computer for playing chess". Sin embargo, al limitar la búsqueda, no podemos estar seguros de que el movimiento elegido sea el óptimo.

Por tanto, para optimizar minimax puede limitarse la búsqueda por nivel de profundidad o por tiempo de ejecución. Otra posible técnica es el uso de la poda alfa-beta, descrita en la siguiente sección.

2.2.2 Algoritmo Alfa-beta

La poda alfa beta es una optimización del algoritmo minimax, pues reduce el número de nodos evaluados en el árbol de juego de dicho algoritmo.

2.2.2.1 Desarrollo del algoritmo

En la poda alfa-beta se utilizan dos parámetros que actuarán de cotas sobre el valor minimax de un nodo (α,β):

- Para un nodo MAX:

  α valor de la mejor elección para MAX (valor más alto) que hemos encontrado en el camino hasta ahora

- Para un nodo MIN:

  β valor de la mejor elección para MIN (valor más bajo) que hemos encontrado en el camino hasta ahora

El algoritmo actualiza los valores de a y B conforme se ejecuta, recorriendo el árbol.

¿En qué condiciones se realiza la poda? Sea v el valor de un nodo:

- Si el nodo es MAX y $v \geq \beta$

  Se puede podar por debajo de dicho nodo.

- Si el nodo es MIN y $v \leq \alpha$

  Se puede podar por debajo del nodo.

En la siguiente subsección aparece un breve ejemplo que ayudará a comprender su funcionamiento.

El pseudocódigo del algoritmo será el siguiente:

```
function ALPHA-BETA-SEARCH(state) returns an action
    v = MAX-VALUE(state, -∞, +∞, depth)
    return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state,α,β, depth) return a utility value
     if node is a terminal node or depth == 0
        return the heuristic value of node
    v := -∞
    for each child of node
        v := max(v, MIN-VALUE(child, α, β, depth-1))
```

```
        if v ≥ β then return v
        α := max(α, v)
    return v
```

```
function MIN-VALUE(state,α,β,depth) return a utility value
     if node is a terminal node or depth == 0
        return the heuristic value of node
    v := +∞
    for each child of node
        v := min(v, MAX-VALUE(child, α, β, depth-1))
        if v ≤ α then return v
        α := min(α, v)
    return v
```

2.2.2.2 Ejemplo

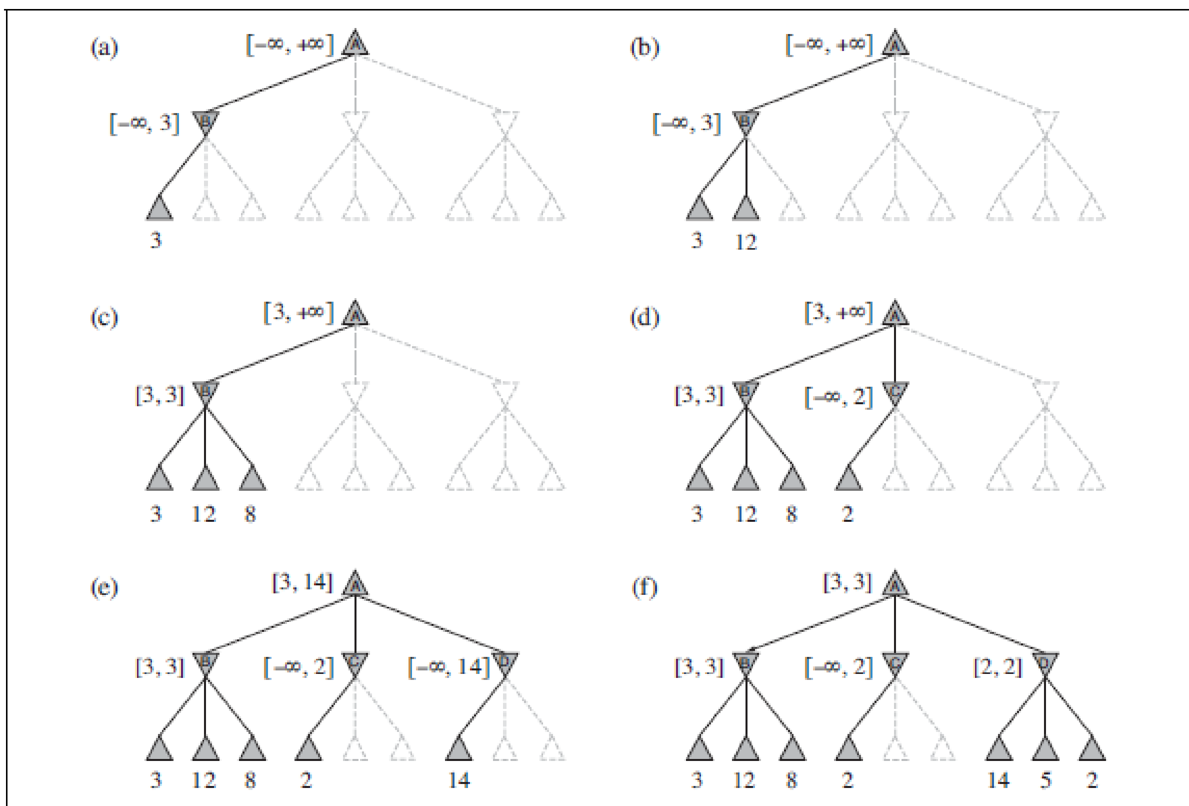

Ilustración 3: Ejemplo de ejecución del algoritmo Alfa-Beta

a) A partir del nodo B, se genera su primer hijo, que es terminal y su función de utilidad nos da el valor 3. Por tanto, como B es un nodo MIN, sabremos que su valor final será menor o igual a 3, es decir, estará en el rango [-∞, 3]. Por ello, β = 3.

b) Se genera el siguiente hijo de B, cuyo valor es 12. Como 12 es mayor que el β actual, el valor de β no cambia.
c) Se genera el siguiente hijo del nodo B, con valor 8 y ocurre lo mismo que en caso anterior. Como ya se han generado todos los hijos del nodo B, ya podemos darle definitivamente el valor de β, en este caso su valor será 3. Al establecer un valor para el nodo B, ya podemos saber que el valor del nodo A es 3 o más, pues es un nodo MAX. Por tanto, estará en el rango [3, ∞], y α = 3.
d) Se generan el primer hijo de C, el segundo hijo del nodo raíz A. Su valor es 2, por tanto, como C es un nodo MIN, sabremos que su valor definitivo será menor o igual que 2, es decir, está en el rango [-∞,2]. Por ello, β = 2.
   Como sabemos que el nodo A tendrá un valor superior a 3, porque el nodo B es mejor, podemos descartar el nodo C, pues tendrá un valor menor que 3. Por eso no es necesario que generemos sus hijos restantes, es decir, se poda el árbol.

e) Pasamos a generar los hijos del nodo D, tercer hijo del nodo raíz. Su valor es 14, por tanto, como C es un nodo MIN, sabremos que su valor definitivo será menor o igual que 14, es decir, está en el rango [-∞,14]. Por ello, β = 14.
   Como el valor de D será menor de 14, el valor de A, como mucho podrá valer 14, por tanto, estará en el rango [3,14].

f) Al generar los siguientes hijos de D, le daremos el valor 2, pues es el menor valor de sus hijos. Finalmente, como el nodo A es MAX, seleccionará el valor 3, del nodo B, pues es el mayor.

Como vemos, se ha conseguido eliminar una parte del árbol.

2.2.2.3 Eficiencia

La eficiencia de la poda Alfa-Beta depende del orden en el que se examinan los sucesores, es decir, el algoritmo se comportará de forma más eficiente si examinamos primero los mejores sucesores.

Tal y como hemos indicado antes, la complejidad del algoritmo minimax es $O(b^m)$. Si lográramos ordenar los sucesores, el algoritmo Alfa-Beta tendrá solamente que examinar $O(b^{m/2})$. Esto implica que Alfa-beta podrá alcanzar un nivel aproximadamente dos veces mayor que Minimax en la misma cantidad de tiempo.

Complejidad	*Minimax*	*Poda Alfa-Beta*
Tiempo	$O(b^m)$	$O(b^{m/2})$*
Espacio	O(bm)	O(bm)

Tabla 1: Comparativa de la complejidad entre Minimax y Expectimax

*Con una ordenación perfecta en los sucesores

Existen otras variaciones del algoritmo como:

1) Búsquedas con aspiración
2) Búsqueda sobre la variación principal
3) Profundización progresiva
4) Tablas de transposición
5) Procedimiento MTG(f)

2.2.3 Algoritmos expectiminimax y expectimax

Existen juegos que incluyen un elemento aleatorio, juegos estocásticos. Uno de los juegos más comunes de este tipo es el Backgammon, pero también podemos incluir el 2048 en este tipo de juegos, como veremos a partir de su descripción en el siguiente capítulo.

Por ello nos son de interés algoritmos que tengan en cuenta esta aleatoriedad.

2.2.3.1 Algoritmo expectiminimax

El expectiminimax es una variante del algoritmo Minimax. En el Minimax tradicional tenemos nodos “MIN” y nodos “MAX”, mientras que en el expectimax tendremos además nodos “CHANCE”, o “posibilidad”.

Dichos nodos “CHANCE” se intercalan con los nodos MIN y MAX. En dichos nodos, en lugar de tomar el máximo o el mínimo de los valores de utilidad de sus hijos, como se hace en el Minimax, se toma una media ponderada de los valores de sus hijos, en la cual los pesos serán las probabilidades de los sucesores.

La forma de intercalar dichos nodos depende de cada juego. Cada turno del juego se evalúa como un nodo “MAX”, que representa el turno del jugador, un nodo “MIN”, que representa al adversario, o un nodo “CHANCE”, que representa un jugador o efecto aleatorio.

Por ejemplo, consideremos un juego en el que cada ronda consta de un solo tiro de dados y, a continuación, las decisiones tomadas por el primer jugador de la IA, y luego otro oponente inteligente. El orden de los nodos en este juego se alterna entre " CHANCE ", " MAX " y " MIN ".

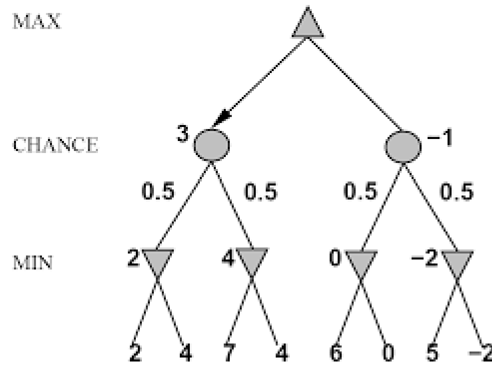

Ilustración 4: Ejemplo del algoritmo Expectiminimax

Explicaremos cómo funciona el algoritmo mediante el ejemplo anterior.

1. En primer lugar, se genera el árbol, al igual que en el algoritmo Minimax. Mediante una función de utilidad se da valor a las hojas de este, situadas en el nivel 4.

2. En el nivel 3 tenemos nodos MIN, por lo que cada nodo seleccionará el menor valor de los valores de sus hijos, siendo 2,4,0 y -2 los valores escogidos.

3. En el nivel 2, los nodos son de tipo CHANCE. ¿Cómo se calculará su valor? Situémonos en el nodo de la izquierda de nivel dos. Calcularemos su valor de la siguiente manera:

$$expected_value = 0.5 * 2 + 0.5 * 4 = 1 + 2 = 3$$

Por tanto, su valor esperado será 3. Para el nodo de la derecha tendremos:

$$expected_value = 0.5 * 0 + 0.5 * (-2) = -1$$

Por lo que su valor será -1.

4. El nodo de nivel 1, la raíz, es de tipo MAX, por lo que seleccionará el mayor valor de ambos, el 3.

Este ejemplo es bastante sencillo y, además las probabilidades son las mismas para todos los hijos, lo que correspondería a una media simple, sin embargo, no siempre es así, pues la probabilidad dependerá del juego en concreto.

2.2.3.1.1 Pseudocódigo

```
function expectiminimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        // Return value of minimum-valued child node
        let α := +∞
        foreach child of node
            α := min(α, expectiminimax(child, depth-1))
    else if we are to play at node
        // Return value of maximum-valued child node
        let α := -∞
        for each child of node
            α := max(α, expectiminimax(child, depth-1))
    else if random event at node
        // Return weighted average of all child nodes' values
        let α := 0
        for each child of node
            α := α + (Probability[child] * expectiminimax(child, depth-1))
    return α
```

2.2.3.2 Algoritmo expectimax

En el caso del expectimax, sólo tendremos nodos MAX y nodos CHANCE, no habrá nodos MIN.

2.2.3.2.1 Pseudocódigo

```
function expectimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the agent is MAX
        // Return value of maximum-valued child node
        let α := -∞
        foreach child of node
            α := max(α, expectimax(child, depth-1))
    else if random event at node
        // Return weighted average of all child nodes' values
        let α := 0
        foreach child of node
```

```
        α := α + (Probability[child] * expectimax(child, depth-1))
    return α
```

Un inconveniente de estos algoritmos es que no se le puede aplicar la poda alfa-beta debido a que debemos expandir todos los hijos de un nodo para calcular su valor, ya que dicho valor será una media ponderada de los valores de los hijos.

3 Inteligencia artificial aplicada al 2048

Una vez vista la forma de definir formalmente los juegos y los algoritmos usados para resolverlos, pasamos a definir el 2048 basándonos en los conceptos anteriores y posteriormente a aplicar los algoritmos vistos. Para ello, antes es necesario conocer dicho juego.

3.1 Juego 2048

El 2048 es un juego desarrollado en marzo de 2014 por el joven desarrollador italiano Gabriele Cirulli que alcanzó una gran popularidad.

El objetivo del juego es conseguir una casilla cuyo valor sea 2048, de ahí su nombre, deslizando y combinando las casillas del tablero.

En su versión original, el tablero contiene 4x4 casillas las cuales, a su vez, contienen un número, de manera que las casillas tendrán distintos colores dependiendo del número que contengan.

Inicialmente el tablero contendrá sólo dos casillas, con valor 2 o 4. A partir de ellas podremos hacer algún movimiento hacia alguna dirección.

Mediante las teclas de dirección del teclado podemos mover las casillas, deslizándolas por la cuadrícula. De esta manera podemos mover las casillas deslizándolas hacia posiciones vacías.

Si dos casillas de igual valor colisionan en un movimiento se combinan en una nueva casilla cuyo valor será la suma de los valores colisionados. Por ejemplo, si dos baldosas con el número 8 colisionan, se formará una con valor 16.

Después de realizar un movimiento aparece una casilla nueva en un lugar vacío del tablero, que contendrá el número 2 (en un 90% de los casos) o el número 4 (en el 10% restante).

Se acumula una puntuación, que comienza en cero y se incrementa al combinar dos casillas, con el valor de dicha combinación.

El usuario ganará el juego si consigue obtener una casilla con el número 2048. Sin embargo, si no se puede hacer movimientos, es decir, ya no quedan lugares vacíos y no existen casillas adyacentes con el mismo valor, el juego termina.

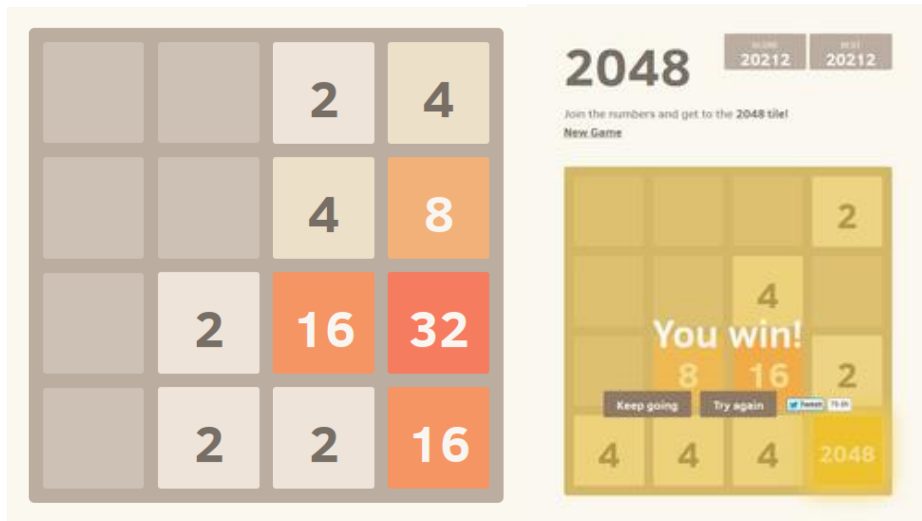

Ilustración 5: Ejemplos de tableros del 2048

Inicialmente el juego fue desarrollado en el lenguaje Javascript, estando disponible su código en el siguiente enlace:

https://github.com/gabrielecirulli/2048

Posteriormente se lanzó para smartphones, en Android e iOS.

Debido a que alcanzó una popularidad abrumadora han sido numerosas las diferentes implementaciones, en múltiples lenguajes y las diferentes versiones diseñadas a partir de este juego, en parte ambas posibilitadas por el fácil acceso al código original de Gabriel Cirulli.

Como digo, existen múltiples implementaciones, desde las que representan el tablero con un sólo número, hasta las que consideran que el tablero es una matriz. Este es un claro ejemplo de que existen diferentes formas de representar el espacio de estados de los juegos.

3.2 Definición formal del 2048

Puesto que ya conocemos cómo se juega y sabemos qué tenemos que definir para aplicar los algoritmos, pasamos a definir formalmente el 2048.

Debemos definir el conjunto de estado y los movimientos para el 2048.

- Conjunto de estados: matriz de 4x4 que contenga casillas cuyos números sean las potencias de 2, excluyendo el 0.

  En el estado inicial aparecen dos casillas, las cuales pueden contener los valores 2 o 4.

  Un estado será objetivo si alguna de las casillas contiene el número 2048.

- Movimientos: En este juego los movimientos del usuario son distintos a los del computador. Los movimientos del usuario serán mover las casillas hacia arriba, derecha, abajo o izquierda. Sin embargo, para evitar bucles infinitos en la ejecución de algoritmos de búsqueda, considero que no siempre es posible que el jugador realice los cuatro movimientos. Realizar un movimiento será posible siempre que al realizarlo se deslicen las casillas en el tablero o se combinen, de manera que "cambie" algo en el tablero. Por tanto, si realizar dicho movimiento no supondría ningún cambio, ni se combinan celdas ni se deslizan, ese movimiento no estaría disponible.

  En el caso del computador, el movimiento a realizar será introducir una casilla con valor 2 o 4 en un lugar vacío.

- Test terminal.

  Un estado será terminal si cumple una de estas dos condiciones:

  - alguna de las casillas contiene el número 2048 (estado objetivo)
  - no se pueden aplicar movimientos (no existen reglas aplicables), es decir, no quedan lugares vacíos y no existen casillas adyacentes con el mismo valor que puedan agruparse al mover, de manera que no se pueden hacer movimientos.

Es necesario también establecer una función de utilidad que dé un valor a los estados terminales, de manera que podamos aplicar los algoritmos vistos en el primer capítulo.

Además, hemos visto que en la práctica el método minimax es impracticable excepto en supuestos sencillos pues vemos que realizar la búsqueda completa requerirían cantidades excesivas de tiempo y memoria. Por ello se propone limitar la profundidad de la búsqueda y determinar el valor de los nodos hoja (que en este caso pueden ser no terminales) mediante una función de evaluación heurística.

3.3 Funciones de evaluación: heurísticas

En este proyecto usamos dicha técnica, es decir, limitamos la profundidad de la búsqueda en el espacio de estados, por lo que es necesario definir una buena heurística que nos permita un alto porcentaje de partidas ganadas.

La búsqueda de una buena heurística es una de las principales tareas en la definición de juegos, siendo también bastante complicada, pues de ella dependerá lo bien o mal que funcione el algoritmo.

La heurística será una función matemática que asocia un valor a cada nodo en función de lo “bueno” o “malo” que sea dicho estado para alcanzar un estado objetivo a partir de él.

3.3.1 Heurísticas para 2048

Para idear las heurísticas podemos utilizar las estrategias que intuitivamente usaría el usuario al jugar al juego.

- Score (puntuación):
  Una forma muy intuitiva de evaluar el estado podría ser la puntuación actual, pues cuando mayor sea más piezas habremos combinado. Sin embargo, no nos vale con combinar casillas sin ninguna estrategia, debemos considerar otras características deseables como las siguientes.

- Casillas vacías:
  Otra medida de interés que nos puede ayudar a determinar cómo de bueno es un estado es el recuento de las casillas libres o vacías. No nos interesa tener pocas casillas vacías, pues en el momento que el tablero se llene y no podamos combinar ninguna casilla perderemos el juego. Sin embargo, en estados iniciales es más común tener muchas casillas vacías y en estados más avanzados lo normal es que queden menos. Por ello podemos ponderar el número de casillas vacías por la puntuación actual.

- Similitud entre valores vecinos:
  También nos interesa que las celdas cercanas tengan valores iguales, o al menos parecidos, para poder combinarlos. Por ello es de interés esta heurística, la cual calcula una media de la diferencia del valor entre las casillas vecinas, tratando de minimizar este recuento ya que nos interesa que la diferencia sea pequeña.

- Monotonía:
  Uno de los trucos principales que la mayoría de jugadores sigue al jugar al juego es colocar la casilla con mayor valor en una esquina y, a partir de ella ir decreciendo su valor. Normalmente también evitaremos que las casillas con valores más pequeños queden aisladas. Esta heurística nos permite tender a tener un tablero más organizado. Para ello, sumamos los valores de las casillas del tablero ponderados de la manera siguiente

7	6	5	4
6	5	4	3
5	4	3	2
4	3	2	1

  De esta manera procuramos tener el mayor valor en la esquina superior izquierda y que los valores del tablero tengan cierta monotonía.

- Valor máximo. Otra simple característica que intuitivamente podemos considerar importante es tener en cuenta el valor máximo, pues cuanto mayor sea más cerca estaremos del objetivo.

En función a estas características, la aplicación dispondrá de 4 heurísticas en las que éstas se combinan, pues por separado no funcionarán tan bien como unidas.

Las cuatro heurísticas utilizadas son las siguientes:

Heurística 1:

Score + Monotonía - Similitud + log(Score)*Celdas vacías

Combinamos:

- la puntuación (score)
- la medida de la monotonía (suma de los valores de la matriz ponderados)
- Similitud: obtenemos la media de las diferencias, como lo que queremos es minimizarlo, lo ponemos en negativo.
- log(Score)*Celdas vacías: tenemos en cuenta el número de celdas vacías, pero ponderado con el logaritmo del score. Es bastante importante tener casillas

vacías, lo que nos da un mayor margen para realizar movimientos y continuar jugando, sobre todo esto es de valorar en estados avanzados, pues en estados iniciales es más común encontrarnos con muchas celdas vacías. Una manera de valorar esto es ponderar el número de celdas vacías por el logaritmo de la puntuación actual. Usamos el logaritmo para tener en cuenta la puntuación sin que se alcancen valores muy altos ya que si esto ocurre prácticamente sólo se tendría en cuenta esta característica.

Heurística 2:

Monotonía - Similitud + log(Score)*Celdas vacías

En este caso no tenemos en cuenta el Score en la suma, por ello es más rápida que la heurística anterior, pero según los resultados no obtenemos mejor solución que la heurística 1. Es conveniente usarla cuando prima la velocidad.

Heurística 3:

Score - Similitud + log(Score)*Celdas vacías

En este caso no tenemos en cuenta la monotonía en la suma.

Heurística 4:

Score + (Celdas vacías * log2(max)) + Monotonía

Siendo log2(max) el logaritmo en base 2 del valor máximo del tablero. Utilizamos el logaritmo por la misma razón que en los casos anteriores ya que si utilizáramos el valor máximo en lugar de su logaritmo, dicho sumando alcanzaría valores altos lo que provocaría que sólo éste fuera tomado en cuenta.

3.3.1.1 Ejemplos

Para entender mejor las heurísticas anteriores las calcularemos para varios estados.

Ejemplo 1

Supongamos que tenemos el siguiente estado:

Ilustración 6: Ejemplo para heurísticas 1

Procedamos a calcular el valor de las heurísticas propuestas.

En primer lugar, vamos a calcular las características antes mencionadas para dicho estado.

- Score.
  Se puede ver en la figura 6, la puntuación actual es 8.
- Casillas vacías
  El número de casillas vacías es 12.
- Similitud
  Esta medida calcula una media de la diferencia del valor entre las casillas vecinas. Puesto que nos interesa que los números iguales o, al menos, parecidos estén cercanos, se obtienen las distancias entre los vecinos de cada celda, acumulando la media de las distancias para cada celda con sus vecinos.
  - $Celda\ (0,0)$: $\frac{|4-4|+|4-2|+|4-4|}{3} = \frac{2}{3} = 0.66666$
  - $Celda\ (0,1)$: $\frac{|2-4|+|2-2|+|2-2|+|2-4|}{4} = \frac{4}{4} = 1$
  - $Celda\ (0,2)$: $\frac{|2-2|+|2-2|}{2} = \frac{0}{2} = 0$
  - $Celda(1,0)$: $\frac{|4-4|+|4-2|+|4-4|}{3} = \frac{2}{3} = 0.66666$

$$Similitud = \frac{7}{3} = 2.3333333$$

- Monotonía
  Suma de los valores del tablero ponderada mediante los valores de la tabla:

7	6	5	4
6	5	4	3
5	4	3	2
4	3	2	1

$$Monotonía = 4*7 + 4*6 + 2*6 + 0*5 + 0*5 + 2*5 + 0*4 + 0*4 + 0*4 + 0*4 + 0*3 + 0*3 + 0*3 + 0*2 + 0*2 + 0*1 = 74$$

Heurísticas:

$$\text{Heurística 1} = \text{Score} + \text{Monotonía} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías} = 8 + 74 - 2.333 + \log(8) * 12 = 104.6199651668247 \approx 104.62$$

$$\text{Heurística 2} = \text{Monotonía} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías} = 74 - 2.333 + \log(8) * 12 = 96.6199651668247 \approx 96.62$$

$$\text{Heurística 3} = \text{Score} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías} = 8 - 2.333 + \log(8) * 12 = 30.619965166824695 \approx 30.62$$

$$\text{Heurística 4} = \text{Score} + (\text{Celdas vacías} * \log_2 max) + \text{Monotonía} = 8 + (12 * \log_2 4) + 74 = 8 + (12 * 2) + 74 = 106$$

Ejemplo 2

Supongamos ahora que tenemos el siguiente estado:

64	8	4	2
16	4		
4			
			2

Score: 380

Ilustración 7: Ejemplo para heurísticas 2

- Score.
  Se puede ver en la figura 7, la puntuación actual es 380.
- Casillas vacías
  El número de casillas vacías es 8.
- Similitud
  - $Celda\ (0,0): \frac{|64-64|+|64-8|+|64-16|+|64-4|}{4} = \frac{164}{4} = 41$
  - $Celda\ (0,1): \frac{|8-64|+|8-8|+|8-4|+|8-16|+|8-4|}{5} = \frac{72}{5} = 14.4$
  - $Celda\ (0,2): \frac{|4-8|+|4-4|+|4-2|+|4-4|}{4} = \frac{6}{4} = 1.5$
  - $Celda\ (0,3): \frac{|2-4|+|2-2|}{2} = \frac{2}{2} = 1$
  - $Celda(1,0): \frac{|16-64|+|16-8|+|16-16|+|16-4|+|16-4|}{5} = \frac{80}{5} = 16$
  - $Celda\ (1,1): \frac{|4-64|+|4-8|+|4-4|+|4-16|+|4-4|+|4-4|}{6} = \frac{76}{6} = 12.66666$
  - $Celda\ (2,0): \frac{|4-16|+|4-4|+|4-4|}{3} = \frac{12}{3} = 4$
  - $Celda\ (3,3): \frac{|2-2|}{1} = \frac{0}{1} = 0$

$$Similitud = 90.56666666666668$$

- Monotonía

$$\begin{aligned} Monotonía = {} & 64*7 + 16*6 + 8*6 + 4*5 + 4*5 + 4*5 + 0*4 + 0*4 \\ & + 0*4 + 2*4 + 0*3 + 0*3 + 0*3 + 0*2 + 0*2 + 2*1 \\ & = 662 \end{aligned}$$

<u>Heurísticas:</u>

$$\begin{aligned} \text{Heurística 1} &= \text{Score} + \text{Monotonía} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías} \\ &= 380 + 662 - 90.56666666666668 + \log(380) * 8 \\ &= 998.9547033550967 \approx 998.95 \end{aligned}$$

$$\begin{aligned} \text{Heurística 2} &= \text{Monotonía} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías} \\ &= 662 - 90.56666666666668 + \log(380) * 8 = 618.9547033550967 \\ &\approx 618.95 \end{aligned}$$

$$\begin{aligned} \text{Heurística 3} &= \text{Score} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías} \\ &= 380 - 90.56666666666668 + \log(380) * 8 = 336.9547033550968 \\ &\approx 336.95 \end{aligned}$$

$$\begin{aligned} \text{Heurística 4} &= \text{Score} + (\text{Celdas vacías} * \log_2 max) + \text{Monotonía} \\ &= 380 + (8 * \log_2 64) + 662 = 1090.0 \end{aligned}$$

Ejemplo 3

Supongamos ahora que tenemos el siguiente estado:

Ilustración 8: Ejemplo para heurísticas 3

- Score.
  Se puede ver en la figura 8, la puntuación actual es 9228.
- Casillas vacías
  El número de casillas vacías es 7.
- Similitud
  - $Celda\ (0,0): \frac{|1024-1024|+|1024-32|+|1024-16|+|1024-4|}{4} = \frac{3020}{4} = 755$
  - $Celda\ (0,1): \frac{|32-1024|+|32-32|+|32-4|+|32-16|+|32-4|}{5} = \frac{1064}{5} = 212.8$
  - $Celda\ (0,2): \frac{|4-32|+|4-4|+|4-2|+|4-4|}{4} = \frac{30}{4} = 7.5$
  - $Celda\ (0,3): \frac{|2-4|+|2-2|}{2} = \frac{2}{2} = 1$
  - $Celda(1,0): \frac{|16-1024|+|16-32|+|16-16|+|16-4|+|16-8|}{5} = \frac{1044}{5} = 208.8$
  - $Celda\ (1,1): \frac{|4-1024|+|4-32|+|4-4|+|4-16|+|4-4|+|4-8|}{6} = \frac{1064}{6} = 177.3333$
  - $Celda\ (2,0): \frac{|8-16|+|8-4|+|8-8|+|8-2|}{4} = \frac{18}{4} = 4.5$
  - $Celda\ (3,0): \frac{|2-8|+ +|2-2|}{2} = \frac{6}{2} = 3$
  - $Celda\ (3,2): \frac{|4-4|}{1} = \frac{0}{1} = 0$

$$Similitud = 1369.9333$$

- Monotonía

$$Monotonía = 1024*7 + 16*6 + 32*6 + 8*5 + 4*5 + 4*5 + 2*4 + 0*4 + 0*4 + 2*4 + 0*3 + 0*3 + 0*3 + 4*2 + 0*2 + 0*1 = 7560$$

Heurísticas:

$$\text{Heurística 1} = \text{Score} + \text{Monotonía} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías}$$
$$= 9228 + 7560 - 1369.9333333333332 + \log(9228) * 7$$
$$= 15481.976650001723 \approx 15481.98$$

$$\text{Heurística 2} = \text{Monotonía} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías}$$
$$= 7560 - 1369.9333333333332 + \log(9228) * 7 = 6253.976650001722$$
$$\approx 6253.98$$

$$\text{Heurística 3} = \text{Score} - \text{Similitud} + \log(\text{Score}) * \text{Celdas vacías}$$
$$= 9228 - 1369.9333333333332 + \log(9228) * 7 = 7921.976650001722$$
$$\approx 7921.98$$

$$\text{Heurística 4} = \text{Score} + (\text{Celdas vacías} * \log_2 max) + \text{Monotonía}$$
$$= 9228 + (7 * \log_2 1024) + 7560 = 16858$$

3.4 Aplicación de algoritmos al juego

Una vez hechas las anteriores definiciones podemos aplicar los algoritmos vistos en el segundo capítulo a este juego.

Como sabemos, estos algoritmos generan un árbol de juego. Veíamos anteriormente, a modo de ejemplo, parte del árbol del juego de las tres en raya. ¿Cómo sería el árbol para 2048?

El jugador MAX sería el usuario, o la IA mientras que el jugador MIN será el computador encargado de colocar una nueva casilla tras cada movimiento del jugador MAX. Como realizan movimientos alternativamente cada nivel del árbol se corresponderá a uno de los jugadores.

¿De qué forma será dicho árbol? ¿Cuántos hijos tendrá cada nodo?

Un nodo MIN tendrá tantos hijos como casillas vacías multiplicado por dos, ya que, tras tu turno, el computador podrá poner una ficha en cada posición vacía con valor 2 o con valor 4.

Un nodo MAX podrá tener, a lo sumo, cuatro hijos, correspondientes a los cuatro movimientos posibles (arriba, derecha, abajo e izquierda). Sin embargo, si al realizar un movimiento sobre un tablero no se produce ningún cambio, dicho movimiento se considerará no válido para evitar ciclos.

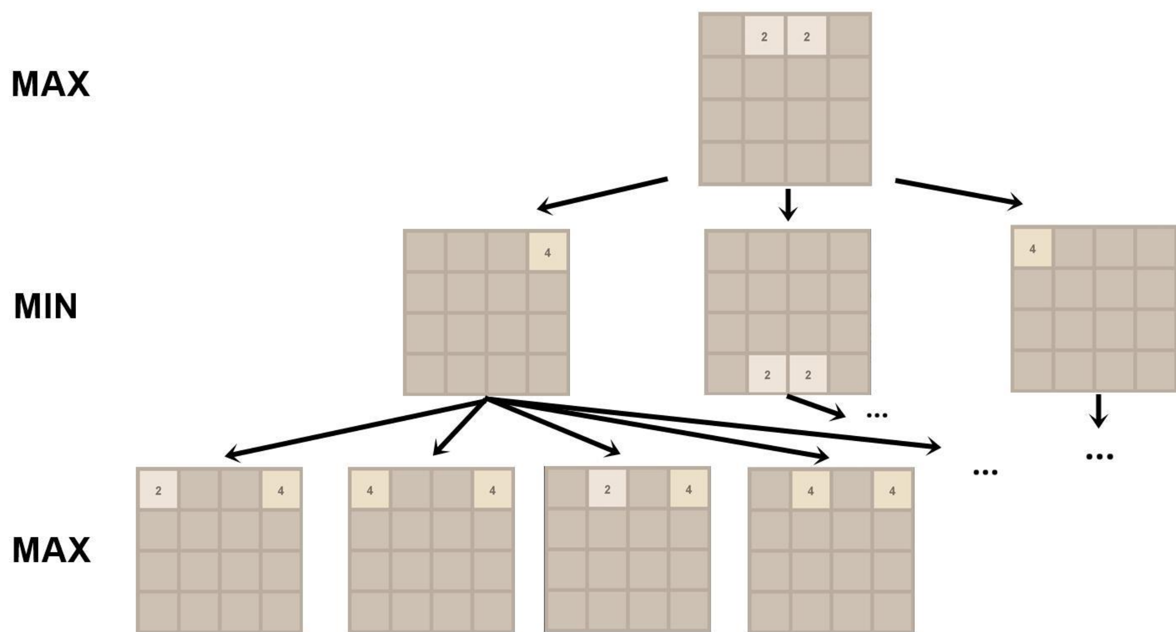

Ilustración 9: Parte del árbol de juego de 2048

La figura 9 muestra un ejemplo de árbol del juego, en el que sólo se muestran 8 estados pero que nos ayuda a ver cómo se generaría dicho árbol. Además, nos permite intuir las dimensiones del árbol completo.

En el capítulo anterior hemos descrito además del algoritmo Minimax y la poda Alfa-beta, el algoritmo expectimax, utilizado en juegos que incluyen un elemento aleatorio. Sin lugar a dudas el 2048 es de este tipo de juegos, ya que la introducción de una nueva casilla tras cada movimiento se realiza de manera aleatoria.

Por tanto, es posible utilizar el algoritmo expectimax en el juego 2048, pues en el turno del jugador, estaríamos en un nodo MAX, y en el turno del ordenador, que añade un valor (2 o 4) en una celda aleatoria, estaríamos en un nodo CHANCE.

Recordemos que el valor de un nodo CHANCE vendrá dado por una media ponderada de los valores de sus sucesores en la cual los pesos serán las probabilidades de estos.

¿Cuál es la probabilidad de cada movimiento? La probabilidad de que el computador coloque la nueva casilla en un determinado lugar es la misma para todas las casillas. Sin embargo, la probabilidad de que sea 2 es del 90% y de que sea 4 es del 10%. Por tanto, la probabilidad de un movimiento será.

- Si se añade un 2:

$$probabilidad = \frac{1}{Número\ de\ casillas\ vacías} * 0.9$$

- Si se añade un 4:

$$probabilidad = \frac{1}{Número\ de\ casillas\ vacías} * 0.1$$

Hay otra cosa a tener en cuenta con el algoritmo Expectimax, pues la función de evaluación heurística debe tener unos requisitos especiales. Con Minimax, la escala de los valores no importa, sin embargo, Expectimax necesita utilizar valores grandes, ya que, al realizar las medias ponderadas progresivamente hacia arriba, los valores se van a ir reduciendo, de manera que si a los nodos terminales se les asocia un valor muy pequeño el valor que llegará a la raíz será siempre 0. Por ello, como para Minimax y Alfa-Beta, no importa la escala, podemos seleccionar funciones de evaluación que asocien números grandes, de manera que las descritas en la sección anterior cumplen esta propiedad.

4 Diseño de la aplicación

El presente trabajo está diseñado de manera que pueda ser de utilidad en la explicación de los contenidos de la asignatura Sistemas Inteligentes.

En la parte práctica de dicha asignatura se utiliza el paquete AIMA, que es un paquete de clases Java que permite definir y resolver problemas de Sistemas Inteligentes de manera que el diseño permite separar la representación del problema de los algoritmos.

Por ello, la implementación del 2048 abarcada en este trabajo está elaborada en función de la biblioteca AIMA, de manera que pueda servir como práctica de la asignatura anteriormente citada.

Pasamos a mostrar el diagrama UML del proyecto. Sin embargo, por su tamaño es imposible analizarlo en su totalidad, por lo que lo haremos por partes.

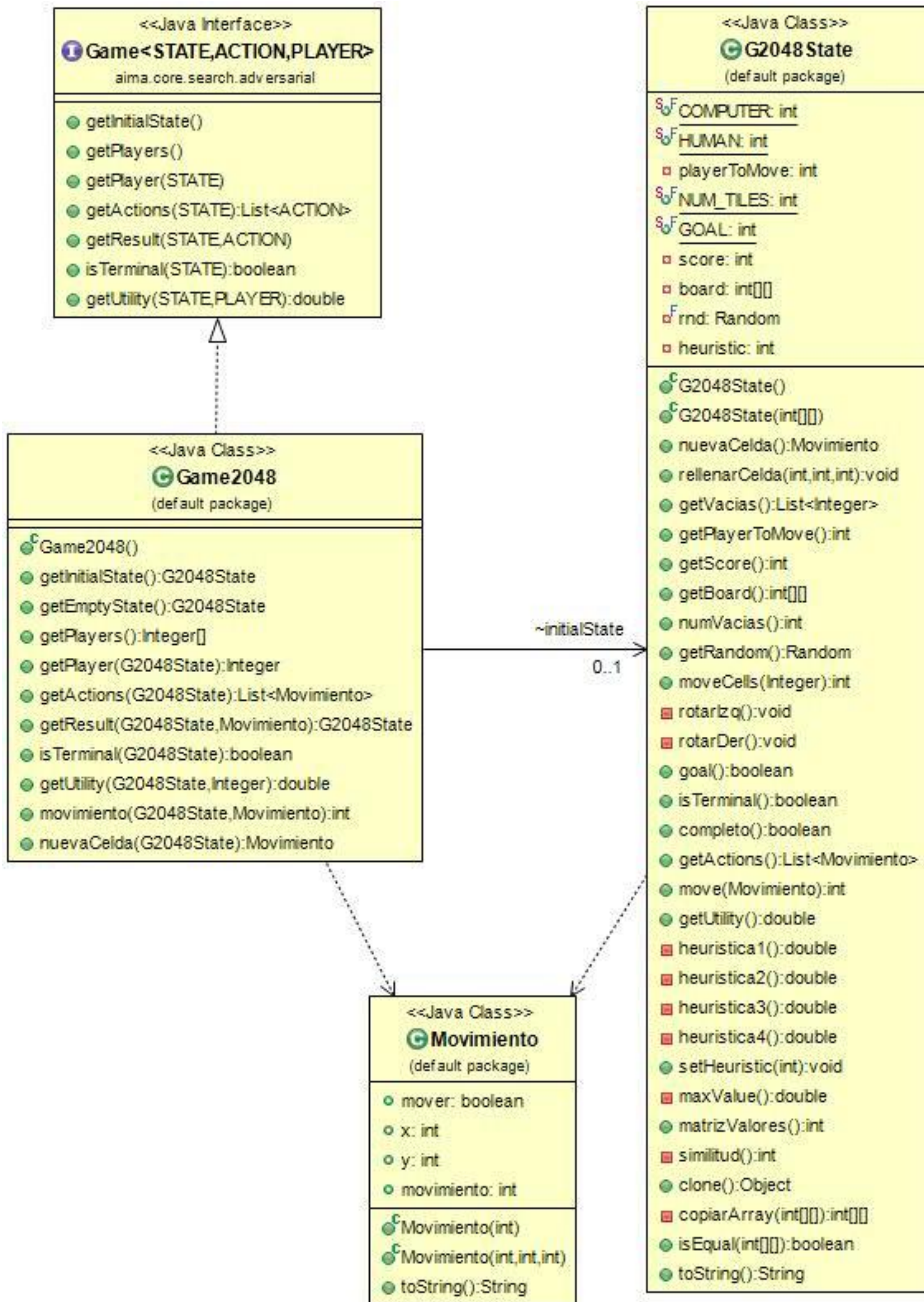

Ilustración 10: Diagrama UML del Juego

La figura 10 muestra las clases necesarias para modelar el juego en sí. La clase Movimiento nos permitirá identificar y diferenciar las diferentes acciones del juego.

La clase G2048State identificará el estado del tablero y contendrá los métodos necesarios para realizar los movimientos, calcular las heurísticas...

La clase Game2048 implementará la interfaz Game<STATE, ACTION, PLAYER>, del paquete AIMA. En este caso, los estados son de tipo G2048State, las acciones de tipo Movimiento y los jugadores de tipo Integer. En dicha clase se implementan los métodos de la interfaz.

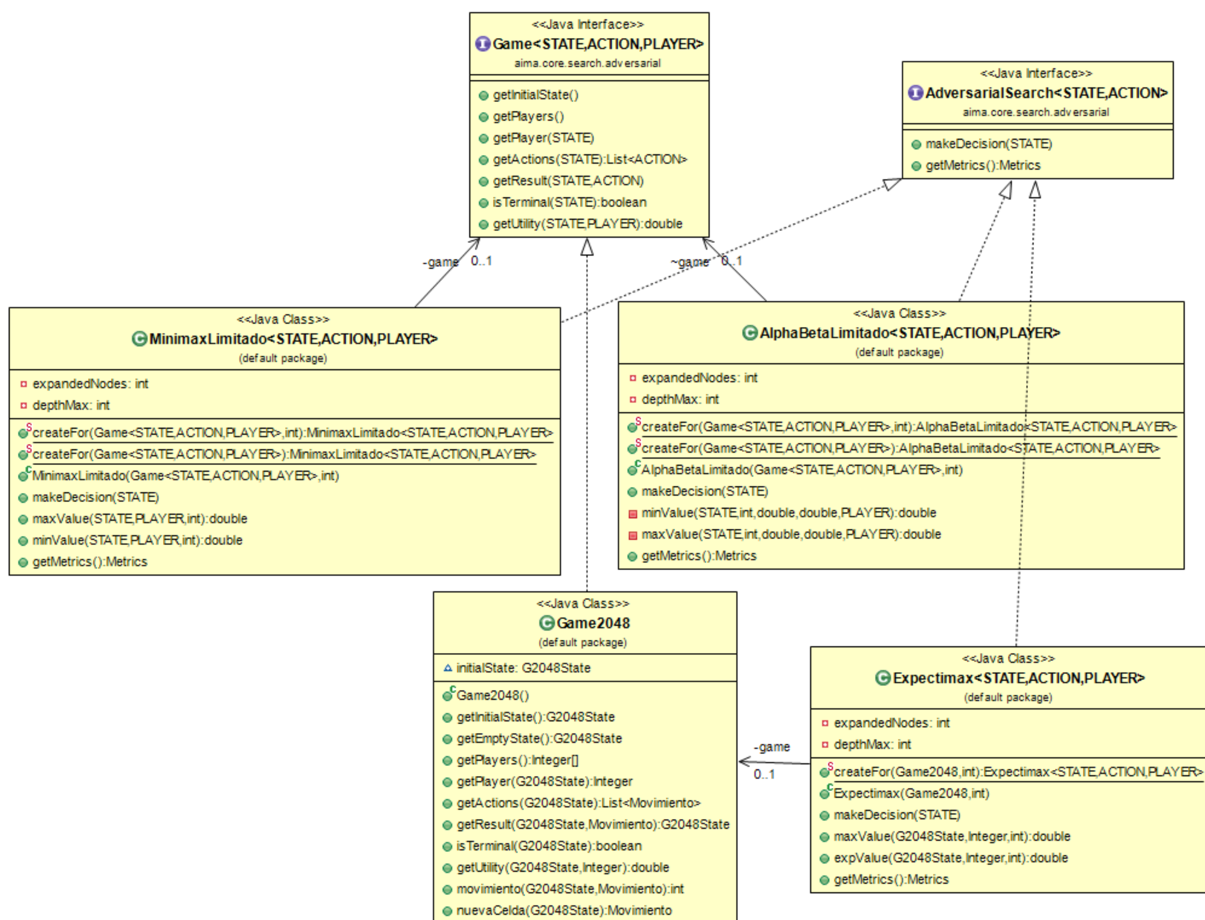

Ilustración 11: Diagrama UML de los algoritmos

Además, tal y como hemos comentado antes, el algoritmo Minimax es imposible de utilizarlo en la práctica en problemas grandes, pues la complejidad es demasiado alta.

Una manera de optimizar dicho algoritmo era limitando la profundidad, por lo que he creado las clases necesarias para usar los algoritmos Minimax y Alfa-beta con limitación de la profundidad.

Otro de los algoritmos comentados, el Expectimax, también ha sido implementado con el mismo objetivo.

En la figura 11 podemos ver las relaciones entre los algoritmos implementados. Las clases AlphaBetaLimitado, MinimaxLimitado y Expectimax implementan la interfaz AdversarialSearch, implementado los métodos de ésta. En ellas se ejecutan los algoritmos descritos en el presente documento. Para ello hacen uso de un objeto de la clase Game, pues necesitarán una partida. Sin embargo, en la clase Expectimax será de tipo Game2048, pues se usan probabilidades que dependerán del juego en concreto.

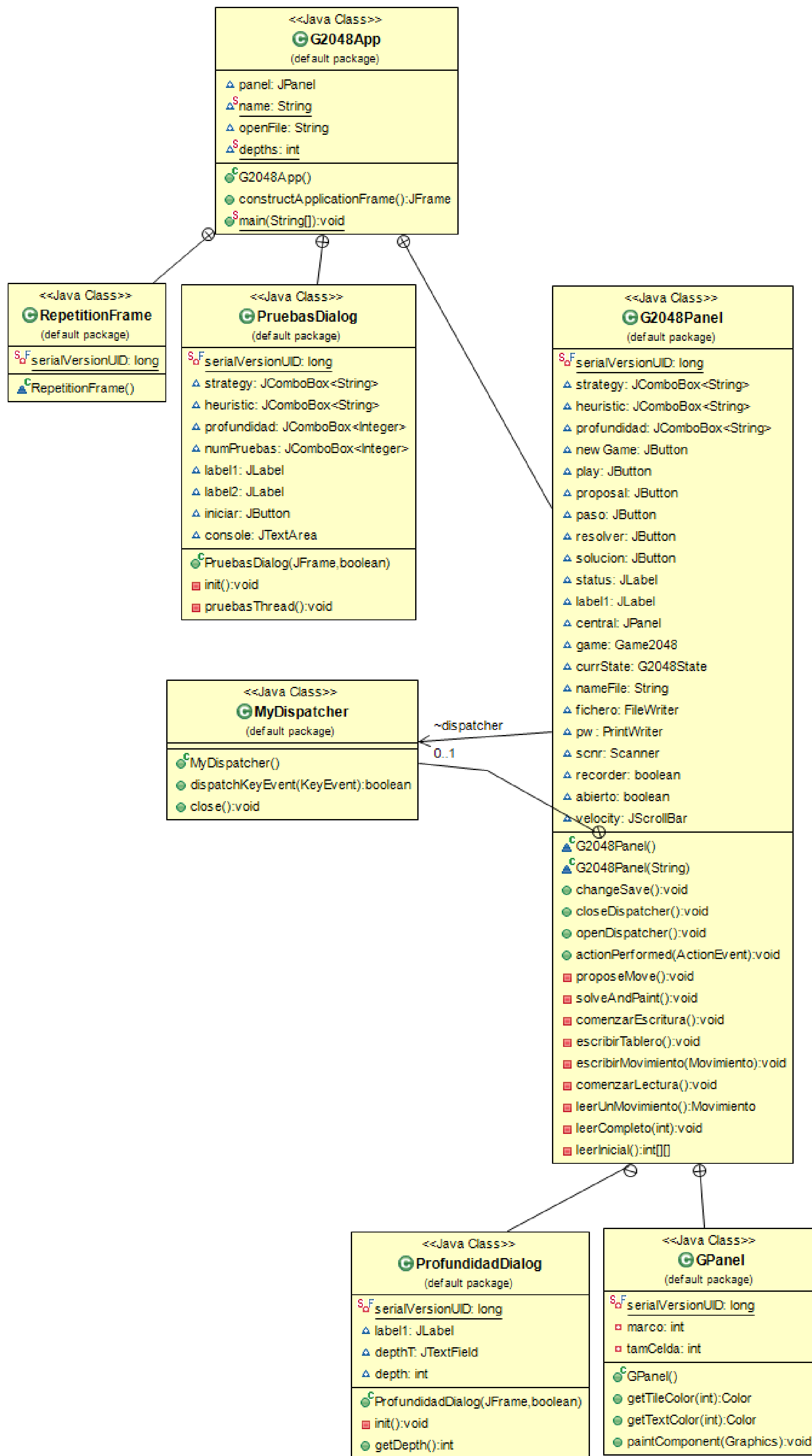

Ilustración 12: Diagrama UML de la interfaz gráfica

Con el fin de implementar la interfaz gráfica se han creado las clases de la figura 12. La clase G2048App implemente la interfaz gráfica y contiene diferentes clases como paneles, frames y ventanas de diálogo, necesarias en la aplicación y se hace del uso de un KeyEventDispatcher, para recoger la entrada por teclado.

5 Resultados

Los resultados obtenidos dependerán de diversos factores:

- Algoritmo utilizado. Tal y como he indicado en el presente documento, la poda Alfa-Beta es una optimización del algoritmo Minimax. Por tanto, dicho algoritmo obtendrá los mismos resultados que el minimax, pero en un tiempo menor. Por su parte, no podemos aplicar la poda alfa-beta al algoritmo Expectimax ya que debemos expandir todos los hijos de un nodo para calcular su valor. Por ello, el algoritmo más rápido será la poda Alfa-beta.
- Heurística. Lógicamente, la heurística es uno de los factores más importantes. Cuanto mejor sea la heurística, mejores serán los resultados obtenidos, es decir, el porcentaje de partidas que la IA consigue alcanzar será mayor.
- Profundidad de búsqueda. Como he comentado varias veces, es necesario limitar la profundidad al aplicar los algoritmos anteriormente mencionados. Evidentemente, el valor al que limitemos la búsqueda influye bastante en los resultados que obtengamos.

Para la obtención de los resultados, se ejecutan 100 partidas para cada caso, obteniendo el porcentaje de partidas ganadas y una media de duración de las partidas.

Las pruebas se han realizado para cada algoritmo, cada heurística y con límites de profundidad desde 1 hasta 7 y las 100 partidas de prueba se han generado de manera aleatoria, usándose las mismas partidas en todos los casos.

Alfa-Beta			
Heurística	Profundidad	Porcentaje de partidas ganadas	Tiempo medio por partida (s)
Heurística 1	1	0	0.001794711
	2	0	0.01560342
	3	51	0.139458503
	4	40	0.573280988
	5	62	2.706512032
	6	69	12.069926438
	7	73	43.486167867
Heurística 2	1	0	0.001767663
	2	0	0.014403909
	3	3	0.10498618
	4	17	0.447963035
	5	61	2.292564486
	6	59	9.381812238
	7	76	34.973646577
Heurística 3	1	0	0.00100925
	2	0	0.013722845
	3	7	0.109398463
	4	35	0.685844045
	5	32	3.055214546
	6	56	16.89068889
	7	59	59.879851354
Heurística 4	1	0	0.001548159
	2	0	0.012754492
	3	30	0.133171757
	4	40	0.513441004
	5	57	2.689431297
	6	53	11.029131179
	7	65	48.601640667

Tabla 2: Resultados del algoritmo Alfa-Beta

Minimax			
Heurística	Profundidad	Porcentaje de partidas ganadas	Tiempo medio por partida (s)
Heurística 1	1	0	0.00190301
	2	0	0.015617527
	3	51	0.1881203765
	4	41	1.255284443
	5	64	8.8477722355
	6	65	63.884055994
	7	75	376.912523012
Heurística 2	1	0	0.00160323
	2	0	0.014963952
	3	5	0.137765874
	4	15	0.932783362
	5	68	7.471190727
	6	71	50.547010795
	7	72	295.228987382
Heurística 3	1	0	0.001080002
	2	0	0.017641039
	3	5	0.1457578025
	4	40	1.285479457
	5	49	8.98162815
	6	57	65.250421968
	7	62	393.458308591
Heurística 4	1	0	0.001565401
	2	0	0.011273029
	3	29	0.164911558
	4	38	1.043750885
	5	54	8.753824515
	6	62	56.27655640
	7	66	364.923192912

Tabla 3: Resultados del algoritmo Minimax

Expectimax			
Heurística	Profundidad	Porcentaje de partidas ganadas	Tiempo medio por partida (s)
Heurística 1	1	0	0.0056195
	2	0	0.019205236
	3	56	0.235435689
	4	84	1.921496353
	5	87	11.918400834
	6	94	107.034087053
	7	97	646.254418221
Heurística 2	1	0	0.00569638
	2	0	0.018926718
	3	12	0.185022204
	4	58	1.508970255
	5	80	10.242626022
	6	93	90.854049336
	7	95	645.943074468
Heurística 3	1	0	0.00338204
	2	0	0.012725246
	3	10	0.148787463
	4	54	1.656921287
	5	57	10.20525689
	6	78	91.598982443
	7	87	645.666991384
Heurística 4	1	0	0.00504987
	2	0	0.014774837
	3	63	0.224199239
	4	80	1.569982214
	5	87	11.049808607
	6	88	83.185857268
	7	94	586.569852

Tabla 4: Resultados del algoritmo Expectimax

Evidentemente, el número de partidas ganadas dependerá de cada ejecución en concreto, pues el computador actúa de manera aleatoria.

Sólo se ha realizado baterías de pruebas en aquellos casos cuya duración por partida no es superior a 10 minutos aproximadamente. Sin embargo, si no pusiéramos esta restricción, es decir, si estableciéramos un límite de profundidad mayor, los resultados serían mejores, incrementándose el porcentaje de partidas ganadas.

Para realizar las pertinentes comparaciones nos apoyaremos en las siguientes gráficas.

Las próximas cuatro figuras nos muestran el porcentaje de partidas ganadas para cada heurística en función del algoritmo usado y la profundidad de búsqueda.

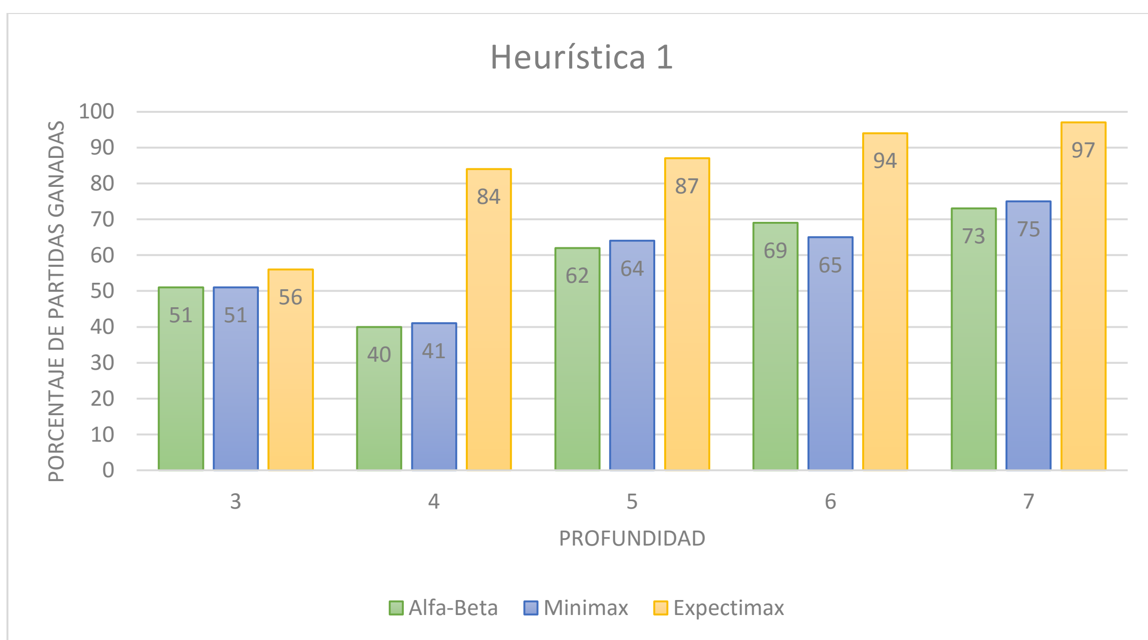

Ilustración 13: Porcentaje de partidas ganadas usando la heurística 1

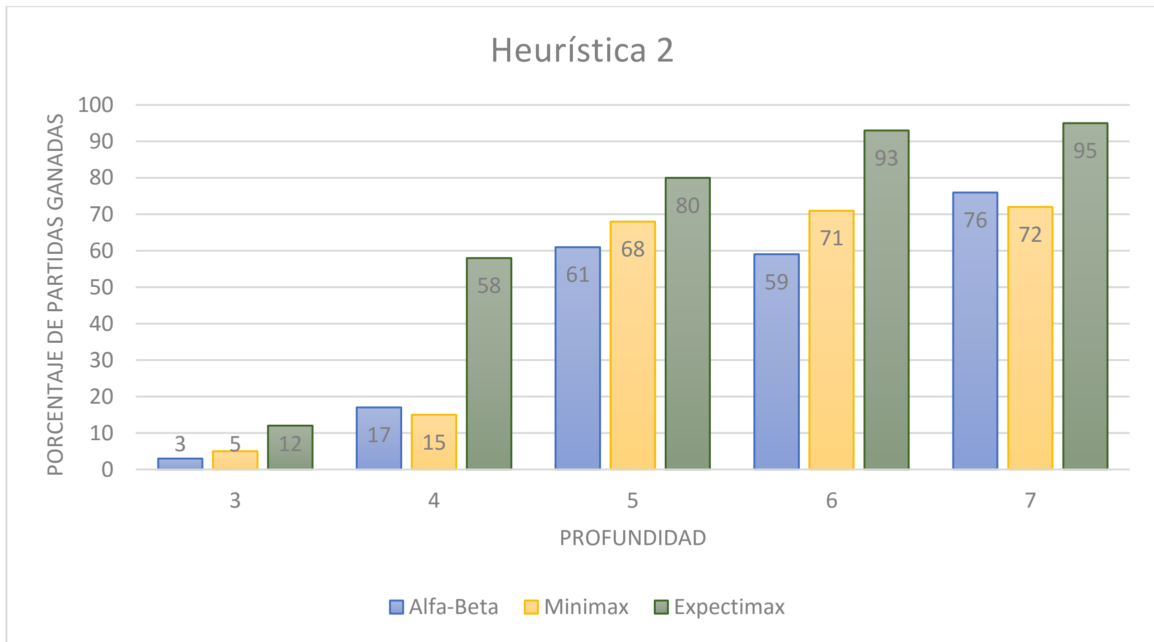

Ilustración 14: Porcentaje de partidas ganadas usando la heurística 2

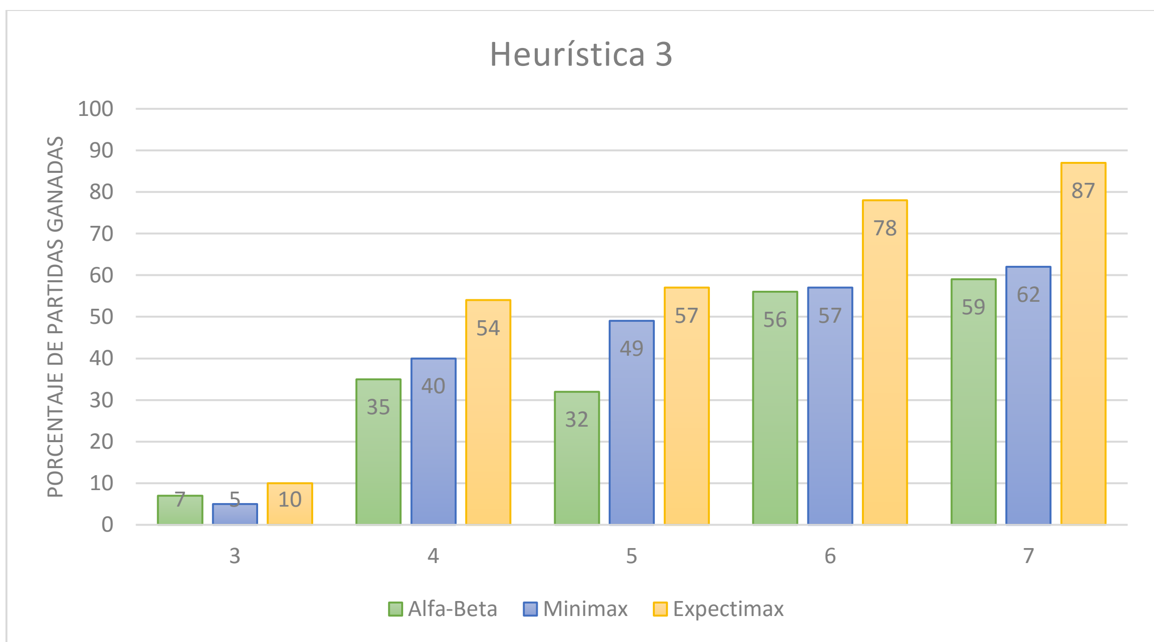

Ilustración 15: Porcentaje de partidas ganadas usando la heurística 3

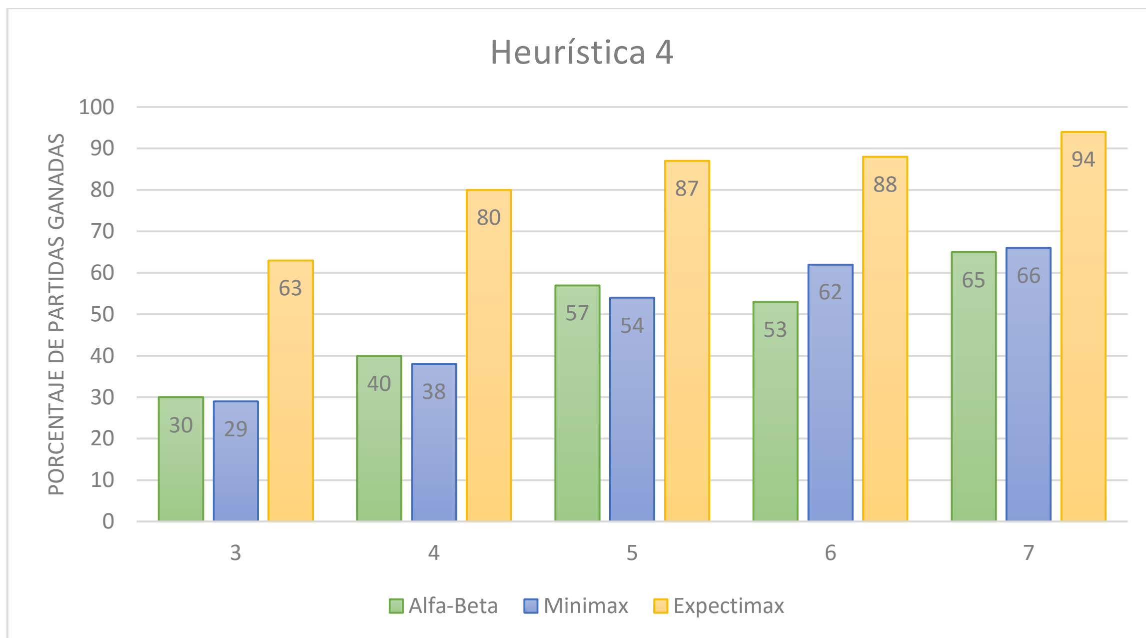

Ilustración 16: Porcentaje de partidas ganadas usando la heurística 4

Podemos ver cómo, sin lugar a dudas, para las cuatro heurísticas, el algoritmo que mejor funciona, obteniendo un mayor número de partidas ganadas, es el espectimax con diferencia.

Con un límite de profundidad 3 vemos cómo claramente empieza a despuntar este algoritmo con la heurística 4 y, para las demás heurísticas basta con establecer el límite de profundidad en 4 niveles para alcanzar un número aceptable de partidas ganadas, destacando sobre los algoritmos Minimax y Alfa-Beta.

Vemos cómo, usando el expectimax, en profundidades como 6 y 7 obtenemos un porcentaje muy cercano a 100, cosa que no ocurre con los demás algoritmos. Esto nos muestra lo bien que funciona este algoritmo con este juego, es el más adecuado.

Los resultados de Minimax y Alfa-Beta, evidentemente son similares, variando en pocas partidas ya que, como sabemos, el algoritmo Alfa-Beta devuelve el mismo resultado que el Minimax. Sin embargo, debido a la aleatoriedad del turno del computador no siempre se tomarán las mismas decisiones a medida que la partida avanza, de ahí las diferencias. Incluso, para un mismo estado inicial cada ejecución será distinta.

Básicamente, con el expectimax podemos alcanzar los mismos resultados que con los demás algoritmos estableciendo un límite de profundidad más pequeño.

Las siguientes figuras nos servirán para comparar entre las heurísticas.

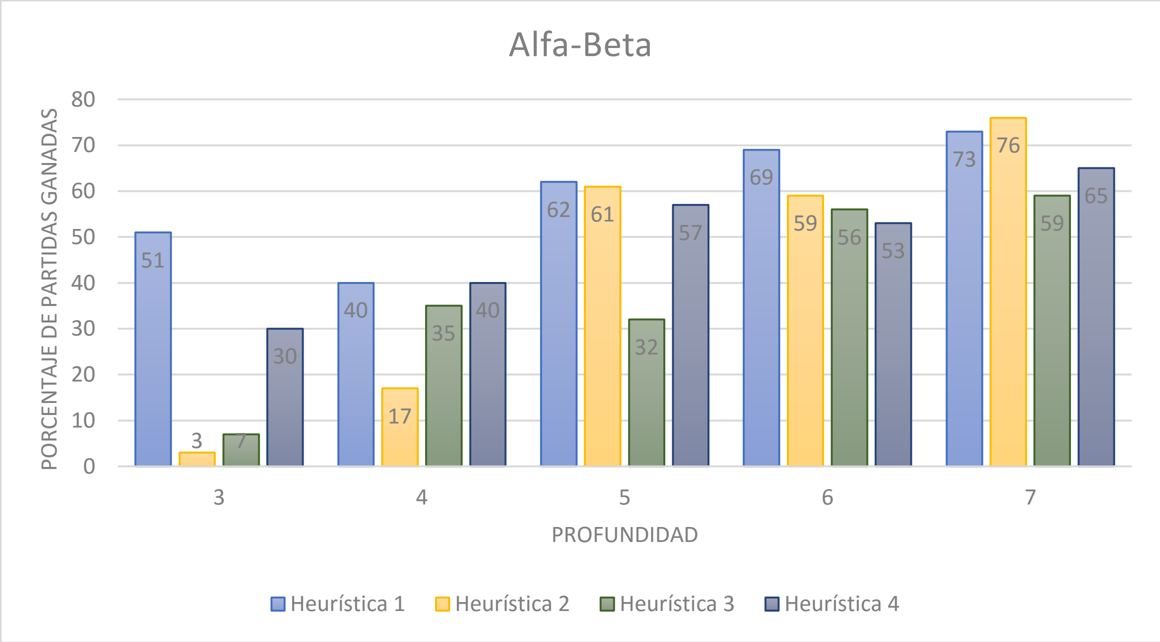

Ilustración 17: Porcentaje de partidas ganadas usando el algoritmo Alfa-Beta

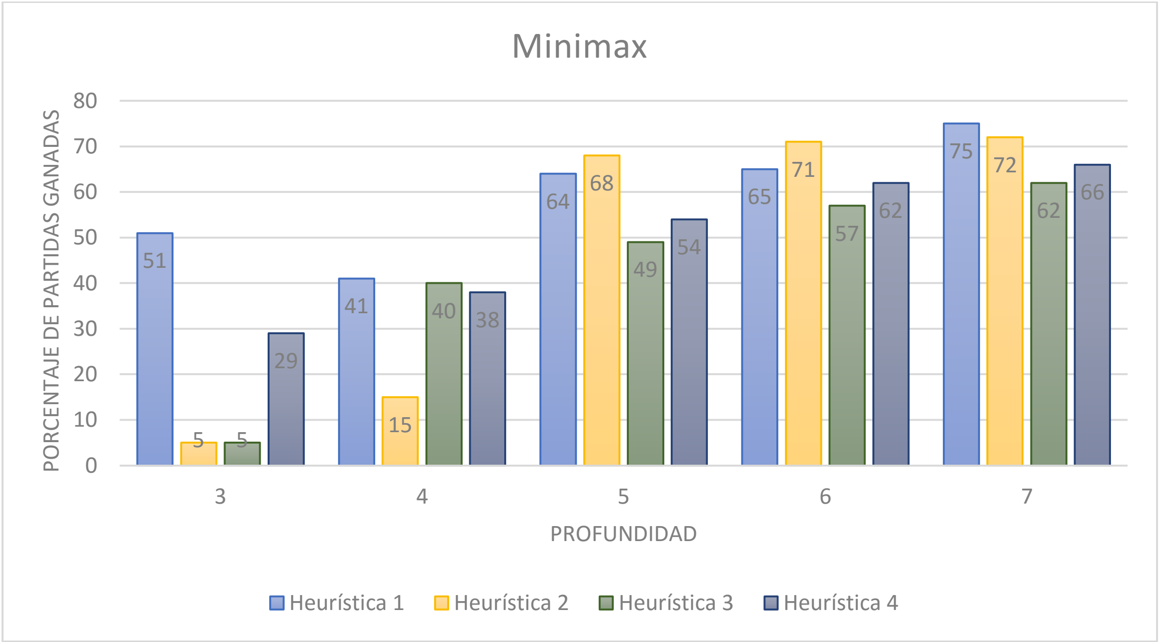

Ilustración 18: Porcentaje de partidas ganadas usando el algoritmo Minimax

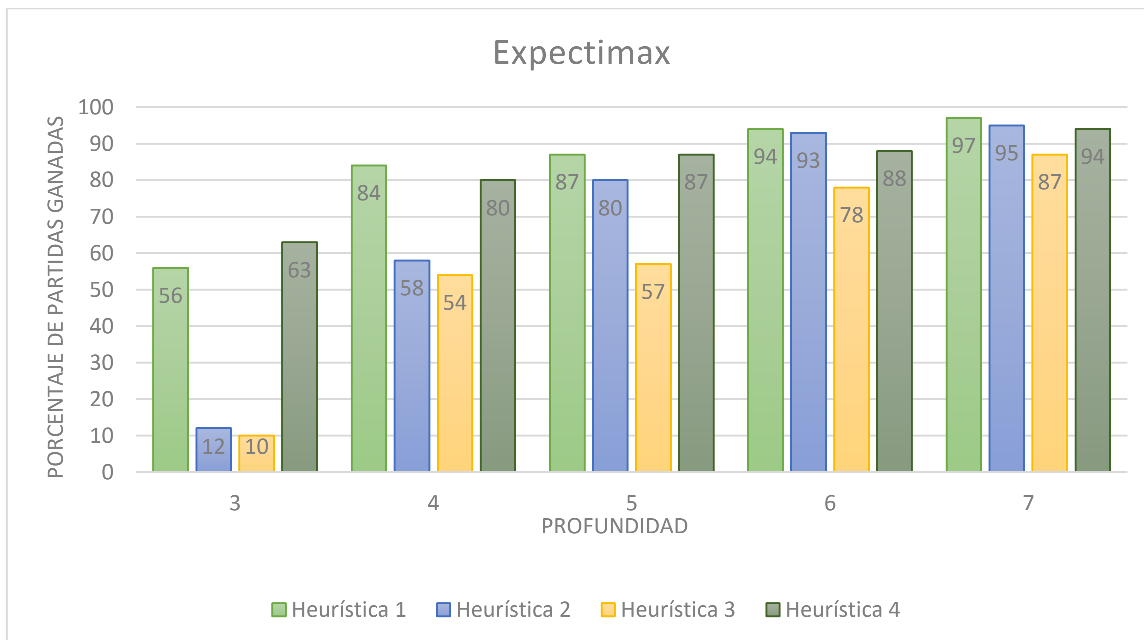

Ilustración 19: Porcentaje de partidas ganadas usando el algoritmo Expectimax

Como vemos, una de las mejores heurísticas es la primera, pues destaca sobre las demás en la mayoría de los casos. Cuando el límite de profundidad es más alto, las heurísticas 1 y 2 están bastante igualadas. Sin embargo, cuando el límite de profundidad es bajo (3 o 4), sin duda es mucho mejor la primera.

Vemos también que en la mayoría de los casos la peor heurística es la tercera, por lo que no nos conviene usarla si lo que queremos es obtener un porcentaje alto de victorias.

Todo esto nos indica que debemos seleccionar la heurística en función de la profundidad y algoritmo que vayamos a usar. Por ejemplo, si queremos usar el Minimax con un límite de profundidad de 3, sin duda deberíamos usar la primera heurística.

Hemos comparado los algoritmos y las heurísticas en función al número de partidas ganadas. Sin embargo, otro parámetro de gran interés a analizar es el tiempo que tarda en resolverse la partida.

Como se puede intuir, el algoritmo Alfa-Beta, será el más rápido. Para comparar los tiempos entre los diferentes algoritmos según la heurística podemos analizar las siguientes imágenes.

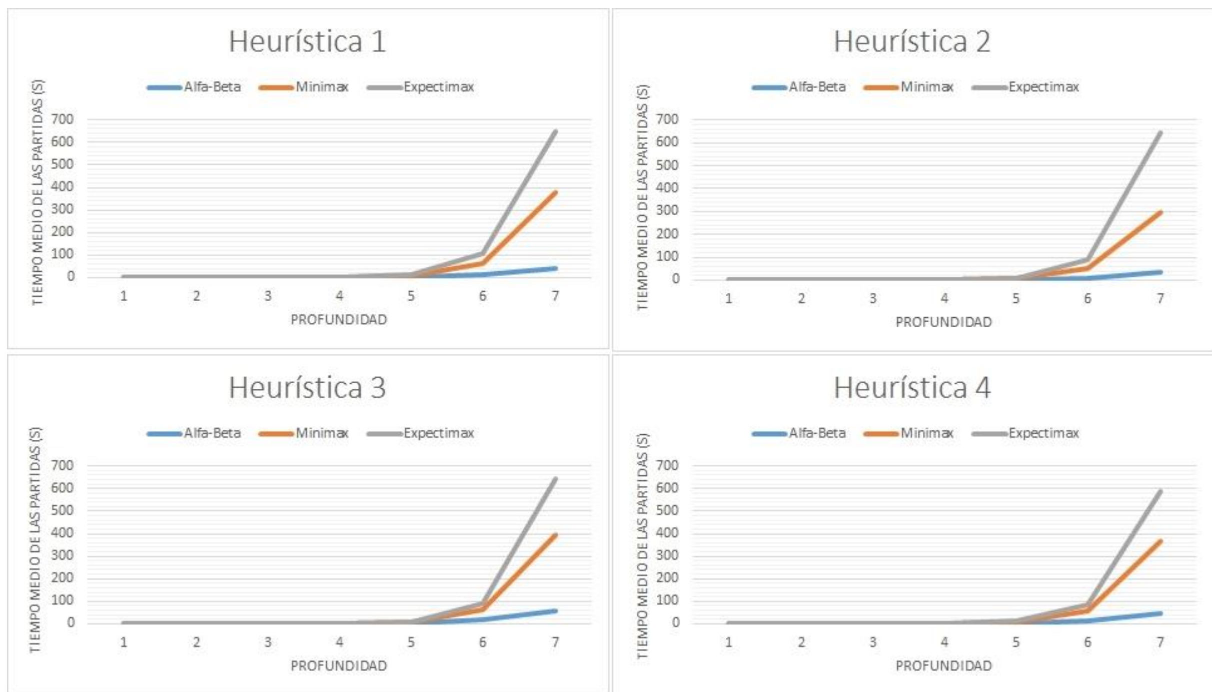

Ilustración 20: Tiempo medio de las partidas para las diferentes heurísticas en función del algoritmo y la profundidad

Se ve claramente que el algoritmo más rápido es, tal y como se esperaba, el Alfa-Beta. Mientras que el Minimax y el Expectimax tienen unos tiempos muy similares, siendo éste último algo más lento, pues tiene que hacer un número mayor de cálculos correspondientes con las probabilidades.

Podemos analizar también cuál de las heurísticas es más rápida o al contrario mediante las siguientes figuras.

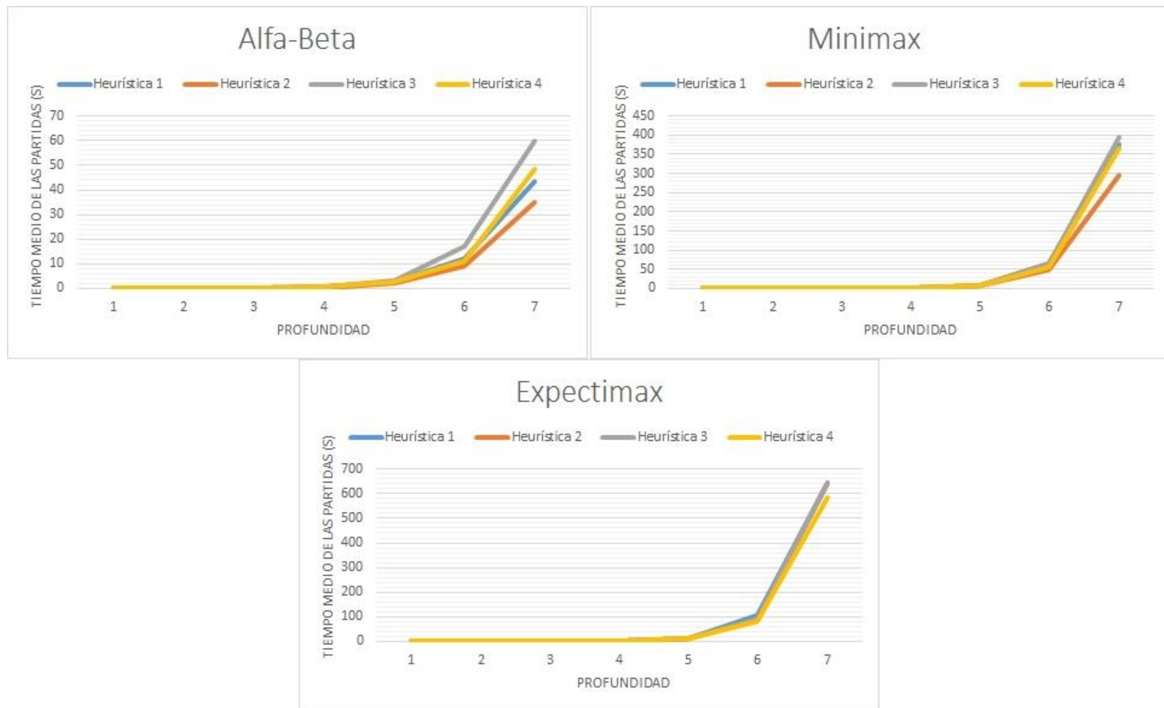

Ilustración 21: Tiempo medio de las partidas para cada algoritmo

Por ejemplo, podemos ver que la heurística más costosa en cuanto al tiempo es la heurística número 3, que además era la que peores resultados obtenía. Por lo que esto nos puede indicar que es mejor descartar dicha heurística, lo que nos indica que la monotonía es una característica muy importante a mantener en el tablero, ya que esta heurística no la tiene en cuenta.

6 Conclusiones

Durante el presente proyecto se ha desarrollado una aplicación que nos permite aplicar diferentes algoritmos y técnicas de la Inteligencia artificial al juego 2048.

Como bien comentábamos en la primera sección, los juegos siempre han sido un reto para la Inteligencia Artificial desde sus inicios, aplicándose cada día tanto a los juegos más novedosos como a los antiguos. Por ello, la asignatura Sistemas Inteligentes se encarga de mostrar dichas técnicas, para lo que hace uso de prácticas de laboratorio.

Con el objetivo de actualizar dichas prácticas, se proponía aplicar todos los conceptos estudiados al juego 2048 de manera que la aplicación sirviera como apoyo a la enseñanza de éstos. Esto nos obligaba a definir varios conceptos como los estados, los movimientos, la función de utilidad y una función de evaluación heurística. En este caso han sido propuestas cuatro funciones de evaluación con el objetivo posterior de probar su funcionamiento.

Por tanto, para aplicar las técnicas estudiadas a este juego, éste debe definirse de manera formal, definiendo los conceptos antes mencionados, lo que hace necesario un análisis del juego.

Una vez definido formalmente el juego, podemos aplicar los algoritmos, en este caso el Minimax, la poda Alfa-Beta y el Expectimax, aplicable únicamente en juegos estocásticos, que incluyan un elemento aleatorio como el 2048.

La aplicación creada es una forma de ver lo que nos permiten estos algoritmos. Sus funcionalidades aparecen en el manual de usuario, en el Anexo B.

Con el fin de estudiar los resultados y comparar los algoritmos utilizados, se ha realizado una batería de pruebas. A partir de su análisis podemos concluir que el algoritmo que mejores resultados ofrece es el Expectimax, pues con límite de profundidad 7 consigue ganar casi todas las partidas. Sin embargo, es el más lento, siendo el más rápido el Alfa-Beta, como era de esperar. Además, también se hace un análisis de las heurísticas propuestas, dependiendo su buen funcionamiento del límite de profundidad establecido. Como ampliación del trabajo podrían proponerse nuevas heurísticas.

Por tanto, se cumplen los objetivos previstos, ya que se ha elaborado la aplicación propuesta, de manera que sirva como práctica de la asignatura y todos los análisis y definiciones que su desarrollo conlleva.

Referencias bibliográficas

[1] Russell, S. and Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3th ed.). Upper Saddle River, NJ: Prentice Hall.: http://aima.cs.berkeley.edu/

[2] *What is the optimal algorithm for the game 2048?*. *Stackoverflow.com*. [Fecha de consulta: 10 agosto 2016] Disponible en: http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

[3] Cirulli, G. *gabrielecirulli/2048*. *GitHub*. [Fecha de consulta: 10 agosto 2016]. Disponible en: https://github.com/gabrielecirulli/2048?files=1

[4] Zettlemoyer, L. (2011). *Artificial Intelligence: Adversarial Search*. [Fecha de consulta: 10 agosto 2016]. Disponible en: https://courses.cs.washington.edu/courses/cse473/11au/slides/cse473au11-adversarial-search.pdf

[5] *Java Platform SE 7*. (2016). *Docs.oracle.com*. [Fecha de consulta: 10 agosto 2016]. Disponible en: https://docs.oracle.com/javase/7/docs/api/

Anexos técnicos

Anexo A. Implementación

El presente trabajo está diseñado de manera que pueda ser de utilidad en la explicación de los contenidos de la asignatura Sistemas Inteligentes.

En la parte práctica de dicha asignatura se utiliza el paquete AIMA, que es un paquete de clases Java que permite definir y resolver problemas de Sistemas Inteligentes de manera que el diseño permite separar la representación del problema de los algoritmos.

Por ello, la implementación del 2048 abarcada en este trabajo está elaborada en función de la biblioteca AIMA, de manera que pueda servir como práctica de la asignatura anteriormente citada.

Implementación del juego

Clase Movimiento

Nos servirá para identificar las acciones del juego.

En este caso vamos a diferenciar dos tipos de movimientos: el que hace el usuario y el que hace el ordenador.

1. Mover las fichas: puede ser en 4 direcciones distintas (arriba, derecha, abajo e izquierda)
2. Añadir una ficha nueva.

La clase contendrá 4 variables que nos permitirá identificar el movimiento:

- boolean mover: true si movemos, false si añadimos
- int x: contendrá la coordenada x de la casilla en el caso de añadir una celda nueva
- int y: contendrá la coordenada y de la casilla en el caso de añadir una celda nueva
- int movimiento:

- si el movimiento trata de mover las fichas, esta variable contendrá la dirección
- si el movimiento trata de añadir una celda nueva, contendrá el valor (2 o 4) de ésta

Por lo tanto, tenemos dos constructores, uno para cada tipo de movimiento.

```
public class Movimiento  {
    public boolean mover; //true si movemos false si añadimos
    public int x; //contendrá la coordenada x de la casilla en el caso de añadir una celda
    public int y; //contendrá la coordenada y de la casilla en el caso de añadir una celda
    public int movimiento;

    public Movimiento(int m){
        mover = true;
        movimiento =m;
        x=-1;
    }

    public Movimiento (int i, int j, int v){
        mover = false;
        x=i;
        y=j;
        movimiento = v;
    }

    public String toString(){
        if(mover){
            switch(movimiento){
            case 0: return "Arriba";
            case 1: return "Derecha";
            case 2: return "Abajo";
            case 3: return "Izquierda";
            }
        }
        return ("Anadir "+ movimiento + " en ["+x+"]["+y+"].");

    }
}
```

Clase G2048State

Nos permite identificar el tablero de la partida y contiene los métodos necesarios para su tratamiento.

```
public static final int COMPUTER = 1; //Jugador computer
public static final int HUMAN = 0; //Jugador humano

private int playerToMove = HUMAN;

public static final int NUM_TILES = 4; //Tamaño del tablero (normalmente 4x4)

public static final int GOAL = 2048; //Objetivo

private int score=0;

private int[][] board; //Contendrá los valores del tablero, de tipo entero

private final Random rnd;

private int heuristic =1;
```

Constructores

Se inicializa la tabla con 0 (casillas vacías) y se añaden dos nuevas con el método nuevaCelda().

```
public G2048State() {
    board = new int[NUM_TILES][NUM_TILES];
    rnd = new Random();

    nuevaCelda();
    nuevaCelda();
}
```

Hay otro constructor que recibe como parámetro una matriz de enteros, un tablero.

```
public G2048State(int [][] b){
    board = new int[b.length][b.length];
    for(int i=0;i<b.length;i++) {
        for(int j=0;j<b.length;j++) {

            board[i][j] = b[i][j];
        }
    }
    rnd = new Random();
}
```

Nueva celda

Este método añade una celda en una posición vacía. Primero se obtiene una lista de celdas vacías mediante el método getVacias(). Si dicha lista no está vacía podremos añadir. Seleccionamos un índice aleatorio entre las vacías y se le añade 2 o 4, de manera que el 2 tendrá una probabilidad de salir del 90% y el 4, un 10%.

```java
public Movimiento nuevaCelda() {
    List<Integer> vacias = getVacias(); //Obtenemos la lista de celdas vacías
    if(vacias.isEmpty()){
        return null;  //Si no hay ninguna no podemos añadir
    }

    int index= vacias.get(rnd.nextInt(vacias.size())); //Obtenemos un índice aleatorio
    //Obtenemos un valor con 90% de probabilidad de que sea 2 y 10% de probabilidad de que sea 4
    int value = (rnd.nextDouble() < 0.9) ? 2:4;

    //Calcular coordenadas
    int x = index/NUM_TILES;
    int y = index%NUM_TILES;

    rellenarCelda(x,y,value); //Colocar la celda nueva en la vacía

    playerToMove = (playerToMove == HUMAN ? COMPUTER : HUMAN); //Se cambia el jugador
    return new Movimiento(x,y,value);

}
```

Rellenar celda

Simplemente se cambia el valor de la celda, se rellena.

```java
public void rellenarCelda(int i, int j, int value) {
    if(board[i][j]==0) board[i][j]=value;  //Coloca una celda en una vacía
}
```

Obtener vacías (getVacias())

Se recorre la tabla y se añade a una lista los índices de las casillas vacías.

```java
public List<Integer> getVacias() {
    List<Integer> vacias = new ArrayList<Integer>();
    for(int i =0; i < NUM_TILES; i++){
        for(int j =0; j < NUM_TILES; j++){
            if(board[i][j] == 0) {
                vacias.add(i*NUM_TILES+j); // así conseguimos numerar del 0 al 15
            }
        }
    }
    return vacias;
}
```

Goal

Devuelve true si se ha alcanzado en alguna casilla el valor GOAL.

```
public boolean goal() {
    for(int i =0; i < NUM_TILES; i++){
        for(int j =0; j < NUM_TILES; j++){
            if(board[i][j]>=GOAL) return true;
        }
    }

    return false;
}
```

Mover celdas

Este método realiza los movimientos que puede hacer el usuario en el tablero (arriba, derecha, abajo e izquierda). Por simplificar, he realizado el algoritmo para hacer el movimiento hacia la izquierda, de manera que si se requiere mover en otra dirección, se gire el array y se use el mismo algoritmo en todos los casos.

1. Rotamos tantas veces como sea necesario.
2. Algoritmo de movimiento de las celdas hacia la izquierda.
3. Se recorre y se ejecuta el algoritmo para cada una de las filas del tablero. Sintéticamente, podemos decir que lo que hacemos es comprobar qué hay delante de la celda para ver si podemos moverla (al ser celdas vacías las anteriores o sumarla con la anterior si tiene el mismo valor).
4. Volvemos a rotar para obtener la orientación original

```
	public int moveCells(Integer direction) {

	int sumas = 0;

	//Rotamos el tablero para hacer sólo los movimientos como si la direccion fuera la izquierda
	if(direction==0) { //arriba -> una rotación a la izquierda
		rotarIzq();
	}else if(direction==1) {//derecha -> dos rotaciones
		rotarIzq();
		rotarIzq();
	}else if(direction==2) { //abajo -> una rotación a la derecha
		rotarDer();
	}

	for(int i=0; i < NUM_TILES; i++) {
```

```
            int indiceAux=0;
            for(int j=1; j < NUM_TILES; j++) {
            if(board[i][j]!=0) {
                  //Comprobamos lo que hay antes de la celda para ver si
podemos mover o sumar
                  int anterior = j-1;
                  while(anterior  >  indiceAux  &&  board[i][anterior]  ==0)
anterior--;

                  if(board[i][anterior]==0){    //todas    las    posiciones
anteriores están vacias
                        board[i][anterior]= board[i][j];
                        board[i][j]=0;
                  }  else  if(board[i][anterior]  ==  board[i][j]){  //se
encuentra con uno anterior del mismo valor -> podemos unirlos
                        board[i][anterior]= board[i][anterior]*2;
                        board[i][j]=0;
                        sumas = sumas+board[i][anterior];
                        indiceAux=anterior+1; //indicará hasta donde hemos
mezclado

                  }else if(anterior+1!=j){ //el anterior no es del mismo
valor -> lo colocamos al lado
                        board[i][anterior+1]=board[i][j];
                        board[i][j]=0;
                  }
            }

            }
      }


      score+=sumas;

      //Rotamos de nuevo para volver a la orientación original
      if(direction==0) {
            rotarDer();
      }
      else if(direction==1) {
            rotarDer();
            rotarDer();
      }
      else if(direction==2) {
            rotarIzq();
      }
      return sumas;
      }
```

Rotar

Rotar las matrices.

```
private void rotarIzq() {
    int[][] newBoard = new int[NUM_TILES][NUM_TILES];

    for(int i =0; i < NUM_TILES; i++){
        for(int j =0; j < NUM_TILES; j++){
            newBoard[NUM_TILES- 1 - j][i] = board[i][j];
        }
    }

    board = newBoard;
}


private void rotarDer() {

    int[][] newBoard = new int[NUM_TILES][NUM_TILES];

    for(int i =0; i < NUM_TILES; i++){
        for(int j =0; j < NUM_TILES; j++){
            newBoard[i][j]=board[NUM_TILES-1-j][i];
        }
    }

    board = newBoard;
}
```

Completo

Devuelve true si no hay celdas adyacentes con el mismo valor ni casillas vacías.

```
public boolean completo(){
    for (int x = 0; x < NUM_TILES; x++) {
            for (int y = 0; y < NUM_TILES; y++) {
                if (board[x][y] ==0||
                        (x>0 && board[x][y] == board[x-1][y]) ||
                        (x<NUM_TILES-1&& board[x][y] == board[x+1][y])||
                        (y>0&& board[x][y] == board[x][y-1])||
                        (y<NUM_TILES-1&& board[x][y]== board[x][y+1])) {
                        return false;
                }
            }
    }
    return true;
}
```

Terminal

Método que devuelve true si el estado es terminal, es decir, si no se puede hacer ningún movimiento.

En este juego hay dos posibilidades para que un estado sea terminal:

1. Se alcance el valor objetivo. Lo comprobamos mediante el método goal()

2. No haya movimientos disponibles. En el caso de que haya celdas vacías, siempre habrá movimientos. Además, existirán movimientos disponibles si podemos sumar algunas casillas del tablero, de manera que queden agrupadas. Para comprobar eso utilizamos el método anterior completo().

```
public boolean isTerminal()  {
    if(goal()) {
        return true;
    }else {
        return completo(); //si esta completo será terminal
    }

}
```

getActions

Devuelve una lista de movimientos que se pueden hacer desde el tablero actual.

En el caso de que el jugador al que le toca el turno sea el humano:

Comprobamos, para cada dirección, si al realizar ese movimiento sobre una copia, el tablero cambia, pues en ese caso sí se puede realizar el movimiento. En caso de que el tablero se quede igual no podremos realizar dicho movimiento. Es decir, si al mover hacia la izquierda no varía nada, no será un movimiento posible.

En el caso de que sea el turno del ordenador:

Los movimientos disponibles serán añadir un 2 o un 4 a cada una de las celdas vacías.

```java
public List<Movimiento> getActions() { //devuelve una lita de movimientos que puedo hacer
    List<Movimiento> lista = new ArrayList<Movimiento>();
    if(playerToMove == HUMAN){
        G2048State copyBoard;

        for(int i =0; i < 4; i++){ //comprobar para las 4 direcciones
            copyBoard = (G2048State) this.clone(); //se hace una copia
            copyBoard.moveCells(i); // se realiza el movimiento
            //si ha cambiado el tablero es que puedo hacer dicho movimiento
            if(!copyBoard.isEqual(this.board)) lista.add(new Movimiento(i));
        }

    }else if(playerToMove == COMPUTER){
        //Debo añadir, para cada celda vacía el movimiento de añadirle una celda con 2 o 4.
        List<Integer> vacias = this.getVacias(); //celdas vacias
        int[] valores = {2, 4}; //posibles valores

        int x,y; //coordenadas
        for(Integer celda : vacias) {
            x = celda/NUM_TILES;
            y = celda%NUM_TILES;
            for(int value : valores) {
               lista.add(new Movimiento(x,y, value));
            }
        }

    }else{
        System.out.println("ERROR: Jugador no existe");
    }
    return lista;
}
```

Move

Este método recibe una variable de tipo Movimiento, que indicará el movimiento a realizar sobre el tablero.

En el caso de que la variable mover del objeto action sea true, llamaremos al método moveCells con la dirección que almacena dicho objeto. En el caso de que sea false, significa que tendremos que añadir una nueva celda, por lo que llamamos al método rellenarCelda, en las coordenadas y con el valor que nos indica action.

```java
public int move(Movimiento action) {
    int m=0;
    if(action == null) System.out.println("Recibo un movimiento nulo");
    if(action.mover){  //Mover en una direccion
        m=moveCells(action.movimiento);
    }else{ //Rellenar celda con coordenadas y valor dado.
        rellenarCelda(action.x, action.y, action.movimiento);
    }

    playerToMove = (playerToMove == HUMAN ? COMPUTER : HUMAN);
    return m;
}
```

Getters

```
public int getPlayerToMove() {
    return playerToMove;
}

public int getScore() {
    return score;
}

public int[][] getBoard() {
    return copiarArray(this.board);
}

public int numVacias() {
    return getVacias().size();
}

public Random getRandom() {
    return rnd;
}
```

getUtility()

Da un valor de “cómo de bueno es el estado”. En este caso, si el estado no es terminal he usado una función heurística.

```
public double getUtility() {
    /*
     * Da un valor de utilidad al estado
     */
    double valor;
    if(isTerminal()) {
        if(goal()) {
            valor=Double.MAX_VALUE;
        }
        else {
            valor=0;
        }
    }else{ //si no es final le damos el valor de una heurística

        switch(this.heuristic){
            case 1: valor=heuristica1(); break;
            case 2:valor=heuristica2(); break;
            case 3:valor=heuristica3(); break;
            case 4:valor=heuristica4(); break;
            default: valor =0;
        }

        if(valor < 0) valor =0;

    }
    return valor;
}
```

Similitud

Puesto que nos interesa que los números iguales o, al menos, parecidos estén cercanos, obtengo las distancias entre los vecinos de cada celda, acumulando la media de las distancias para cada celda con sus vecinos.

```
private int similitud() {
   int similitud=0;

   for(int i =0; i < NUM_TILES; i++){
        for(int j =0; j < NUM_TILES; j++){
            if(board[i][j]!=0) {
                int numVecinos=0;
                int sum=0;
                for(int k =-1; k <= 1; k++) {
                    int x=i+k;
                    if(x >= 0 && x<board.length) {
                        for(int l =-1; l <= 1; l++) {
                            int y = j+l;
                            if(y >= 0 && y<board.length) {
                                if(board[x][y]>0) {
                                    numVecinos++;
                                    sum+=Math.abs(board[i][j]-board[x][y]);
                                }
                            }
                        }
                    }
                }

                similitud+=sum/numVecinos;
            }
        }
    }

    return similitud;
}
```

Matriz valores

Como se ha comentado en la sección de Heurísticas, uno de los trucos principales que la mayoría de jugadores sigue al jugar al juego es colocar la casilla con mayor valor en una esquina y, a partir de ella ir decreciendo su valor. Normalmente también evitaremos que las casillas con valores más pequeños queden aisladas. Esta heurística nos permite tender a tener un tablero más organizado. Para ello, con este método, sumamos los valores de las casillas del tablero ponderados de la manera siguiente:

7	6	5	4
6	5	4	3
5	4	3	2
4	3	2	1

```
public int matrizValores(){
    int suma = 0;
    int ponderacion = 7;
    for(int f =0; f < NUM_TILES; f++){
        for(int i =f; i >= 0; i--){
            int j = f-i;
            suma += board[i][j]*ponderacion;
        }
        ponderacion--;
    }


    int y2 = 1;
    while(y2 < NUM_TILES){
        int x= NUM_TILES-1;
        int y = y2;
        while(x < NUM_TILES && y < NUM_TILES && x >= 0 && y >= 0 ){
            suma += board[x][y]*ponderacion;
            x--;
            y++;
        }
        y2++;
        ponderacion--;
    }

    return suma;
}
```

Heurísticas

Calculan las diferentes heurísticas propuestas a partir de los métodos anteriores.

```
private double heuristica1() {
    double actualScore = getScore();
    double smoothness = similitud();
    double matrix =(double) matrizValores();

    double suma =
            actualScore+matrix-smoothness+ Math.log(actualScore)*numVacias();
    return suma;

}
private double heuristica2() {
    double actualScore = getScore();
    double smoothness = similitud();
    double matrix =(double) matrizValores();

    double suma =
                 matrix-smoothness+ Math.log(actualScore)*numVacias();

  return suma;

}
 private double heuristica3() {
   double actualScore = getScore();
   double smoothness = similitud();

   double suma =
              actualScore-smoothness+ Math.log(actualScore)*numVacias();

   return suma;

 }

 private double heuristica4() {
   double actualScore = getScore();
   double max = this.maxValue();
   double matrix2 =(double) matrizValores();
   double suma =
              actualScore + (numVacias() * max) + matrix2;

   return suma;

 }
```

Clone

```
    @Override
    public Object clone()  {
        G2048State copy=null;
        try {
            copy = (G2048State)super.clone();
            copy.board = copiarArray(board);
        } catch (CloneNotSupportedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        return copy;
    }
    private int[][] copiarArray(int[][] tablero) {
        int[][] copy = new int[tablero.length][tablero.length];
        for(int i=0; i < tablero.length; i++) {
            copy[i] = tablero[i].clone();
        }
        return copy;
    }
```

isEqual

```
    public boolean isEqual(int[][] newboard) {
        for(int i=0;i<this.board.length;i++) {
            for(int j=0;j<this.board.length;j++) {
                if(this.board[i][j]!= newboard[i][j]) {
                    return false;
                }
            }
        }
        return true;
    }
```

toString

```
@Override
public String toString() {
    StringBuilder strBuilder = new StringBuilder();
    for (int row = 0; row < NUM_TILES; row++) {
        for (int col = 0; col < NUM_TILES; col++) {
            strBuilder.append(board[row][col] + "\t");
        }
        strBuilder.append("\n");
    }
    return strBuilder.toString();
}
```

Clase Game2048

Esta clase implementará la interfaz Game<STATE, ACTION, PLAYER>.

En este caso, el estado vendrá dado por la clase G2048State. Las acciones será de la clase Movimiento. Los jugadores (humano y ordenador) serán representados por enteros.

```
public class Game2048 implements Game<G2048State, Movimiento,Integer> {

    G2048State initialState;

    public Game2048(){
        initialState = new G2048State();
    }

    public Game2048(int [][]b){
        initialState = new G2048State(b);
    }
    @Override
    public G2048State getInitialState() {
        return initialState;
    }

    public G2048State getEmptyState() {
        int [][] board = new int[G2048State.NUM_TILES][G2048State.NUM_TILES];
        G2048State newState = new G2048State(board);
        return newState;
    }

    @Override
    public Integer[] getPlayers() {
        return new Integer[] { G2048State.HUMAN, G2048State.COMPUTER };
    }

    @Override
    public Integer getPlayer(G2048State state) {
        return state.getPlayerToMove();
    }
```

```java
    public List<Movimiento> getActions(G2048State state) {
        return state.getActions();
    }

    @Override
    public G2048State getResult(G2048State state, Movimiento action) {
        G2048State result = (G2048State) state.clone();
        result.move(action);
        return result;
    }

    @Override
    public boolean isTerminal(G2048State state) {
        return state.isTerminal();
    }

    @Override
    public double getUtility(G2048State state, Integer player) {
        return state.getUtility();
    }

    public int movimiento(G2048State state,Movimiento action){
        return state.move(action);

    }

    public Movimiento nuevaCelda(G2048State state) {
          return state.nuevaCelda();
    }


}
```

Implementación de los algoritmos

Además, tal y como hemos comentado antes, el algoritmo Minimax es imposible de utilizarlo en la práctica en problemas grandes, pues la complejidad es demasiado alta.

Una manera de optimizar dicho algoritmo era limitando la profundidad, por lo que he creado las clases necesarias para usar los algoritmos Minimax, Alfa-beta y Expectimax con limitación de la profundidad.

El código de los algoritmos Minimax y Alfa-beta es similar al utilizado en las clases que proporciona AIMA, simplemente se le introduce la opción del límite de profundidad.

Para ello debemos añadir un argumento a las funciones minValue, maxValue, la profundidad e ir decrementándolo en cada llamada que se haga a las funciones, de manera que cuando se alcance una profundidad igual a 0, se considere que dicho estado es un nodo hoja y se devuelva su evaluación, como si se tratara de un estado terminal.

```
if (game.isTerminal(state) || depth==0 ) return game.getUtility(state, player);
```

En el caso del algoritmo Expectimax, se usa la misma función maxValue del algoritmo Minimax limitado y debe implementarse la función expValue, pues no es proporcionado por AIMA.

```
public double expValue(G2048State state, Integer player, int depth) { // returns an
    //expected value

    expandedNodes++;

    if (game.isTerminal(state) || depth ==0)  return game.getUtility(state, player);

    double value = 0;
    for (Movimiento action : game.getActions(state)){
        double probability = ((action.movimiento == 2)?0.9:0.1)/state.numVacias();
        value = value +
                probability*maxValue(game.getResult(state, action), player, depth-1);
    }
    return value;
}
```

Implementación gráfica

La clase G2048App contiene el código asociado a la interfaz gráfica. A su vez contiene clases asociadas a los paneles, frames y dialogs de los que hace uso. Describo a continuación lo más relevante del proyecto: la clase GPanel, encargada de dibujar el tablero y el método solveAndPaint cuyo objetivo es resolver la partida e ir mostrando la ejecución de la misma.

Clase GPanel

Los métodos getTileColor y getTextColor devolverán el color del fondo de la celda y del número de la misma en función de dicho valor.

Para dibujar el tablero usaremos dos bucles for anidados.

En las variables *x* e *y* tendremos la posición a dibujar, y variarán en función del tamaño del marco (variable *marco*) y del tamaño de la celda (*tamCelda*).

Mediante el método anterior getTileColor, obtendremos el color de la celda, para posteriormente usar la función fillRect en la posición (x,y).

Para dibujar el valor dentro de la casilla, en primer lugar lo pasamos a String y calculamos, mediante funciones de FontMetrics su anchura y altura, para poder dibujarlo centrado en la celda.

Una vez conocidos dichos parámetros, pasaremos a dibujar su valor mediante la función drawString.

Finalmente, tras dibujar el tablero, se comprueba si es un estado terminal para mostrar los mensajes "¡Has ganado!" o "¡Has perdido!" en los casos correspondientes.

Debajo del tablero se muestra la puntuación actual.

Método solveAndPaint()

Otro método relevante del proyecto es el solveAndPaint, cuyo objetivo es, como su nombre indica, resolver y pintar el tablero a medida que va eligiendo los movimientos. Por ello es necesario que se lance una hebra.

En primer lugar se eligen el algoritmo y la heurística a utilizar accediendo a la opción seleccionada en un ComboBox. Posteriormente, hasta que se alcance un estado terminal,se hace uso del método makeDecision que tienen las clases de los algoritmos. Dicho método devolverá el movimiento a realizar. Además también se llama al método nuevaCelda, para colocar una celda nueva después de ejecutar el movimiento devuelto por el método makeDecision.

En cada iteración del bucle se llamará a repaint(), de manera que se volverá a dibujar el tablero, actualizándose.

Existe las opciones de escribir los movimientos en un fichero, siempre que la opción *recorder*, esté activada.

El método proposeMove() es similar a este, sin embargo sólo devuelve un movimiento, no es necesario un bucle y, por tanto, no es necesario lanzar una hebra.

El código restante hace referencia a los diferentes elementos de las ventanas para las distintas funcionalidades de la aplicación.

Existen métodos correspondientes a la lectura y escritura de los tableros en ficheros, que permiten la funcionalidad de guardar y reproducir partidas.

Hay varias clases correspondientes a los JFrames, JDialog y JPanel utilizados, que hacen uso de los múltiples diversos componentes de Java Swing para la composición de la interfaz como JMenu, JFileChooser, JButton, JComboBox, JCheckBoxMenuItem, JLabel, JScrollPanel… entre otros.

Para recoger la entrada de teclado mediante las flechas en el juego, se hace uso de la clase KeyEventDispatcher ya que KeyListener, utilizado en las prácticas de la asignatura, no funciona al tener varios paneles. De esta manera también sirve de ejemplo, pues es una manera alternativa de recoger la entrada por teclado.

En caso de querer consultar el código al completo, se recomienda acceder al proyecto Java.

Anexo B. Manual de usuario

La aplicación desarrollada permite al usuario una serie de funcionalidades:

- Jugar mediante las flechas de teclado.
- Pedir un movimiento recomendado.
- Resolver la partida.
- Guardar una partida.
- Reproducir una partida guardada.
- Realizar pruebas

Al ejecutar la aplicación, en la parte central de la ventana podemos ver el tablero de 4x4 celdas, inicialmente vacío.

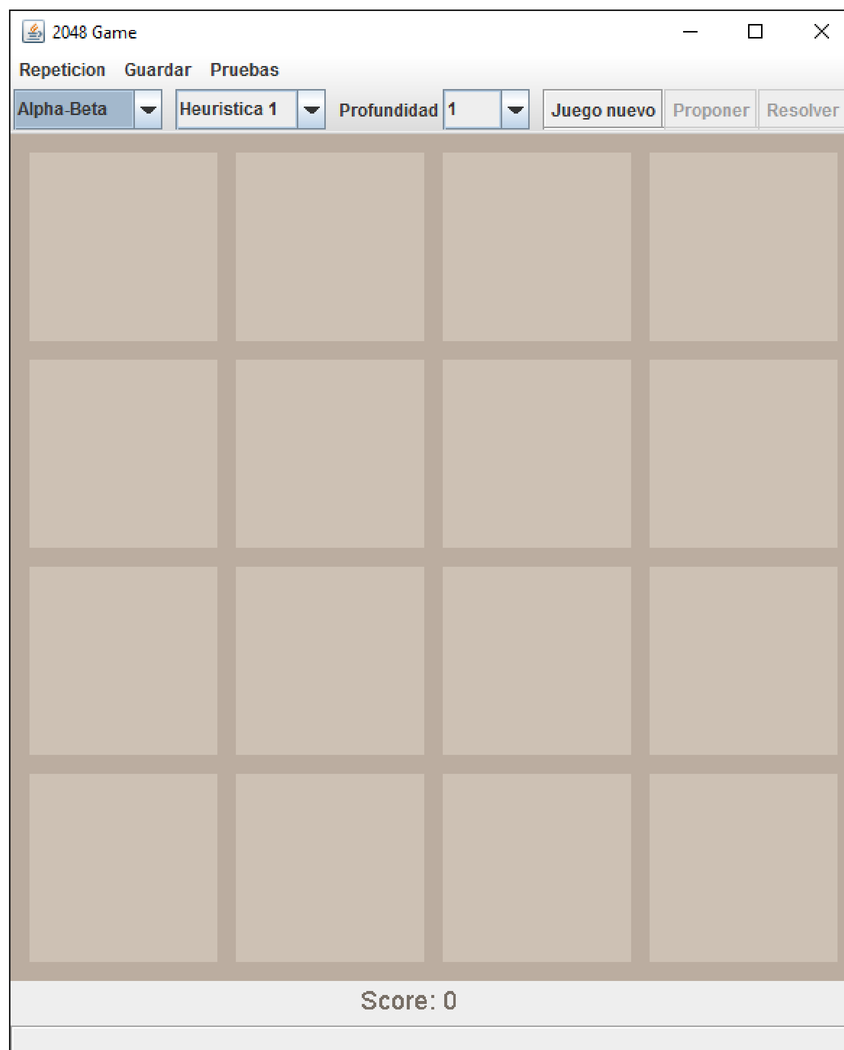

Ilustración 22: Aspecto inicial de la aplicación

Para empezar una partida nueva debemos clicar en el botón “Juego nuevo” en la parte superior de la ventana (Figura 22). Antes de pulsar dicho botón, podemos ver en la

imagen 22, que las opciones de “Proponer” o “Resolver” están desactivadas ya que primero debe iniciarse el juego.

Una vez hayamos pulsado sobre “Juego nuevo”, el tablero se inicializará y aparecerán valores en dos de las casillas. A partir de ese momento ya sí se nos permite pulsar en “Proponer” o “Resolver”, al igual que usar las flechas del teclado para jugar de la manera tradicional al juego.

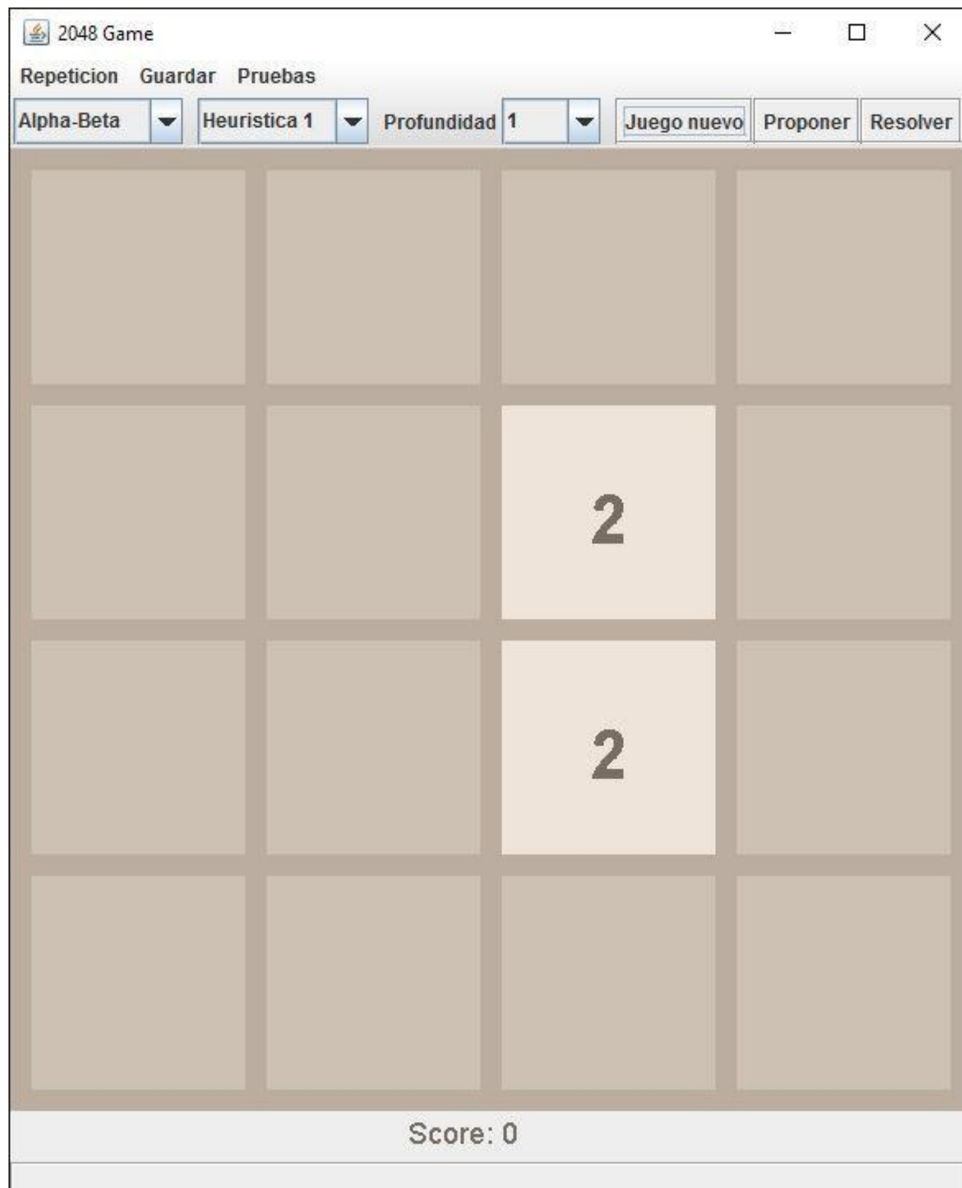

Ilustración 23: Aspecto de la aplicación al iniciar un nuevo juego

Antes de usar los botones “Proponer” y “Resolver”, debemos seleccionar:

- El algoritmo a usar: Alpha-Beta, Minimax o Expectimax.
- La heurística a usar entre las cuatro propuestas en el presente documento.
- Profundidad de la exploración.

Las dos primeras opciones podremos seleccionarlas mediante dos desplegables situados en la parte superior (Figura 24).

Ilustración 24: Desplegables para elegir el algoritmo y la heurística a utilizar

Para elegir la profundidad, tenemos otro desplegable (figura 25) en el cual podremos elegir una profundidad entre 1 y 7, de manera que si queremos seleccionar una profundidad mayor deberemos seleccionar la opción “Más” que abrirá un cuadro de diálogo donde podremos escribir la profundidad deseada (figura 26).

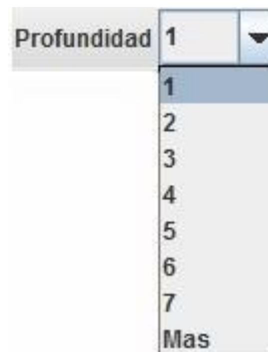

Ilustración 25: Desplegable para elegir la profundidad

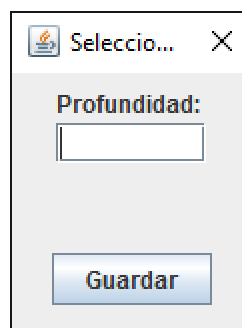

Ilustración 26: Ventana para introducir el valor de la profundidad personalizado

Una vez elegidos los parámetros deseados, podremos pedir que se efectúe un solo movimiento tomando la decisión aplicando el algoritmo elegido o pedir que se resuelva el juego al completo. Para ello sirven los botones “Proponer” y “Resolver”.

Por ejemplo, tenemos el siguiente juego (figura 27) con los parámetros que aparecen seleccionados:

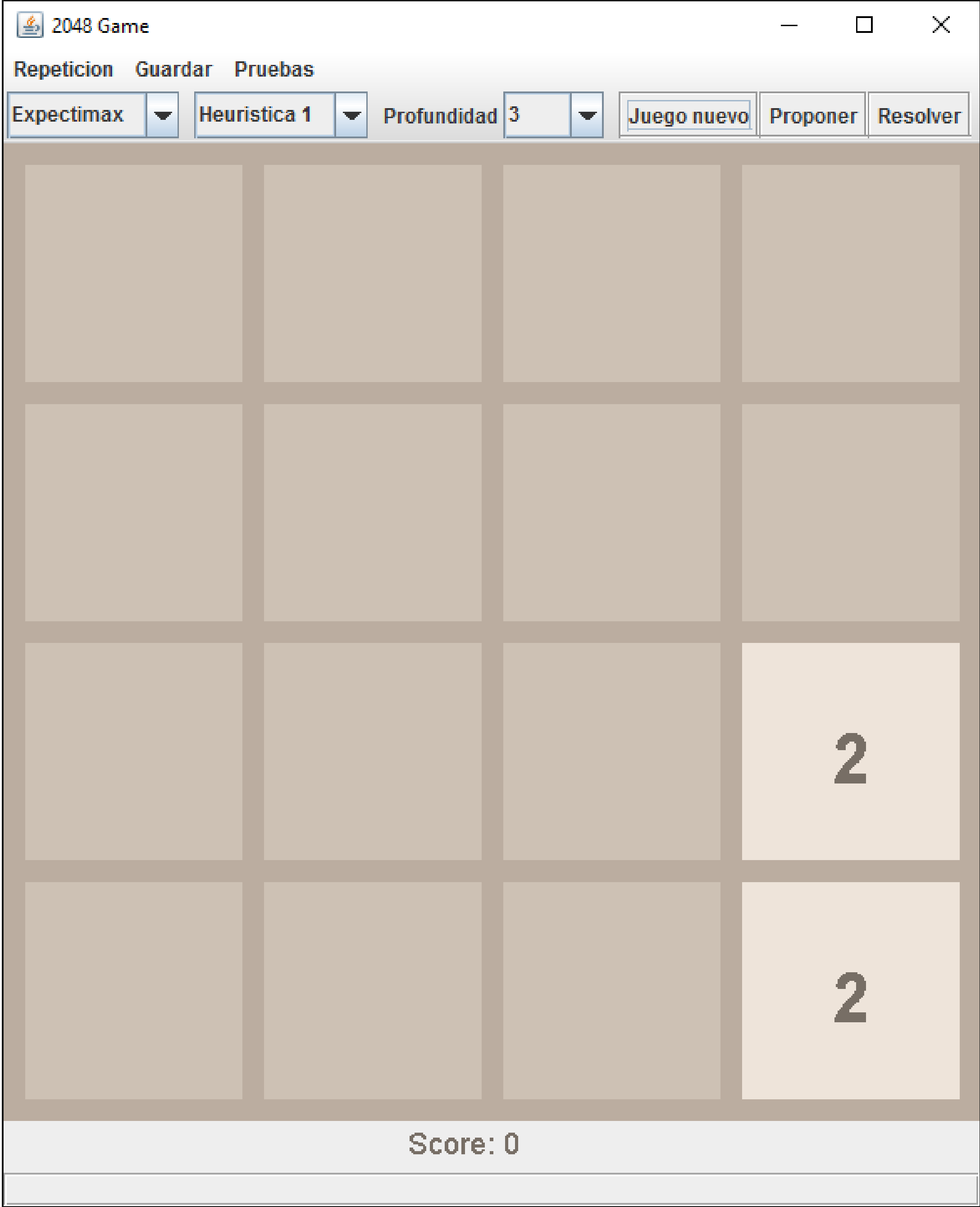

Ilustración 27: Ejemplo de tablero inicial

Pulsamos “Proponer” y obtenemos:

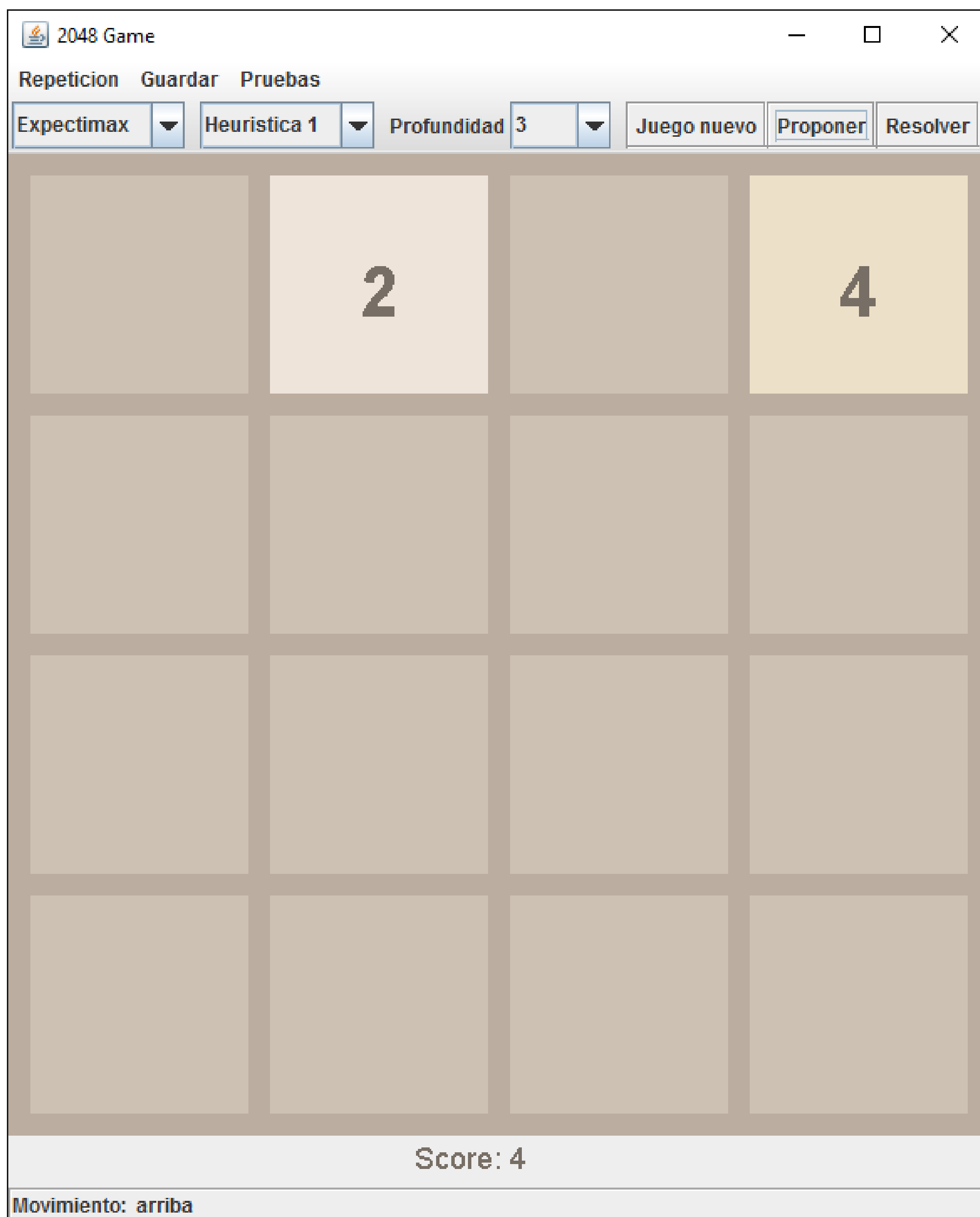

Ilustración 28: Tablero obtenido tras pulsar la opción “Proponer” con los parámetros seleccionados

Como se puede ver en la imagen 28, el movimiento seleccionado por el algoritmo es desplazar las casillas hacia arriba. El movimiento es efectuado, viéndose así en el nuevo aspecto del tablero y en la actualización de la puntuación, que pasa a ser 4. Además, en la barra inferior de la ventana podemos ver cómo, efectivamente, el movimiento escogido fue “arriba”.

Ahora tenemos tres opciones para continuar la partida: continuar jugando mediante las flechas, volver a pedir un movimiento o pedir que se resuelva por completo.

Si elegimos esta tercera opción debemos pulsar el botón “Resolver”. Tras pulsarlo, el tablero comenzará a resolverse, de manera que la velocidad entre los movimientos dependerá del algoritmo utilizado y la profundidad seleccionada. Finalmente se obtendrá un tablero en el cual aparezca la casilla 2048 o en el que no se puedan realizar más movimientos.

Ilustración 29: Tablero resuelto tras pulsar la opción “Resolver”

También tenemos la opción de guardar partidas (figura 30) para posteriormente reproducirlas. Si queremos guardar la partida debemos seleccionar la opción de “Guardar partida”, en el menú “Guardar”, que inicialmente no está seleccionada.

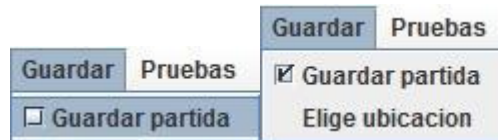

Ilustración 30: Menús para guardar partidas

Podemos elegir el nombre del fichero y dónde guardarlo, mediante “Elegir ubicación” en el mismo menú (figura 30). En ese caso se abrirá un explorador de archivos que nos permitirá elegir dónde vamos a guardar el archivo y el nombre de este (figura 31).

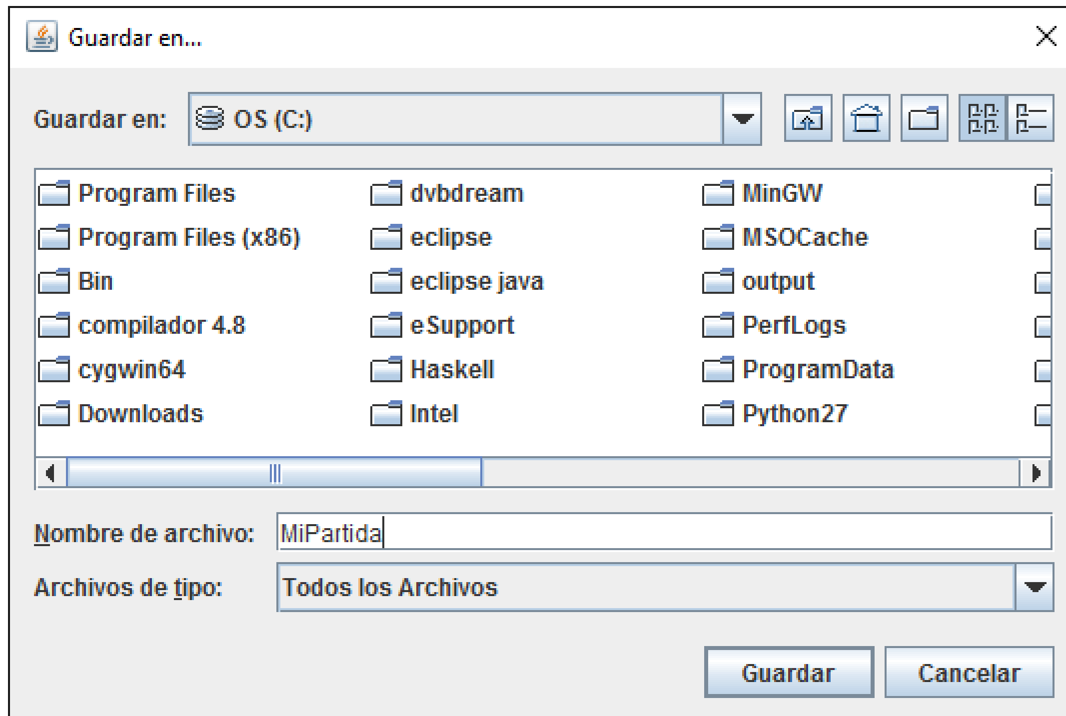

Ilustración 31: Explorador para elegir ruta y nombre del fichero

Si no elegimos ningún nombre ni ubicación, el fichero tendrá como nombre la fecha y hora en el que es creado y se almacenará en la ruta de la aplicación.

Este proceso podemos realizarlo en cualquier momento de la partida, no es necesario que se comience a guardar la partida desde el inicio.

Cuando queramos reproducir una partida almacenada debemos pulsar “Elegir archivo” dentro del menú “Reproducir” (figura 32).

Ilustración 32: Menú para repetir partidas

Una vez pulsado se abrirá un explorador de archivos que nos permitirá seleccionar el fichero que contenga la partida que deseamos reproducir. Al seleccionar el archivo de la partida se abrirá una ventana nueva con el aspecto mostrado en la figura 33.

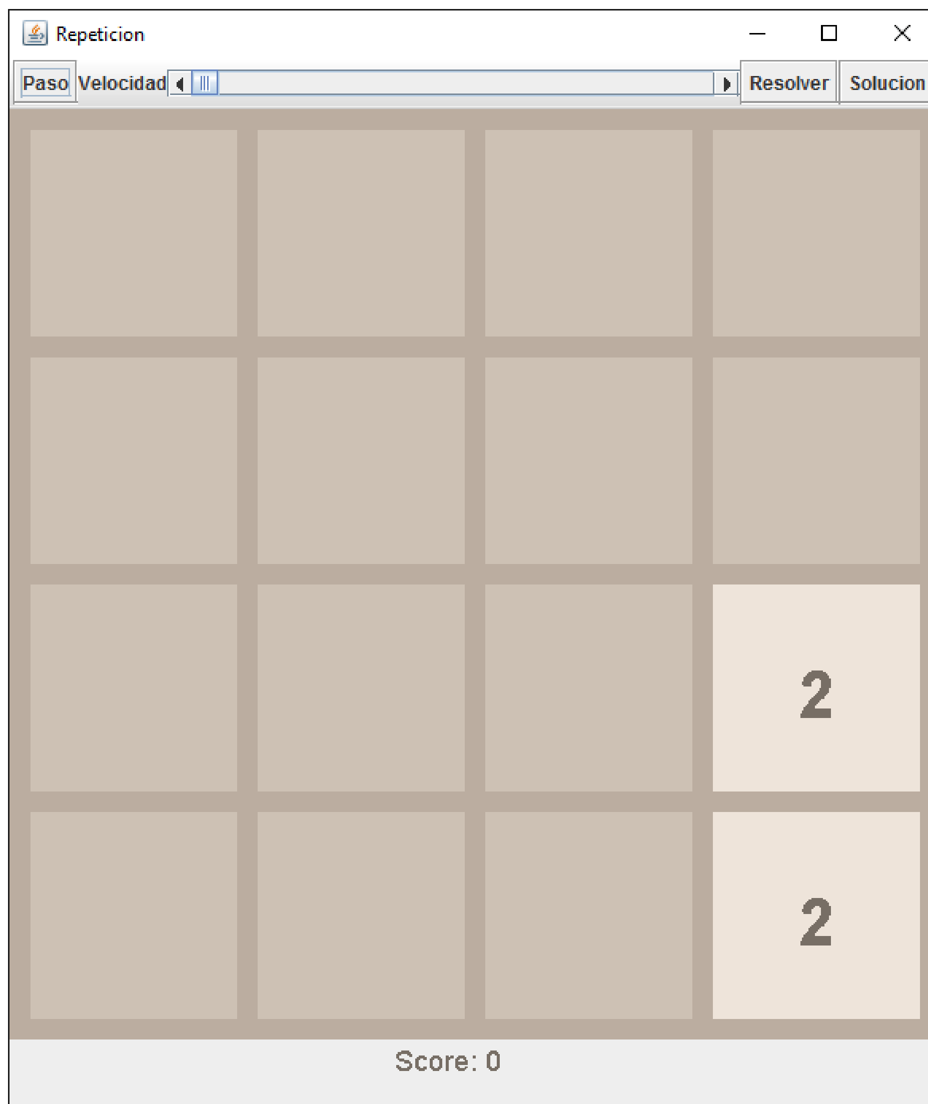

Ilustración 33: Ventana de repetición de una partida

Veremos el tablero inicial, el primero que se grabó. En este caso se trata del tablero inicial del ejemplo anterior.

A partir de éste podremos reproducir la partida de diferentes formas.

Al clicar en el botón “Paso” se ejecutará un solo paso, tal y como indica su nombre. Si queremos reproducir la partida al completo, podemos elegir la velocidad entre sus pasos mediante la barra central y posteriormente pulsar en “Resolver”. Cuanto más a la derecha desplacemos la barra más rápido se reproducirá. Si únicamente queremos

ver la solución obtenida debemos pulsar “Solución”. Además, a medida que se reproduce la partida también veremos el Score actualizado.

Pulsando “Solución” en el ejemplo anterior obtendremos:

Ilustración 34: Tablero obtenido tras reproducir la partida guardada

Podemos ver que es la misma solución obtenida durante la ejecución de la partida.

Otra funcionalidad de la aplicación es la posibilidad de realizar pruebas. Para ello debemos seleccionar la opción “Realizar pruebas” del menú Pruebas.

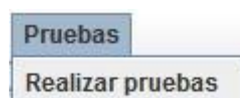

Ilustración 35: Menú para realizar pruebas

Una vez seleccionado se abrirá una ventana nueva con el aspecto de la figura 36. Como vemos en la figura 37, podremos seleccionar el algoritmo, la heurística, la profundidad y el número de pruebas a realizar. Una vez seleccionadas las opciones deseadas pulsamos “Iniciar pruebas”. Se realizarán tantas partidas como se haya

seleccionado, y se indicará si se ha ganado o perdido y el porcentaje de ganadas. Cuando se concluyan todas, además aparecerá una media del tiempo de las partidas en segundos.

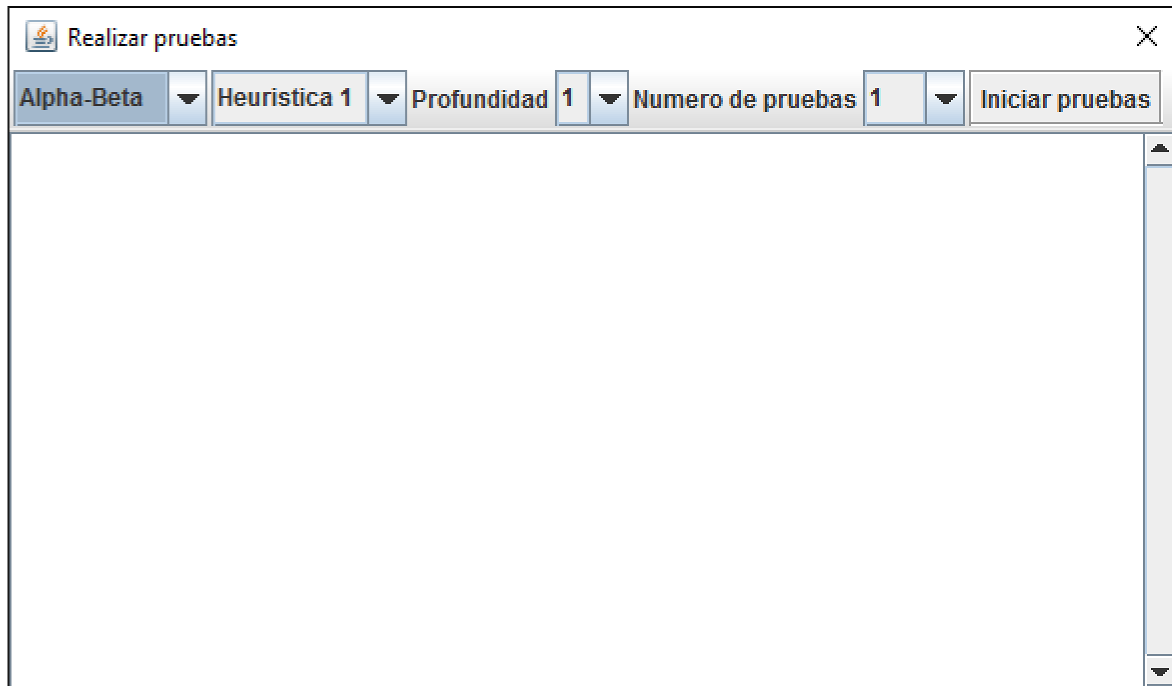

Ilustración 36: Ventana para la realización de pruebas

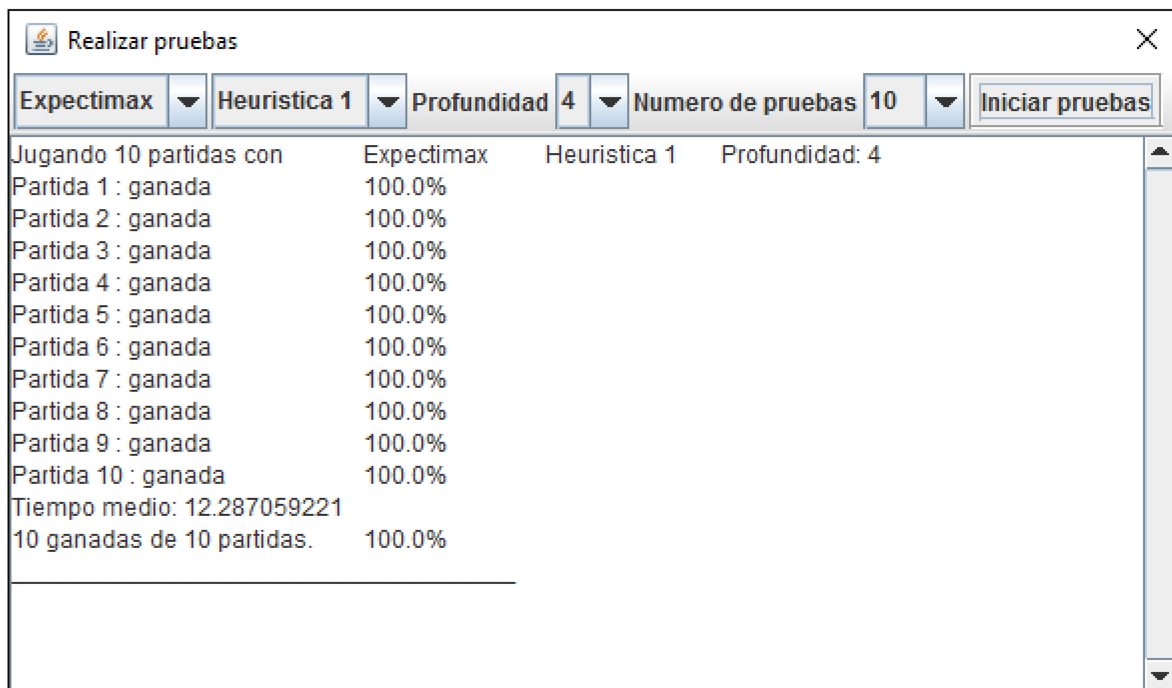

Ilustración 37: Ejemplo de una batería de pruebas

Mientras se están ejecutando las pruebas podemos seguir jugando en la ventana principal del juego.