Universidad Carlos III de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Jugando al 2048 con Inteligencia Artificial

Autor: Ignacio Sáez Lahidalga

Tutor: Moisés Martínez Muñoz y Nerea Luís Mingueza

Resumen

En este trabajo se presenta el proceso de análisis, diseño, implementación y experimentación de un jugador automático para el juego *2048*. El objetivo principal de este trabajo consiste en desarrollar un jugador automático para el juego *2048* mediante la utilización de Inteligencia Artificial. Para ello se han seleccionado dos algoritmos sobre los cuales se han realizado pequeñas implementaciones para adaptarlos al funcionamiento del juego. Con el fin de realizar una comparativa entre el rendimiento de ambos algoritmos se han definido un conjunto de funciones heurísticas.

Abstract

This dissertation presents the analysis, design, implementation and experimentation process of an automatic player for the game *2048.* The main goal of this project is to develop an automatic player for the game *2048* by using Artificial Intelligence. In order to do this, two algorithms have been selected and adapted with small modifications to make them suitable for the game. With the objective to compare the performances of both algorithms two different heuristics have been defined.

Índice General

Índice de figuras

Índice de tablas

Capítulo 1: Introducción

Para poder hablar de la construcción de un software inteligente, es preciso primero empezar con una definición de inteligencia. La inteligencia puede definirse como un conjunto de propiedades de la mente. Entre estas propiedades se incluyen las habilidades de planificar, resolver problemas y en general, razonar. Otra definición más simple sería que la inteligencia es la habilidad de tomar la decisión correcta dados una información o inputs iniciales (Jones, 2008).

A lo largo de la historia, se han definido distintos enfoques a la dotación de "inteligencia" para un software a través de distintas técnicas, es decir, crear una inteligencia artificial. La inteligencia artificial es un campo de estudio de múltiples disciplines, como las ciencias de la computación, las matemáticas, la lógica y la filosofía, que se centran en estudiar y diseñar sistemas que puedan resolver problemas de manera autónoma, utilizando como base la inteligencia humana. Uno de los primeros en plantearse la cuestión fue Alan Turing que se preguntó si la respuesta de un ordenador fuera indistinguible de la de un humano, entonces el ordenador podría ser considerado como inteligente (Turing, 1950).

Una de las primeras aplicaciones de la Inteligencia Artificial (IA) se centró en el desarrollo de sistemas orientados a la resolución de juegos de tablero (Shaw, et al., 1959) o la resolución genérica de problemas (Turing, 1953). En este momento, se consideraba como una maquina inteligente aquella que fuera capaz de realizar alguna acción que las personas pudieran realizar, y que, además, les resultara difícil. En 1950, Claude Shannon propuso que el juego del ajedrez era fundamentalmente un problema de búsqueda, si bien la fuerza bruta no es práctica debida al vasto espacio de búsqueda existente en el ajedrez. Los problemas de búsqueda son aquellos en los que se debe encontrar una solución a un problema sin tener un algoritmo para resolverlo, sino que solo sabemos cómo es la solución. Mediante búsquedas, heurísticas y bases de datos de movimientos de apertura y de fin de partida ofrecen una manera mucho más eficiente de jugar al ajedrez.

A lo largo del tiempo se han ido aplicando distintas técnicas de IA en una gran variedad juegos con distintos grados de complejidad. En este trabajo se va a tratar de implementar una serie de IAs que sean capaces de jugar al 2048 y de llegar pasarse el juego.

1.1 Descripción del problema

Desarrollado por Gabriel Cirulli, e inspirado en el *1024* y por *threes,* en 2014 un popular juego de tablero de 4x4, el *2048*, consistente en fusionar casillas con números idénticos (todos son múltiplos de dos) deslizando todas las casillas hacia una dirección, hasta conseguir alcanzar la casilla de 2048 o hasta que no puedan realizarse más movimientos.

La siguiente figura es una captura de pantalla de un tablero de una partida en curso del *2048.*

Figura 1 Partida en Curso

Una partida del juego comienza con un tablero vació salvo por dos casillas seleccionadas aleatoriamente. Cada una de estas casillas será o bien un 2, con una probabilidad del 90%, o un 4 con una probabilidad de un 10%. Además, se iniciará la puntuación del tablero a 0.

El jugador puede realizar 4 movimientos. Deslizar todas las casillas hacia arriba, abajo, derecha o izquierda. Si como consecuencia de realizar un movimiento, dos casillas con el mismo número chocan, estas se fusionan creando una nueva casilla con valor la suma de ambas casillas iguales. Como consecuencia de que se fusionen las casillas la puntuación del tablero se verá afectada, de tal forma, que esta aumentara en el mismo número que la nueva casilla que se ha formado. Además, tras cada movimiento, se añadirá, de forma aleatoria también un 2 o un 4 en una casilla vacía, con la misma probabilidad que cuando se inicializa el tablero, es decir, 90% y 10% respectivamente.

Si el jugador intenta realizar un movimiento que no es posible realizar, por ejemplo, mover hacia abajo y que no haya ninguna casilla susceptible de ser movida, entonces no se generará movimiento alguno, el tablero quedará igual, y tampoco se añadirá una casilla al azar.

El juego continúa hasta que no queden más movimientos posibles, es decir, este todo el tablero lleno, sin celdas adyacentes con números idénticos, en cuyo caso, la partida se pierde. Si por el contrario se llega a formar una casilla con 2048, la partida se da por ganada.

1.2 Motivación

El desarrollo de algoritmos de inteligencia artificial para la toma de decisiones es una tarea interesante ya que una vez aplicados estos algoritmos en entorno sencillos, pueden aplicarse para sistemas más complejos para resolver problemas de mayor envergadura, sobre todo hoy en día cuando cada vez se confía más en los ordenadores y máquinas para realizar nuestras tareas del día a día.

Antes de poder confiar a los ordenadores tareas más complejas de nuestra vida diaria, como por ejemplo conducir, tenemos que asegurarnos que son capaces de realizar tareas más sencillas como mínimo tan bien, si no mejor que las personas.

Con el fin de asegurar la eficacia de los programas de ordenador para la realización de una tarea, conviene probar y comparar los resultados arrojados de los distintos métodos para un mismo problema y así obtener el mejor método.

1.3 Objetivos del trabajo

El objetivo principal de este trabajo consiste en implementar un jugador automático para el juego *2048.* Para ello se implementarán diferentes algoritmos de búsqueda cuyos resultados serán analizados y comparados con el fin de discernir cuál es el algoritmo que mejor se adecua al problema. Por tanto, este trabajo es un trabajo de investigación en el que no se conocen los resultados que se van a obtener, ni se sabe si estos serán favorables o no. En caso de que lo sean, se abre la posibilidad a utilizar métodos más complejos para poder mejorar los resultados obtenidos.

Con el fin de definir el conjunto de tareas necesarias para cumplir el objetivo principal de este trabajo de fin de grado, se han realizado una división en objetivos más específicos:

- Estudio y análisis del funcionamiento del juego *2048.* Así, como estudio de las diferentes versiones que hay implementadas con el fin de elegir una o realizar una implementación nueva.

- Estudio u análisis de los diferentes algoritmos de inteligencia artificial utilizados para la resolución del de juegos de tablero.
- Implementación o despliegue de los diferentes algoritmos seleccionados para la resolución de partidas del juego *2048.*
- Definir un conjunto de pruebas con el fin de analizar el funcionamiento de los distintos algoritmos seleccionados.
- Desarrollo de un documento que describa el proceso de análisis, diseño e implementación de este trabajo.

1.4 Estructura del documento

Este documento está dividido en 8 capítulos:

1. El primero de ellos es la introducción al trabajo como la descripción del problema, la motivación del trabajo, los objetivos, y estructura del mismo.
2. El segundo capítulo se compone del estado del arte que tiene como cometido poner al lector en contexto e informarle de las distintas tendencias y avances realizados en la materia.
3. En el tercer capítulo se describen los procesos de análisis e implementación del jugador automático. En primer lugar, se presenta un análisis del sistema, donde se describe el alcance del mismo, los diferentes casos de uso, así como los requisitos funcionales y no funcionales del jugador. A continuación, se presenta el diseño donde, donde se describe detalladamente la arquitectura utilizada, se especifica el funcionamiento de los distintos componentes que forman el sistema y se presenta el flujo de funcionamiento del jugador.
4. El cuarto capítulo de este trabajo se describe la experimentación realizada para evaluar al jugador automático que se ha realizado en este trabajo. Para ello se realiza una descripción de los de los diferentes experimentos realizados. A continuación, se presentan los resultados obtenidos para cada uno de los algoritmos utilizados y finalmente se realiza un análisis entre los resultados obtenidos.
5. El capítulo 5 se presenta la metodología de desarrollo de software que se ha utilizado para el desarrollo de este trabajo. A continuación, se presentan la planificación inicial y final del desarrollo analizando las desviaciones que se han producido. Para finalizar se presenta el coste de desarrollo del jugador automático.

6. El capítulo 6 presenta las conclusiones generales obtenidas tras la realización de este trabajo; las conclusiones obtenidas para cada uno de los objetivos del trabajo, así como las diferentes líneas futuras que podrían aplicarse.
7. El capítulo 7 presenta un resumen en inglés del contenido del trabajo. Este resumen incluye una introducción con los objetivos del trabajo, la experimentación realizada para evaluar al jugador automático, y se presentan las conclusiones del mismo.
8. El capítulo 8 presenta los Anexos de este documento. Incluye un manual de instalación del sistema, un manual de usuario y los diagramas de clases del proyecto.

Capítulo 2: Estado del Arte

Este capítulo presenta un resumen de las diferentes técnicas de Inteligencia Artificial utilizadas en el contexto de este trabajo; de esta manera será más fácil entender la situación en la que se parte en la elaboración de este tipo de proyecto.

2.1 Inteligencia Artificial

Como se ha presentado en la introducción, la inteligencia artificial es un campo de estudio de múltiples disciplines, como las ciencias de la computación, las matemáticas, la lógica y la filosofía, que se centran en estudiar y diseñar sistemas que puedan resolver problemas de manera autónoma, utilizando como base la inteligencia humana. El primero en acuñar dicho término fue John McCarthy en 1956 que la define como la ciencia e ingenio de hacer maquinas inteligentes, especialmente programas de cómputo inteligentes. (McCarthy, 2007)

Los juegos de tablero son aquellos juegos que tienen lugar sobre una superficie de juego limitada, a la que llamamos tablero, en la que se mueven o se colocan una o más piezas de acuerdo a unas reglas prefijadas. Si bien inicialmente los juegos tablero requerían tableros y piezas físicas, con el avance de la tecnología se han creado versiones digitales de los juegos que han permitido que sean estudiados en el campo de la inteligencia Artificial.

Los juegos de tablero han sido usados ampliamente como instrumento de desarrollo de inteligencias artificiales. Esto se debe a que, en los juegos de tablero, las reglas están perfectamente definidas, el rango de posibles acciones está limitado, los resultados posibles son claros y el nº de estados del tablero, si bien puede llegar a ser inmensamente grande, como en el caso del ajedrez, es finito, lo que facilita la implementación de los algoritmos de inteligencia artificial. El trabajo de Shannon sobre ajedrez en computadoras resultó en lo que se conoce como el número Shannon, 10^{120} que es el límite inferior de la complejidad del árbol de juego del ajedrez (Shannon, 1950). Además, en juegos como las damas y el ajedrez el intelecto, en contraste con la suerte, juega un papel muy importante. El *2048,* que es el juego del que trata este trabajo, se puede catalogar como un juego de tablero.

Algunos juegos de tablero que han sido estudiados por la inteligencia artificial, a parte de las damas y el ajedrez que han sido mencionados anteriormente, son: el tres en raya, Otelo, Go, Backgammon, Póker, Scrabble. Algunos de estos juegos son de información completa y

deterministas (tres en raya, ajedrez), otros sin embargo son de información imperfecta y conllevan un factor suerte (Póker, Scrabble).

Los juegos de tablero suelen modelarse como problemas de búsqueda. Como se detalló en la introducción, los problemas de búsqueda son aquellos en los que se debe encontrar una solución a un problema sin tener un algoritmo para resolverlo, sino que solo sabemos cómo es la solución. Los problemas de búsqueda constan de un espacio de estados, entre los que se incluye un estado inicial y un estado final y un conjunto de operadores o acciones, con costes.

En las siguientes secciones de este capítulo se explicarán en mayor detalle los distintos tipos de búsqueda.

2.2 Búsqueda no informada

Son algoritmos de búsqueda no informada aquellos que no utilizan ninguna información específica del problema que ayude a decidir cuál es el mejor operador para continuar la búsqueda.

Los algoritmos de Búsqueda no informada se modelan como un árbol de búsqueda. En estos árboles, cada nodo representa un estado del problema, siendo el nodo raíz el inicial, y los nodos hoja estados terminales, entre los que se puede incluir el objetivo. A continuación, se muestra una imagen ilustrativa de un árbol de búsqueda.

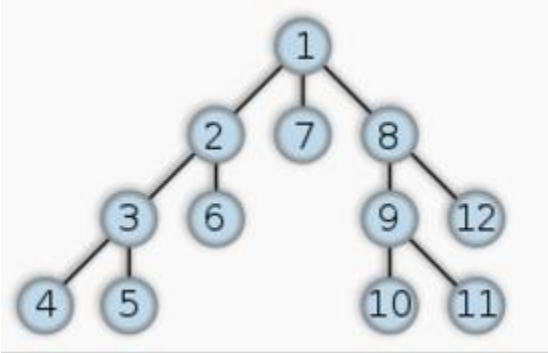

Figura 2 Árbol de Búsqueda

Dentro de esta categoría destacan dos algoritmos, Primero en Profundidad y Primero en Anchura.

El algoritmo de primero en profundidad se caracteriza por ir expandiendo los nodos hasta que estos no tienen ningún sucesor. Una vez que se llegue a un nodo terminal, se retrocede a su padre y si este tiene más sucesores, se expanden. A continuación, se presenta un funcionamiento más detallado:

1. Se selecciona el estado inicial como nodo raíz
2. Mientras haya nodos sin expandir
    1. Si el nodo actual es un nodo terminal
        1. Si es solución, fin del algoritmo
        2. Si no es solución o todos sus hijos han sido expandidos, el nodo actual pasa a ser el padre.
    2. Expandir nodo
    3. El nodo actual pasa a ser el primer nodo sin expandir de sus sucesores

El algoritmo de primero en anchura, se diferencia del primero en profundidad en que expande primero todos sus nodos sucesores, y hasta que no se hayan expandido todos los nodos sucesores del nodo inicial no se expanden los sucesores de estos. Se implementa mediante una cola tipo FIFO con el siguiente funcionamiento:

1. Se expande el nodo inicial y se encolan en orden
2. Mientras que la cola no este vacía
    1. Se desencola nodo y este pasa a ser el actual
    2. Si este es nodo solución, entonces devolver nodo
    3. En caso contrario expandir nodo y encolar sus sucesores

Debido al gran tamaño del espacio de estados de los juegos de tablero y al ser los algoritmos de búsqueda no informada de fuerza bruta, no resulta práctico utilizar estas técnicas.

2.3 Búsqueda por Contienda

Se conoce como búsqueda por contienda a aquellas técnicas de búsqueda que se aplican teniendo en cuenta las distintas acciones que pueda realizar un adversario. Este tipo de técnicas resultan especialmente adecuados para los juegos de mesa o de tablero.

Los juegos de tablero, tradicionalmente son juegos de dos jugadores (el *2048* no lo es, pero se explicará más adelante en el documento su adaptación a uno de dos jugadores) que compiten entre sí. También son conocidos por como juegos de “Suma Cero” (en inglés *zero-sum games*) en ámbito de la teoría de juegos.

Estos juegos deben definirse formalmente para poder adaptarse a los algoritmos, para ello:

- Se define un estado inicial, que incluye la posición de las fichas o piezas en el tablero y una indicación de a quien le toca jugar.
- Un conjunto de operadores, que determinan cuales son los movimientos permitidos a cada jugador.
- Una prueba, estado o condición de juego terminal, es decir, cuando se acaba la partida
- Una función de utilidad, que también se conoce como función de evaluación o de resultado, que sirve para comparar los distintos estados del tablero.

Los algoritmos de búsqueda por contienda se modelan utilizando un árbol de búsqueda. En dichos árboles, cada nodo representa un estado del tablero, y las ramas de cada nodo, representan las acciones que llevan a un nuevo estado. Si se trata de un estado terminal, bien porque se ha llegado a la profundidad máxima de búsqueda o por las condiciones del juego, entonces se representa mediante un nodo hoja.

En la figura 2.1 se muestra el árbol de búsqueda del tres en raya. Se observa que el nodo raíz, o inicial, representa el estado inicial del tablero, esto es, un tablero vacío. Las ramas de este nodo representan los movimientos de colocar el círculo que tiene el primer jugador, y el nodo al que llegan es el estado en el que queda el tablero tras realizar el movimiento del primer jugador. A continuación de cada nodo sale una rama que representa los movimientos que podría realizar el segundo jugador al colocar al x en el tablero, llegando al siguiente nodo, que es el estado en el que queda el tablero tras jugar el segundo jugador. El resto del desarrollo del árbol es análogo hasta que se llega a un estado final, es decir, gana uno de los dos jugadores o se quedan ambos sin movimientos sin que ninguno haya ganado.

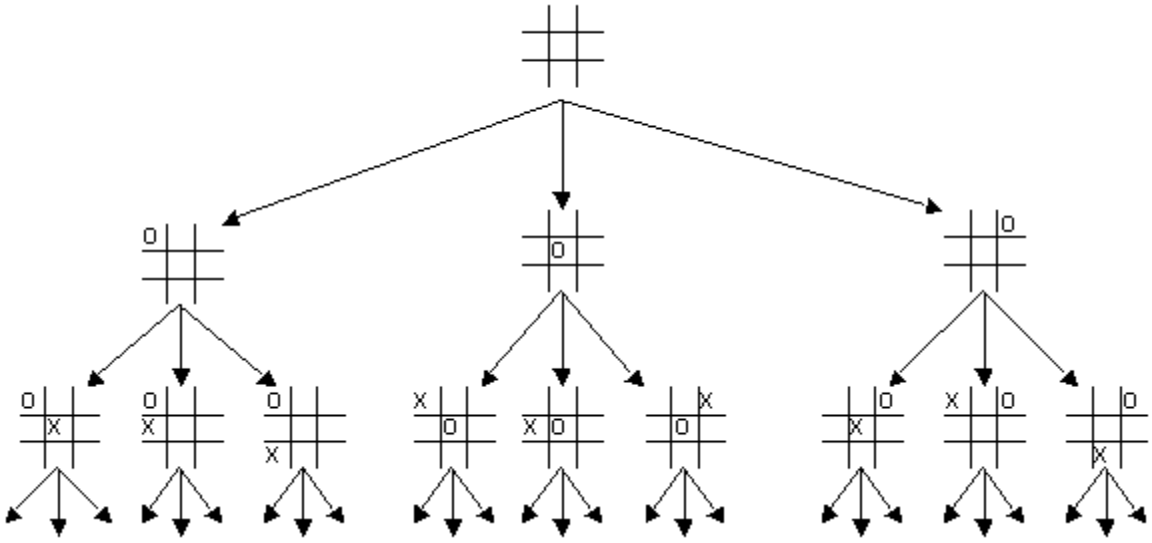

Figura 3 Árbol de Búsqueda del Tres en Raya

Para este tipo de juegos resulta óptimo la utilización de algoritmos como el minimax o sus variantes, que se explican a continuación.

El minimax (Neumann, 1928) es un método para seleccionar el mejor movimiento por el que el jugador que lo realiza maximiza su beneficio (o minimiza su coste) a la vez que minimiza el beneficio del contrincante (o maximiza su coste).El minimax es un algoritmo de búsqueda de primero en profundidad, esto es, que expande cada nodo hasta la profundidad máxima antes de volver atrás y expandir otros nodos. Minimax puede utilizar dos estrategias:

- En la primera, se busca en el árbol de búsqueda de la partida hasta llegar a los nodos hoja del árbol (nodos terminales).
- En el segundo, se expande el árbol hasta llegar a una determinada profundidad que se había predefinido anteriormente.

El algoritmo funciona de la siguiente manera:

1. Se llama al algoritmo con el estado, o nodo, inicial, la profundidad máxima de búsqueda y si hay que maximizar la jugada o no.
2. Si la profundidad es 0, o el nodo es un nodo terminal, se devuelve el valor de la función de evaluación para ese nodo. En caso contrario, continua el algoritmo.
3. Si se está maximizando la jugada, se inicia como mejor valor del tablero en menos infinito, o el valor más bajo posible. Para cada sucesor del nodo inicial:

    1. Se vuelve a llamar a la función solo que, con el sucesor correspondiente, un grado de profundidad menos y sin maximizar la jugada.
    2. Se compara el resultado de esta llamada y se compara con el mejor valor del tablero, y si este es mayor entonces, pasa a ser el mejor valor del tablero.
    3. Se devuelve el mejor valor del tablero.
4. Si no se maximiza la jugada, es decir el caso de la jugada del adversario se inicia el mejor valor de tablero en infinito, o el mayor valor posible. Para cada sucesor:
    1. Se vuelve a llamar a la función solo que, con el sucesor correspondiente, un grado de profundidad menos maximizando la jugada.
    2. Se compara el resultado de esta llamada y se compara con el mejor valor del tablero, y si este es menor entonces, pasa a ser el mejor valor del tablero.
    3. Se devuelve el mejor valor del tablero.

La poda alfa-beta (Newell, et al., 1958) nació para solucionar el principal problema del minimax que es que el número de estados es exponencial al número de movimientos. La poda alfa-beta devuelve el mismo resultado que el algoritmo minimax, esto es, porque los nodos que poda o que no explora son aquellos que nunca iban a ser elegidos por el minimax, y por tanto, se ejecuta en menor tiempo, debido a que el árbol de búsqueda es de menor tamaño.

Se caracteriza por tener dos parámetros: Alfa, que es el valor más alto encontrado mientras se explora el camino de maximización, y Beta, que es el valor más bajo encontrado durante la exploración del camino de minimización del oponente. Después se comparan los parámetros entre sí y se poda la rama.

El funcionamiento del algoritmo es el siguiente:

Se llama al algoritmo con el estado, o nodo, inicial, la profundidad de máxima de búsqueda, el valor de alfa igual a menos infinito, valor de beta igual a infinito y si hay que maximizar la jugada o no.

1. Si la profundidad es 0, o el nodo es un nodo terminal, se devuelve el valor de la función de evaluación para ese nodo. En caso contrario, continua el algoritmo.
2. Si se está maximizando la jugada, se inicia como mejor valor del tablero en menos infinito, o el valor más bajo posible. Para cada sucesor del nodo inicial:

    1. Se vuelve a llamar a la función solo que, con el sucesor correspondiente, un grado de profundidad menos, valores actuales de alfa y beta, y sin maximizar la jugada.
    2. Se compara el resultado de esta llamada y se compara con el mejor valor del tablero, y si este es mayor entonces pasa a ser el mejor valor del tablero.
    3. Se compara este nuevo valor con alfa, y se elige como nueva alfa el valor máximo entre los dos.
    4. Si beta es menor o igual a alfa entonces se poda esta rama, en caso contrario, se devuelve el mejor valor del tablero.
3. Si no se maximiza la jugada, es decir el caso de la jugada del adversario se inicia el mejor valor de tablero en infinito, o el mayor valor posible. Para cada sucesor:
    1. Se vuelve a llamar a la función solo que, con el sucesor correspondiente, un grado de profundidad menos, valores actuales de alfa y beta, y maximizando la jugada.
    2. Se compara el resultado de esta llamada y se compara con el mejor valor del tablero, y si este es menor entonces pasa a ser el mejor valor del tablero.
    3. Se compara el nuevo valor con beta, y se elige como nueva beta la menor entre las dos.
    4. Si beta es menor que alfa, entonces se poda la rama y se devuelve el mejor valor del tablero.

Existen otras modificaciones al minimax, como por ejemplo, el expectiminimax (Michie, 1966). Este algoritmo se diferencia del minimax en que se utilizan para juegos que tienen una componente aleatoria, es decir, en los que se generan nodos de forma aleatoria, como puede ser el Backgammon. Para usarlo hay que saber con qué probabilidad sucederá dicho evento. A parte del transcurso normal del minimax, cuando sucede el evento aleatorio se devuelve la media ponderada, por la probabilidad del suceso, de todos los sucesores del nodo.

Otra modificación bastante simple sobre el minimax, es el Negamax (Breuker, 1998), que consiste en que cada nodo devuelve el valor máximo de sus hijos, solo que en los nodos a maximizar se le cambia de signo. De esta forma se obtiene el mismo resultado, ya que si tomar el mayor valor, pero cambiando el signo es lo mismo que tomar el menor valor.

Otro factor importante de los algoritmos de búsqueda por contienda es como se clasifican los distintos estados de los tableros de juego para poder compararlos a la hora de ejecutar los algoritmos. En algunos casos, el juego lleva internamente algún tipo de puntuación, que pude

servir como función de evaluación. Normalmente, se suele dar un valor a las distintas piezas del juego, si hay más de un tipo, y a la posición y/o distribución de las mismas sobre el tablero. Para sistemas que deban aprender de partidas pasadas, es popular también el uso de redes neuronales como función de evaluación.

A continuación, se exponen algunos ejemplos de búsqueda por contienda aplicados al mundo de los juegos de tablero.

En 1952, Arthur Samuel diseño una IA que jugaba a las damas y que incluía aprendizaje y generalización. Su programa permitía que dos copias del software jugaran entre ellas, aprendiendo la una de la otra. Como resultado el programa fue capaz de derrotar a su creador. En 1962, el programa de Samuel derrotó al excampeón de damas del estado de Connecticut. El programa de Samuel utilizaba un minimax con poda alfa beta

Dentro del campo de las Damas, aparte del trabajo de Samuel, hay otros trabajos dignos de mención. Un ejemplo de ellos es *Chinook* (Schaeffer, 2009), desarrollado por Jonathan Schaeffer y sus colegas de la Universidad de Alberta. Fue el primer programa de ordenador al que se le permitió competir en un torneo de humanos. El algoritmo de Chinook consta de lo siguiente:

- Una base de datos de movimientos de apertura o iniciales de partidas jugadas por grandes maestros.
- Un algoritmo de búsqueda, basado en minimax
- Una heurística de evaluación de movimientos
- Una base de datos de para todos los estados del tablero con 8 piezas o menos

Es importante destacar que, en este sistema, todos los conocimientos fueron programados por sus desarrolladores, es decir, la IA de *Chinook* no aprendió nada por sí misma.

Inspirado en los éxitos del trabajo de Schaeffer, surgió *Blondie24* (Fogell, 2002) creado por David B. Fogell y Kumar Chellapilla. El diseño de este sistema está basado en un algoritmo minimax que utiliza como heurística o función de evaluación una red neuronal. Dicha red neuronal obtenía sus pesos de un algoritmo evolutivo. En dicho algoritmo, las redes neuronales jugaban torneos entre ellas, las que ganaban un mayor número de partidas, evolucionaban, mientras que aquellas que perdían eran eliminadas y sustituidas por otras de nueva generación. Lo interesante de este programa es que aprendió por si solo a jugar a las damas, es decir, en ningún momento se le

programo conocimiento sobre el juego, todo el conocimiento que tenía, lo obtuvo de las partidas que jugaba.

Otro importante ejemplo de uso de inteligencia artificial en juegos de tablero es *Deep Blue* (Newborn, 2002) desarrollado por IBM y que se centra en el Ajedrez. Este sistema fue capaz de derrotar a Garry Kasparov, campeón mundial de ajedrez, en mayo de 1997. El proyecto comenzó en 1985 en Carnegie Mellon University y el equipo fue contratado posteriormente por IBM. La eficacia del algoritmo de este programa radicaba en su función de evaluación, aparte de sus bases de datos de apertura y clausura de partida, que evaluaba el valor total de las piezas ponderado por las casillas en las que se encontraba.

2.4 Búsqueda Heurística

Los métodos de búsqueda heurística son aquellos orientados a reducir la cantidad búsqueda requerida para encontrar una solución. Estos problemas suelen modelarse de la siguiente manera:

- La búsqueda se modela mediante un árbol de búsqueda.
- Se modelan un estado inicial u origen y un estado final u objetivo. Además, habrá estados intermedios. Todos los estados se modelan en el árbol de búsqueda como nodos, tanto raíz, intermedios y nodos hoja o terminales
- Las acciones o pasos de un estado a otro tienen un coste asociado. Estas acciones se representan con las ramas del árbol de búsqueda.
- Usan una función heurística, para estimar el coste de ir del estado actual al estado final. Esta función varía en función del problema que se trata de resolver.

En estos problemas, los algoritmos de búsqueda heurística tratan de reducir el tamaño del árbol, cortando aquellos nodos que no resultan prometedores. Estos métodos se denominan búsqueda informada, ya que utilizan las heurísticas para decidir el camino del árbol que siguen y no la fuerza bruta.

Como característica de estos algoritmos es que no se garantice que se encuentre una solución, si existe, y tampoco se garantiza encontrarla en el menor tiempo posible.

Existen varios algoritmos de búsqueda heurística. A continuación, se explican algunos de ellos.

El algoritmo de primero el mejor se caracteriza por expandir primero aquellos nodos con menor valor en la función heurística. Este algoritmo no tiene en cuenta el coste necesario para llegar a la solución. El algoritmo funciona con una lista de nodos abiertos y de nodos cerrados. La lista de nodos abiertos funciona como una cola con prioridad. La prioridad se da en función del valor de la heurística, dando mayor prioridad a aquellos valores con menor valor heurístico. La lista cerrada puede es una cola sin prioridad. El funcionamiento del algoritmo el siguiente:

1. Se encola el estado inicial a la lista abierta y se inicializa la cerrada como una lista vacia.
2. Mientras la lista abierta no está vacía:
    1. Se desencola el primer elemento de la lista abierta y se le llama N. Se encola N a la lista cerrada
    2. Si N es el estado final, se rastrea el camino seguido hasta llegar a N y se devuelve.
    3. Se crean los sucesores de N.
    4. Para cada sucesor de N:
        1. Si no está en la lista cerrada y tampoco en la abierta, encolarla en la abierta y registrar su nodo padre
        2. En caso contrario, si mejor que el anterior, cambiar el nodo padre registrado
            1. Si no está en la lista abierta, se encola en ella
            2. En caso contrario ajustar su prioridad en esta lista usando esta nueva evaluación

El algoritmo de búsqueda Primero el Mejor, se ha utilizado para resolver el conocido problema de las N-Reinas, que es un problema derivado del ajedrez que consiste en colocar en un tablero de N*N un número N de Reinas de manera que las reinas no puedan atacarse las unas a las otras.

Una técnica de búsqueda heurística muy popular es el algoritmo A* (Hart, et al., 1968). Este algoritmo desarrollado por Peter Hart, Nils Nilsson y Bertram Raphael está basado en el algoritmo de Dijstra de 1959 y obtiene mejores resultados usando heurísticas para guiar su búsqueda.

El algoritmo A* se caracteriza por usar en su función de evaluación tanto el coste hasta llegar al nodo actual *G(n)* como el coste estimado para llegar desde el nodo actual al nodo objetivo *H(n),* de tal manera que se obtiene la siguiente función de coste:

$$\boldsymbol{F(n) = G(n) + H(n)}$$

El algoritmo utiliza dos listas: una lista de nodos abiertos, inicializada con el nodo inicial, y una lista de nodos cerrados. La lista abierta esta implementada como una cola de prioridad, ordenada por el valor ascendente de *F(n).* El funcionamiento del algoritmo es el siguiente:

1. Se inicializa la lista cerrada vacía y se encola en la lista abierta el nodo inicial.
2. Mientras que la lista abierta no está vacía:
    1. Se desencola el nodo de la lista abierta y se le llama N
    2. Si N es el objetivo entonces se devuelve el camino recorrido hasta llegar a él.
    3. Se encola en la lista cerrada
    4. Para cada sucesor de N
        1. Si sucesor en lista cerrada, se ignora
        2. Se calcula *NuevaG* como la suma del G(N) + distancia (N, sucesor)
        3. Si el sucesor no está en la lista abierta, se encola
        4. En caso contrario, si *NuevaG >= G(sucesor)* se ignora, ya que no es el camino óptimo.
        5. Se registra de que nodo viene el sucesor
        6. Se cambia actualiza el valor de *G(N) = NuevaG*
        7. Se calcula *F(Sucesor) = G(Sucesor) + H(Sucesor)*
3. Si se llega a este punto sin solución devolver fracaso

El A* es una técnica de búsqueda popular que ha sido utilizada para resolver juegos de tablero simples como puede ser el N-puzle.

Una variante interesante de este algoritmo es IDA*. Se diferencia del A* en que se establece un coste máximo para el cual se detiene el algoritmo. El algoritmo funciona como una sucesión de búsquedas tipo A*. En cada una de estas iteraciones, en lugar de hasta que se llegue el estado objetivo, se realiza hasta que se alcanza el coste límite. Cada vez que se alcanza el coste límite sin llegar al nodo final se repite el proceso aumentando el límite de coste. Ese nuevo límite, bien dado por el menor de los límites de corte, es decir, por el menor valor del coste de los nodos que tenían un valor superior en la anterior iteración. Explicado en detalle por pasos esto sería:

1. Se define un umbral de coste.
2. Mientras no se encuentre solución:

1. Ejecutar A*. Con la modificación en el paso 2.2 de que si *G(N) >= Limite de Coste* se sale del algoritmo.
2. Si no se ha encontrado solución, incrementar profundidad

Existe otro grupo de algoritmos llamados algoritmos de búsqueda local. Estos algoritmos comienzan en un nodo e iterativamente se mueven a nodo vecino. Esto solo es posible si existe una relación de vecindad definida. A esta categoría pertenece el algoritmo Hill-Climbing.

El algoritmo Hill-Climbing tiene el siguiente funcionamiento:

1. Se establece como nodo actual al nodo inicial
    1. Para cada vecino del nodo actual
        1. Elegir el que tenga mejor valor heurístico, que será el siguiente
    2. Si el nodo siguiente tiene un mejor valor heurístico entonces, este pasa a ser el nodo actual. En caso contrario devolver el nodo actual

Otro método de búsqueda heurística que se utiliza para procesos de toma de decisiones, sobre todo en juegos, es el Árbol de búsqueda Montecarlo. Este método se basa en analizar los movimientos más prometedores, y amplia el árbol de búsqueda basado en una muestra aleatoria del espacio de búsqueda. Para aplicarlo en juegos, hay que realizar muchas series de rondas eliminatorias. En cada ronda se juega hasta el final de la partida realizando movimientos al azar. El resultado final de cada ronda se utiliza para ponderar los nodos del árbol de búsqueda, de manera que aquellos modos que sean mejores serán los que tendrán mayor probabilidad de ser elegidos en futuras rondas. Este método ha sido usado recientemente para el juego de Go, en un sistema llamado Alpha Go desarrollado por Google, que además también utiliza aprendizaje profundo., que son técnicas de aprendizaje automático.

2.5 Inteligencia Artificial en el 2048

Debido a que el *2048* es un juego bastante reciente, se publicó a principios de 2014, no ha sido un problema que haya sido estudiado en exceso, si bien, sí que existen algunos ejemplos los cuales se exponen a continuación.

El primer ejemplo de uso de inteligencia artificial aplicado al problema del *2048*, fue el de Matt Overlan que implementa un minimax con poda alfa-beta (Overlan, 2014). Según información publicado por el mismo autor, el algoritmo presenta una tasa de victorias de entorno al 90%, lo

cual es un porcentaje bastante alto. En cuanto a la heurística o función de evaluación utiliza una que potencia la monotonía en filas y columnas, es decir que estén en orden creciente o decreciente a la vez que trata de alinear casillas con el mismo valor. También minimiza el número de casillas ocupadas en el tablero.

Robert Xiao (Xiao, s.f.) implementó otro jugador automático que utilizaba una variante del algoritmo minimax, el expectiminimax. La principal diferencia con el minimax, es que el expectiminimax, no elige el máximo de los mínimos del oponente, si no que elige el máximo de los valores esperados de los siguientes estados. El valor esperado de un estado es la suma del valor de cada uno de los estados multiplicado por la probabilidad de que eso suceda. Es decir, toma decisiones en base a lo que es probable que suceda. Esta implementación conseguía jugar un movimiento cada 150ms y consiguió una puntuación media de 157.652 puntos.

Otro ejemplo interesante es el desarrollado por Philip Rodger y John Levine, que implementaron un jugador que utilizaba un árbol de búsqueda Monte-Carlo (Rodgers & Levine, 2014). Decidieron utilizarlo debido a las siguientes propiedades clave del algoritmo:

1. Puede estar ejecutando durante cualquier cantidad de tiempo devolver un resultado, de hecho, cuanto más tiempo ejecute mejor será el resultado.
2. No requiere una función de evaluación, ya que las rondas eliminatorias realizan este trabajo. La naturaleza aleatoria se asegura de que se valoren todos los movimientos
3. Produce arboles asimétricos, podando ramas poco prometedoras permitiendo una mayor profundidad de búsqueda.

En cuanto a resultado se refiere, los autores afirman que sus sistema mejora el de Robert Xiao, salvo que cada movimiento tarda de media 3 segundos por turno lo que es aproximadamente 22 veces más que el suyo.

Capítulo 3: Descripción del sistema

En este capítulo se realiza una descripción del sistema que se va a implementar. Primero, se realiza y se presenta un análisis del sistema, donde se describe el alcance del mismo, los diferentes casos de uso, así como los requisitos funcionales y no funcionales del jugador. En segundo lugar, se presenta el diseño donde, donde se describe detalladamente la arquitectura utilizada, se especifica el funcionamiento de los distintos componentes que forman el sistema y se presenta el flujo de funcionamiento del jugador.

3.1 Introducción

El sistema implementado tiene como base el juego *2048*. Incorpora la mecánica interna del juego, además de una interfaz gráfica para poder visualizar el desarrollo de la partida.

El sistema se puede utilizar de dos maneras distintas. Bien jugando manualmente utilizando las teclas de dirección del teclado y siguiendo la partida en la interfaz gráfica, o bien configurando el jugador automático con el algoritmo deseado, viendo el desarrollo de la partida en la interfaz gráfica para después ver en un archivo generado las estadísticas y resultados de la misma.

3.2 Análisis del sistema

En este apartado quedan recogidos todos los detalles y tareas relativas del análisis del sistema del *2048*. El primer subapartado se corresponde con la definición del alcance del trabajo. A continuación, se muestran los requisitos del sistema, los cuales son de vital importancia para acotar el proyecto y acercarlo a su fase de diseño. Finalmente se concluye este apartado detallando los casos de uso de este sistema

3.2.1 Alcance del trabajo

Para la realización de este proyecto, primero habrá que familiarizarse correctamente con las normas y el entorno del juego *2048.* Una vez hayan quedado claras o bien se habrá que implementar una versión del juego sobre la cual implementar el jugador automático o bien encontrar una versión del juego ya implementado sobre la cual se pueda modificar y posteriormente implementar el jugador automático.

Para la realización de este proyecto se cuenta con un ordenador equipado con software para el desarrollo de programas en distintos lenguajes de programación (IDEs, compiladores…)

De lo hablado en las primeras reuniones que se mantuvieron acerca del proyecto, se llegó a las siguientes conclusiones y requisitos de usuario que sirven para establecer los límites del trabajo:

- El sistema *2048* deberá tener algún tipo de modo para seguir la evolución de la partida.
- El sistema deberá poder usarse en modo manual, además del jugador automático
- El jugador automático deberá poder elegir entre dos algoritmos de búsqueda.
- Cada uno de los algoritmos de búsqueda deberán poderse configurar con dos funciones de evaluación o heurísticas y distintas profundidades de búsqueda en el árbol del algoritmo.

3.2.2 Diagrama de Casos de Uso

En esta sección se mostrará el diagrama de casos de uso que servirá como ilustración de los distintos casos de uso para este sistema.

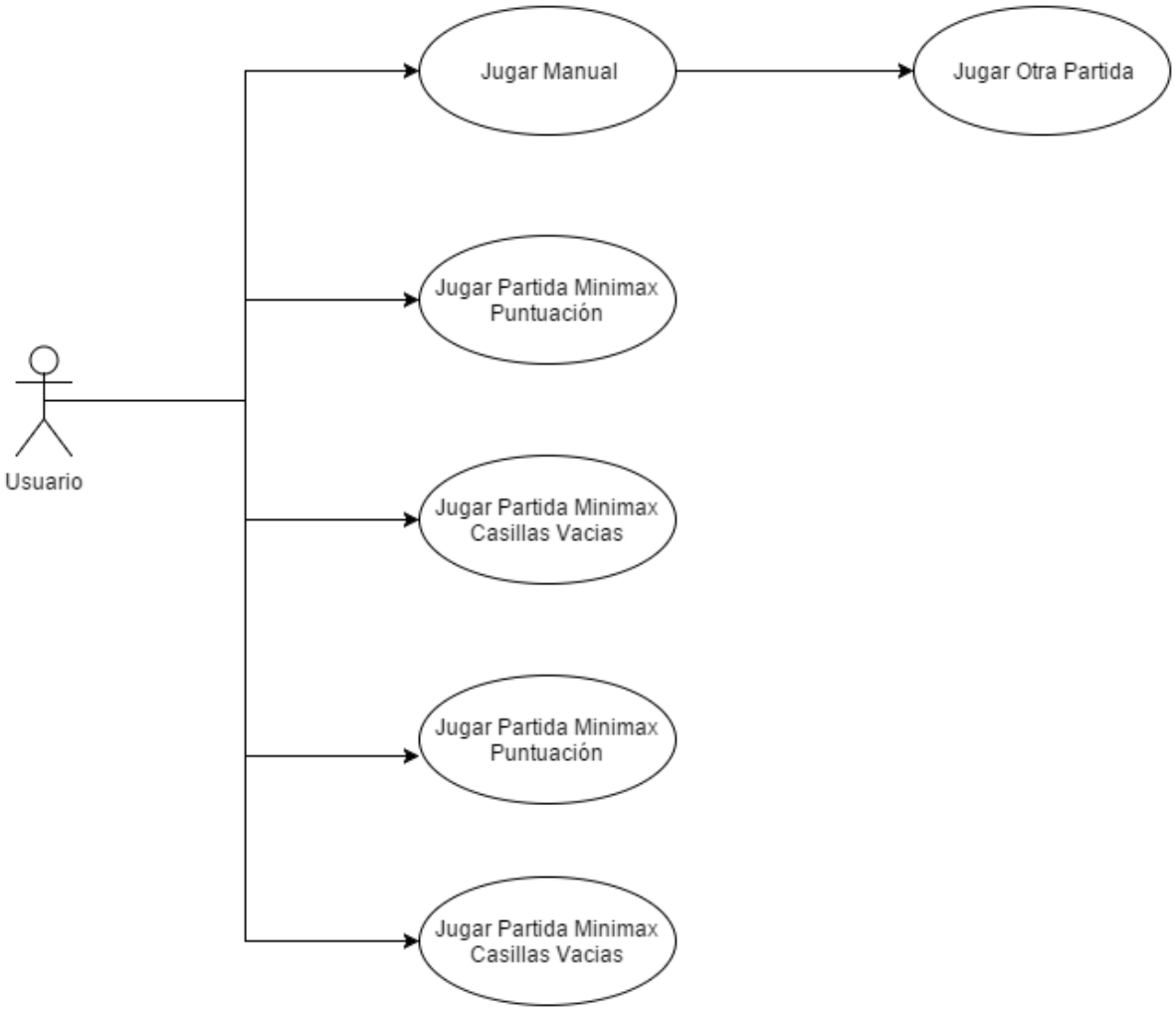

Figura 4 Diagrama de Casos de Uso

Tal como se describe en el diagrama, se establecen 6 casos de uso distintos, 5 de ellos independientes y uno de ellos que depende de que el sistema se encuentre en la situación de uno de ellos.

Para el caso del sistema de este trabajo, solo existe un actor para el que se realizará la descripción de los casos de uso y este será, lógicamente, el usuario final del sistema.

3.2.3 Especificación de casos de uso

Para la realización de la descripción textual de los distintos casos de uso, se han seleccionado una serie de atributos que describen cada uno de los casos de uso. A continuación, se realiza una descripción del significado de cada uno de los atributos utilizados para la descripción de los casos de uso.

- Código: Identificación unívoca abreviada del caso de uso, se construye mediante CU seguido de un - y de tres dígitos. Por ejemplo, CU-001.
- Nombre: Identificación extendida del caso de uso.
- Actores: Conjunto de entidades que interactúan con el caso de uso. El caso de uso representa una funcionalidad demandada por un actor.
- Descripción: Se realiza una descripción básica de la funcionalidad o funcionalidades del caso de uso.
- Precondiciones y poscondiciones: Se realiza una descripción de las condiciones que deben cumplirse para poder realizar una operación, y el estado en el que queda el sistema tras realizar una operación.
- Escenario: Se realiza una descripción básica de las acciones que se ejecutaran paso a paso en el caso de uso.

Caso de uso	
Código	CU-001
Nombre	Jugar Manual
Actores	Usuario
Descripción	El usuario quiere jugar una partida de 2048
Precondiciones	El sistema no puede estar en ejecución
Poscondiciones	Partida con GUI en la que el usuario puede jugar usando las teclas de dirección
Escenario	Ejecutar el programa y cuando salga el menú dando elegir el modo elegir jugar manual. Usar las flechas de dirección para jugar

Tabla 1 CU-001

Caso de uso	
Código	CU-002
Nombre	Jugar otra Partida
Actores	Usuario
Descripción	El usuario quiere jugar otra partida o reiniciar su partida
Precondiciones	Una partida manual debe estar en ejecución
Poscondiciones	Partida con GUI en la que el usuario puede jugar usando las teclas de dirección
Escenario	Durante una partida manual o al terminar una partida, el usuario quiere reiniciar su partida, para ellos debe pulsar la teclas Esc

Tabla 2 CU-002

Caso de uso	
Código	CU-003
Nombre	Jugar Partida Minimax Puntuación
Actores	Usuario
Descripción	El usuario quiere jugar una partida con el jugador automático usando el algoritmo minimax y como heurística la puntuación del tablero
Precondiciones	Establecer el parámetro del algoritmo a 0 y el de la heurística a 0. Establecer la profundidad deseada
Poscondiciones	Al finalizar la partida e obtienen los datos estadísticos de la partida al igual que el resultado
Escenario	El usuario quiere realizar una prueba con minimax y usando como heurística la puntuación. Ejecuta el programa con los parámetros necesarios y selecciona la opción jugar automático. El usuario verá el desarrollo de la partida y al finalizar esta tendrá un resumen con los datos de la misma

Tabla 3 CU-003

Caso de uso	
Código	CU-004
Nombre	Jugar Partida Minimax Casillas Vacías
Actores	Usuario
Descripción	El usuario quiere jugar una partida con el jugador automático usando el algoritmo minimax y como heurística el número de casillas vacías
Precondiciones	Establecer el parámetro del algoritmo a 0 y el de la heurística a 1. Establecer la profundidad deseada
Poscondiciones	Al finalizar la partida e obtienen los datos estadísticos de la partida al igual que el resultado
Escenario	El usuario quiere realizar una prueba con minimax y el número de casillas vacías. Ejecuta el programa con los parámetros y selecciona la opción jugar automático. El usuario verá el desarrollo de la partida y al finalizar esta tendrá un resumen con los datos de la misma

Tabla 4 CU-004

Caso de uso	
Código	CU-005
Nombre	Jugar Partida A* Puntuación
Actores	Usuario
Descripción	El usuario quiere jugar una partida con el jugador automático usando el algoritmo A* y como la puntuación
Precondiciones	Establecer el parámetro del algoritmo a 1 y el de la heurística a 0. Establecer la profundidad deseada
Poscondiciones	Al finalizar la partida e obtienen los datos estadísticos de la partida al igual que el resultado
Escenario	El usuario quiere realizar una prueba con A* y la puntuación. Ejecuta el programa con los parámetros y selecciona la opción jugar automático. El usuario verá el desarrollo de la partida y al finalizar esta tendrá un resumen con los datos de la misma

Tabla 5 CU-005

Caso de uso	
Código	CU-006
Nombre	Jugar Partida A* Casillas Vacías
Actores	Usuario
Descripción	El usuario quiere jugar una partida con el jugador automático usando el algoritmo A* y como heurística el número de casillas vacías
Precondiciones	Establecer el parámetro del algoritmo a 1 y el de la heurística a 1. Establecer la profundidad deseada
Poscondiciones	Al finalizar la partida e obtienen los datos estadísticos de la partida al igual que el resultado
Escenario	El usuario quiere realizar una prueba con A* y el número de casillas vacías. Ejecuta el programa con los parámetros y selecciona la opción jugar automático. El usuario verá el desarrollo de la partida y al finalizar esta tendrá un resumen con los datos de la misma

Tabla 6 CU-006

3.2.4 Requisitos del Sistema

Para la realización de la descripción textual de los distintos requisitos que han sido identificados, se han seleccionado una serie de atributos que describen cada uno de los requisitos. A continuación, se realiza una descripción del significado de cada uno de los atributos utilizados para su descripción:

- Código: Identificación unívoca abreviada del requisito, se construye mediante el código del requisito seguido de un - y de tres dígitos. Los requisitos serán divididos en funciones y no funcionales y sus códigos son RF para los requisitos funciones y RNF para los requisitos no funcionales. Por ejemplo, RF-001.
- Nombre: Identificación extendida del requisito.
- Descripción: Se realiza una descripción básica del requisito que ha sido identificado.
- Fuente: Indica a través de que fuente ha sido identificado el requisito. Normalmente este valor se corresponderá con uno o varios códigos de los casos de uso.
- Necesidad: Determina el grado de implementación del requisito. Los valores que puede tomar este atributo son los siguientes:
    - Esencial: El requisito tiene que ser implementado.
    - Deseable: Es preferible implementar el requisito, pero no es obligatorio.
    - Opcional: El requisito se podrá implementar, pero no es importante ni obligatorio.
- Prioridad: Define la importancia del requisito, de forma que permita definir el orden en el cual serán incluido en el proceso de diseño y el orden de implementación. Los valores que puede tomar este atributo son los siguientes:
    - Alta: El requisito debe ser implementado en las fases iniciales del desarrollo.
    - Media: El requisito debe ser implementado una vez que hayan sido implementados los requisitos de prioridad alta.
    - Baja: El requisito debe ser implementados en las fases finales del desarrollo. Estos requisitos no influirán en el correcto funcionamiento del sistema.
- Estabilidad: Define la estabilidad del requisito durante la vida útil del software. Esto implica si el requisito podrá ser o no modificado durante el ciclo de vida. Los valores que puede tomar este atributo son los siguientes:
    - Estable: El requisito no puede variar durante el ciclo de vida del sistema.

  - Inestable: El requisito puede variar a lo largo del ciclo de vida del sistema.
- Verificabilidad: Define el grado de verificabilidad de un requisito, es decir indica en qué grado es posible comprobar que el requisito se ha incorporado en el sistema desarrollado. Los valores que puede tomar este atributo son los siguientes:
  - Alta: Se puede verificar que el requisito ha sido implementado en el sistema. Este tipo de requisitos se corresponden con las funcionalidades básicas del sistema.
  - Media: Se puede verificar que el requisito ha sido implementado en el sistema. Pero requiere de una comprobación compleja o del código fuente del sistema.
  - Baja: Es difícil verificar si el requisito ha sido implementado en el sistema o en algunos casos no es posible.

3.2.4.1 Requisitos Funcionales

Requisito del sistema			
Código	**RF-001**	**Fuente**	CU-001, CU-003, CU-004, CU-005, CU-006
Nombre	Ejecutar Sistema		
Descripción	El sistema se ejecutara a través de un ejecutable .Jar que se genera al exportar el sistema como JAR ejecutable, configurando en los argumentos del programa los parámetros de ejecución (Algoritmo, Heurística)		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 7 RF-001

Requisito del sistema			
Código	**RF-002**	**Fuente**	CU-001
Nombre	Jugar Manual		
Descripción	El sistema deberá dar lugar a que el usuario pueda elegir jugar de manera manual.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 8 RF-002

Requisito del sistema			
Código	**RF-003**	**Fuente**	CU-001,CU-002
Nombre	Control por teclado		
Descripción	El sistema deberá permitir al usuario utilizar el teclado para controlar la partida manual y reiniciarla		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 9 RF-003

Requisito del sistema			
Código	**RF-004**	**Fuente**	CU-001, CU-002, CU-003, CU-004, CU-005, CU-006
Nombre	Interfaz Gráfica		
Descripción	El sistema deberá permitir al usuario seguir la partida, tanto manual como automática, a través de una interfaz gráfica		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 10 RF-004

Requisito del sistema			
Código	**RF-005**	**Fuente**	CU-003, CU-004,
Nombre	Minimax		
Descripción	El sistema deberá permitir al usuario ejecutar el jugador automático para que use el algoritmo minimax.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 11 RF-005

Requisito del sistema			
Código	**RF-006**	**Fuente**	CU-005, CU-006
Nombre	A*		
Descripción	El sistema deberá permitir al usuario ejecutar el jugador automático para que use el algoritmo A*.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 12 RF-006

Requisito del sistema			
Código	**RF-007**	**Fuente**	CU-003, CU-005
Nombre	Heurística Puntuación		
Descripción	El sistema deberá permitir al usuario ejecutar el jugador automático con el algoritmo que desee y la heurística que mide la puntuación del tablero		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 13 RF-007

Requisito del sistema			
Código	**RF-008**	**Fuente**	CU-004, CU-006
Nombre	Heurística Casillas Vacías		
Descripción	El sistema deberá permitir al usuario ejecutar el jugador automático con el algoritmo que desee y la heurística que mide las casillas vacías		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 14 RF-008

Requisito del sistema			
Código	**RF-009**	**Fuente**	CU-003, CU-004, CU-005, CU-006
Nombre	Estadísticas de la Partida		
Descripción	El sistema deberá volcar a un fichero los datos obtenidos durante la ejecución con jugador automático		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 15 RF-009

3.2.4.2 Requisitos No Funcionales

Requisito del sistema			
Código	**RNF-001**	**Fuente**	CU-001, CU-002, CU-003, CU-004, CU-005, CU-006
Nombre	Multiplataforma		
Descripción	El sistema se desarrollará de manera que se pueda se pueda utilizar en cualquier plataforma		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 16 RNF-001

Requisito del sistema			
Código	**RNF-002**	**Fuente**	CU-001, CU-002, CU-003, CU-004, CU-005, CU-006
Nombre	Actualización Interfaz de Usuario		
Descripción	La interfaz de usuario del sistema deberá actualizarse en tiempo real, o con un retardo inapreciable respecto al sistema.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 17 RNF-002

3.3 Diseño del sistema

En este apartado se describe el diseño del jugador desarrollado para este trabajo fin de grado. En primer lugar, se presenta la arquitectura. A continuación, se realiza una descripción de cada uno de los componentes. Para finalizar se presenta una descripción del flujo de control del jugador.

3.3.1 Arquitectura del sistema

El sistema del *2048* está formado por tres elementos clave, el núcleo de la aplicación, el jugador automático y la interfaz de usuario. El núcleo de la aplicación lleva la gestión del tablero e incluye el jugador manual. El jugador Automático incluye los dos algoritmos implementados junto a las heurísticas. Finalmente, la interfaz de usuario que se encarga de pintar el estado del tablero tras cada movimiento. A continuación, se muestra un diagrama con la arquitectura del sistema.

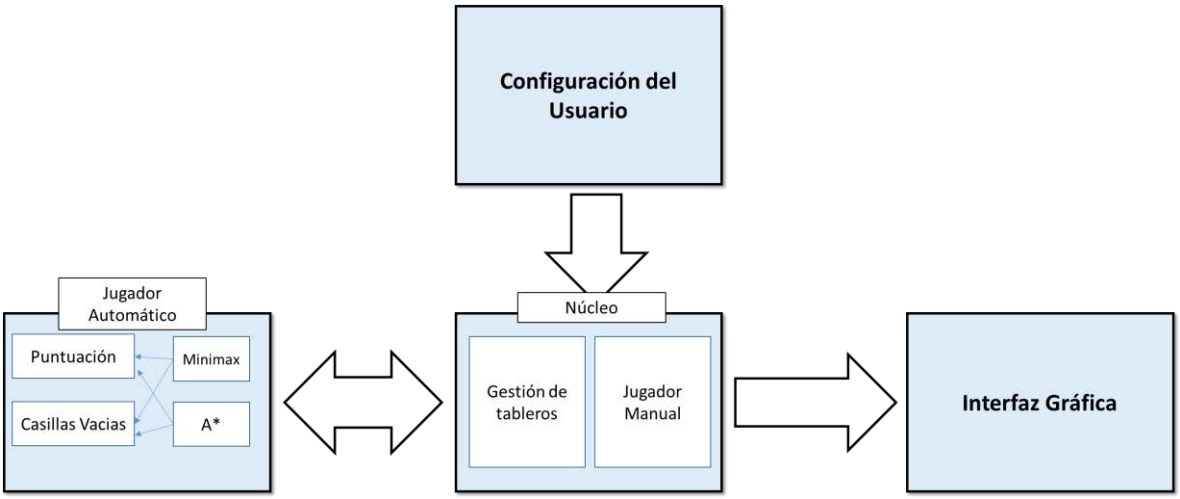

Figura 5 Arquitectura del Sistema

El funcionamiento del sistema es el siguiente:

1. El usuario introduce los parámetros de configuración, eligiendo el algoritmo y la heurística, pasándoselo al núcleo del sistema, en concreto a la Gestión de tableros.
2. Gestión de tableros crea el estado inicial.
3. La gestión de tableros le pasa el estado, tanto a la interfaz gráfica, como al jugador automático, a este último le pasa también la configuración con la que debe ejecutarse. La interfaz gráfica pinta el estado recibido.
4. El jugador automático recibe el estado del tablero y la configuración, esto es, el algoritmo y heurística a utilizar. Ejecuta el algoritmo seleccionado (con la heurística elegida) con el estado que se le ha pasado y le devuelve a la Gestión de Tableros el movimiento a realizar.
5. Gestión de tableros recibe el movimiento a realizar y lo ejecuta, obteniendo un nuevo estado.

Los pasos 3-5 se repiten hasta que gestión de tableros compruebe que o bien no se pueden realizar más movimientos o se haya llegado a 2048.

3.3.2 Descripción general del sistema

El sistema está formado por dos paquetes:

1. El paquete principal que contiene el código del jugador manual, el jugador automático con el algoritmo Minimax y las heurísticas empleadas y el jugador del A* que implementa funciones del segundo paquete. Las clases del paquete y las relaciones entre ellas se muestran en el capítulo de Anexos.

   El juego principal y el manual están contenidos en la clase *Game2048.* Esta clase almacena el tablero como un array de 16 elementos de la clase *Tile.* La clase *Estado* sirve para generar un objeto que almacena para cada instancia del tablero los valores de cada casilla, la puntuación, el rango de movimientos disponibles y un entero representando el último movimiento realizado. La clase *AISolver* llama a los diferentes algoritmos de búsqueda; bien el minimax que esta implementado en la misma clase o bien el A* a través de la clase *AstarSolver* y los métodos de otras clases del otro paquete que implementa. La clase *Boardmaker* sirve de clase auxiliar para los algoritmos para la generación de sucesores del árbol de búsqueda. La clase *movimiento*, sirve de apoyo al algoritmo minimax para almacenar el movimiento y la puntuación del nodo anterior.

2. El paquete de búsqueda que contiene el algoritmo del A* además del resto de clases necesarias para el correcto funcionamiento del algoritmo de búsqueda. En la sección 8.2 de los anexos, se muestra un gráfico con las clases que lo contienen y con qué clases del otro paquete interactúan.
   La clase *AISolver,* del paquete principal, llama a la clase *AStarSolver* que extiende a la clase *AStar* que incorpora el algoritmo de búsqueda del A*. El algoritmo explora los nodos, que se modelan con la clase *SearchNode*, y para cada nodo sus sucesores se almacenan como una lista de nodos sin explorar de la clase *Succesor.* El algoritmo devuelve un Objeto de la clase *Result* que entre otros datos devuelve el camino explorado a través de una lista de estados.
   Para este paquete se ha utilizado los paquetes de búsqueda heurística utilizados en la asignatura Inteligencia Artificial en la Industria del Entretenimiento, los cuales han sido implementados por uno de los directores de este trabajo.

3.3.3 Descripción de módulos

A continuación, se explica más en detalle los distintos módulos en los que se compone el sistema y como estos interactúan entre ellos. Para facilitar el entendimiento y poder desarrollar adecuadamente el sistema, se divide el sistema en tres módulos. Estos módulos se corresponden con los presentados en el punto 3.3.1:

1. Núcleo
2. Interfaz Gráfica
3. Jugador Automático

3.3.3.1 Núcleo

El módulo del Núcleo del sistema está formado por dos partes, el Gestor de Tableros y el jugador manual. Ambas partes están basadas en el código original de Konstantin Bulenkov (Bulenkov, 2014). En el apartado 8.3 de los Anexo se muestra un diagrama de clases de este módulo. El modulo se ejecuta desde la clase *Game2048*.

El gestor de tableros es el encargado de aplicar las reglas y lógica del juego y de realizar los movimientos. Almacena el estado del tablero en todo momento y lo distribuye a los demás módulos del sistema cuando lo necesitan. Este gestor recibe la configuración inicial, es decir, los parámetros de ejecución. Estos parámetros son, si se usará el jugador automático o el manual (la entrada de esta elección es a través de la interfaz de usuario), la elección del algoritmo a usar (0 para minimax y 1 para A*) y la heurística (0 para la puntuación y 1 para el nº de casillas vacías).

Si se ha elegido el jugador automático, se le pasa el estado del tablero y el algoritmo y heurística seleccionado al módulo del Jugador Automático y este devuelve el siguiente movimiento a realizar, y el gestor de tableros lo realiza. Mientras sucede esto, el módulo de la interfaz de usuario está a la espera de que vaya cambiando el estado del tablero para actualizar la interfaz. Al finalizar la partida, este módulo recopila la siguiente información de la partida y la muestra por la salida estándar y la guarda en un fichero:

- Tiempo transcurrido en cada turno, en milisegundos (Solo volcado a fichero)
- Tiempo total de ejecución, en milisegundos
- Número total de movimientos
- Tiempo medio por movimiento, en milisegundos
- Puntuación

- Si la partida se ha ganado o no

Si se ha seleccionado el jugador manual, solo se interactúa con el módulo de la interfaz de usuario. El Gestor del tablero le pasa el tablero al jugador manual. El jugador manual espera a que el usuario teclee el movimiento a realizar y se lo pasa al gestor de tableros, que realiza el movimiento. De igual modo que con el jugador automático, el módulo de la interfaz de usuario va actualizando la interfaz.

El Jugador Manual no se explicará en más detalle debido a que no es el objeto de este trabajo, sino que venía implementado en el código base.

3.3.3.2 Interfaz de Usuario

El módulo de la interfaz de usuario venía en su mayoría implementada en el código base, si bien se han realizado algunas pequeñas modificaciones que afectan a la elección del modo de juego y la actualización de la interfaz cuando se usa el jugador automático.

Cuando se ejecuta el programa, el interfaz de usuario abre una ventana preguntándole al usuario que jugador quiere ejecutar, si el automático o el manual, y le pasa esta información en forma de boolean al módulo Núcleo. En la siguiente figura se muestra una captura de pantalla de este menú.

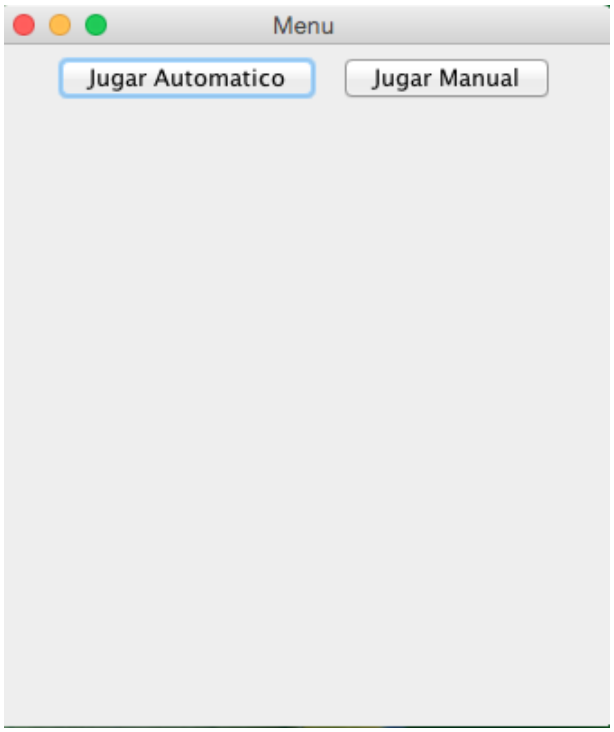

Figura 6 Menú Interfaz de Usuario

En el caso del jugador manual, no hizo falta modificar el funcionamiento de la interfaz de usuario, esta esperaba a que hubiera un evento (pulsación de una tecla) y después iba al gestor de tableros y se actualizaba.

Sin embargo, para el jugador automático, al no producirse un evento con cada actualización del tablero, este no se actualizaba hasta que acababa la ejecución del programa. Para solucionar esto, se puso este módulo a ejecutar en el hilo principal y los otros dos en un hilo secundario, de esta manera, sí que se actualizaba la interfaz gráfica en tiempo real.

A continuación, se muestra una captura de pantalla de la interfaz gráfica mientras ejecuta el jugador automático.

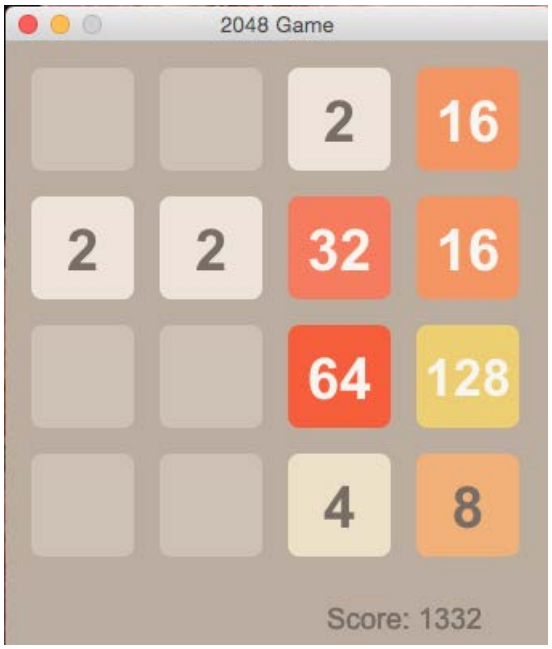

Figura 7 Partida en Ejecución

Al finalizar la partida, la interfaz de usuario también se modifica en función del resultado obtenido. En las dos siguientes figuras se pueden ver cómo queda la interfaz. Primero, cuando se pierde y después cuando se gana.

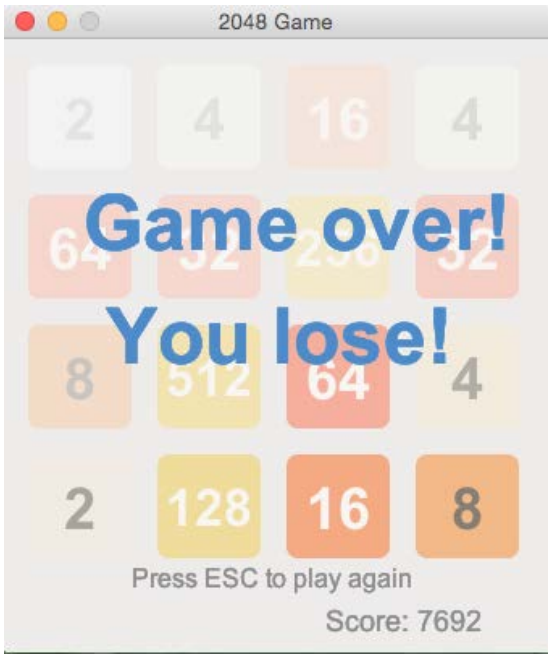

Figura 8 Partida Perdida

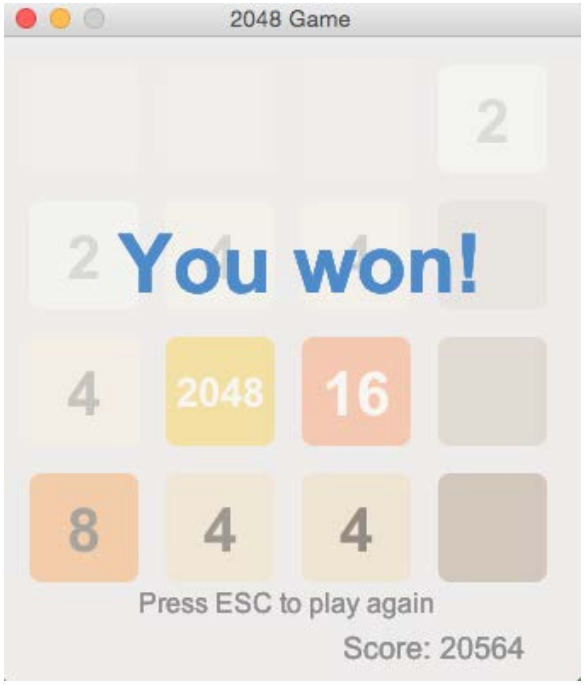

Figura 9 Partida Ganada

3.3.3.3 Jugador Automático

Este módulo es el que implementa al jugador automático. Este módulo tiene como clase principal la clase *AISolver*, que tiene implementadas las dos heurísticas y los dos algoritmos: el minimax y el A*. El A* está implementado mediante una llamada al paquete de Búsqueda, que está descrito en el apartado 3.3.2, y es la clase *AstarSolver* que interactúa tanto con el paquete mencionado y la clase *AISolver*. Ambos algoritmos se pueden utilizar indistintamente con cualquiera de las dos heurísticas.

A continuación, se describirán las dos heurísticas implementadas y posteriormente se procederá a explicar cómo se han implementado los algoritmos.

3.3.3.3.1 Heurística: Puntuación

Esta heurística evalúa el tablero en función de la puntuación que este tenga. Recibe un objeto de la clase *BoardMaker*, que es una clase auxiliar que almacena el tablero, la puntuación y además puede gestionar los movimientos del tablero, lo cual se usa para la generación de sucesores del estado actual. De este objeto, extrae la puntuación y la devuelve. A continuación, se muestra el código de esta función.

```
1    public static int puntuacionHeuristica(BoardMaker tablero){
2            int puntuacion = tablero.myScore;
3            return puntuacion;
4    }
```

Tabla 18 Código Heurística Puntuación

Por ejemplo, en el caso de la Figura 7 Partida en Ejecución, que la puntuación es de 1332 puntos, la heurística devolvería este valor.

3.3.3.3.2 Heurística: Casillas Vacías

Esta heurística evalúa el tablero en función de las casillas vacías que tenga. Recibe como entrada lo mismo que la anterior heurística, un objeto de la clase *BoardMaker.* La función saca el tablero del objeto y comprueba las casillas una a una, de manera que, si la casilla que está evaluando se encuentra vacía, aumenta en uno el valor del contador de casillas vacías. Cuando acaba de recorrer tablero, devuelve el valor de este contador. A continuación, se muestra el código de esta función.

```
1    public static int casillasVaciasHeuristica(BoardMaker tablero){
2            int casillasVacias = 0;
3            for(int i = 0; i < tablero.getMyTiles().length; i++){
4                   if(tablero.getMyTiles()[i].isEmpty()){
5                          casillasVacias++;
6                   }
7            }
8            return casillasVacias;
9    }
```

Tabla 19 Código Heurística Casillas Vacías

Por ejemplo, en el caso de la Figura 7 Partida en Ejecución, que muestra un tablero con 6 casillas vacías, la heurística devolvería un 6.

3.3.3.3.3 Minimax

Desde la clase principal, la clase *AISolver*, en concreto el método *getNextMove()*. Para que el jugador sepa con que algoritmo, heurística y tableros ejecutarse, recibe los siguientes parámetros:

- El estado de la partida
- La profundidad de ejecución en forma de entero

- La heurística a utilizar en forma de entero a elegir entre dos opciones: 0 para la heurística de la puntuación y 1 para la heurística del número de casillas vacías)

Hay dos versiones del minimax implementadas, una por cada heurística, ya que una de ellas Maximiza el tablero del jugador (el de la puntuación) y el otro minimiza el tablero (el de las casillas vacías, ya que en realidad utiliza el inverso, esto es el de las casillas ocupadas). Por lo demás el algoritmo es simétrico para ambas heurísticas.

Es importante destacar, que al contemplar el minimax dos jugadores y al ser el *2048* un juego de un solo jugador se ha tenido que asumir que la casilla que se rellena aleatoriamente tras cada movimiento es el movimiento que realiza la maquina u el otro jugador, y que por tanto es el que debemos minimizar. Para el movimiento de la máquina, para un tablero dado, se genera un sucesor por cada casilla vacía. En cada uno de estos sucesores, se coloca un 2 en una única casilla vacía. De tal manera que, todos los sucesores tendrán un dos en una casilla que antes estuviera vacía distinta a los otros sucesores, es decir, los sucesores deberán ser todos distintos.

El minimax es un algoritmo recursivo, es decir para cada posible movimiento o nodo del árbol de decisión se vuelve a llamar a sí mismo, reduciendo a su vez la profundidad en una unidad, debido a que se va acercando a la profundidad objetivo con cada iteración.

Como ya se ha descrito en el capítulo dos, el comportamiento general del algoritmo, no se expondrán en este apartado el funcionamiento general del algoritmo, simplemente se comentará como se va expandiendo el árbol para la implementación realizada.

Cuando es el turno del jugador, para el tablero inicial se generan como máximo 4 nuevos tableros, uno para cada movimiento que esté disponible. Se comienza primero hacia abajo, luego a la derecha, luego arriba y finalmente hacia la izquierda. Para cada movimiento se vuelve a llamar al algoritmo con el tablero resultante, cambiando el modo de jugador al Ordenador o Maquina.

En el turno del ordenador o maquina el abanico de movimientos posibles es mucho mayor. En vez de 4 movimientos posibles como máximo, hay uno por cada casilla vacía. Se recorre el tablero y se coloca un 2 en cada casilla generando un tablero nuevo cada vez, llamando de nuevo al algoritmo. Para simplificar el árbol de búsqueda se ha decidido simplificar el problema, ya que el juego originalmente coloca un 2 o un 4 de manera aleatoria con una probabilidad del 90% y del 10% respectivamente.

Finalmente, para ambos modos de juego, si se llega un nodo terminal (no hay más movimientos) o se ha llegado a la profundidad objetivo, se devuelve el valor heurístico, y este se va propagando hacia arriba, comparando así las distintas ramas del árbol, eligiendo así la mejor.

El modulo devuelve un entero que representa el movimiento a realizar al gestor de tableros y este lo ejecuta.

3.3.3.3.4 A*

El segundo algoritmo implementado para el jugador automático es el A*. Una de las diferencias de la implementación de este algoritmo con el anterior es que no contempla un movimiento realizado por la máquina. Es decir, cuando se genera un movimiento nuevo, este se genera con un 2 aleatorio colocado entre las casillas vacías, para asegurar poder comparar ambos algoritmos se ha simplificado esta versión también para no incluir la posibilidad del 4 aleatorio. Este algoritmo al igual que el del minimax comienza desde la clase *AISolver*, en el método *getNextMove()*, que recibe el estado de la partida, un entero representando la profundidad de búsqueda del algoritmo, un entero que representa la heurística a utilizar (0 para la puntuación y 1 para el número de casillas Vacías) y un entero representando el algoritmo a utilizar (para A* es un 1).

Como ya se ha expuesto en el capítulo dos el comportamiento general del algoritmo, simplemente se comentarán las particularidades de la implementación.

Cuando se expande un nodo de la lista cerrada y se generan los nuevos nodos se llama al método *generateSuccesors()* que recibe el estado y genera un *Succesor* nuevo por cada movimiento posible, que se devuelven en forma de lista y que pasan a formar parte de la lista cerrada. En este caso se ha decidido que todos los movimientos tengan un mismo coste, por tanto, la única medida de evaluación son las heurísticas, que son las mismas que en el anterior método. De hecho, la clase *AstarSolver* calcula las heurísticas llamando a la clase *AISolver*.

A diferencia del A* original que sigue hasta que no quedan nodos por visitar en la lista abierta o se ha llegado a un nodo terminal, este algoritmo se ha modificado de manera similar al IDA* que se comentó en el apartado 2.4. Cuando se ha llegado a una profundidad objetivo, (al tener todos los movimientos coste 1, el coste de llegar al nodo actual es el mismo que la profundidad actual), entonces el algoritmo devuelve un objeto de la clase *Result* que incluye una lista con el camino recorrido en el árbol. De esta lista se extrae el movimiento realizado el primer lugar, se le pasa al gestor de tableros y este lo ejecuta.

A diferencia del IDA*, no se aumenta la profundidad en la siguiente iteración, pero se ejecuta el algoritmo desde el nuevo estado generado tras realizar el movimiento.

3.3.4 Diagrama de Flujo del Sistema

A continuación, se presenta el diagrama de flujo del sistema.

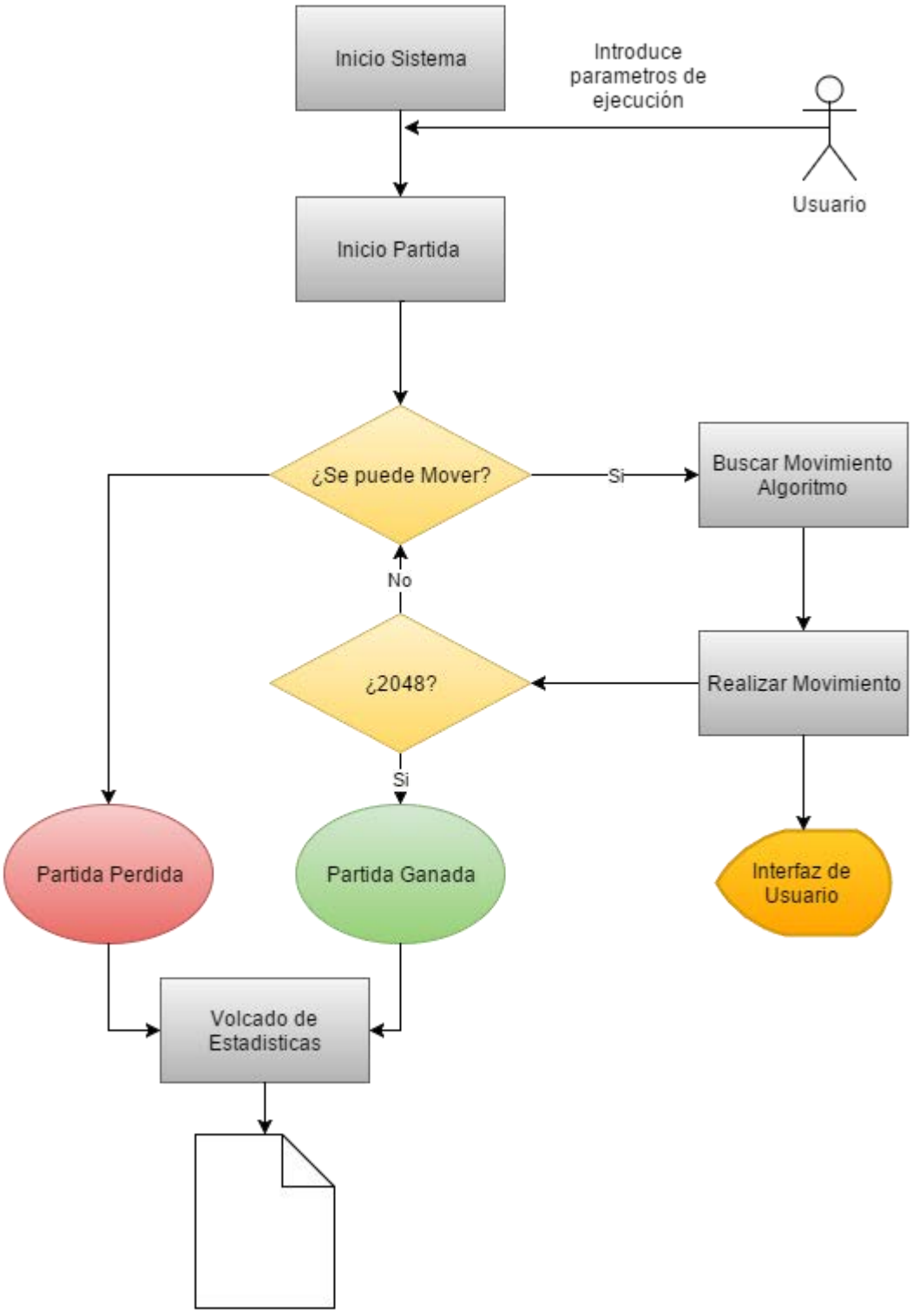

Figura 10 Diagrama de Flujo del Sistema

Capítulo 4: Experimentación

Este capítulo describe el conjunto de experimentos que han sido realizados sobre el sistema que se ha desarrollado en este trabajo. Primero, se presenta en qué consisten los experimentos realizados y sobre qué sistema y plataforma se han realizado. A continuación, se describen los resultados obtenidos durante la experimentación con el algoritmo minimax. Posteriormente, se presentan los resultados obtenidos con el A*. Finalmente, se explica una serie de experimentaciones adicionales que se han realizado, y el motivo por el cual se realizaron.

4.1 Descripción de los Experimentos

En primer lugar, se considera como un experimento una ejecución del sistema, usando el jugador automático, utilizando un algoritmo, heurística y profundidad determinada. Estos parámetros iniciales forman el input del experimento. El output del experimento es, como se explicó en el apartado 3.3.3.1, un fichero con la siguiente información:

- Tiempo transcurrido en cada turno (Ms)
- Tiempo total de ejecución (Ms)
- Número total de movimientos
- Tiempo medio por movimiento(Ms)
- Puntuación
- Si la partida se ha ganado o no

Para determinar que algoritmo presenta un rendimiento mejor para el problema de este trabajo, se han realizado una batería de 90 pruebas con cada algoritmo, la mitad de ellas, 45, utilizando como función de evaluación la puntuación del tablero, y la otra utilizando como heurística el número de casillas vacías. Estas heurísticas que se han utilizado para los experimentos, son las que se explicaron en el punto 3.3.3.3.

Para la elaboración de las estadísticas que se van a mostrar a continuación se ha utilizado el archivo que genera el sistema al ejecutar el algoritmo. Tal como se ha explicado anteriormente, el archivo contiene el tiempo en milisegundos de cada turno juagado, el tiempo total de partida (en ms), el número de movimientos realizado durante la partida, el tiempo medio por movimiento, la puntuación obtenida y si la partida se ha ganado o no.

Para facilitar la comprensión de esta sección se denominará a la heurística basada en la puntuación como heurística 1 (H1) y a la heurística basada en el número de casillas vacías como heurística 2 (H2)

Todas las pruebas se han realizado en el siguiente equipo:

Fabricante	Apple
Modelo	MacBook Pro 15” 2010
Sistema Operativo	Mac OS X 10.10.5 (Yosemite)
Procesador	Intel Core i5 2.53Ghz 2 núcleos
Memoria	4 Gb

Tabla 20 Equipo Utilizado

Se comenzará esta sección comentando los resultados obtenidos por el algoritmo Minimax y después se procederá con los resultados del A*.

4.2 Experimentación Minimax

En esta sección se describe la experimentación realizada sobre el juego *2048* cuando el jugador automático ha sido configurado para que use el algoritmo minimax. Estas pruebas se han realizado para cada una de las heurísticas con tres profundidades del árbol de búsqueda distinta, siendo estas profundidades 5, 7 y 9. Debido a que con este algoritmo el segundo movimiento (y luego el cuarto y el sexto, es decir, los pares) corresponden a movimientos de la maquina o el tablero, como se expuso en el apartado 3.3.3.3.3, la colocación del 2 aleatorio, se ha elegido un número impar de profundidad para dar mayor importancia y tener más en cuenta los movimientos realizados por el jugador. Al elegir un número impar de profundidad, siempre habrá un movimiento más del jugador que de la máquina, salvo que se llegue a un nodo terminal. Para cada profundidad se han realizado 15 ejecuciones. Se ha omitido la profundidad 1 debido a que lo convertiría en un algoritmo reactivo y no tendría en cuenta los futuros posibles estados del tablero a la hora de elegir el movimiento a realizar. Tampoco se valorado los experimentos con profundidad 3 debido a que en esta profundidad no se consigue ganar. Se han intentado profundidades mayores que la 9, por ejemplo, la 11, pero el tiempo de ejecución era demasiado alto para el número de experimentos que se querían realizar.

En primer lugar, analizando el rendimiento del algoritmo independientemente de la profundidad y la heurística, se han ganada un total de 20 Partidas de las 90 partidas jugadas, si bien es cierto que la distribución de las partidas aumenta conforme aumenta la profundidad.

La figura 11 ilustra la distribución del total de las partidas realizadas por la casilla máxima obtenida durante ellas. En primer lugar, tenemos que el número de partidas ganadas representan el 22% del total ejecutado. En segundo lugar, el 58% de las partidas, esto es 52 partidas, han acabado con una casilla con un 1024. Este dato es importante, ya que si bien, no este porcentaje de las partidas no se ha ganado, al obtener esta casilla se puede decir que han estado francamente cerca de ganar. Por tanto, si sumamos ambos porcentajes, tenemos que un 80% de las partidas o se han ganado o han estado cerca de ganar. Por otro lado, el 20% restante corresponden a partidas con un 512 o 256 como casilla máxima. De este último grupo, solo hay 2 partidas, mientras de las que han alcanzado 512, hay 16 instancias. No hay ninguna partida que haya tenido una casilla máxima con menor valor que 256.

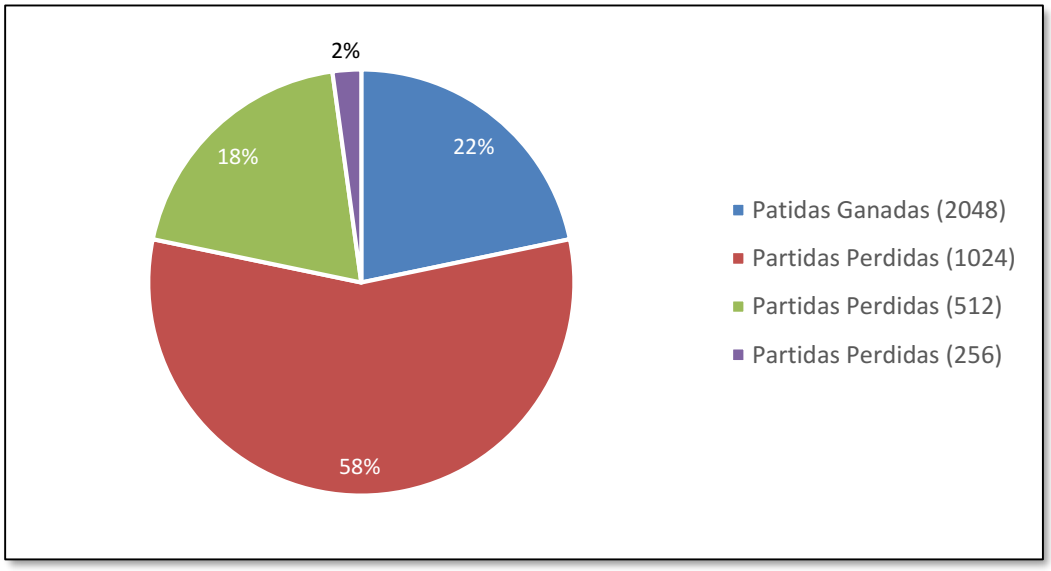

Figura 11 Distribución Partidas

Además, tenemos que la duración media de cada partida ha sido de unos 699.000 milisegundos que llevado a minutos hace una media de duración de las partidas de 11 minutos, creciendo el tiempo de ejecución exponencialmente según se aumentaba la profundidad. La duración total de las pruebas de esta fase han sido un total de 17 horas y 28 minutos.

En la tabla 21 se muestra una media de las estadísticas de la ejecución de del minimax por su profundidad, independientemente de la heurística utilizada. En esta tabla se ve claramente una mejora en la media de puntuación obtenida por partida al aumentar la profundidad, sobre todo al cambiar entre la profundidad 5 a 7, donde la diferencia media es de 4.500 puntos y un incremento del 13% de las partidas ganadas. En el caso de del paso de profundidad 7 a la 9 la diferencia es también significativa al ser de casi 1000. El aumento más significativo es en el porcentaje de victorias que aumenta en casi un 17%. Si observamos la desviación típica de la puntuación observamos que esta va decreciendo, de manera que las puntuaciones van siendo más consistentes conforme aumenta la profundidad

Profundidad	*5*	*7*	*9*	***Media***
Tiempo total(Ms)	**3.834,8**	72.312,3	2.020.577,3	**698.908,1**
Desviación típica tiempo total (Ms)	**616,5**	9.598,6	401.507,7	**137.240,9**
Nº movimientos	748,7	975,3	**1.000,2**	**908**
Desviación Típica Nº movimientos	235,3	172,9	**139,9**	**182,7**
Tiempo medio turno(Ms)	**5,0**	75,0	2.036,1	**705,3**
Desviación Típica Tiempo Medio (Ms)	**1,1**	10,4	363,4	**124,9**
Puntos	11.568,7	16.073,2	**17.057,8**	**14.899,8**
Desviación Típica Puntos	4.736,4	4.028,1	**3.681,8**	**4.148,7**
¿Ganada?	6,67%	20,00%	**36,67%**	**21,11%**

Tabla 21 Estadísticas Minimax por profundidad

La figura 12 compara las puntuaciones medias por profundidad. Si nos fijamos en las barras del gráfico, vemos que el crecimiento en la media de puntuación va siendo menor conforme se aumenta la profundidad. Esto se puede explicar por el límite de puntuación del juego.

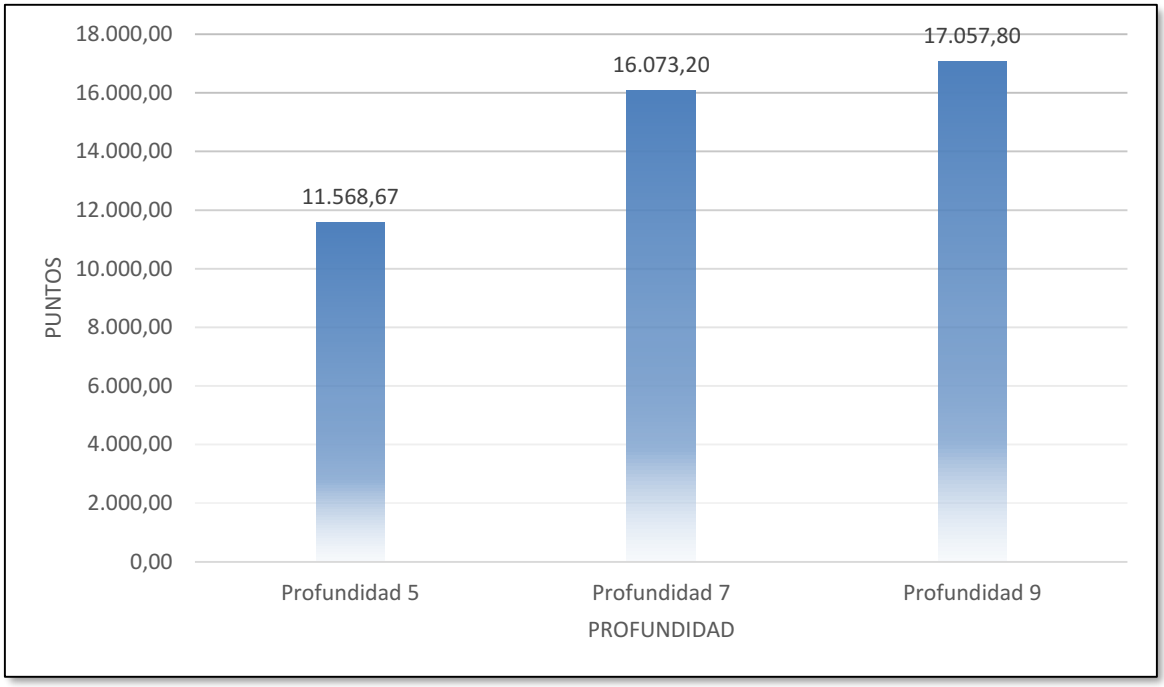

Figura 12 Evolución Puntos por Profundidad

Si observamos la siguiente tabla, vemos que la media de puntuación de las partidas ganadas está en 20.814,9 puntos. Es comprensible que la puntuación media, para cualquier profundidad, como máximo esté limitada por esa puntuación, o una similar, en el caso de que se ganen todas las partidas de los experimentos.

	Tiempo total	*Nº movimientos*	*Tiempo medio turno*	*Puntos*
Media	**1.157.562,95**	**1.072,40**	**1.075,75**	**20.814,90**

Tabla 22 Media Partidas Ganadas

Otra conclusión interesante que sacamos es que el jugador automático necesita unos 1072 movimientos para llegar a formar la casilla del 2048.

Ahora entraremos en más detalle valorando el papel que juega la heurística. En la siguiente tabla se muestran las medias de las estadísticas de las partidas jugadas, distribuidas por profundidad y por la heurística utilizada.

Heurística	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
Profundidad	5		7		9	
Tiempo total(Ms)	**3.799,3**	3.870,3	72.864,7	71.759,9	2.171.985,6	1.869.169,0
Desviación típica tiempo total (Ms)	636,8	**615,8**	11.601,2	7.456,9	480.218,8	232.804,2
Nº movimientos	744,1	753,3	959,5	991,1	**1.041,7**	958,6
Desviación típica Nº movimientos	260,4	216,5	204,6	139,8	**68,7**	179,2
Tiempo medio turno(Ms)	5,2	**4,9**	77,3	72,7	2.080,2	1.992,0
Desviación típica Tiempo Medio (Ms)	1,4	**0,7**	11,7	8,7	405,1	324,5
Puntos	11.491,2	11.646,1	15.362,7	16.783,7	**18.487,2**	15.628,4
Desviación típica Puntos	5.132,4	4.485,0	4.227,8	3.828,1	**2.686,2**	4.060,2
¿Ganada?	6,67%	6,67%	13,33%	26,67%	**53,33%**	20,00%

Tabla 23 Estadísticas por Heurística y Profundidad Minimax

Como se ha explicado anteriormente, al aumentar la profundidad aumenta la puntuación media. Cuando tenemos en cuenta las heurísticas y comparamos el rendimiento de cada heurística respecto la profundidad inmediatamente inferior, se observa que esto también se cumple excepto para los experimentos de H2 en la profundidad 9. En las demás profundidades, la H2 tiene mejor puntuación media que la H1, excepto en la profundidad 9, donde se ve superada por la H1 y además su puntuación medio es inferior a la que obtuvo en la anterior profundidad. En el siguiente gráfico, que muestra la evolución de la media de puntuación por heurística, se ve ilustrada esta situación.

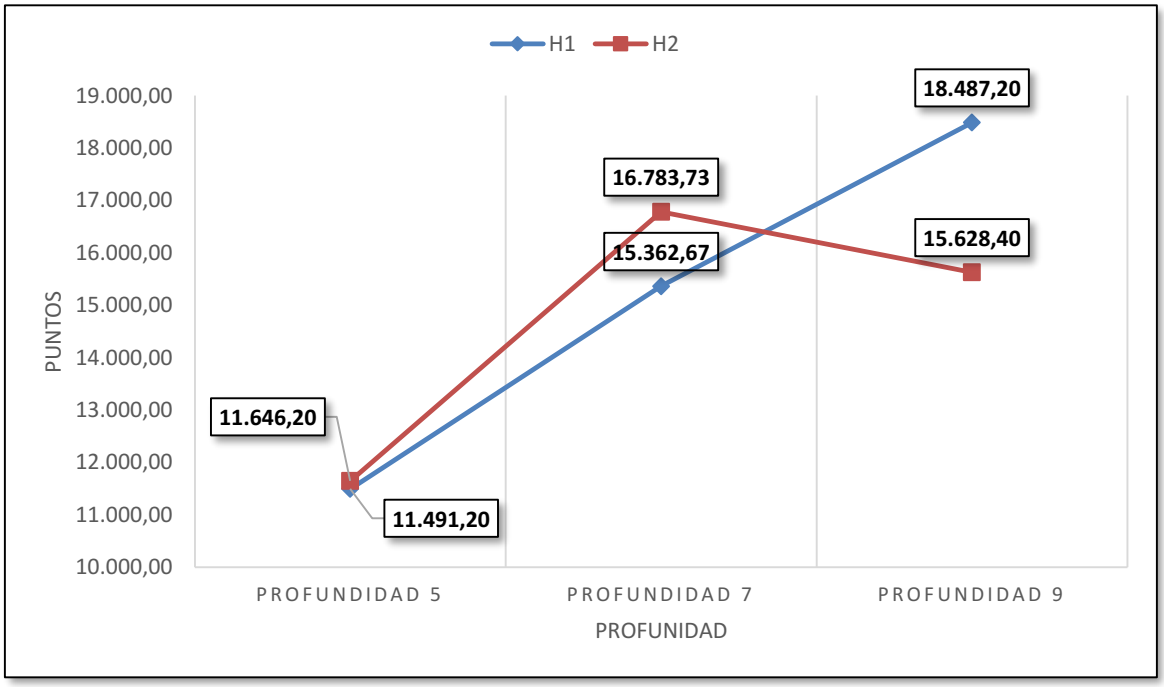

Figura 13 Evolución Media de Puntos por Heurística Minimax

Se puede ver que H1 sigue una progresión casi lineal en aumento de puntuación respecto a sí misma en la profundidad anterior. Por el contrario, la H2 reduce significativamente su media de puntuación en la profundidad 9, creando una brecha entre las dos heurísticas en esta profundidad de casi 3000 puntos.

Tal y como se explicó en la sección 1.1, conseguir formar una casilla de 2048 la puntuación aumenta en 2048 puntos. Teniendo en cuenta que la mayoría de las partidas llegan a formar una casilla de 1024, formar una nueva casilla con 1024 (con la que juntar la otra y formar una de 2048) aumenta la puntuación en al menos 1024 puntos. Teniendo esto en cuenta, ganar la partida aporta un incremento de 3000-4000 puntos. Debido a esto, se han analizado las partidas, pero sin incluir las partidas ganadas, solo las que han perdido, ya que salvo en la primera profundidad experimentada, el número de partidas ganadas varia significativamente de una heurística a otra. El motivo de esto puede ser simplemente por la muestra elegida y no por el rendimiento del algoritmo.

La siguiente tabla muestra las medias de las estadísticas de las partidas jugadas, distribuidas por profundidad y por la heurística utilizada, sin incluir las partidas que se han ganado, es decir, se han quitado 20 partidas a la muestra.

Heurística	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
Profundidad	5		7		9	
Tiempo total(Ms)	**3.757,1**	3.830,7	71.071,8	69.880,9	2.285.694,6	1.835.136,5
Desviación típica tiempo total (Ms)	638,7	**619,0**	21.173,7	7.416,7	1.234.369,9	239.245,1
Nº movimientos	721,3	732,4	946,2	940,8	**1.013,4**	926,8
Desviación típica Nº movimientos	254,2	**208,4**	228,0	129,0	133,2	175,5
Tiempo medio turno(Ms)	5,3	**4,9**	76,8	74,5	2.237,3	2.035,0
Desviación típica Tiempo Medio (Ms)	1,4	**0,7**	22,7	7,4	1.175,9	329,2
Puntos	10.833,4	11.008,6	14.561,8	14.619,6	**15.916,0**	14.332,7
Desviación típica Puntos	4.623,7	3.885,4	4.169,2	**2.365,8**	2.433,3	3.216,2

Tabla 24 Estadísticas Partidas No Ganadas Minimax

Una vez eliminadas las partidas ganadas, observamos que las estadísticas son más parejas que en la tabla anterior, pero aun así sucede el fenómeno que se comentó en el punto anterior, si bien más suavizado. En principio, podríamos asumir que la H2 no mejora necesariamente sus resultados al aumentar la profundidad, también podría ser que las pruebas realizadas con H2 y

profundidad 9 no sean representativas de la capacidad del algoritmo, al no ser lo suficientemente grande.

En la figura 14 se muestra la evolución de la media de puntuación por heurística sin tener en cuenta las partidas ganadas. Se puede observar en esta gráfica algo distinto a lo que sucedía si se tenía en cuenta las partidas ganadas. Con las partidas ganadas H2 aumenta la su diferencia en puntuación media con H1 en la profundidad 7 respecto a la 5. Sin las partidas ganadas, sucede lo contrario, la H1 reduce la brecha con H2 en esta profundidad. Finalmente, en la profundidad 9, H1 obtiene una mayor puntuación media, con una diferencia significativa.

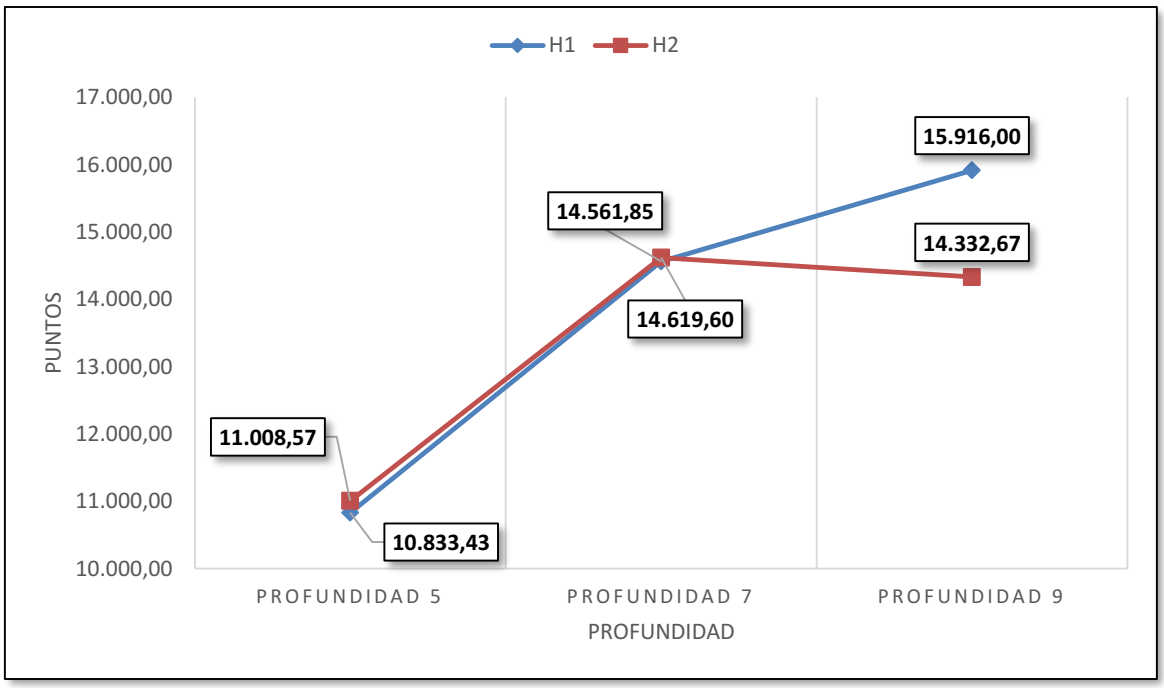

Figura 14 Evolución Media de Puntos por Heurística Partidas No Ganadas

Durante la realización de los experimentos, se observó que los primeros turnos requerían mucho más tiempo de computación respecto a los movimientos. Esto resulta más aparente en las ejecuciones con profundidad 7 y 9. Esto se puede explicar porque al principio, al estar el tablero vacío hay más casillas vacías. Al haber más casillas vacías, aumenta el número de sucesores para los nodos o estados que corresponden a movimientos de la máquina, es decir, la colocación del dos aleatorio tras cada movimiento. Al aumentar el número de sucesores, aumenta el espacio de búsqueda, por lo que es lógico que aumente el tiempo de ejecución del algoritmo.

En las figuras 15, 16 y 7, se muestra para cada profundidad una gráfica con los tiempos empleados en cada movimiento por una partida ganadora con cada una de las heurísticas. Estas gráficas sirven para ilustrar el decrecimiento del tiempo medio de ejecución de cada turno que se ha explicado anteriormente. Para elaborar estas gráficas se han utilizado datos de partidas ganadas, para asegurar una mayor comparabilidad de los experimentos. En estas graficas se muestra en el eje horizontal el número del turno medido, y en el eje vertical mide el tiempo empleado en el turno en milisegundos. En cuanto al eje horizontal, no hay diferencias significativas, entre profundidades, sin embargo, el eje vertical tiene valores mayores según aumenta la profundidad, ya que si aumenta la profundidad aumenta también el espacio de búsqueda.

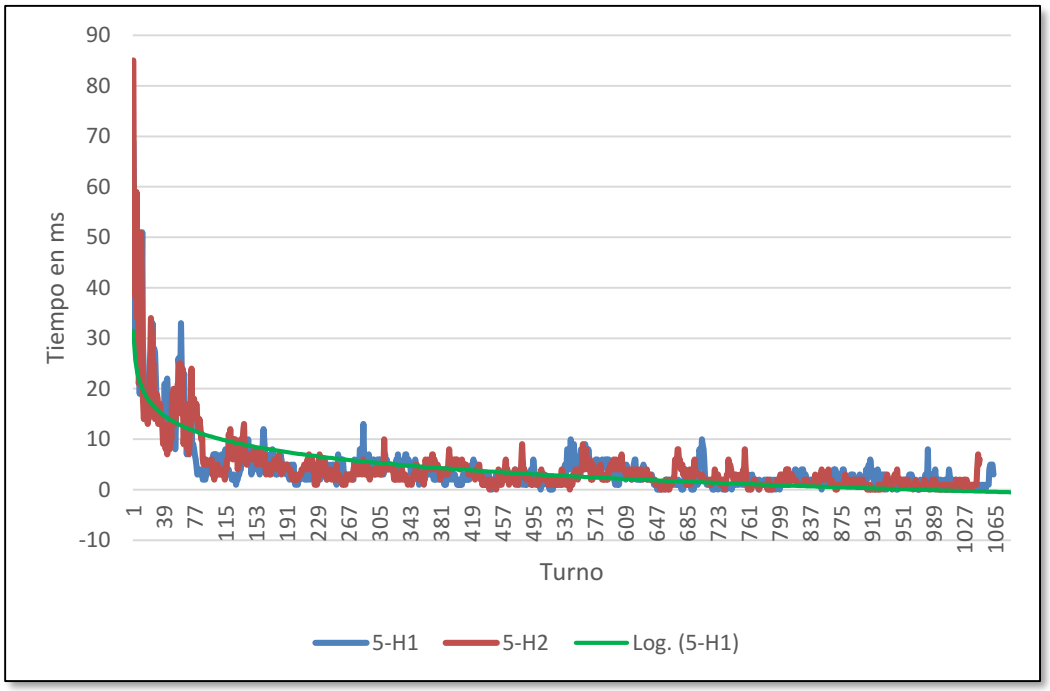

Figura 15 Tiempos por Número de Movimiento Profundidad 5

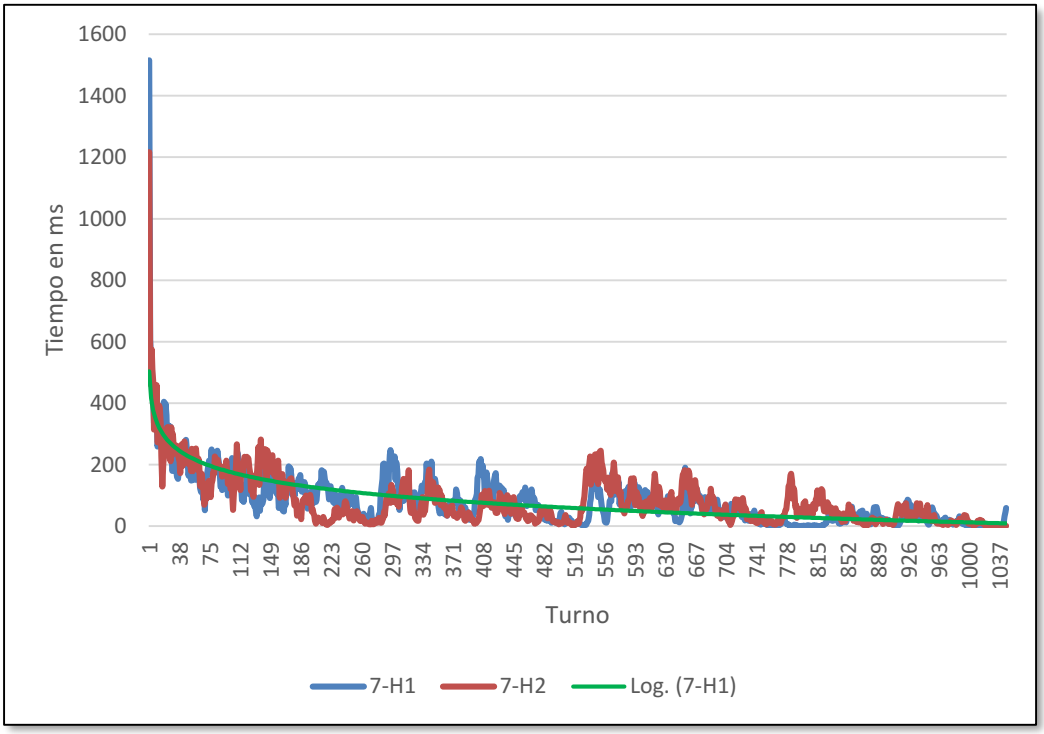

Figura 16 Tiempos por Número de Movimiento Profundidad 7

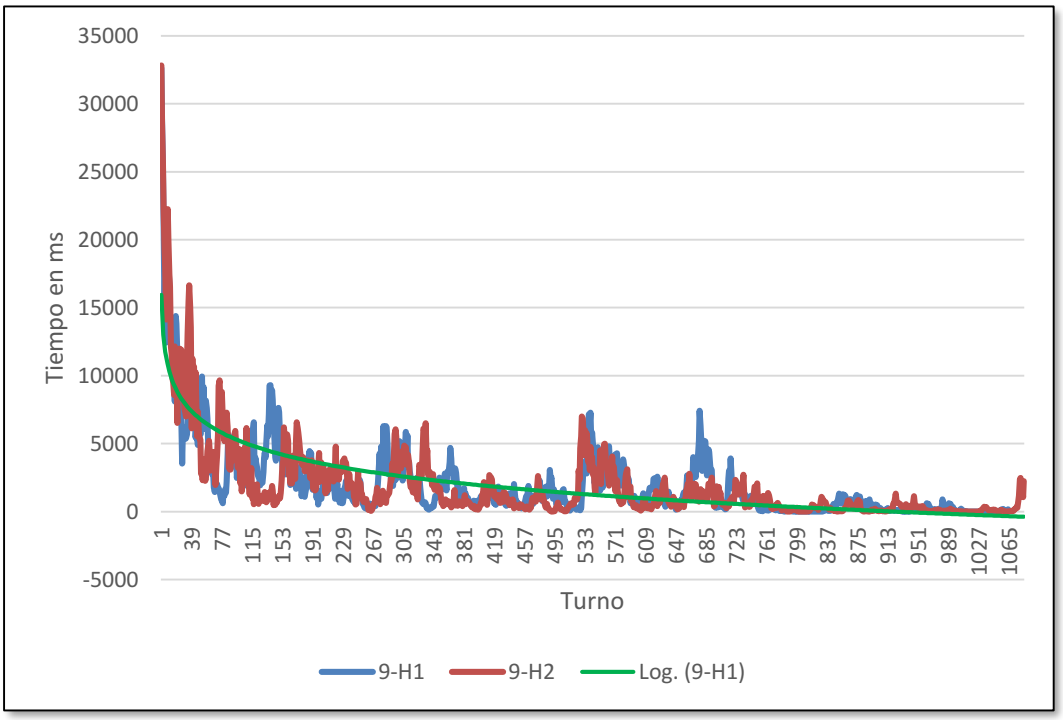

Figura 17 Tiempos por Número de Movimiento Profundidad 9

4.3 Experimentación A*

En este apartado se describe de forma detallada los resultados obtenidos durante las pruebas realizadas durante el A*. Para esta sección se han realizado 15 pruebas para cada una de las dos heurísticas, H1 y H2, con 3 profundidades distintas. A Diferencia del algoritmo del minimax, el A* no es un algoritmo que tenga en cuenta un posible movimiento de la maquina por lo que no tiene en cuenta las posiciones donde se pueda colocar el dos automáticamente tras cada movimiento. Esto afecta significativamente al rendimiento del algoritmo, ya que hay una gran cantidad de estados que se quedan fuera del espacio de búsqueda del algoritmo. Para estas pruebas se han probado las profundidades 5, 8 y 10. Estas profundidades son distintas a las del minimax, ya que al ejecutar en profundidad 5 y comprobar que los resultados no eran comparables a los de minimax, se intentó aumentar estas para ver si así conseguía mejorar la situación

Primero analizamos los experimentos independientemente de las heurísticas y profundidad. De las 90 partidas jugadas no se han obtenida ninguna victoria, de hecho, ninguna de las partidas se puede considerar que hayan estado cerca de ganar.

En la tabla 25 se muestra una media de las estadísticas obtenidas durante la ejecución las partidas, teniendo solo en cuenta la profundidad, es decir sin tener en cuenta la heurística utilizada. Observamos que las puntuaciones no son altas. Esto se puede deber, tal y como se indicó anteriormente, a que faltan nodos en el espacio de búsqueda, unido además a que probablemente las heurísticas usadas, y descritas en el apartado 3.3.3.3, no estiman apropiadamente el coste necesario para llegar al nodo objetivo. Debido a que no hay partidas ganadas, se ha omitido la columna de porcentaje de partidas ganadas que sí que estaba presente en las tablas de los experimentos de minimax.

Profundidad	*5*	*8*	*10*	***Media***
Tiempo total(Ms)	**264,9**	1.253,4	5.793,1	**2.437,1**
Desviación típica tiempo total (Ms)	**98,6**	729,5	5.003,6	**1.943,9**
Nº movimientos	64,2	**76,4**	74,0	**71,5**
Desviación típica Nº movimientos	**14,7**	22,0	27,1	**21,3**
Tiempo medio turno(Ms)	**3,7**	16,7	78,8	**33,1**
Desviación típica Tiempo Medio (Ms)	**1,4**	10,0	55,9	**22,4**
Puntos	355,7	**487,5**	455,9	**433,0**
Desviación típica Puntos	**140,8**	214,4	270,3	**208,5**

Tabla 25 Estadísticas A* por Profundidad

Observamos que según aumentamos la profundidad de búsqueda del algoritmo no se consigue que aumente significativamente los puntos obtenidos en la partida. Sí que va aumentando el tiempo total de ejecución, aproximadamente 5 veces por cada vez que se aumenta la profundidad, pero al no aumentar el número de movimientos significativamente esto se debe al aumento de la profundidad que hace que el algoritmo tarde más. Además, al contrario de lo que sucedía con minimax, la desviación típica de la media de puntos no decrece según aumenta la profundidad, por lo que las puntuaciones no convergen.

En el gráfico 18 se comparan las medías de puntuación obtenidas en cada profundidad.

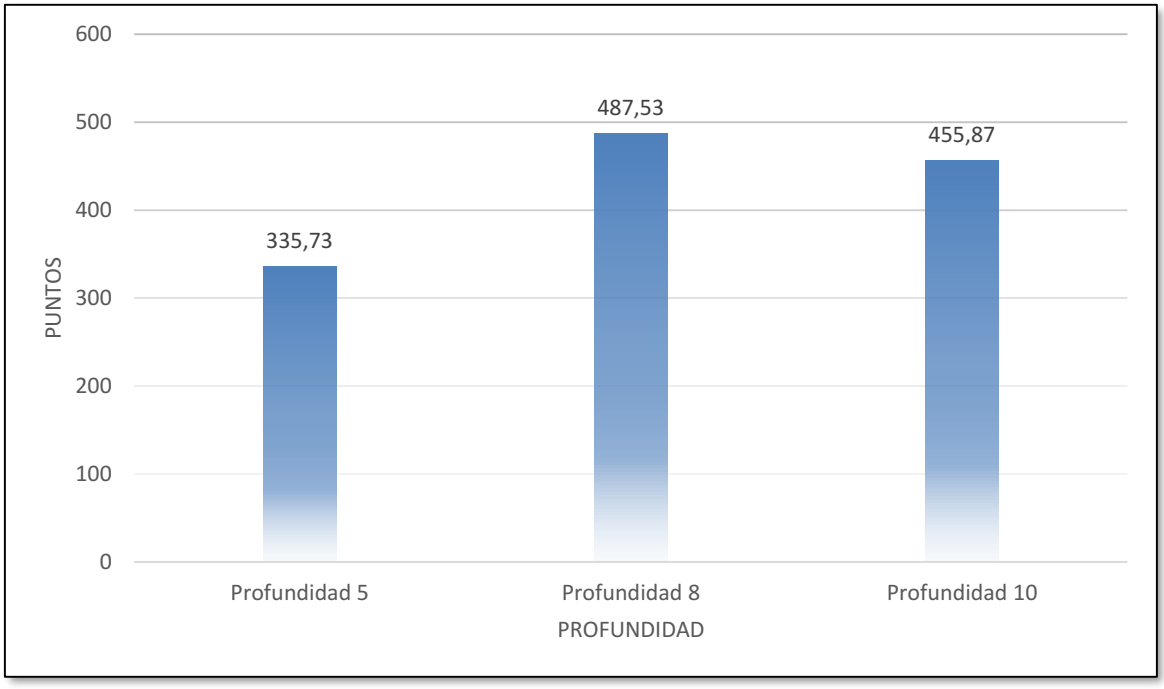

Figura 18 Puntuación Media por Profundidad A*

Observamos que al pasar de la primera a la segunda profundidad si se obtiene un aumento en la puntuación obtenida pero después al aumentar de nuevo, este aumento se corta e incluso decrece, y no se puede justificar como en el caso del minimax que estaba acotado por la puntuación máxima de ganar la partida.

Ahora entramos a valorar el papel que juega la heurística en el algoritmo. La tabla 26 muestra una media de las estadísticas de ejecución de los experimentos, desglosado por heurísticas.

Heurística	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
Profundidad	5		8		10	
Tiempo total(Ms)	**181,7**	348,1	598,9	1.907,9	1.848,3	9.738,0
Desviación típica tiempo total (Ms)	**36,7**	63,1	212,7	373,1	571,3	4.264,7
Nº movimientos	65,0	63,4	75,8	77,0	69,7	**78,3**
Desviación típica Nº movimientos	17,6	**11,8**	21,9	22,8	23,3	30,6
Tiempo medio turno(Ms)	**2,5**	4,9	7,7	25,7	28,1	129,6
Desviación típica Tiempo Medio (Ms)	**0,6**	0,9	2,3	5,2	11,3	28,6
Puntos	361,1	350,4	**493,1**	482,0	425,2	486,5
Desviación típica Puntos	168,5	**112,4**	209,8	226,2	252,6	292,5

Tabla 26 Estadísticas por Heurística y Profundidad A*

Observamos que la H1 es la que tiene unas puntuaciones más bajas en la profundidad 10 mientras que la H2 mantiene sus puntuaciones más estables. Observamos también que la H2, la que evalúa el tablero en función de las casillas vacías, tiene un tiempo de ejecución significativamente mayor al de la H1 sin tener mejores puntuaciones, por lo que, al menos en tiempo de ejecución, la heurística H2 no es la más apropiada para el algoritmo A*.

En el siguiente gráfico observamos que ambas heurísticas tienen un comportamiento similar en la profundidad 5 y en la 8, estas divergen, como se ha comentado anteriormente al analizar los datos de la tabla 4.6.

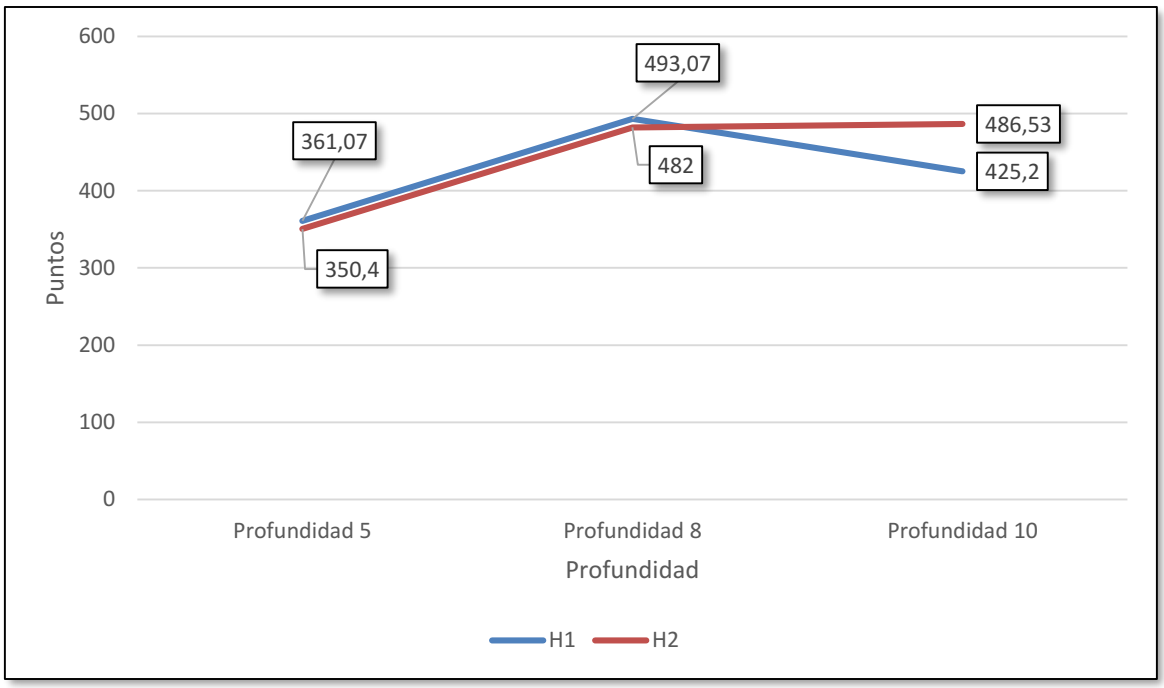

Figura 19 Evolución Media Puntos por Heurística A*

4.5 Experimentación Adicional A*

Como los resultados obtenidos no eran los esperados, se alteraron las heurísticas, respecto a las explicadas originalmente en el punto 3.3.3.3, para ver si así se obtenían mejores resultados.

Teniendo en cuenta que el algoritmo A* elige los nodos de la lista abierta en función del resultado de la ecuación $f(n) = g(n) + h(n)$ siendo *g(n)* el coste hasta llegar al nodo y *h(n)* el valor de la heurística, que debe ser una estimación de lo que falta para llegar al estado objetivo, son proponen los siguientes cambios.

En el caso de la H1, se ha modificado de tal manera para que devuelva como valor de la heurística la diferencia entre la puntuación del tablero y 20000 puntos, que es una estimación, en base a los resultados obtenidos en los experimentos del minimax, de los puntos necesarios para ganar la partida.

A continuación, se muestra el código que implementa la modificación. La función recibe el valor heurístico de la heurística original y realiza la modificación y la devuelve. Por ejemplo, si un tablero tuviera una puntuación de 1452 puntos, su valor heurístico antiguo era ese mismo valor. Con la nueva modificación seria 18.548 puntos.

```
1    puntHeuristica = AISolver.puntuacionHeuristica(new BoardMaker(estado.get_myTiles(),estado.get_myScore()));
2    puntHeuristica = 20000 - puntHeuristica;
```

Tabla 27 Modificación H1

En el caso de la H2, se ha seguido un procedimiento similar, solo que en esta ocasión se ha puesto la diferencia que sale entre el número de casillas vacías y 16, el número total de casillas del tablero, de tal forma que idealmente esta heurística devuelva 0, es decir, todas las casillas vacías, pero esto es imposible debido a las casillas rellenas aleatoriamente después de cada movimiento.

A continuación, se muestra el código que implementa la modificación. La función recibe el valor heurístico de la heurística original y realiza la modificación y la devuelve. Por ejemplo, si un tablero antes tuviera 4 casillas vacías, su valor heurístico antes era de 4. Con la nueva modificación seria de 12.

```
1    puntHeuristica = AISolver.casillasVaciasHeuristica(new BoardMaker(estado.get_myTiles()));
2    puntHeuristica = 16 - puntHeuristica;
```

Tabla 28 Modificación H2

Se han realizado diez pruebas con cada una de las heurísticas tras las modificaciones, y los resultados obtenidos son los mostrados en la tabla 29:

Heurística	*H1*	*H2*
Profundidad	10	10
Tiempo total(Ms)	13.871,8	318.563,4
Desviación típica tiempo total (Ms)	4.709,8	170.045,9
Nº movimientos	449,6	259,9
Desviación típica Nº movimientos	187,5	178,3
Tiempo medio turno(Ms)	31,5	1.368,9
Desviación típica Tiempo Medio (Ms)	15,5	739,2
Puntos	6.296,8	4.532,4
Desviación típica Puntos	2.965,0	2.536,2

Tabla 29 Estadísticas Heurística V2

Los resultados obtenidos en estas nuevas pruebas son significativamente mejores que los obtenidos con la misma profundidad, pero aún no se logra ganar. También es importante destacar que los tiempos de ejecución son mayores con H1 V2 e inmensamente más grandes para el caso de H2 V2.

4.6 Conclusiones Experimentación

Después de los resultados obtenidos queda claro que el algoritmo más eficiente, y que mejor consigue jugar al *2048,* es el Minimax. El A* no consigue unos resultados que se puedan considerar decentes principalmente porque no tiene en cuenta el rango de lugares donde colocar el dos aleatorio, restándole gran parte del espacio de búsqueda al problema.

En cuanto a heurísticas, con los resultados obtenidos con el Minimax, ambas tienen un rendimiento similar en cuanto a puntos se refiere, pero al usar la heurística de la puntuación se obtiene un porcentaje mayor de victorias, un 24% frente a un 17%.

Pinchar aquí para ver un video de una ejecución del sistema.

Capítulo 5: Gestión del proyecto

En este capítulo se realiza una descripción del proceso de desarrollo que se ha realizado para la implementación de este trabajo. Además, se realiza una descripción de las distintas fases del proyecto y se explica la planificación que se ha seguido a lo largo del proyecto. Finalmente se elabora el presupuesto necesario para desarrollar el proyecto.

5.1 Descripción de las fases del proyecto

Para llevar a la realización de este proyecto se ha dividido el proceso en 6 fases diferentes. Las fases son:

- Análisis: Esta fase tiene como objetivo detallar que debe hacer el sistema y de qué manera. En esta fase se creó una serie de casos de uso y requisitos.
- Diseño: En esta fase se determina la arquitectura del sistema, identificando los distintos módulos y cómo interactúan entre ellos
- Implementación: El objetivo de esta fase es desarrollar y llevar a cabo el diseño realizado en la fase anterior.
- Pruebas: Una vez que se tiene una versión funcional, se llevan a cabo pruebas para determinar al correcto funcionamiento del sistema y que cumple con los requisitos que se fijaron en la primera fase.
- Evaluación: Para comparar los dos algoritmos implementados en el sistema, se realizan una serie de experimentos, para luego recopilar sus datos y analizarlos.
- Documentación: Esta fase consiste en la elaboración de este documento.

5.2 Planificación

El objetivo de esta sección es dar una estimación de la dedicación de tiempo a cada fase del proyecto.

Tal y como se comentó en la sección anterior, el proyecto consta de 6 fases distintas: Análisis, diseño, implementación, pruebas, evaluación y documentación. Cada una de estas fases no puede realizarse sin que se haya completado la anterior, por lo que deben realizarse en orden.

El proyecto comenzó el 1 de octubre de 2015 y ha finalizado el 22 de junio, si bien debido a otros proyectos y compromisos laborables la dedicación al mismo no ha sido exclusiva.

Para la realización de dicha planificación se ha tenido en cuenta que cada jornada de dedicación al proyecto consiste de 5 horas. Los días trabajados son independientes del día de la semana, ya que para este proyecto todos los recursos utilizados estaban disponibles los 7 días de la semana, festivos incluidos.

A continuación, se muestra una tabla con la planificación detallada por tareas de cada etapa además de un diagrama de Gantt ilustrando dicha planificación.

Nombre de Tarea	Duración	Comienzo	Fin	Rol Encargado
Desarrollo del Trabajo	**190 días**	**Thu 01/10/15**	**Wed 22/06/16**	
Análisis	**5 días**	**Thu 01/10/15**	**Wed 07/10/15**	
Análisis del 2048	2 días	Thu 01/10/15	Fri 02/10/15	Analista
Estudio de los diferentes algoritmos	1 días	Mon 05/10/15	Mon 05/10/15	Analista
Descripción de Requisitos del Sistema	2 díass	Tue 06/10/15	Wed 07/10/15	Analista
Diseño	**5 días**	**Thu 08/10/15**	**Wed 14/10/15**	
Diseño de la Arquitectura	1 días	Thu 08/10/15	Thu 08/10/15	Analista
Diseño Modulo Núcleo	1 días	Fri 09/10/15	Fri 09/10/15	Analista
Diseño Modulo Interfaz de Usuario	1 días	Mon 12/10/15	Mon 12/10/15	Analista
Diseño Modulo Jugador Automático	2 días	Tue 13/10/15	Wed 14/10/15	Analista
Implementación	**27 días**	**Wed 20/04/16**	**Thu 26/05/16**	
Implementación Minimax	15 días	Wed 20/04/16	Tue 10/05/16	Programador
Implementación A*	12 días	Wed 11/05/16	Thu 26/05/16	Programador
Pruebas	**4 días**	**Thu 26/05/16**	**Tue 31/05/16**	
Pruebas Minimax	1 días	Thu 26/05/16	Thu 26/05/16	Programador
Pruebas A*	4 días	Thu 26/05/16	Tue 31/05/16	Programador
Experimentación	**6 días**	**Tue 31/05/16**	**Sun 05/06/16**	
Experimentación Minimax	3 días	Tue 31/05/16	Thu 02/06/16	Testers

Experimentación A*	3 días	Thu 2/05/16	Sun 05/06/16	Testers
Documentación	**14 días**	**Sun 05/06/16**	**Wed 22/06/16**	**Analista**

Tabla 30 Panificación del Proyecto

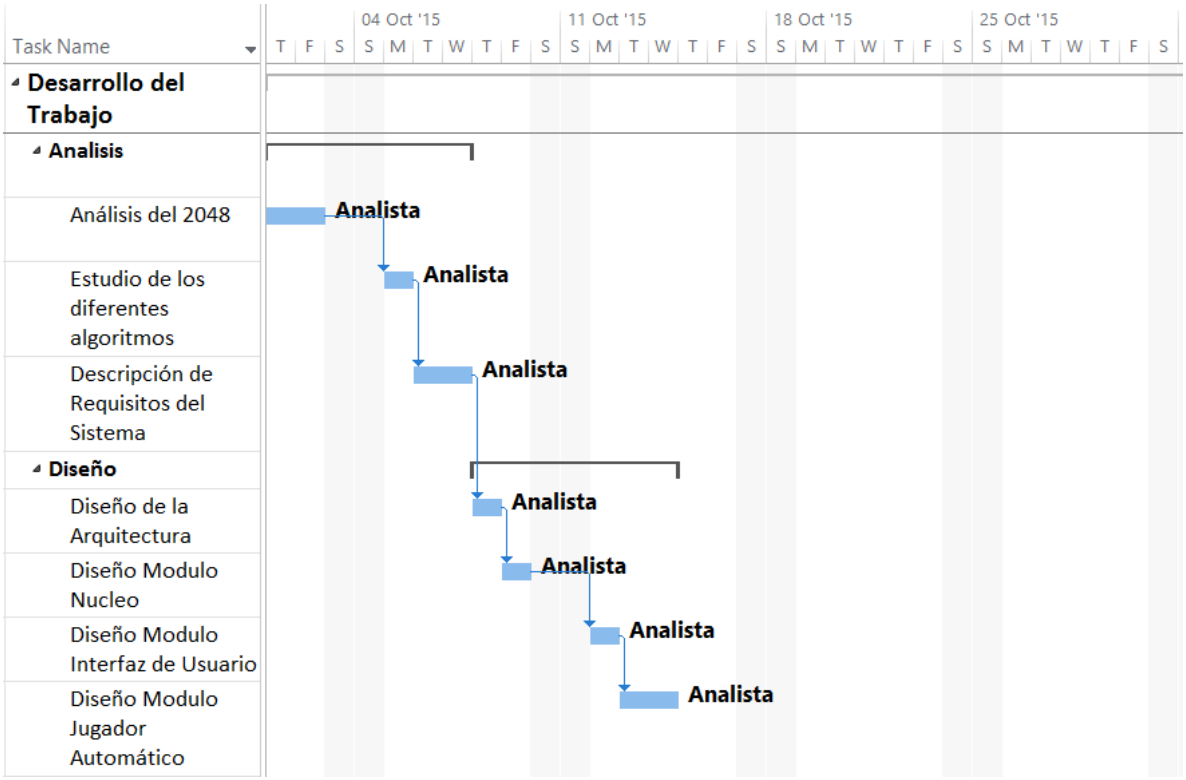

Figura 20 Diagrama Gantt Fases 1-2

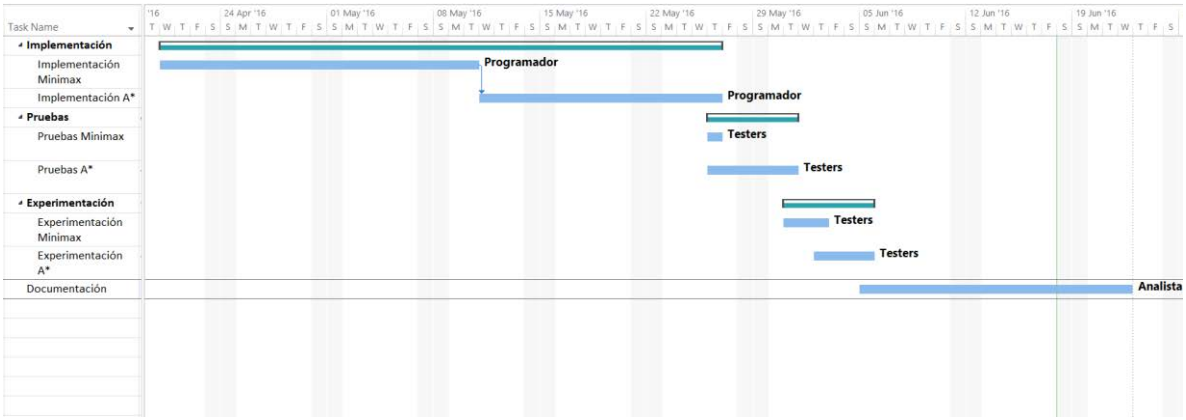

Figura 21 Diagrama Gantt Fases 3-6

5.3 Presupuesto

El capítulo del presupuesto se divide en dos partes, la primera relativa al coste del personal utilizado para el proyecto y la segunda correspondiente al material necesario para el proyecto.

Finalmente se realiza un resumen del presupuesto total incluyendo impuestos, margen de beneficios y cálculo del beneficio final.

Debido al tamaño de este proyecto se consideró que solo sería necesaria una persona para la realización del proyecto. Como se comentó en la sección anterior se considera que cada jornada consiste en 5 horas aproximadamente, debido a que tiene otros compromisos laborales. En la siguiente tabla se muestran los costes asociados a esta persona. El salario anual se ha obtenido del Boletín Oficial del Estado según grupos profesionales y funciones.

PUESTO	SALARIO ANUAL	COSTE/HORA	HORAS/DIA	DIAS	COSTE (EUROS)
ANALISTA	24.600 €	11,8 €	5	24	1.416 €
PROGRAMADOR	15.800 €	7,6 €	5	31	1.178 €
TESTER	10.000 €	4,8 €	5	6	144 €
TOTAL PERSONAL					**2.738 €**

Tabla 31 Presupuesto Personal

En cuanto al material utilizado, se han utilizado dos equipos, ya que uno de ellos tuvo una avería durante el desarrollo. Para calcular el coste del equipo se ha usado el valor contable del equipo, que es el valor de compra menos la amortización acumulada del equipo. Al tratarse de equipos informáticos se estima la vida útil de los activos en 4 años, por tanto, los equipos informáticos se amortizan un 25% de su valor original anualmente. La fórmula que calcula el valor contable de los equipos es:

$$Valor\ contable = Precio - Precio\ x\ Tasa\ amortización\ x\ nº\ de\ Periodos\ Amortizados$$

En la tabla 5.3 se muestran los equipos usados y su coste proporcional para el proyecto.

EQUIPO	VALOR CONTABLE	COSTE IMPUTABLE
PC (I7 2.8 GHZ RAM 12GB)	937,50 €	156,67 €
MACBOOK PRO (I5 2.5 GHZ RAM 4GB)	1.500 €	250,68€
TOTAL		**407,35€**

Tabla 32 Presupuesto Equipos Utilizado

Por tanto, el presupuesto final para este proyecto sería el siguiente:

CONCEPTO	PRECIO	IVA(21%)	TOTAL
PERSONAL	2.738 €	574,98 €	3.312,98€
EQUIPOS	407,35€	85,54€	492,89€
LICENCIA JAVA	0€	0€	0€
LICENCIA ECLIPSE	0€	0€	0€
MARGEN DE BENEFICIO (30%)	1.044,75€	219,38€	1.264,13€
TOTAL			**5.070€**

Tabla 33 Presupuesto Total

El precio del presupuesto de este proyecto para un cliente seria de 5.070 €

Capítulo 6: Conclusiones y trabajos futuros

Este capítulo presenta las conclusiones obtenidas tras la realización de este trabajo. En primer lugar, se presentan las conclusiones generales sacadas lo largo del trabajo. En el segundo apartado, se comparan los resultados obtenidos con los objetivos que se marcaron en le sección 1.3 de este trabajo. Finalmente se proponen posibles mejoras y distintos enfoques que se le pueden dar en un futuro a este trabajo para mejorar sus resultados.

6.1 Conclusiones generales

En primer lugar, durante la realización de este proyecto se ha aprendido a implementar un proyecto de software completo mediante una definición inicial de funcionalidades, un diagrama de casos de uso, una definición de requisitos, diseñando la solución, la implementación del sistema y finalmente las pruebas y experimentación de las diferentes configuraciones. Este proceso incluye también la documentación de todas sus fases.

En segundo lugar, tras las investigaciones realizadas sobre las distintas técnicas de Inteligencia artificial, se ha descubierto que los juegos de tablero han sido objeto de estudio desde las primeras investigaciones en el campo de la inteligencia artificial y que resultan sencillos de modelizar ya que las reglas están perfectamente definidas, el rango de posibles acciones está limitado, los resultados posibles son claros y el número de estados del tablero es finito.

Aparte del conocimiento adquirido durante la realización del proyecto, se ha considerado que ha sido un proyecto interesante de realizar e incluso se puede decir que ha sido entretenido, si bien también ha habido momentos de frustración durante el mismo, como por ejemplo conseguir que la interfaz de usuario se actualizara con cada movimiento del jugador automático, que como se indica en el apartado 3.3.3.2 se solucionó ejecutando la interfaz en el hilo principal ejecutando la interfaz en el hilo principal y el jugador automático en otro hilo.

6.2 Conclusiones referentes a los objetivos

Para esta sección se enumeran los objetivos que se plantearon al inicio del trabajo, y en cada uno se explicará si se han conseguido o no.

- Estudio y análisis del funcionamiento del juego *2048:* Cumplir este objetivo fue uno de los primeros hitos de este trabajo, ya que era necesario para poder avanzar con el resto.

Finalmente, tal y como se expone en el apartado 3.3.2 se decidió usar un código ya existente, de Konstantin Bulenkov, como base del trabajo.

- Estudio u análisis de los diferentes algoritmos de inteligencia artificial utilizados para la resolución del de juegos de tablero: Tras analizar las características del juego del *2048* se analizaron los trabajos realizados en otros juegos de tablero y se estudiaron los métodos usados en estos trabajos, para ver que enfoques se adecuaba a este problema.
- Implementación o despliegue de los diferentes algoritmos seleccionados para la resolución de partidas del juego *2048.* Se implementó el algoritmo del Minimax y se adaptó una versión existente del A*, tal y como se describe en el apartado 3.3.3.3.
- Definir un conjunto de pruebas con el fin de analizar el funcionamiento de los distintos algoritmos seleccionados. Como se expuso en el capítulo cuarto, se realizaron experimentos con los distintos algoritmos. En el caso de los experimentos de minimax, estos se pueden considerar satisfactorios, ya que, si bien no tiene un 100% de victorias, un porcentaje alto de las partidas ganaron o quedaron cerca de ganar. En el caso del A*, los resultados no fueron los esperados, debido a que tal y como se implementó el algoritmo, dejaba una parte importante del espacio de estados sin explorar.
- Desarrollo de un documento que describa el proceso de análisis, diseño e implementación de este trabajo. Este objetivo consistía en el desarrollo de este documento.

6.3 Trabajos futuros

Una vez realizado el proyecto y tras conocer los resultados obtenidos se han pensado posibles mejoras o distintos enfoques que se pueden aplicar al proyecto.

A nivel general se ha pensado que podrían aplicarse técnicas de aprendizaje automático. Para ello habría que jugar un número significativo de partidas, recopilando información durante ellas en ficheros, para después utilizar un conjunto de experimentos y mediante algoritmos de aprendizaje supervisado o no supervisado, aprender modelos de comportamiento que nos permitan tomar decisiones.

Como se puede observar tras las explicaciones del apartado 3.3.3.3, las heurísticas utilizadas en este trabajo son bastante simples. Para mejorar el rendimiento de los algoritmos se podría utilizar una función más elaborada que tenga en cuenta más factores del tablero. Por ejemplo, se podría

usar una función que combine ambas, es decir, valore tanto la puntuación del tablero, como el número de casillas vacías. También se puede usar una función que tenga en cuenta la distribución de las casillas con valores altos en el tablero. De tal forma, que, por ejemplo, valore mejor los tableros que tengan los valores altos en las casillas inferiores de la parte derecha del tablero.

Para el caso del A* se pueden modificar dos aspectos. El primero de ellos, modificar la generación de sucesores de manera que tenga en cuenta las distintas posiciones donde puede aparecer un dos aleatorio, de manera que luego el algoritmo busque en un espacio de estados completo. Además, se pueden modificar las heurísticas en la misma línea que se hizo en la batería de pruebas adicionales que se explicó en la sección 4. Esto es, tratando de convertir la heurística en una aproximación del coste necesario para llegar a formar la casilla del 2048.

Finalmente, para el minimax, aparte de los cambios mencionados para las heurísticas, se puede modificar el algoritmo para reducir su tiempo de ejecución mediante la poda de ciertas ramas del árbol de búsqueda y así poder ejecutarlo en mayores profundidades y así aumentar los resultados. Una forma de conseguir esto es aplicar poda alfa-beta que como se explicó en el capítulo 2 de este documento reduce el tiempo de computación sin sacrificar la posibilidad de encontrar la solución. También resultaría interesante quitar el límite de ejecución, es decir, dejar que juegue hasta más de 2048 y ver que puntuaciones obtiene.

Capítulo 7: English Summary

The purpose of this chapter is to serve as a brief overlook of the whole project in English. This chapter will be divided in the following sections:

- Introduction in which we will be introducing the problem to be solved during this project and its objectives.
- Experimentation, during which the results of the tests carried out will be explained in detail.
- The last section of this chapter will consist of the conclusions of the project as well as future improvements to it.

7.1 Introduction

This document contains the scope, achieved results and conclusions of the project *Jugando al 2048 con Inteligencia Artificial* during which we try to solve the game using different techniques of Artificial Intelligence (AI).

To be able to talk about developing intelligent software we must first start by giving a definition of intelligence. Intelligence can be defined as a group of abilities such as planning, problem solving and in general, reasoning. A simpler definition of this concept would be that intelligence is the capacity to make the right decision given a set of initial inputs. (Jones, 2008)

Throughout history, there have been different approaches to giving "intelligence" to a software using a variety of techniques, thus, creating an artificial intelligence. Artificial intelligence is a field of study in computer science, math, logic and philosophy that concentrates on studying and designing systems capable of solving problems on their own, using human intelligence as base. One of the first to do so was Alan Turing, who asked himself if the computer's answer would be undistinguishable of a human's answer, then the machine or computer could be considered as intelligent. (Turing, 1950)

One of the first uses for AI was focused on games and general problem solving. At the time, a machine was considered to be intelligent if it was able to do some kind of action a human could do and if this action was difficult enough. In 1950, Claude Shannon proposed that Chess was basically

a search problem showing that brute force techniques are not practical approaches for this game because of its huge search spectrum. Heuristic searches and opening and end game databases provide a much more efficient way of playing chess. After Shannon's work on computer chess resulted in what is known as Shannon's number, 10^{120} which happens to be the inferior limit to the complexity of chess' search tree. (Shannon, 1950)

In 1952, Arthur Samuel approaches an IA that played checkers including learning and problem generalization. His program allowed copies of the software to play against each other whilst learning the one from the other. As a result, the software was able to defeat its own creator and in 1962, the Samuel's program defeated Connecticut's former checkers champion.

There were other attempts to apply AI techniques to a huge variety of games with different complexities. In this dissertation we have attempted to implement a series of AIs capable of playing the *2048* game and even beat it.

The game was developed by Gabriel Cirulli in 2014 which's goal is to fuse tiles with the same number on it, moving all tiles in the same direction at once and while summoning a new tile in a random location with every movement, until the 2048 tile is obtained or there are no more possible movements.

The approaches implemented in this dissertation must choose, for every given state of the board, the movement that leads it to the best possible scenario. These algorithms are based on different search algorithms. The empirical evaluation consists on an analysis of the achieved results in order to discover which algorithm fits problem best.

Therefore, this dissertation can be considered research project in which we do not know if the results achieved will be favorable or not. If they were favorable, there is an opportunity for using more complex methods to improve the achieved results.

The dissertation's objectives are:

- Analysis of the *2048* game
- Analysis of different techniques and algorithms used in AI to solve different board games
- Implementation of the selected algorithms.
- Definition of a set of benchmarks to analyze the performance of the different approaches used

- Development of a document that compiles the whole process.

7.2 Experimentation

In order to compare the performance of both algorithms, 90 executions have been done using each algorithm, Minimax and A*. For each of those 90 executions, half of them will be done using the score of the game as the boards evaluation function. The other half will be done using the number of empty tiles as heuristic.

In order to elaborate the statistics that are going to be shown, a file generated by the system while executing the algorithm that saves the elapsed time between turns, full execution time (in milliseconds), number of movements, the score and the result of the game.

To make it easier to understand, the first heuristic, based on the score, will be called H1 from now on, and the second one, the amount of empty tiles, H2.

This section will start talking about the results of the minimax algorithm, followed by the A*'s.

All tests have been done using the following hardware configuration:

Vendor	Apple
Model	MacBook Pro 15" 2010
Operating System	Mac OS X 10.10.5 (Yosemite)
Processor	Intel Core i5 2.53Ghz 2 cores
Memory	4 Gb

Tabla 34 System Used

First of all, I start talking about the results obtained using the Minimax. These tests have been done using both heuristics and different values of depth of the search tree, being these depths 5, 7 and 9. Because this algorithm in its second move (and then the fourth the sixth, this is every even number) corresponds to the machines turn. During the machines turn, the board will randomly place a number 2 in an empty cell. By choosing an uneven number for the depth, there will at least one more movement by the player, unless a terminal node is reached. For each depth and heuristic, we have run the program 15 times.

First of all, if we analyze the algorithm as a whole, it has achieved the 2048 tile in 20 out of the 90 test, although the amount of won games is much higher the deeper the algorithm searches.

If we observe figure 22, we can see that the percentage of won games has been 22%. However, the percentage of games which came close of winning, the ones that achieved the 1024 tile, is also significant since it represents a 58% of the games.

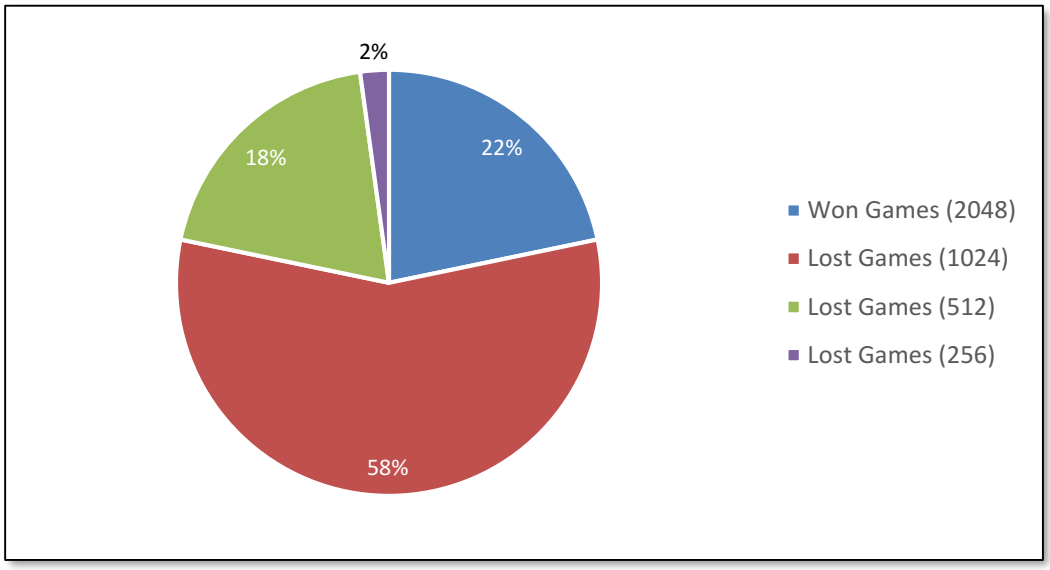

Figura 22 Game Distribution

We also know that the average length of each game is about 699.000 milliseconds, which is 11 minutes. Though the runtime increases exponentially as the maximum depth is increased. The total duration of all the test in this phase is 17 hours and 28 minutes. Table 35 shows a summary of the minimax's performance distributed by the selected depth.

Depth	*5*	*7*	*9*	***Average***
Total Time(Ms)	**3.834,8**	72.312,3	2.020.577,3	**698.908,1**
Total time Standard Deviation (Ms)	**616,5**	9.598,6	401.507,7	**137.240,9**
Nº of movements	748,7	975,3	**1.000,2**	**908**
Nº of movements Standard Deviation	235,3	172,9	**139,9**	**182,7**
Average time per turn(Ms)	**5,0**	75,0	2.036,1	**705,3**
Average time per turn Standard Deviation(Ms)	**1,1**	10,4	363,4	**124,9**
Score	11.568,7	16.073,2	**17.057,8**	**14.899,8**
Score Standard Deviation	4.736,4	4.028,1	**3.681,8**	**4.148,7**
¿Won?	6,67%	20,00%	**36,67%**	**21,11%**

Tabla 35 Statistics by Depth Minimax

Table 35 shows an improvement in the average score obtained in the games as when the maximum depth is increased. This variation is bigger specially when switching from depth 5 to 7, where the difference in points is 4.500 points and there is an increment in the won games of 13%. When switching from depth 7 to 9 the difference is also significant, since it is close to 1000 points. The most significant change is the percentage of games won which is approximately 17%.

On figure 23, which shows the average score for different depth values, we can observe that increment of the columns stabilizes as they approach to depth 9.

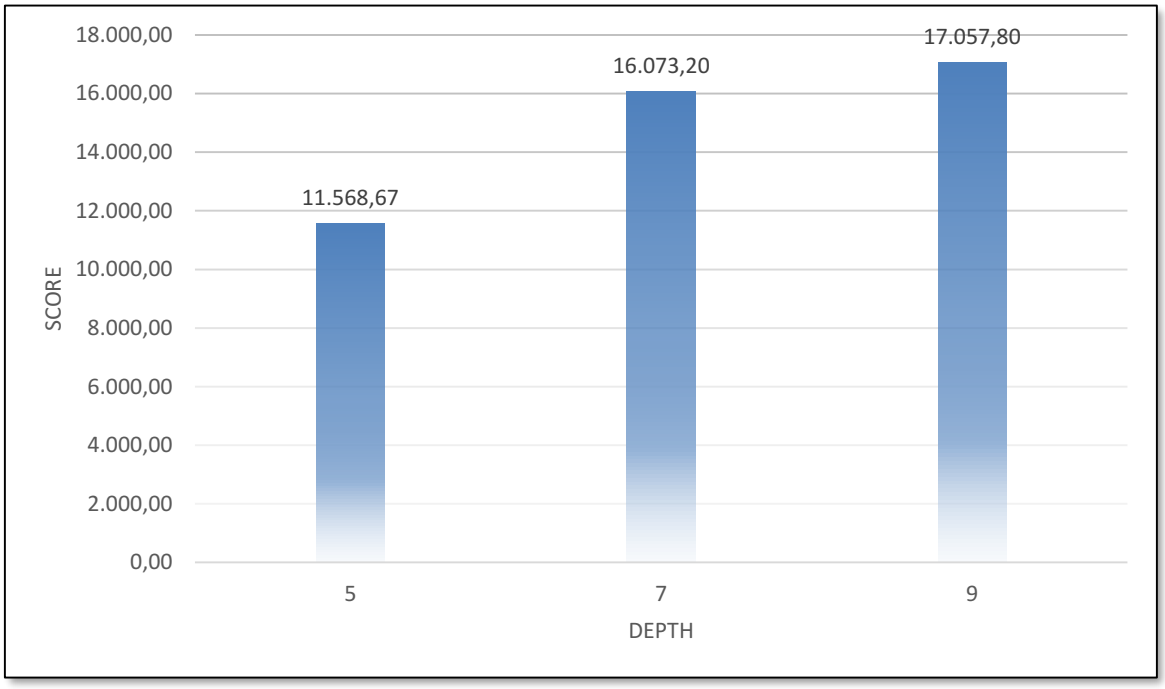

Figura 23 Average Score by Depth

As shown in the following table, the average of won games is 2814,9 points. Therefore, the average score's maximum value is limited by this value.

	Total Time	*Nº of Moves*	*Average time per turn*	*Points*
Average	**1.157.562,95**	**1.072,40**	**1.075,75**	**20.814,90**

Figura 24 Won Games Average Statistics

Another interesting conclusion is that the automatic player needs about 1072 moves to get to form a 2048 Tile.

We are going to analyze the effect of the heuristic function in the progress. Table 34 shows the statistics of the played games, distributed by depth and the heuristic used.

Heuristic	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
Depth	5		7		9	
Total Time(Ms)	**3.799,3**	3.870,3	72.864,7	71.759,9	2.171.985,6	1.869.169,0
Total time Standard Deviation (Ms)	636,8	**615,8**	11.601,2	7.456,9	480.218,8	232.804,2
Nº of movements	744,1	753,3	959,5	991,1	**1.041,7**	958,6
Nº of movements Standard Deviation	260,4	216,5	204,6	139,8	**68,7**	179,2
Average time per turn(Ms)	5,2	**4,9**	77,3	72,7	2.080,2	1.992,0
Average time per turn Standard Deviation(Ms)	1,4	**0,7**	11,7	8,7	405,1	324,5
Score	11.491,2	11.646,1	15.362,7	16.783,7	**18.487,2**	15.628,4
Score Standard Deviation	5.132,4	4.485,0	4.227,8	3.828,1	**2.686,2**	4.060,2
¿Won?	6,67%	6,67%	13,33%	26,67%	**53,33%**	20,00%

Tabla 36 Statistics by Heuristic and Depth Minimax

As we mentioned before, increasing the maximum depth does affect and the average score. While using H2 (Empty Tiles) the score increases in depth 7 compared to depth 5, but in depth 9 it decreases (compared to depth 7) in a significant amount, creating a big difference in score between H1 and H2 in depth 9, since H1's score doesn't decrease when increasing the depth.

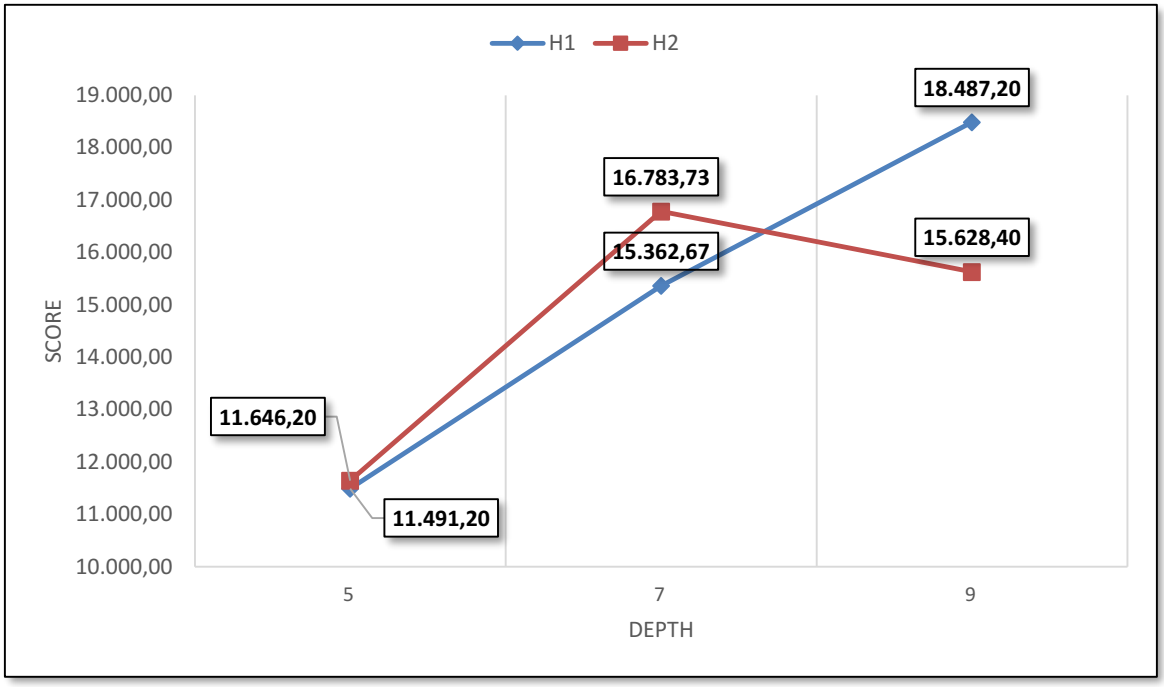

Figura 25 Average Score By Heuristic

Because achieving the 2048 tile gives a 2048-point bonus just for making this tile, plus all the extra point obtained while making the other 1024 tile needed to fuse the other 1024 tile, winning the game gives an extra 3000-4000 points. With this in mind we analyze the test we ran but taking out the games that ended up in a victory, because the wins make the results swing significantly. The following table summarizes these non-winning games.

Heuristic	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
Depth	5		7		9	
Total Time(Ms)	**3.757,1**	3.830,7	71.071,8	69.880,9	2.285.694,6	1.835.136,5
Total time Standard Deviation (Ms)	638,7	**619,0**	21.173,7	7.416,7	1.234.369,9	239.245,1
Nº of movements	721,3	732,4	946,2	940,8	**1.013,4**	926,8
Nº of movements Standard Deviation	254,2	208,4	228,0	**129,0**	133,2	175,5
Average time per turn(Ms)	5,3	**4,9**	76,8	74,5	2.237,3	2.035,0
Average time per turn Standard Deviation(Ms)	1,4	**0,7**	22,7	7,4	1.175,9	329,2
Score	10.833,4	11.008,6	14.561,8	14.619,6	**15.916,0**	14.332,7
Score Standard Deviation	4.623,7	3.885,4	4.169,2	**2.365,8**	2.433,3	3.216,2

Tabla 37 Statistics by Heuristic Non-Won Games Minimax

Although we took out the victories and the results obtained among executions using the same depth are closer, the phenomenon described previously still happens. At first glance this could be because H2 doesn't necessarily improve its results by increasing the depth, although it could also

be that the test carried out with H2 and depth 9 aren't necessarily representative of the algorithm's capacity, being the sample maybe not big enough.

The following chart exemplifies the issue described:

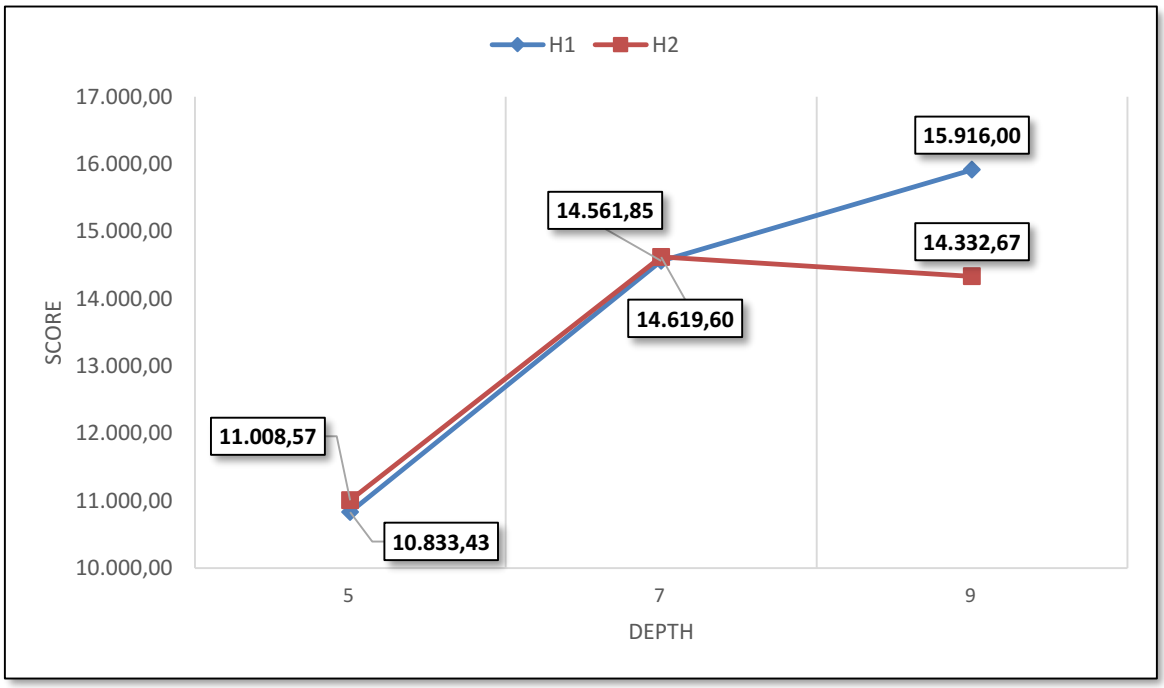

Figura 26 Average Score by heuristics (lost games)

While running the tests it was observed that the first moves required much more computation time than the end game's moves. This was much more apparent in depths 7 and 9. This can be explained because since the board is empty at the beginning of the game, not as much as for the player but for the machine's turn, there is a wider range of possible moves for the machine, this is, more empty tile to put a random two after each move, expanding the search tree in a great measure.

In graphs 27,28 and 29 we can see this explained for each depth and heuristic in games that ended with in a victory.

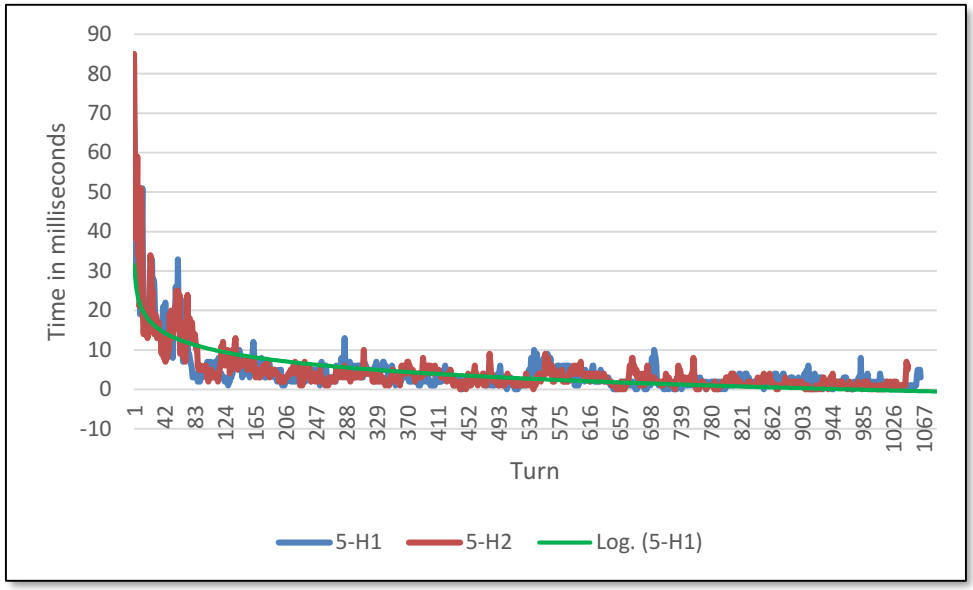

Figura 27 Time per Number of Move Depth 5

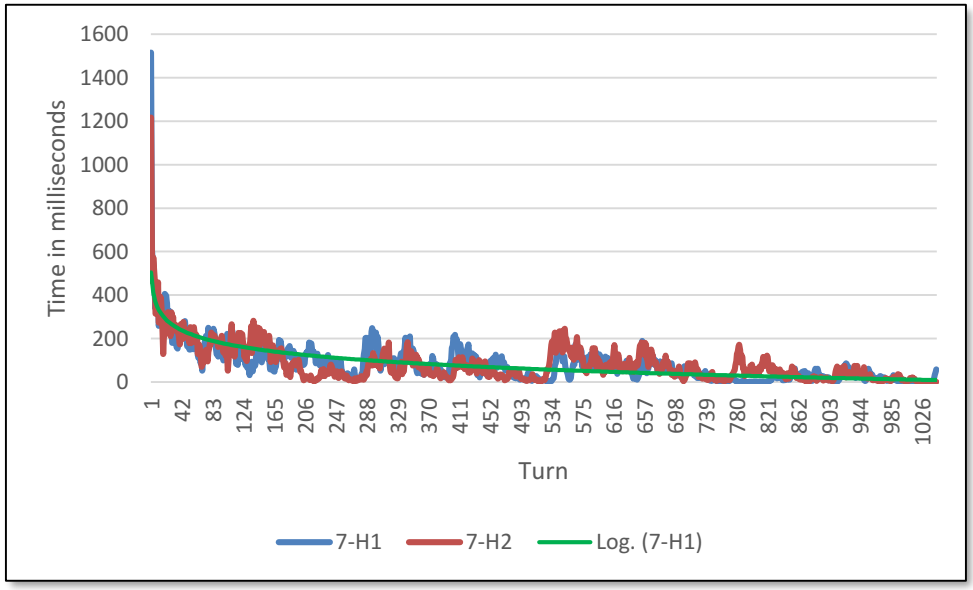

Figura 28 Time per Number of Move Depth 7

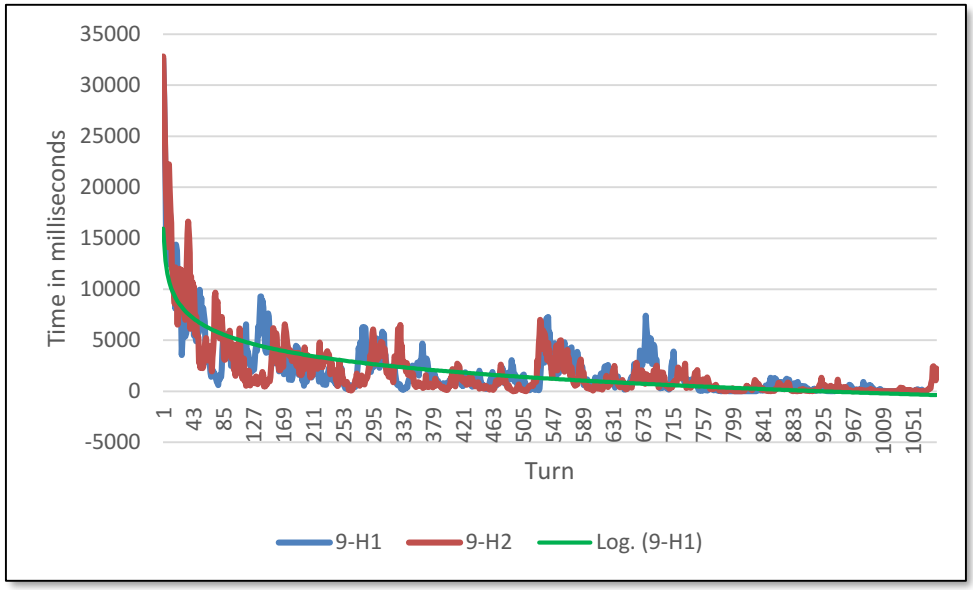

Figura 29 Time per Number of Move Depth 9

After completing the minimax tests, it was time to start with the A* algorithm's tests. Just like with minimax there have been 15 tests for each of the heuristics, both H1 and H2, for three different depths. Unlike minimax, the A* algorithm doesn't take into account a possible move made by the machine, therefore it doesn't considers the empty tiles where a 2-tile can be inserted after every move. For these test we have used the following depths: 5, 8 and 10

First of all, we are going to analyze the data as a whole. Out of the 90 games played, none of them ended in a victory, in fact, none of the games can be considered to have come even close to winning. In table 30 we can see a summary of the game play statistics. The scores aren't too high. Since there have been no won games the winning percentage column has been omitted.

Depth	*5*	*8*	*10*	***Average***
Total Time(Ms)	**264,9**	1.253,4	5.793,1	**2.437,1**
Total time Standard Deviation (Ms)	**98,6**	729,5	5.003,6	**1.943,9**
Nº of movements	64,2	**76,4**	74,0	**71,5**
Nº of movements Standard Deviation	14,7	22,0	**27,1**	**21,3**
Average time per turn(Ms)	3,7	16,7	**78,8**	**33,1**
Average time per turn Standard Deviation(Ms)	**1,4**	10,0	55,9	**22,4**
Score	355,7	**487,5**	455,9	**433,0**
Score Standard Deviation	**140,8**	214,4	270,3	**208,5**

Figura 30 Statistics by Depth A*

It is easily observable that as we increase the depth search of the algorithm there isn't a very significant increase in the average score per game. The execution does increase significantly with the depth increase, approximately 5 times per every increase. However, since there isn't a significant increase in the number of moves this is explained because of increasing the depth search makes the algorithm take longer.

After passing from the first depth to the second one, there is an increase of the obtained score, but after increasing it again the growth stabilizes, in fact it decreases, as shown in figure 31, and unlike the minimax, it is not because of the limitation of the winning score.

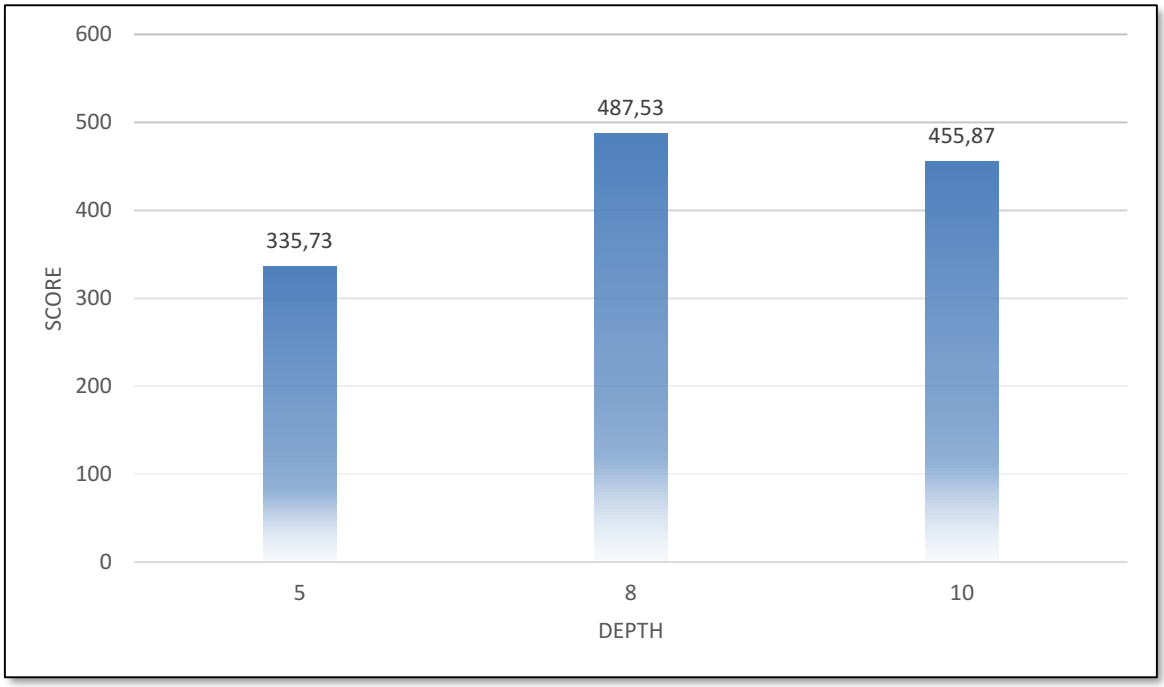

Figura 31 Average Score by Depth A*

Now we take a look at the performance but looking at how the heuristic affects the performance. Table 38 shows the statistics distinguishing between heuristics.

Heuristic	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
Depth	5		8		10	
Total Time(Ms)	**181,7**	348,1	598,9	1.907,9	1.848,3	9.738,0
Total time Standard Deviation (Ms)	**36,7**	63,1	212,7	373,1	571,3	4.264,7
Nº of movements	65,0	63,4	75,8	77,0	69,7	**78,3**
Nº of movements Standard Deviation	17,6	**11,8**	21,9	22,8	23,3	30,6
Average time per turn(Ms)	**2,5**	4,9	7,7	25,7	28,1	129,6
Average time per turn Standard Deviation(Ms)	**0,6**	0,9	2,3	5,2	11,3	28,6
Score	361,1	350,4	**493,1**	482,0	425,2	486,5
Score Standard Deviation	168,5	**112,4**	209,8	226,2	252,6	292,5

Tabla 38 Statistics by Heuristic and Depth A*

H1 is the heuristic that obtains the lowest score on depth 10 while H2 is able to keep its score up, although it also decreases. It is also observed that H2 has a much longer time of execution compared to H1 but without better scores, therefore, at least in time of execution, H2 heuristic isn't the most appropriate for the A* algorithm.

Figure 32 describe how both heuristics have similar behaviors in both depth 5 and 8, but afterwards H1 plunges its average score while H1 stops growing.

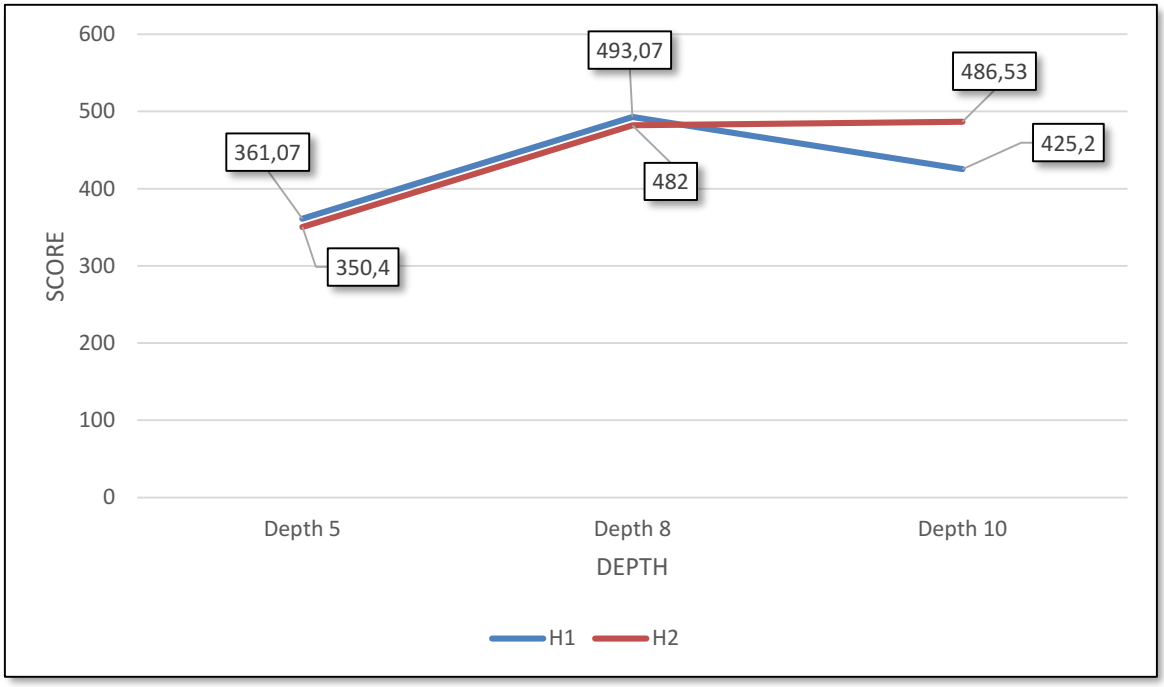

Figura 32 Average Score Evolution by Heuristic A*

Since the results achieved were not the ones expected, some alterations to the heuristics were made.

Since the A* algorithm chooses the nodes from the open list based on the result of the equation $f(n) = g(n) + h(n)$ with *g(n)* being the cost to get to the node and *h(n)* the value of the heuristic, which should be an estimation of how much it is going to take to get to the desired objective, the changes have been the following:

- For H1, the result of evaluating the board will be the difference between the board's score and 20.000 points, which is an estimation of the score needed to win the game.
- For H2, the procedure was similar, only that in this case the difference will be between the total amount of empty tiles and 16, which is the total amount of tiles in the board. Therefore, the ideal return of the heuristic is 0, this is, and all the tiles empty, but this is impossible because of the random tiled filled after each move.

10 tests have been carried out with each of the heuristics after implementing the modifications and the results obtained are in the following table.

Heuristic	*H1*	*H2*
Depth	10	10
Total Time(Ms)	**13.871,8**	318.563,4
Total time Standard Deviation (Ms)	**4.709,8**	170.045,9
Nº of movements	**449,6**	259,9
Nº of movements Standard Deviation	**187,5**	178,3
Average time per turn(Ms)	**31,5**	1.368,9
Average time per turn Standard Deviation(Ms)	**15,5**	739,2
Score	**6.296,8**	4.532,4
Score Standard Deviation	2.965,0	**2.536,2**

Tabla 39 H1 and H2 V2 Statistics

The results achieved in these new tests are significantly better than the old test with the same depth. It is also important to highlight that the execution times are higher in both heuristics, specially using version 2 of H2.

7.3 Conclusions

This chapter's section serves as a summary of the conclusions achieved during the realization of this project. First, some general conclusions will be discussed. Secondly the conclusions regarding the initial objectives of this document will be presented. Finally, this section will end with a description of possible future improvements to the system developed during this project.

7.3.1 General Conclusions

First of all, the development of this project has helped to understand how to implement complete software project by defining the functionalities, a user story diagram, definition of the initial requirements, designing the solution, implementing the system and finally testing the different run configurations, as well as documenting all the process.

Secondly, after investigating several artificial intelligence techniques, it was discovered that board games had been subject of study from the very first investigations in the field, since they are fairly simple to model.

Other than the knowledge acquired during the project, it has been considered to be interesting to carry out, even amusing at some times, although there have been moments of frustration, such as trying to get the GUI to refresh every time the automatic player made a move. This was solved, as explained in section 3.3.3.2 by running the GUI in the main thread and the rest of the system in a secondary thread.

7.3.2 Conclusions Regarding the Objectives

For this section the objectives set at the beginning of the project will be listed and discussed:

- Analysis of the *2048* game: Fulfilling this goal was one of the first milestones achieved during this project, since it was necessary in order to continue with the rest of the project. Finally, as exposed in section 3.3.2 it was decided to use some preexisting code, developed by Konstantin Bulenkov., to use as the projects base.
- Analysis of the different algorithms used in the for using artificial intelligence in board games. After studying *2048* game and its characteristics, other works in board games were analyzed as well as studying how these could be adapted to our problem.
- Implementation of the selected algorithms: This objective was achieved as described in section 3.3.3.3 implementing minimax and the A*.
- Definition of series of experiments: As explained in chapter 4, a series of experiments were carried out in order to compare both algorithms' performance.
- Development of a document that compiles the whole process.

7.3.3 Future Improvements

Once the project has concluded and after analyzing the achieved results, some improvements to apply in the future have been thought of.

At the higher level it would be interesting to apply machine learning techniques. To do so, an important number of experiments must be carried out, compiling and storing information during them to afterwards apply supervised and unsupervised learning algorithms to learn behavior models that will allow the system to make decisions.

Since the heuristics used in this project could be considered rather simple, it would be interesting to use a more complex functions, like one that combines both the ones used, score and empty tiles, or a function that takes into account where the high numbered tiles are placed.

For the A* algorithm it would be interesting to change the successor generating sequence so that it bears in mind the random tile placed after every move, therefore providing a more complete search space.

Finally, for the minimax algorithm it would be interesting to do two things. The first of them, applying alpha-beta pruning so that the algorithm can prune non-promising branches of the search

tree and therefore find a solution quicker. The second thing would be to remove the 2048 limit, and see for how long it can continue playing and what score does it achieve.

Capítulo 8: Anexos

8.1 Manual de instalación

Debido a que el sistema se ha diseñado para ser ejecutado a partir de un archivo .jar, no hace falta realizar una instalación del sistema como tal. Sin embargo, para poder ejecutarlo, el usuario debe tener una versión de JAVA instalada en su equipo.

Para ello hay que visitar la Página Web Oficial de Java y en función del equipo donde se desee realizar la instalación elegir la opción adecuada. La siguiente figura muestra la página de descargas de Java de la web Oficial.

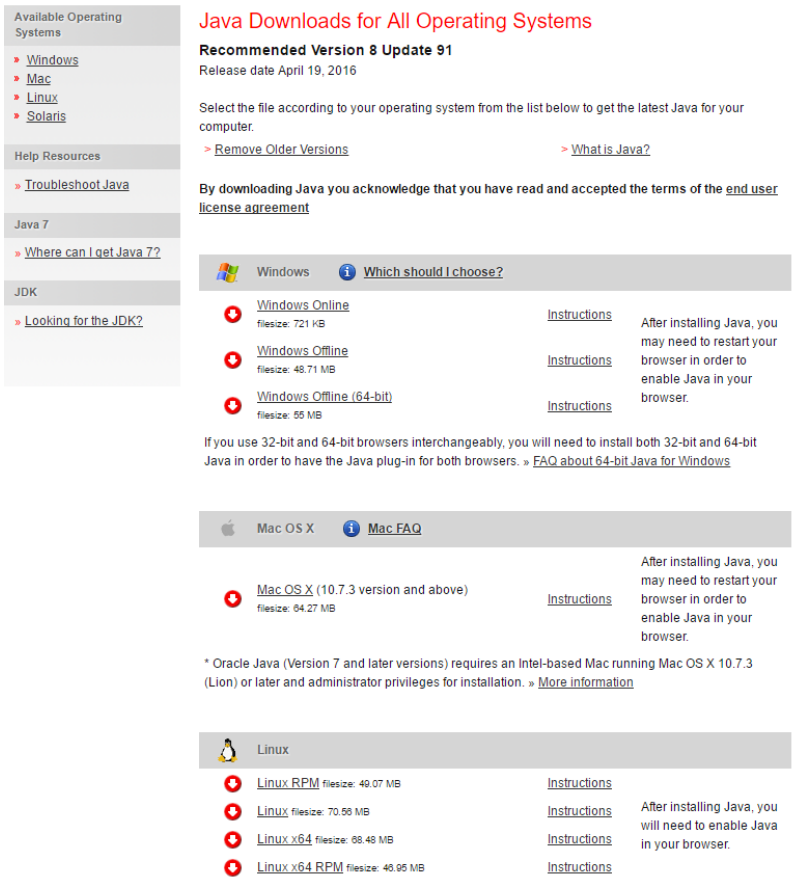

Figura 33 Página Web de Oracle

Al seleccionar la versión elegida, se descargará automáticamente el instalador de java para sistema operativo elegido. Se deberá proceder a ejecutar dicho instalador y debería salir algo similar a lo mostrado en la siguiente imagen.

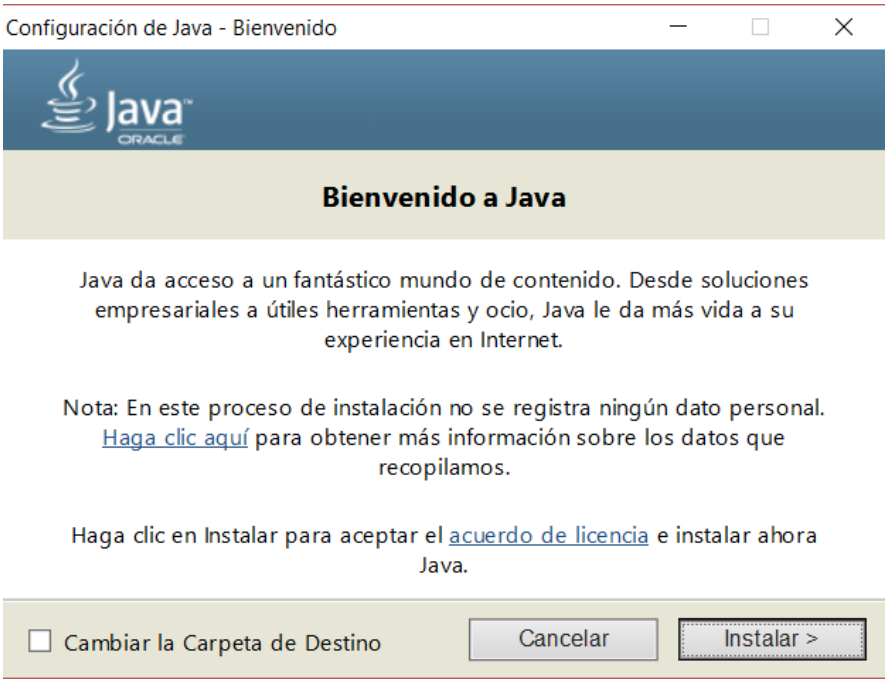

Figura 34 Instalación Java 1

En este punto, debemos hacer clic en instalar. Una vez hecho esto se mostrará una nueva ventana que muestra el progreso de la instalación, como la que se muestra a continuación.

Figura 35 Instalación Java 2

Finalmente, cuando se haya completado la instalación saldrá una pantalla como la que se muestra a continuación. Se deberá hacer clic en el botón "Cerrar" y la instalación habrá terminado.

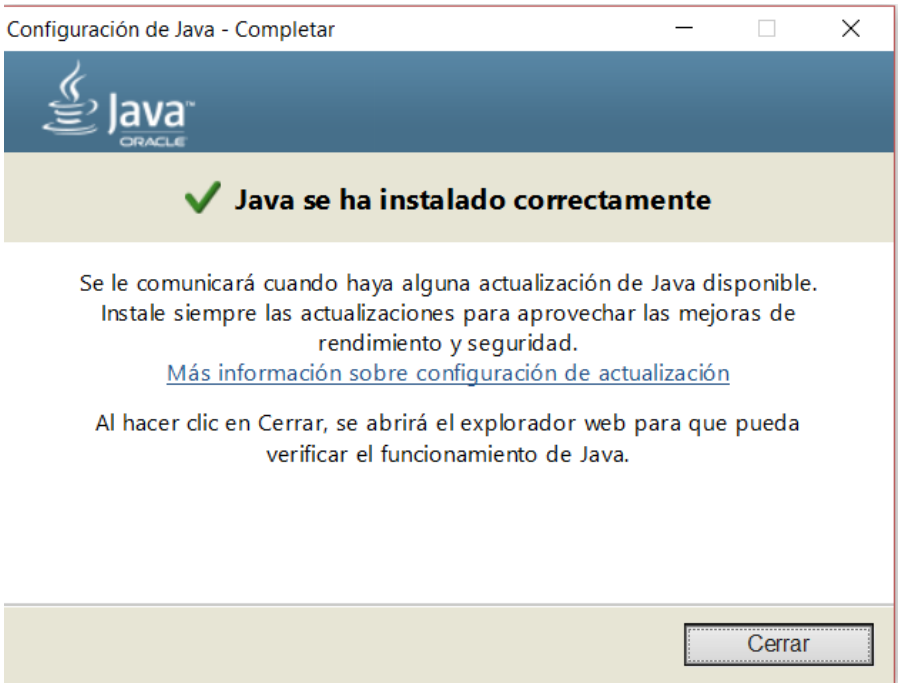

Figura 36 Instalación Java 3

8.2 Manual de usuario

Para poder usar el sistema hace falta dos elementos, el archivo *2048_ISL.jar* y tener Java instalado en el equipo. Si no se tiene Java, ver apartado 8.1.

Existen dos formas de ejecutar el sistema:

- Haciendo Doble clic en el fichero. De este modo se podrá acceder a las dos funcionalidades del sistema, el jugador manual o el jugador automático, con una limitación en este último caso, ya que, al no introducir parámetros de entrada, se ejecutará directamente para que use el minimax, usando como heurística la puntuación y una profundidad de 5. En el caso de querer usar una configuración distinta es necesario ejecutar el sistema desde la terminal (si se usa Linux o OS X) o el cmd (para Windows). En el caso del jugador manual, mediante este modo de ejecución no existe limitación alguna.
- Mediante la terminal (Linux o OS X) o el cmd (Windows). Los pasos para este modo son los siguientes.

1. Debemos colocar el terminal o el cmd en el mismo directorio en el que se encuentra el archivo. Una vez ahí se ejecuta el siguiente comando:

   Java –jar 2048_ISL.jar [-algoritmo] [–heurística] [–profundidad]

   Para poder configurar los parámetros de ejecución debemos especificar los argumentos. Para cada uno de ellos las opciones son:

   -*algoritmo*: Para usar minimax se introduce un 0 y para A* se introduce un 1.

   -*heurística*: Para utilizar como heurística la puntuación, introducir 0. Para usar como heurística el número de casillas vacías, introducir un 1.

   -*profundidad*: Para elegir la profundidad del algoritmo, introducir un número entero con el valor deseado.

   Si no se introduce ningún argumento, se ejecutará con la configuración por defecto. A continuación, se muestra un ejemplo de cómo ejecutar en terminal.

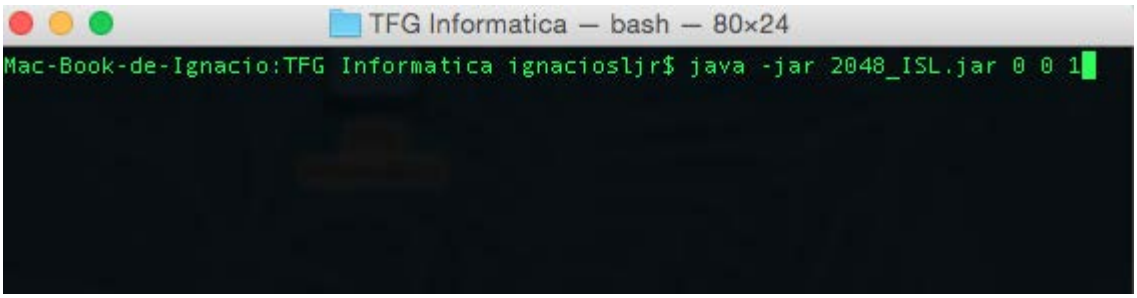

Figura 37 Comando Para Ejecutar en Terminal

2. Una vez ejecutado saldrá una nueva ventana dando a elegir entre usar el jugador automático, con la configuración introducida, o el jugador manual. Para el jugador automático, deberá hacerse clic en “Jugar Automático”. A continuación, se muestra una imagen del menú inicial.

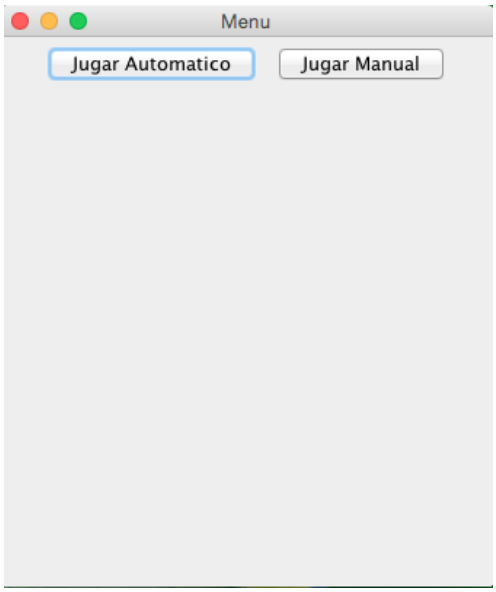

Figura 38 Menu del Sistema

3. Si ha elegido el jugador automático saldrá una ventana con el tablero inicial. Para que dé comienzo habrá que cualquier tecla, excepto *ESC*, y comenzará a ejecutarse el sistema, pudiendo seguir el usuario la partida. Si se ha elegido el jugador manual, saldrá el tablero inicial, y el usuario podrá jugar usando las teclas de dirección de su teclado o reiniciar la partida pulsando la tecla *ESC*. A continuación, se muestra una imagen de una partida en ejecución.

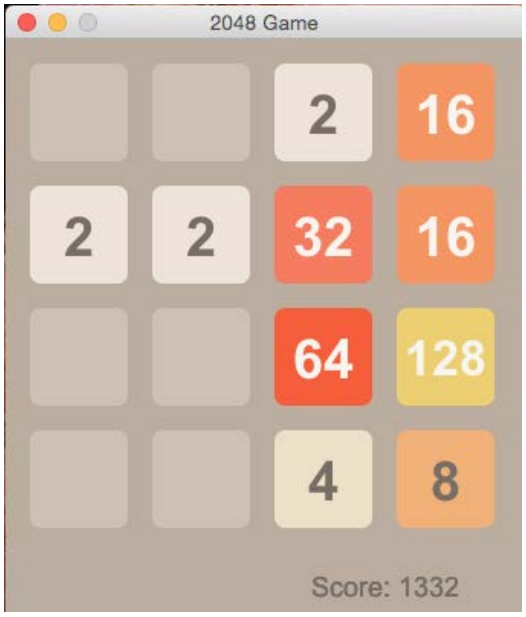

Figura 39 Partida en Ejecución

4. La partida seguirá, bien automáticamente o bien de manera manual según el modo elegido hasta que la partida se pierda o se gane, en la que saldrán, según corresponda una de las siguientes dos pantallas, respectivamente.

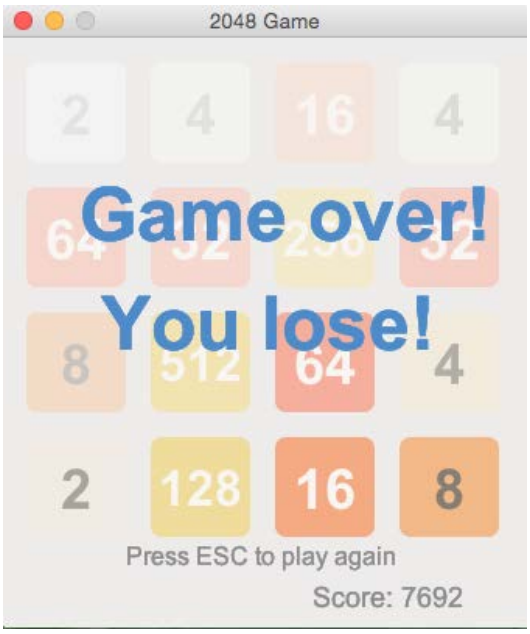

Figura 40 Partida Perdida

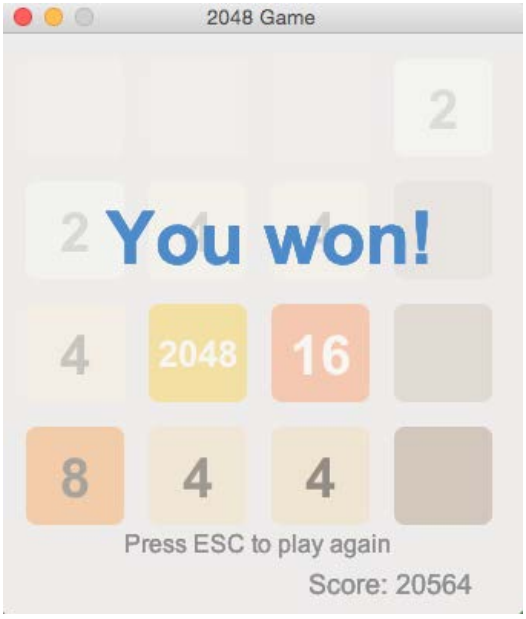

Figura 41 Partida Ganada

5. Para finalizar la ejecución, basta con cerrar la ventana. Para ver los resultados y estadísticas de la ejecución se deberá abrir el fichero “ejecución.txt”

8.3 Diagramas de Clase

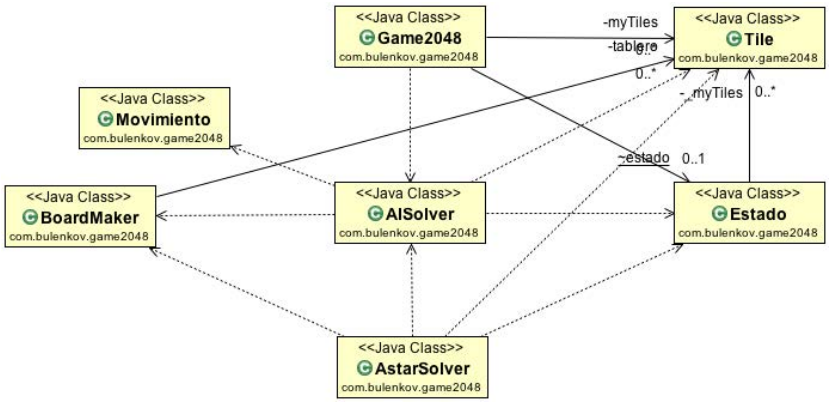

Figura 42 Diagrama de Clase Paquete Principal

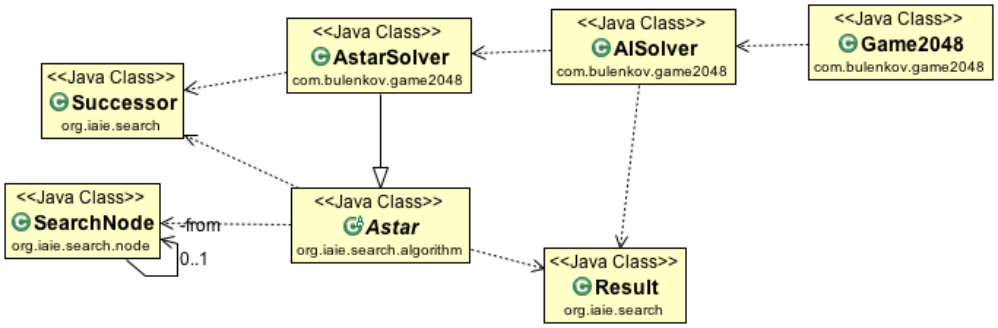

Figura 43 Diagrama de Clases del Paquete A*

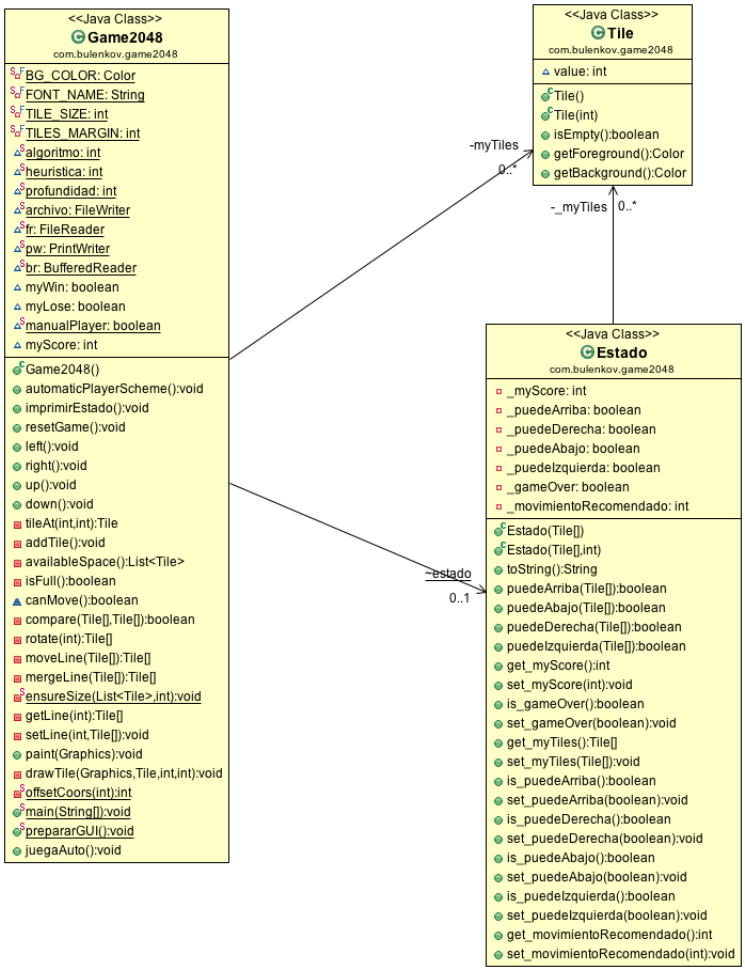

Figura 8.11 Diagrama de clases: Jugador Manual

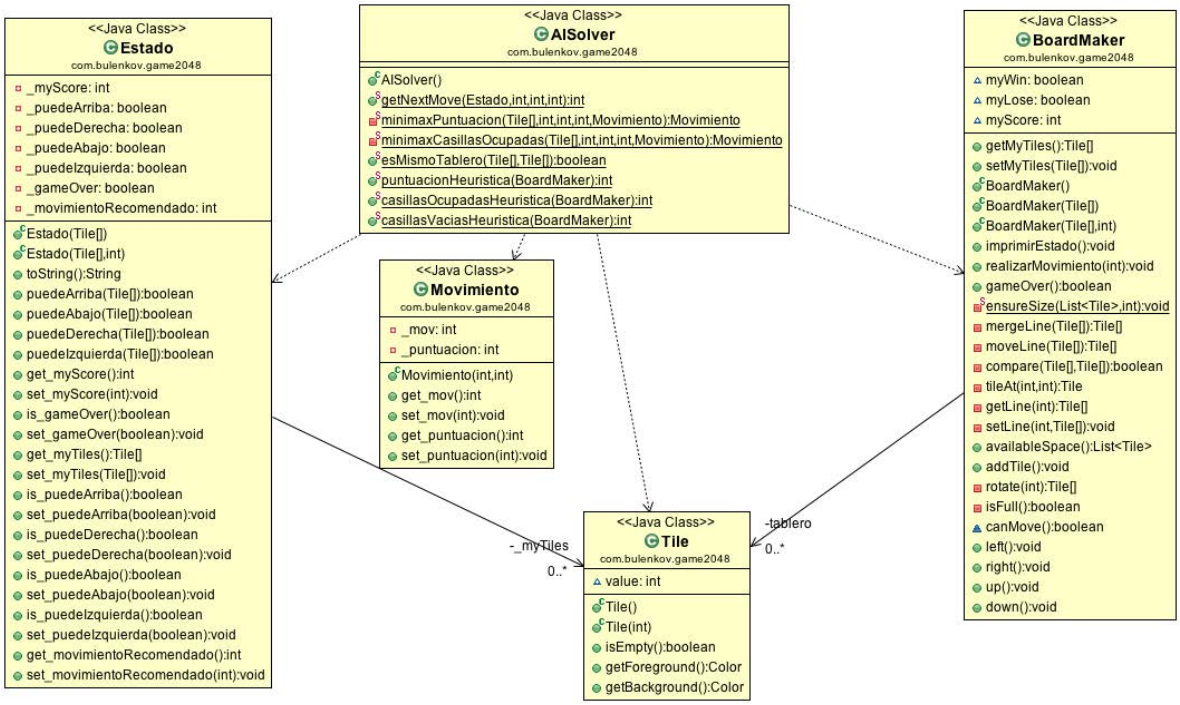

Figura 44 Diagrama de Clases Jugador Automático Minimax

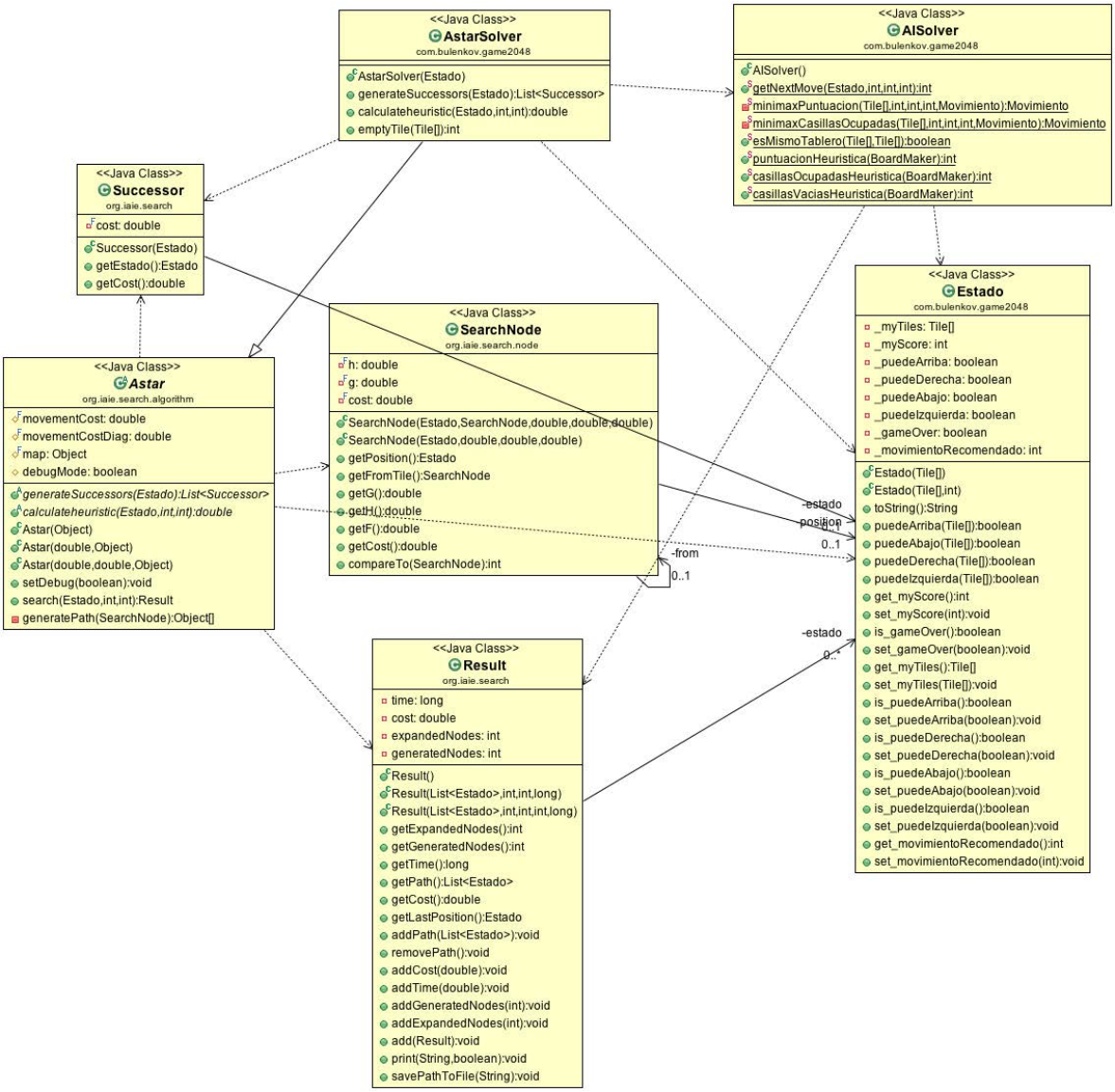

Figura 45 Diagrama de Clases Jugador Automatico A*

Bibliografía

1. Breuker, D. M., 1998. *Memory Versus Search in Games,* s.l.: Maastricht University.

2. Bulenkov, K., 2014. *bulenkov/2048 - Github Repository.* [En línea]
   Available at: https://github.com/bulenkov/2048/branches
   [Último acceso: 10 2015].

3. erikiado, 2015. *Método de Montecarlo.* [En línea]
   Available at: https://inteligenciaartificial101.wordpress.com/2015/02/14/metodo-de-montecarlo/
   [Último acceso: 19 06 2016].

4. Fogell, D. B., 2002. *Blondie24:Playing at the edge of AI.* s.l.:Morgan Kauffman Publishers.

5. González, C. G., s.f. *Busqueda Heurística.* [En línea]
   Available at: http://www.monografias.com/trabajos75/busqueda-heuristica/busqueda-heuristica.shtml
   [Último acceso: 17 06 2016].

6. Hart, P. E., Nilsson, N. J. & Raphael, B., 1968. A Formal Basis for the Heuristic Determitacion of Minimun Cost Paths. *IEEE Transactions on Systems Science and Cybernetics,* IV(2), pp. 100-107.

7. Jones, M. T., 2008. *Artificial Intelligence A Systems Approach.* s.l.:Jones & Bartlett Publishers.

8. McCarthy, J., 2007. *Basic Questions.* [En línea]
   Available at: http://www-formal.stanford.edu/jmc/whatisai/node1.html
   [Último acceso: 15 06 2016].

9. Michie, D., 1966. Game-playing and game-learning automata. *Adavances in Programming and Non-Numerical Computation,* pp. 183-200.

10. Neumann, J. v., 1928. Zur Theorie der Gessellschaftsspiele. *Mathematische Annalen.*

11. Newborn, M., 2002. *Deep Blue.* s.l.:Springer.

12. Newell, A., Simon, H. A. & Shaw, J. C., 1958. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development,* Issue 4, pp. 320-355.

13. Nilsson, N. J., 2009. *The Quest for Artificial Intelligence.* s.l.:Cambridge University Press.

14. Oellerman, A., 2006. *Tree Search.* [En línea]
Available at: http://www.oellermann.com/node/70
[Último acceso: 16 06 2016].

15. Overlan, M., 2014. *Mini-max ai.* [En línea]
Available at: http://ovolve.github.io/2048-AI/

16. Rodgers, P. & Levine, J., 2014. *An Investigation into 2048 AI Strategies,* Glasgow: s.n.

17. Russel, S. & Norvig, P., 2009. *Artifical Intelligence: A Modern Approach.* 3rd ed. s.l.:Prentice-Hall.

18. Schaeffer, J., 2009. *One Jump Ahead: Computer Perfection at Checkers.* s.l.:Springer-Verlag.

19. Shannon, C., 1950. Programming a Computer for Playing Chess. *Philosophical Magazine,* Issue 41.

20. Shaw, J. C., Newell, A. & Shaw, H. A., 1959. Report on A General Problem-Solving Program. *Proceedings of the International Conference on Information Processing,* pp. 256-264.

21. Turing, A., 1950. Computing Machinery and Intelligence.

22. Turing, A., 1953. *The Turing Digital Archive.* [En línea]
Available at: www.turingarchive.org/browse.php/B/7
[Último acceso: 18 06 2016].

23. Universidad Polítecnica de Cataluña, s.f. *Busqueda Heurística.* [En línea]
Available at: http://www.cs.upc.edu/~bejar/ia/transpas/teoria/2-BH2-Busqueda_heuristica.pdf
[Último acceso: 17 06 2016].

24. Xiao, R., s.f. *Expectimax ai.* [En línea]
Available at: https://github.com/nneonneo/2048-ai