



UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

PROYECTO FIN DE CARRERA

**DESARROLLO DE UN JUGADOR DE OMWESO
MEDIANTE TÉCNICAS DE INTELIGENCIA ARTIFICIAL**

Autor: José David Torres Montiel

Tutor: Carlos Linares López

Tabla de contenidos

1.Introducción.....	9
2.Estado de la cuestión.....	13
2.1.Origenes del Mancala.....	13
2.2.Características comunes a todos los juegos Mancala.....	14
2.3.La gran familia Mancala. Variantes.....	15
2.4.Interés del estudio de la familia Mancala.....	20
2.5.Un caso particular: Omweso.....	21
2.5.1.Características comunes a la familia Mancala.....	21
2.5.2.Reglas de juego de Omweso.....	22
2.5.3.Características de Omweso en el marco de la IA.....	24
2.5.4.Características que lo hacen interesante para su estudio.....	26
2.6.Estado del estudio del Mancala.....	28
2.7.Algoritmos de Búsqueda.....	30
2.7.1. Algoritmos de búsqueda : conceptos generales.....	31
2.7.2. Algoritmos de búsqueda : Minimax.....	34
2.7.3. Algoritmos de búsqueda : Negamax.....	37
2.7.4. Algoritmos de búsqueda : Alfabeta.....	39
2.7.5. Algoritmos de búsqueda : Scout.....	42
3.Objetivos.....	47
4. Desarrollo.....	51
4.1. Diagrama de casos de uso.....	53
4.1.1.Gestión de partidas.....	54
4.1.2.Caso de uso : Mover. Seleccionar celda.....	57
4.1.3.Caso de uso: Enlazar movimiento.....	58
4.1.4.Caso de uso: Capturar fichas.....	59
4.1.5.Caso de uso: Reentrada de capturas.....	59
4.1.6.Caso de uso: Captura inversa.....	60
4.1.7.Caso de uso: Mostrar tablero.....	61
4.1.8. Gestionar IA.....	62
4.2.Diagrama de clases.....	67

4.4. Búsqueda de la función de evaluación.....	70
4.3.1. Evaluación de la diferencia de material.....	72
4.3.2. Evaluación de la dispersión del material.....	73
4.3.3. Evaluación de la movilidad.....	77
4.3.4. Evaluación del alcance.....	78
4.4.5. Cálculo de los pesos de cada una de las características.....	79
4.4. Entorno de desarrollo.....	80
4.4.1. Sistema Operativo: Linux/Ubuntu.....	80
4.4.2. Análisis y diseño UML: ArgoUML.....	81
4.4.3. Lenguaje de programación: C++/SDL.....	82
4.1.4. Herramientas secundarias: Gimp, Kdevelop, doxygen.	82
4.5. Manual de Usuario.....	84
4.5.1. Requisitos de sistema.....	84
4.5.2. Instalación del programa.....	85
4.5.2.1. Instalación de la librería GNU Readline.....	85
4.5.2.2. Instalación de las librerías SDL, SDL_image, SDL_ttf.....	87
4.5.2.3. Instalación del programa Omweso.....	89
4.5.3. Ejecución de Omweso.....	91
4.5.4. Gestión de partidas.....	92
4.5.4.1. Crear una partida nueva.....	94
4.5.4.2. Configurar las opciones de la partida.....	95
4.5.4.3. Configuración de opciones de IA de los jugadores.....	98
4.5.4.4. Editar los contenidos del tablero.....	101
4.5.4.5. Guardar una partida a disco.....	103
4.5.4.6. Cargar una partida desde disco.....	104
4.5.4.7. Finalizar una partida prematuramente.....	107
4.5.5. Jugar una partida de Omweso.....	107
4.5.5.1. Visualizar el tablero.....	109
4.5.5.2. Visualizar el tablero gráfico.....	110
4.5.5.3. Realizar un movimiento durante un turno.....	110
4.5.5.4. Realizar una captura inversa.....	111
4.5.6. Herramientas para el investigador de IA.....	111

4.5.6.1.Crear un lote de partidas automatizadas.....	112
4.5.6.2.Registrar los resultados de las partidas en un fichero.....	113
4.5.6.3.Registrar los movimientos de las partidas en un fichero.....	115
4.5.6.4.Cambiar el algoritmo de búsqueda.....	117
4.5.6.5.La función de evaluación.....	118
5.Resultados.....	121
5.1.Rendimiento de los algoritmos de búsqueda.....	121
5.2.Posesión del material durante una partida.....	123
5.3.Duración de una partida. Cantidad de turnos y jugadas.....	129
5.4.Análisis de la función de evaluación.....	131
5.4.1.Rendimiento individual de las características.....	131
5.4.2.Combinación lineal de las características.....	133
5.4.3.Eliminación de algunas características.....	140
5.5.Resultados contra jugadores humanos.....	144
6.Conclusiones.....	147
6.1.Repaso de los objetivos.....	147
6.1.1.El tablero.....	147
6.1.2.El interfaz.....	147
6.1.3.Los diferentes agentes de IA.....	148
6.1.4.Lotes de partidas y log de resultados.....	149
6.1.5.Objetivos de IA.....	149
6.2.Implementación del sistema.....	150
6.2.1.Un sistema portable.....	150
6.2.2.Un sistema ampliable.....	151
6.2.3.Un sistema eficiente y rápido.....	151
6.3.Conclusiones desprendidas de los resultados.....	151
6.3.1.Rendimiento de los algoritmos de búsqueda.....	152
6.3.2.Las funciones de evaluación.....	153
6.3.3.El movimiento de siembra y la reentrada de semillas.....	154
7.Líneas futuras.....	157
7.1.Mejoras a la implementación del sistema.....	157
7.2.Mejoras a la IA del sistema.....	158

8. Bibliografía.....	161
Apéndices.....	165
Apéndice A.1. Pseudocódigo del Minimax.....	165
Apéndice A.2. Pseudocódigo del Negamax.....	166
Apéndice A.3. Pseudocódigo del Alfabeta.....	167
Apéndice A.4. Pseudocódigo del Scout.....	168

Capítulo 1

Introducción

1.Introducción.

La Inteligencia Artificial (IA) como objeto de estudio nació sobre los años 50, concretamente en el año 1956, durante la conferencia de Dartmouth, organizada por John MacCarthy, considerado padre de la IA. Fue la primera vez que se acuñó el concepto de *Inteligencia Artificial*. Desde entonces el campo de la IA ha ido avanzando gracias a las teorías y principios desarrollados por investigadores, y aunque la evolución en su historia más reciente ha sido más lenta de lo que se estimó en un principio, aún hoy día, se siguen haciendo avances en este campo.

El objetivo de la IA es intentar comprender el razonamiento humano, intentar representarlo de alguna forma que pueda ser trasladado a una computadora, y de esta forma conseguir que una computadora tome decisiones inteligentes.

Estas computadoras inteligentes, serían capaces de realizar tareas que hasta ahora requerirían la intervención de un ser humano, o incluso servirían para reducir exponencialmente el tiempo necesario para resolver problemas imposibles de resolver, en un tiempo aceptable, por un humano. La máquina no se cansa, la maquina no se frustra, y puede tomar decisiones exponencialmente mas rápido que cualquier humano.

Pero la representación del razonamiento humano puede llegar a ser algo tan abstracto, que se hace necesario algún medio más empírico para materializar de alguna manera más tangible todas las decisiones que puede llegar a tomar un humano para resolver un problema.

Curiosamente, el propio ser humano lleva siglos, desde muchos años antes de que el concepto IA existiera siquiera, creando y jugando a juegos. Juegos de lógica, juegos de tablero, juegos de estrategia, y muchos otros, principalmente con la finalidad de entretenerse a la vez que suponen un reto para su intelecto.

Gracias a esta iniciativa innata del ser humano de proponerse a sí mismo retos en forma de juegos de entrenamiento, tenemos una fuente muy interesante de estudio de la toma de decisiones humanas, pues precisamente la finalidad de estos juegos es tomar las decisiones correctas para resolver el juego o vencer a tu adversario en la partida.

A la hora de aplicar este conocimiento a la resolución de problemas reales, en algunas ocasiones se puede encontrar un juego que se aproxime al problema real, pero en muchas ocasiones no es así. Pero aún en estos casos, una forma fácil de solucionar el problema real es crear un juego que se adapte perfectamente al problema, y de esta forma tendremos una forma computacionalmente manejable de solucionar el problema.

Pero aunque los juegos se puedan crear y/o aprovechar para resolver problemas reales, no podemos dejar de mencionar, que el ser humano, buscador de retos por naturaleza, utiliza la propia IA para retroalimentar a los propios juegos, creando juegos con la única finalidad de entretener, pero con adversarios automatizados inteligentes capaces de suponer el tan buscado reto. Alrededor de esto podemos encontrar toda una industria que va desde juegos de tablero computerizados, hasta videojuegos de todo tipo, simuladores de vida, etc.

Capítulo 2

Estado de la cuestión

2.Estado de la cuestión.

En este capítulo intentaremos explicar la situación actual que rodea a los aspectos que trata el proyecto, estudios que se han hecho al respecto, o características del Omweso que lo hacen digno de estudio.

2.1.Origenes del Mancala.

El nombre Mancala viene de la palabra arabe “naqala”, que literalmente significa mover¹. Este hecho nos puede hacer pensar que el origen del Mancala es Arabia, y ciertamente se han encontrado tableros muy antiguos en esta zona y en la zona de Egipto, pero no se puede confirmar debido a la escasez de restos arqueológicos.

Sin embargo, lo que sí podemos decir es que se ha jugado Mancala extensamente desde hace mucho tiempo en el continente africano, por lo que se considera a África como la cuna del Mancala. Posteriormente, desde África se extendió, debido en gran medida al movimiento de esclavos, hasta el Caribe, y la costa Este de Sur América. En Norte América tuvo menos incursión debido a la opresión que existió sobre los esclavos con la intención de desarraigar sus tradiciones africanas.

El Mancala se extendió tanto, por tantos lugares, tantos países y tantas culturas, que el propio juego fue evolucionando dependiendo del lugar donde se jugara, dando lugar a un espectro tan amplio de juegos diferentes pero a la vez similares, que en muchas ocasiones se utiliza el término “familia Mancala” para referirse a todos ellos.

Las variantes generadas difieren desde modificaciones a las reglas, modificaciones a la disposición y composición del tablero, o incluso restricciones culturales, de modo que hay países, como Siria o Egipto, donde no pueden jugar hombres contra mujeres, o países, como Filipinas, donde solamente juegan las mujeres².

1 Mancala - Wikipedia, the free encyclopedia

2 H.J.R. Murray; A History of Board-Games Other Than Chess; Oxford: The Clarendon Press, 1952, Chapter 7, pages 158-159

Veremos que la existencia de todas estas variantes abrirá un gran abanico de posibilidades de estudio, pues aunque básicamente tienen muchos aspectos comunes, las diferencias harán que su complejidad e interés de estudio varíen entre los muchos juegos de la familia Mancala.

2.2.Características comunes a todos los juegos Mancala.

Cuando se habla de familia de juegos Mancala nos referimos a una serie, bastante numerosa, de juegos caracterizados por jugarse sobre un tablero formado por hileras de hoyos o cubículos. En los cubículos inicialmente se colocan una serie de fichas (semillas) que cada uno de los dos jugadores adversarios irá moviendo por turnos, y mediante un serie de reglas las fichas se irán eliminando del tablero, ganando el jugador que más fichas haya eliminado, o el que más fichas mantenga en sus cubículos al final de la partida.

En algunas variantes, como el Omweso, veremos que todas las fichas siempre permanecen en el tablero, nunca se eliminan y el objetivo es llegar a una situación en la que el adversario no pueda mover sus fichas de acuerdo a las reglas.

El movimiento de las fichas es una de las cosas más características de este tipo de juegos, y se conoce como movimiento de "siembra", pues consiste en coger todas las fichas de un cubículo e ir soltando una en cada uno de los cubículos siguientes, movimiento claramente similar a cuando un agricultor siembra semillas en el campo. De hecho originariamente el Mancala se jugaba con semillas o piedrecillas, y el tablero se dibujaba haciendo pequeños agujeros en el suelo.

Existen una infinidad de juegos variantes de la familia mancala, pero podríamos resumir que todos tienen unas características comunes:

- a) Los juegos son jugados en un tablero formado por hileras de cubículos (casillas). El número de casillas e hileras (filas) es diferente dependiendo de la variante del juego.

- b) Los juegos son jugados con un mismo número de fichas (semillas, piedrecillas, contadores) y un mismo número de casillas por cada jugador.
- c) Las fichas (semillas) se mueven en el tablero mediante el movimiento de cuenta conocido “siembra”. Puede ser en el sentido de las agujas del reloj o el contrario.
- d) Con el movimiento de siembra, y dadas unas reglas, las semillas del contrincante pueden ser “capturadas”. Los juegos Mancala son conocidos como “juegos de cuenta y captura”.
- e) El objetivo es capturar la mayor parte de semillas.
- f) Salvo raras excepciones se trata de un juego de dos jugadores.

Estas, aunque no pocas, son las únicas características comunes de todos los juegos Mancala, aunque veremos luego que las variantes y modificaciones son incluso más numerosas.

2.3.La gran familia Mancala. Variantes.

La gran familia Mancala está formada por un gran número de variantes del juego. Al tratarse de un juego muy cultural y folclórico, las reglas y estilos de juego se han ido transmitiendo de una forma no escrita, lo que ha generado que en cada cultura se ha ido evolucionando hacia un estilo de juego y conjunto de reglas diferente.

Así podemos hacer una primera distinción, por ejemplo, en el estilo de movimientos, encontrando:

- a) **Juegos de movimiento simple:** Se hace un solo movimiento de siembra, que puede acabar con captura o no, y se pasa el turno al contrario. Estos juegos son más comunes en África Oeste, los países árabes, y Asia central. Ejemplos de juegos de movimiento de siembra simple son: Adji-boto, Oware (wari, warri, aualé, awelé, abapa, etc.), Toguz Kumalak, Um Dyar, Vai lung thlan, etc.

b) **Juegos de movimiento multiple/encadenado:** Se hace un movimiento simple, pero generalmente si se termina en una casilla no vacía se encadena otro movimiento. En este tipo de juegos se da un caso especial curioso, que es que al encadenarse movimientos puede darse lugar a una situación de movimiento indefinido, hasta el punto de que en las competiciones suele existir una limitación de tiempo para el movimiento, y si el jugador no ha terminado el movimiento a tiempo se anula el juego. Dentro de los juegos de movimiento multiple podemos distinguir:

(a) **Movimiento multiple estilo africano:** El movimiento acaba cuando la última semilla cae en una casilla vacía. Si no esta vacía, su contenido es usado para llenar las casillas siguientes en el movimiento encadenado. Ejemplos de juegos de movimiento de siembra multiple estilo africano son: Andada, Anywoli, Sungka, Kiela, Omweso, Katro, etc.

(b) **Movimiento multiple estilo indio (pussa-kanawa) :** El movimiento acaba cuando la celda siguiente a la que acaba el movimiento esta vacía. Si no está vacía, su contenido se utiliza para rellenar las casillas siguientes a la casilla no vacía siguiente a donde se acabo el movimiento. Este estilo es habitual en India, Sri Lanka, Yunnan, Myanmar y Vietnam. Ejemplos de juegos de movimiento de siembra multiple estilo indio son: Ali guli mane, Pachgarhwa, Tap-urdy, Ô quan, Bosh, Kisolo, etc.

Veremos más adelante que los juegos de movimiento encadenado, generán una variabilidad tal en el estado del tablero, pues en un solo turno pueden hacer cambiar totalmente el estado de todas las casillas del tralero, haciendo el próximo movimiento mucho más impredecible, que los hace especialmente interesantes para su estudio.

Atendiendo al tipo de captura que pueden realizar los jugadores podemos encontrar:

- a) **Juegos con reentrada de semillas:** Las semillas capturadas no se eliminan del tablero sino que se reincorporan rellenando las casillas siguientes en el lado del tablero del jugador que ha realizado la captura. Este tipo de juegos generalmente termina cuando uno de los dos jugadores no puede realizar movimientos de acuerdo a la reglas de juego.
- b) **Juegos sin reentrada de semillas:** En estos juegos las semillas capturadas se retiran del tablero. Aparecen unos huecos especiales adicionales de tamaño más grande, cuya funcionalidad es almacenar las semillas capturadas que han sido eliminadas, estos huecos son conocidos como almacenes. Generalmente el juego termina cuando uno de los dos jugadores ha capturado la mayoría de las semillas del tablero.
- c) **Juegos con captura inversa:** Habitualmente las semillas se desplazan por el tablero en sentido contrario a las agujas del reloj, pero en algunos juegos se permite el movimiento en el sentido de las agujas del reloj, pero solamente desde los huecos más a la izquierda, y sólo si existe la opción de captura. Este movimiento se conoce como captura inversa.

La reentrada de la semillas capturadas es otra de las características que, al aumentar la complejidad de predecir el estado del tablero tras el siguiente movimiento, hace que los juegos que usan esta variante sean muy interesantes para su estudio.

La gran cantidad de variantes de Mancala, no solamente ha generado una diferenciación en las reglas de juego sino que también ha afectado a otros aspectos del juego. El aspecto más visual son las diferencias entre los diferentes tableros de Mancala dependiendo de las variantes.

Atendiendo al número de filas del tablero podemos clasificarlos de la siguiente manera:

- a) **Tableros de una fila:** No existen juegos Mancala tradicionales con tableros de una sola fila, salvo quizás el *Tchuka Ruma*. Se trata de una variante muy moderna. Ejemplos de juegos de una sola fila son: *55Stones (1x11)* o *Progressive Mancala (1x11)*.
- b) **Tableros de dos filas:** Son los más populares y habituales. Consisten en dos filas de huecos, una fila para cada jugador. Aún dentro de esta clasificación existen muchas variantes dependiendo del número de huecos en cada fila, pudiendo así encontrar tableros de 2x3, 2x4, 2x5, 2x6, 2x7, 2x8, 2x9, 2x10, 2x12, etc. Ejemplos de juegos de dos filas son: *Adji-boto (2x5)*, *Oware (2x6)*, *Ali guli mane (2x7)*, *Sungka (2x7)*, *Anywoli (2x12)* o *Um dyar* con dos filas y un número variable de columnas entre dos y doce.
- c) **Tableros de tres filas:** Aunque los más habituales son tableros de dos o cuatro filas, existe una subfamilia de juegos Mancala que se juegan en tableros de tres filas. Este grupo de juegos se conoce como *Selus*. El tablero *Selus* es de 3x6, y cada contrincante es dueño de la fila más cercana a él, y además los tres huecos de la fila central y a su mano derecha.
- d) **Tableros de cuatro filas:** Aunque menos populares que los tableros de dos filas, existen varios juegos Mancala usando tableros de cuatro filas. En estos tableros habitualmente cada contrincante es dueño de las dos filas más cercanas a su lado del tablero. Ejemplos de juegos de cuatro filas son: *Bao (4x8)*, *Omweso (4x8)*.
- e) **Tableros de más de cuatro filas:** Existen contados juegos que usen tableros de más de cuatro filas, pero podemos encontrar el *Laomuzhuqi* de cinco filas, y el *Katro (6x6)*.

Como se ha podido deducir, no solamente varían en el número de filas que tiene el tablero, sino que también varían el número de columnas o casillas en cada fila, variando desde el tablero más pequeño del Nano-Wari (2x1) hasta el más grande del juego En Gehé (2x50), pasando por juegos curiosos como el Um dyar que se puede jugar con un número variable de huecos por fila.

Otros autores hacen otros tipos de clasificaciones atendiendo a características muy variadas de los tableros:

- a) **Clasificación estética:** Se clasifican de acuerdo a sus formas o diseños. Existen tableros Mancala rectangulares, con forma de disco, con formas de animales, con alzas.
- b) **Clasificación según el material:** Existen clasificaciones según el tipo de madera con el que está hecho el tablero, tableros hechos con metal, otros fabricados con cerámica, y por supuesto los más ancestrales escavados en la roca.
- c) **Clasificación por origen:** Por supuesto, debido a la gran expansión del juego por el mundo, resulta habitual la clasificación por país, continente y zona de origen.
- d) **Clasificación por antigüedad:** Generalmente se diferencia entre juegos Mancala tradicionales, y juegos modernos. Los juegos tradicionales suelen ser aquellos que se han ido transmitiendo socioculturalmente, de los que no existe una constancia de reglas escritas. Y se consideran juegos Mancala modernos, variantes creadas más recientemente, con un conjunto de reglas escritas y creados con fines de estudio, y en algunos casos con fines comerciales.

Todos los juegos nombrados en este capítulo hasta ahora, se consideran juegos Mancala tradicionales. Ejemplos de juegos modernos basados en las reglas de Mancala son: 55Stones, Afrika, Calady, Cow Poke, Geisterfahrer, Nyakun, Partisan Sowing y Tchoukaillon, entre muchísimas otras variantes.

2.4.Interés del estudio de la familia Mancala.

La primera impresión que da un primer vistazo a los juegos de la familia Mancala es que son juegos demasiado simples, y sin demasiado interés para su estudio. Son juegos deterministas, sin aleatoriedad, de información perfecta, y con muy pocas opciones a la hora del movimiento, solamente tenemos la opción de elegir la casilla del tablero en la que empezaremos el movimiento, y en algunos juegos la posibilidad de captura inversa, pero no hay muchas más posibilidades de elección a la hora del movimiento que nos podría hacer pensar que pueda elaborarse alguna estrategia interesante de estudiar.

Lo que hace a la familia Mancala realmente interesante es que solamente con un movimiento, podemos afectar a varias casillas del tablero. En movimientos largos, o en los juegos con movimiento encadenado, el estado del tablero, de todas las casillas, puede cambiar significativamente de forma que es más complicado predecir el próximo estado del tablero, al menos sin la ayuda de una computadora.

Como hemos comentado en los juegos con movimiento encadenado la complejidad y el número de estados del tablero aumenta exponencialmente. En los juegos que utilizan captura con reentrada de semillas esta situación se complica combinatorialmente aún más.

Obviamente, la complejidad también varía con el número de casillas del tablero, y el número de semillas que se utilizan para el juego. Debido a la gran variedad de tableros con diferentes configuraciones utilizados en la heterogénea familia Mancala, las posibilidades son infinitas lo que nos permitiría estudiar muchísimas combinaciones y posibilidades observando incluso como afectan estas variaciones de los tableros y reglas particulares en la complejidad de cada uno de los juegos.

La complejidad es tal, que en los juegos con movimientos encadenados, puede darse la situación de movimiento perpetuo o sin final. Este tipo de movimientos suele ser caso de estudio. Se estudian los patrones de repetición, el periodo con el que se repite, etc. En el juego real, este tipo de movimiento ha de ser acabado por reglas especiales, de tiempo o límite de encadenamiento, para poder finalizar el juego.

Además, podemos añadir que los juegos Mancala, han sido incorporados a los estudios de IA hace relativamente poco tiempo, por lo que aún existen muchas posibilidades por estudiar dentro de la familia. De hecho, las mayoría de las investigaciones se han centrado en sólo algunos juegos, como Kalah y Awari, por lo que aún tenemos otros muchos juegos Mancala interesantes para el estudio, como pueden ser, entre otros muchos, Bao y Omweso.

2.5.Un caso particular: Omweso.

Omweso es el juego tipo Mancala más popular de Uganda. En esta sección intentaremos concretar como adopta Omweso las características de los juegos de la familia Mancala.

2.5.1.Características comunes a la familia Mancala.

Siguiendo el esquema de secciones anteriores podríamos describir a Omweso según las características de la familia Mancala de la siguiente manera:

- a) **El tablero:** Se juega en un tablero de cuatro filas de ocho casillas para cada jugador (4x8). Es un juego de dos jugadores, y a cada jugador le pertenecen las dos filas más cercanas a su posición. Se juega con 64 semillas, no diferenciadas e inicialmente cada jugador posee 32 semillas en su lado del tablero.
- b) **Movimientos:** Como en cualquier juego Mancala, Omweso es un juego de siembra y captura. El movimiento de siembra se hace en sentido contrario a las agujas del reloj y es de tipo multiple/encadenado, en el que, si una siembra acaba en una casilla no vacía, se cogen todas las semillas en esa casilla y se hace una nueva siembra encadenada, siguiendo el estilo de movimiento encadenado africano. El encadenamiento puede dar lugar a movimientos infinitos, que deberán romperse con reglas especiales.

- c) **Captura:** Se trata de un juego de captura con reentrada de semillas. Las 64 semillas permanecen siempre en el tablero, nunca se eliminan, sino que se siembran en la zona del tablero del jugador que ha hecho la captura, no existiendo almacenes adicionales en el tablero para guardar las semillas capturadas.
- d) **Captura inversa:** Omweso admite la posibilidad de captura inversa, es decir, de hacer un movimiento de siembra en el sentido de las agujas del reloj, pero sólo desde las cuatro casillas más a la izquierda de cada lado del tablero, y sólo si el movimiento proporciona una captura.

Con esto tenemos a Omweso ubicado dentro de la familia Mancala. Veremos que de estas características se podrían intuir las reglas de juego de Omweso, y que salvo por un par de reglas especiales que marcan situaciones excepcionales de victoria/derrota, lo único que nos faltaría saber es que el juego termina cuando uno de los jugadores no puede realizar ningún movimiento legal.

2.5.2.Reglas de juego de Omweso.

La gran componente cultural y folclórica que existe en la transmisión no escrita de las reglas de la familia Mancala, sobre todo en juegos antiguos como Omweso hace que las reglas de juego puedan sufrir variaciones dependiendo donde se juegue, pero aún así describiremos las reglas de juego que adoptaremos y usaremos durante este proyecto.

Así pues las reglas que usaremos para Omweso serán las siguientes:

- (1) **Configuración inicial del tablero:** Cada jugador coloca sus 32 semillas distribuidas por las casillas de su lado del tablero de la forma que crea más conveniente.

- (2) **Objetivo del juego:** Capturar las semillas del contrincante, hasta que este no pueda realizar ningún movimiento legal durante su turno.
- (3) **Movimientos:** Los movimientos se realizarán, por turnos. Serán movimientos de “siembra” estilo africano, en sentido contrario a las agujas del reloj.
- (4) **Restricciones de movimiento:** Las casillas con una sola semilla, y obviamente las casillas sin semillas, no pueden iniciar un movimiento.
- (5) **Fin del turno:** Si la última semilla del movimiento cae en una casilla vacía, finalizará el movimiento pasando el turno al jugador contrario, sino se continuará con un movimiento encadenado.
- (6) **Movimiento encadenado:** Si la última semilla de un movimiento cae en una casilla no vacía, el turno no termina, si se puede se realizará una captura, sino se tomarán todas las semillas de la casilla donde ha caído la última semilla del movimiento y se realizará un nuevo movimiento de siembra, encadenándose movimientos hasta que uno de ellos termine en una casilla vacía.
- (7) **Captura:** Se producirá una captura si se cumplen dos condiciones, una, que el movimiento cae en una casilla de la fila interior del tablero, no vacía (servirá de marcador de captura), y dos, que las dos casillas del jugador contrario opuestas de la misma columna están ocupadas. En este caso, se capturarán todas las semillas que existan en esas dos casillas del contrario, eliminándose de su lado del tablero y reentrando en el lado del tablero del jugador que realizó la captura.
- (8) **Reentrada de semillas:** Las semillas capturadas reentrarán en el lado del jugador que ha realizado la captura mediante una siembra estándar, partiendo desde la misma casilla de la que partió el movimiento anterior, el que originó la captura. El propio movimiento de reentrada puede generar otra captura o un movimiento encadenado de la forma habitual.

- (9) **Captura inversa:** Está permitido un movimiento de siembra en sentido de las agujas del reloj siempre que se origine desde las cuatro casillas más a la izquierda del lado del tablero del jugador que realiza el movimiento, siempre y cuando se produzca una captura con dicho movimiento. Las semillas capturadas reentrarán en el tablero de la forma habitual, con una siembra no inversa típica desde la casilla que originó el movimiento que acabo en captura inversa. Como es habitual, la reentrada puede generar otra captura o un movimiento encadenado.
- (10) **Movimiento infinito:** En el caso de producirse un movimiento encadenado infinito, se limitará la duración máxima a 1000 movimientos y se terminará el turno pasándolo al contrario. Esta limitación en el número máximo de movimientos encadenados es la solución que se ha tomado en este proyecto para solucionar el problema computacional del movimiento infinito, no se trata de ninguna regla oficial.

2.5.3.Características de Omweso en el marco de la IA.

La IA clasifica los juegos dentro de diferentes categorías atendiendo a sus características. La finalidad de esta clasificación es que, una vez determinada la categoría a la que pertenece un juego, todos los juegos pertenecientes a esa misma categoría son susceptibles de ser resueltos con los mismos métodos, facilitando y acelerando el estudio de los mismos. Así pues, intentaremos clasificar Omweso dentro de algunas de las categorías más habituales, lo que nos permitirá ser conscientes de qué tipo de métodos deberemos usar para estudiarlo.

- (a) **Juegos de N-agentes:** Dependiendo del número de jugadores que intervienen en una partida, los juegos se pueden clasificar en juegos de un agente, de dos agentes, de N-agentes, etc. Omweso es un juego de dos agentes.
- (b) **Juegos dinámicos:** Un juego se considera dinámico si el fin perseguido por los contrincantes es bien conocido por todos. En este aspecto podríamos decir que Omweso sí pertenece a esta categoría.
- (c) **Juegos de suma nula y simétricos:** Un juego se considera de suma nula, si el balance de ganancias y pérdidas de todos los jugadores es perfecto. Es decir, en el caso más sencillo de juegos de dos agentes, cada ganancia de un jugador genera una pérdida perfectamente simétrica del contrincante. Además, si independientemente de que jugador realice la estrategia, la recompensa es la misma, se considera que el juego es simétrico. Omweso es un juego de suma nula, y además, simétrico.
- (d) **Juegos cooperativos:** Varios de los agentes pueden llegar a algún tipo de acuerdo para actuar de forma que el resultado de sus acciones es beneficiosa para ambos, en su conjunto, y perjudicial para un contrincante común. En el caso de Omweso, al tratarse de un juego de sólo dos agentes, no existe la posibilidad de ninguna cooperación contra un tercer agente, que obviamente no existe.
- (e) **Juegos simultáneos o secuenciales:** Un juego se considera simultáneo cuando todos los jugadores realizan sus acciones a la vez, o aún no siendo a la vez si se cumple que ninguno conoce los movimientos de los otros hasta que todos han realizado su movimiento. Por el contrario un juego es secuencial si cuando un jugador tiene que decidir su movimiento ya conoce el movimiento anterior de su contrincante. Omweso es un ejemplo claro de juego por turnos, por tanto de movimientos secuenciales.

(f) **Juegos de información perfecta:** Son aquellos juegos en los que no existe el azar, cada acción y consecuencia son perfectamente conocidas. Omweso es un juego de información perfecta.

(g) **Juegos de información completa:** Se consideran juegos de información completa aquellos en que todos los jugadores tienen toda la información sobre el juego, posibles acciones, estado de cada uno de los jugadores, etc., teniendo todos los jugadores la misma información del juego.

Podríamos pues resumir la categorización de Omweso de la siguiente manera: Omweso es un juego de dos agentes, de información perfecta y completa, que se juega por turnos con movimientos secuenciales, un juego dinámico, simétrico y de suma nula.

2.5.4. Características que lo hacen interesante para su estudio.

Como ya hemos comentado cuando hablábamos del interés de la familia Mancala, a simple vista parecen juegos demasiado sencillos y sin interés para su estudio.

Sin ir mas lejos, como hemos visto en el punto anterior, Omweso es un juego clasificado por la IA en categorías bastante comunes y estudiadas. Es un juego de dos agentes, campo en el que se lleva años estudiando, un juego de información perfecta y completa, lo que parece que no deja opción a que ocurra nada fuera de lo común, se juega por turnos y es de suma nula, lo que no parece ofrecernos mucho campo de estudio.

¿Cuál es pues el interés del estudio de Omweso? Pues el principal interés es lo que ocurre cuando un jugador hace un movimiento, el movimiento de siembra.

A diferencia de los movimientos de muchos otros juegos, que solamente afectan a la casilla que origina el movimiento y a la casilla destino, el movimiento de siembra, en un solo movimiento, afecta a todas las casillas que encuentra en su recorrido.

Esto hace que el estado del tablero cambie radicalmente de un movimiento al siguiente, cambiando todas las posibilidades de movimiento y captura para el jugador contrincante cuyo turno comienza después.

El porcentaje de tablero que se ve afectado en un movimiento de siembra incrementa proporcionalmente a la concentración de semillas, es decir, al realizar una siembra originada en una celda donde hay muchas semillas el movimiento de siembra será más largo afectando a una porción más grande del tablero.

En este sentido, Omweso, dentro de la familia Mancala, podríamos decir que potencia este aspecto al ser un juego en el que las capturas no eliminan las semillas sino que se reintroducen en el tablero promoviendo que la concentración de semillas en el tablero no se vaya diluyendo tras cada captura como ocurre en otros juegos Mancala, y además se reintroducen mediante una siembra, lo cual hace que incluso una captura, no sólo no contribuya a diluir la concentración de semillas en el tablero, sino que además produce un cambio importante en el estado del tablero, que como vimos le da una visión interesante al estudio del juego.

Además de la concentración de semillas mantenida durante toda la partida, Omweso tiene otra característica que le hace interesante: los movimientos encadenados. Un movimiento de siembra no sólo produce un importante cambio en el estado del tablero, sino que además puede generar otros movimientos encadenados. Si el estado del tablero cambia tras un movimiento, sólo hay que imaginar que tras cuatro, o diez, o veinte movimientos encadenados sin acabar el turno, el estado del tablero no tiene nada que ver con el estado anterior, lo que complica la estrategia de los jugadores. Incluso veremos que existe la posibilidad de que se de un encadenamiento infinito, que habrá que detener con alguna regla especial.

Los movimientos infinitos no se estudiarán durante este proyecto pues están fuera de los objetivos planteados, pero dado su interés se recomienda la lectura de algunos trabajos al respecto como pueden ser los de *J.Donkers, J.Uiterwijk & A.Voogt*³.

3 J.Donkers, J.Uiterwijk & A.Voogt. Mancala Games.Topics in Mathematics & A.Intelligence. 2001.

2.6.Estado del estudio del Mancala.

Como ya hemos comentado el estudio de los juegos de la familia Mancala es bastante reciente, y la familia es tan amplia que el campo investigado hasta el momento es bastante reducido comparado con las posibilidades existentes.

Entre los primeros estudios se encuentran los hechos para resolver juegos como **Kalah** [Irving, Donkers & Uiterwijk. *Solving Kalah. September 2000*]. Algunas características de Kalah lo hicieron más apto para ser uno de los primeros juegos de la familia Mancala en intentar ser resueltos. Kalah en ningún momento genera bucles, es decir nunca se repite una misma posición, y además no tiene reentrada de semillas capturadas, por lo que el estudio usando bases de datos de posiciones finales se simplifica bastante, pero aún así no le quita mérito a su resolución.

Usaron bases de datos completas del juego, bases de datos de posiciones terminales, y algoritmos de búsqueda como el MTD(f), ó Memory-enhanced Test Driver, que es una mejora del tradicional algoritmo alpha-beta usando solamente ventanas de tamaño cero en la búsqueda [Plaat et al., 1996a]. Para mejorar el rendimiento aplicaron optimizaciones como la ordenación de movimientos, tablas de transposición, o *futility pruning*, una poda que elimina nodos que son tan malos que no podrían en ningún caso competir con la variante principal.

Entre otro de los estudios realizados se encuentran por ejemplo los trabajos en la investigación del **Awari**, buscando soluciones mediante análisis retrogrado [Romein & Bal. *Solving the Game of Awari using Parallel Retrograde Analysis. October 2003*].

Utilizando una red de ordenadores conectados por una rápida interconexión, conformando un gran sistema de computación paralela, usando 144 procesadores (cluster de 72 ordenadores dual 1,00Ghz Pentium III), 72GB de memoria principal y una rápida interconexión, una red conmutada de 2Gb/s con enlaces bidireccionales (Myrinet), tuvieron que determinar la evaluación de 889.063.398.406 posiciones para resolver el Awari.

Para la resolución diseñaron un algoritmo paralelo basado en análisis retrogrado, que partiendo de las posiciones terminales del juego, razona hacia atrás hasta la posición raíz del juego. De esta manera consiguen resolver el juego en 51 horas, o incluso menos aplicando algunas optimizaciones.

Otros estudios se produjeron en el campo del **Awari**. Davis y Kendall investigaron en como evolucionar un jugador de Awari, usando una aproximación co-evolutiva donde jugadores computerizados juegan unos contra otros, contra los jugadores supervivientes más fuertes, mutados usando una estrategia evolutiva (ES) [*David & Kendall. An Investigation, using Co-Evolution , to Evolve an Awari Player. 2002*].

Los jugadores eran representados usando una función de evaluación sencilla representando el estado del juego. Cada peso interviniente en la función va siendo evolucionado usando ES. El resultado de la función es utilizado en una búsqueda de tipo minimax, encontrando de esta forma al jugador más evolucionado.

Enfrentaron al jugador más evolucionado contra uno de los juegos shareware más fuertes (Awale) consiguiendo vencerle en tres de sus cuatro niveles de dificultad.

Son especialmente destacables los estudios de Donkers, Uiterwijk y Voogt sobre los juegos de la familia Mancala, pero volcándose en las características matemáticas de los juegos, estudiando si existe alguna peculiaridad matemática que los define o los diferencia, de forma que puedan dar pistas para encontrar alguna forma de resolverlos eficientemente.

Descubriendo que a pesar de que los juegos Mancala son juegos aparentemente sencillos, ya que son bastante deterministas, de información perfecta, sin participación aleatoria y con no demasiadas reglas ni posibles movimientos, hay una característica que los hace especialmente complejos. El hecho de que un simple movimiento puede tener efectos sobre los contenidos de muchas o incluso todas las casillas del tablero, hace especialmente difícil predecir incluso pocos movimientos por adelantado, cuanto

más el resultado final del juego [J.Donkers, J.Uiterwijk & A.Voogt. *Mancala Games. Topics in Mathematics and Artificial Intelligence. 2001*].

También dejan abiertas otras líneas de investigación, dejando planteadas algunas preguntas especialmente orientadas a los especialistas en el campo de las matemáticas como, de qué forma afecta alguna de las reglas en la complejidad del juego Mancala, si es posible predecir la complejidad conociendo el conjunto de reglas, que tipo de heurísticas se pueden usar para que una computadora juegue un juego Mancala, o cómo deben ser manejadas o estudiadas las reglas especiales de algunos juegos Mancala.

Como decíamos, los estudios e investigaciones sobre la familia Mancala son relativamente recientes y no demasiado abundantes. Concretamente en el caso de Omweso no existe ningún estudio, que conozcamos, directamente orientado a su estudio por lo que tenemos un amplio campo prácticamente inexplorado en el que trabajar lo cual lo hace aún más interesante para su estudio.

La principal fuente de información y estudios sobre Omweso podemos encontrarla en la International Omweso Society, donde Brian Wemham se ha encargado de recolectar toda la información disponible para este juego, desde el origen del Omweso, descripción de las reglas, la complejidad del juego o incluso en algunos interesantes estudios sobre los movimientos infinitos. [B.Wemham. *Omweso: The royal Mancala Game of Uganda. A General Overview of Current Research. 2007*].

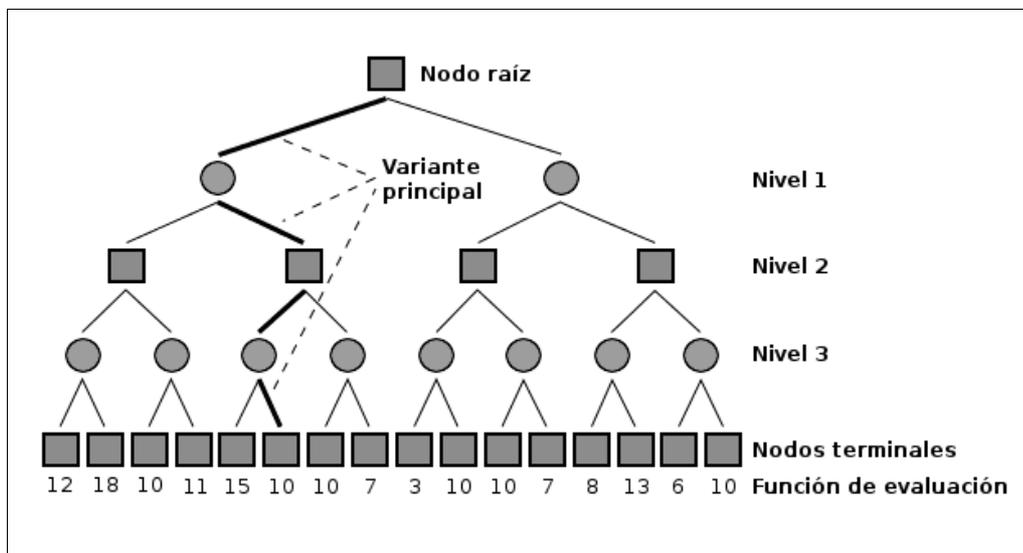
2.7.Algoritmos de Búsqueda.

Durante muchos años los investigadores han buscado algoritmos de búsqueda para la resolución de diferentes problemas. En un proceso de más de 20 años han aparecido diferentes algoritmos, cada uno con sus mejoras y algunos de ellos evolucionando a partir de algoritmos antecesores. [Murray S. Campbell and T.A. Marsland. *A Comparison of Minimax Tree Search Algorithms. University of Alberta, Canada*].

Entre todos ellos el más representativo, en los juegos de suma nula, y en el que se basan muchos de los algoritmos posteriores es el Minimax. Durante este capítulo describiremos el Minimax y otros algoritmos también clásicos de la IA como son: el negamax, el alfabeta y el scout.

2.7.1. Algoritmos de búsqueda : conceptos generales.

Todos los algoritmos de búsqueda manejan unos conceptos básicos idénticos con los que deberemos familiarizarnos y que nos ayudarán a manejar la nomenclatura necesaria para entender o interpretar cualquiera de ellos.



Árbol 1: Conceptos generales

- (a) **Árbol de búsqueda:** Generalmente el desarrollo de un algoritmo de búsqueda genera un árbol en el que cada rama representa cada una de las decisiones posibles a tomar durante el juego.
- (b) **Nodos y posiciones:** Cada vez que se toma una rama (decisión) del árbol el juego pasa a un estado diferente consecuencia de dicha decisión. Al estado que toma el tablero en cada momento es habitual conocerlo como posición. Así la **posición inicial**, sería el estado desde el que comienza la partida. Los **nodos** son donde se toman las decisiones de que jugada realizar.

- (c) **Nodo raíz y nodos terminales:** Así se conoce como nodo inicial o **nodo raíz** del árbol de búsqueda, al primer nodo del árbol desde el que se toma la primera decisión. Los **nodos terminales** son aquellos en los que no se puede tomar ninguna decisión más, habitualmente porque la partida finaliza, o bien existe alguna limitación como puede ser tiempo de búsqueda, profundidad, etc.
- (d) **Nodos padre y nodos hijo:** Desde cada nodo del árbol de búsqueda se generan N ramas que representan las posibles decisiones a tomar (jugadas) desde ese estado del juego. Cada rama (jugada) dará lugar a un nuevo estado del tablero, estos estados generados se conocen como **nodos hijo**, y el nodo desde el que se generaron se considera como **nodo padre**, a su vez los nodos hijos se considerarán como nodos padre de los nodos generados en el siguiente nivel de profundidad.
- (e) **Factor de ramificación:** El número de ramas que se generan desde cada nodo del árbol es lo que se conoce como **factor de ramificación**. Factores de ramificación alto suelen indicar juegos más complejos, o al menos con un mayor número de decisiones entre las que elegir, por lo que habitualmente el factor de ramificación multiplica exponencialmente los recursos necesarios para analizar todas las decisiones a tomar durante el juego.
- (f) **Niveles de profundidad:** Los algoritmos de búsqueda suelen funcionar analizando las posibles jugadas del jugador actual, y posteriormente del jugador contrincante, simulando los turnos de uno y otro jugador hasta encontrar las decisiones correctas para ganar la partida. Con infinitos recursos los algoritmos podrían analizar todos los turnos necesarios hasta simular la partida completa y así encontrar la manera de ganar la partida. Pero ocurre que en la mayoría de los casos los juegos son lo suficientemente complejos para que sea imposible analizar todas las ramas del árbol hasta encontrar todas las soluciones al juego, por lo que habitualmente se limita el análisis a solamente la simulación de unos cuantos turnos por adelantado.

A este número de turnos por adelantado que se analizan para tomar la decisión de juego es a lo que se conoce como **profundidad de búsqueda**.

(g) **Función de evaluación:** La forma en la que se decide si cada una de las decisiones tomadas es correcta o no, es mediante la utilización de una función que evaluará la posición/estado del tablero de juego tras simular cada decisión y devolverá un valor que representará si dicha decisión representa un beneficio o una ventaja en el juego para el jugador que toma dicha decisión. De esta manera veremos que mientras los algoritmos de búsqueda son bien conocidos, el verdadero reto está en encontrar una función de evaluación que discrimine de forma eficiente entre buenas y malas decisiones de manera que nos permita encontrar la solución al juego.

(h) **Variante principal:** Se conoce como variante principal la sucesión de movimientos o decisiones tanto del jugador que esta realizando la búsqueda como del contrincante que llevarían a la resolución del juego mas favorable para el jugador. La variante principal es lo que obtendremos como resultado de la búsqueda del algoritmo.

Podemos pues resumir el proceso de búsqueda utilizando los términos anteriores de la siguiente manera: El algoritmo de búsqueda, partiendo del **nodo raíz** (estado inicial) expandirá todos los **nodos padre e hijos**, hasta llegar a los **nodos terminales**, y analizará todas las ramas del árbol de búsqueda, discriminando entre buenas y malas ramas (decisiones) mediante el valor obtenido por una **función de evaluación**. De esta manera encontrará la sucesión de decisiones, conocida como **variante principal**, que nos llevará a la resolución de la partida más favorable para el jugador que realiza la búsqueda.

2.7.2. Algoritmos de búsqueda : Minimax.

El Minimax es uno de los algoritmos con más peso y más representativo en la IA. De hecho, muchos de los algoritmos posteriores se basan en el Minimax, algunos de

ellos son evoluciones del propio Minimax, y en muchos casos, como mínimo, heredan algunos de sus conceptos y fundamentos.

El Minimax pertenece a lo que podría denominarse como una primera generación de algoritmos de búsqueda. Estos algoritmos se caracterizan porque realizan una búsqueda exhaustiva, lo que podría denominarse como algoritmos de fuerza bruta, analizando todas y cada una de las jugadas posibles en cada momento para encontrar la mejor solución al problema.

Ideado por Von Neumann [*Von Neumann, 28*], se basa en la idea de que tanto el jugador como el contrincante seleccionarán la mejor jugada que encuentren durante su turno.

Es decir, el algoritmo simula tanto los turnos del jugador como del contrincante. Durante los turnos simulados del jugador elige la jugada que **maximiza** la evaluación del tablero para el jugador. Durante los turnos simulados del contrincante, asume que este seleccionará las jugadas que **minimizan** la evaluación del jugador, ya que maximizar tu evaluación implica minimizar la de tu contrincante, ya que estamos hablando de un algoritmo que se aplica a **juegos de suma nula** [*2.5.3.Características de Omweso en el marco de la IA. (d)*].

El proceso Minimax es el siguiente:

- Desde cada nodo P se generan/simulan todas las jugadas posibles, partiendo como primer nodo desde el nodo raíz.
- Los nodos en los que juega el jugador son denominados nodos Max. Los nodos en los que juega el contrincante se denominarán nodos Min.
- Si un nodo es nodo terminal, su valor vendrá dado por el valor devuelto por la función de evaluación seleccionada aplicada al estado del tablero en ese nodo.

- Si no es un nodo terminal, y es un nodo Max, su valor será el mayor de los valores devueltos por sus nodos hijos.

$$\text{valor}(P) = \max_{i=1,n} \{ \text{valor}(H_i), \forall H_i \in \text{Hijos}(P) \}$$

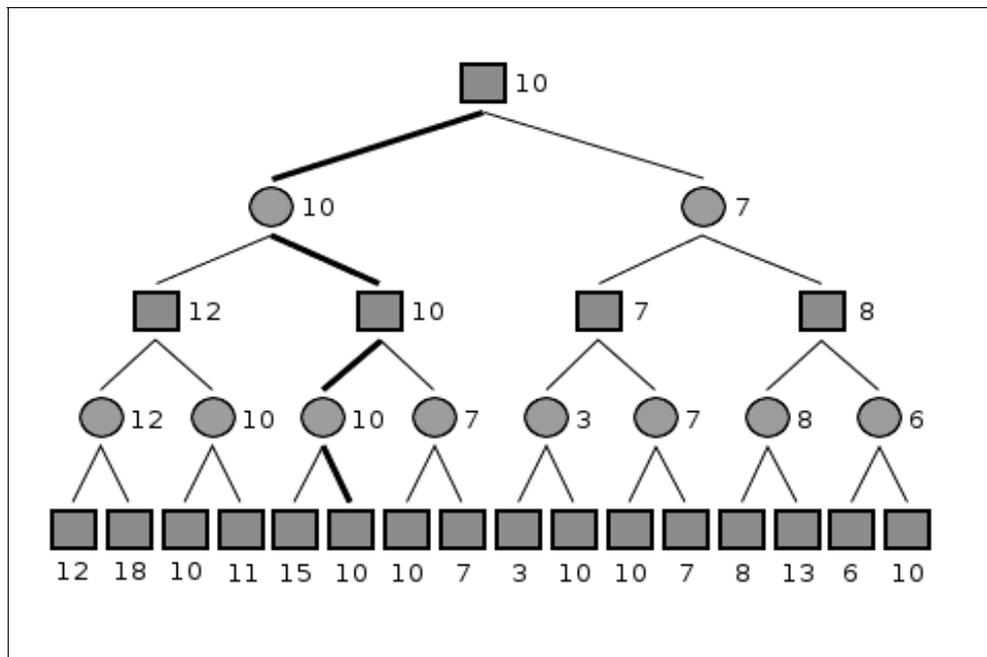
- Si no es un nodo terminal, y es un nodo Min, su valor será el menor de los valores devueltos por sus nodos hijos.

$$\text{valor}(P) = \min_{i=1,n} \{ \text{valor}(H_i), \forall H_i \in \text{Hijos}(P) \}$$

De esta manera, el algoritmo comienza obteniendo los valores de los nodos terminales mediante la función de evaluación, y los valores se van propagando hacia arriba hacia sus padres aplicando las normas Min y Max, hasta llegar al nodo raíz.

Veremos que de esta manera tras la propagación hacia los padres, al nodo raíz que es el nodo padre de todo el árbol, le llegará propagado el mejor de los valores que se pueden obtener tras analizar todas las jugadas.

No sólo obtendremos el mejor valor, sino que si lo planteamos bien podremos obtener la **variante principal** [2.7.1.Algoritmos de búsqueda. Conceptos generales. (h)].



Árbol 2: Algoritmos de búsqueda. Minimax.

Como se puede ver en la figura [Árbol 2], podemos ver como los valores de los nodos terminales se propagan hacia arriba, eligiendo el máximo valor de sus sucesores, en el caso de nodos Max, o el mínimo valor de sus sucesores, en el caso de los nodos Min.

Vemos también, en color más oscuro y con líneas más gruesas, la variante principal que como vemos viene determinada por el recorrido realizado por el valor que finalmente obtiene el nodo raíz.

En el apartado [Apéndice A.1. Pseudocódigo del Minimax] de este mismo documento, se presenta el pseudocódigo del Minimax. Como se ve la forma más sencilla de expandir un árbol de búsqueda es implementarlo como un algoritmo recursivo.

2.7.3. Algoritmos de búsqueda : Negamax.

Fue Baudet quien dándose cuenta de la posibilidad de aplicar un pequeño giro matemático, dio lugar a uno de las variantes más curiosas del Minimax, el algoritmo conocido como Negamax.

Matemáticamente, calcular el valor Min durante las jugadas del contrincante es equivalente al resultado que se obtendría de calcular en su lugar el valor Max cambiando previamente el signo de los valores de los nodos [Baudet, 78].

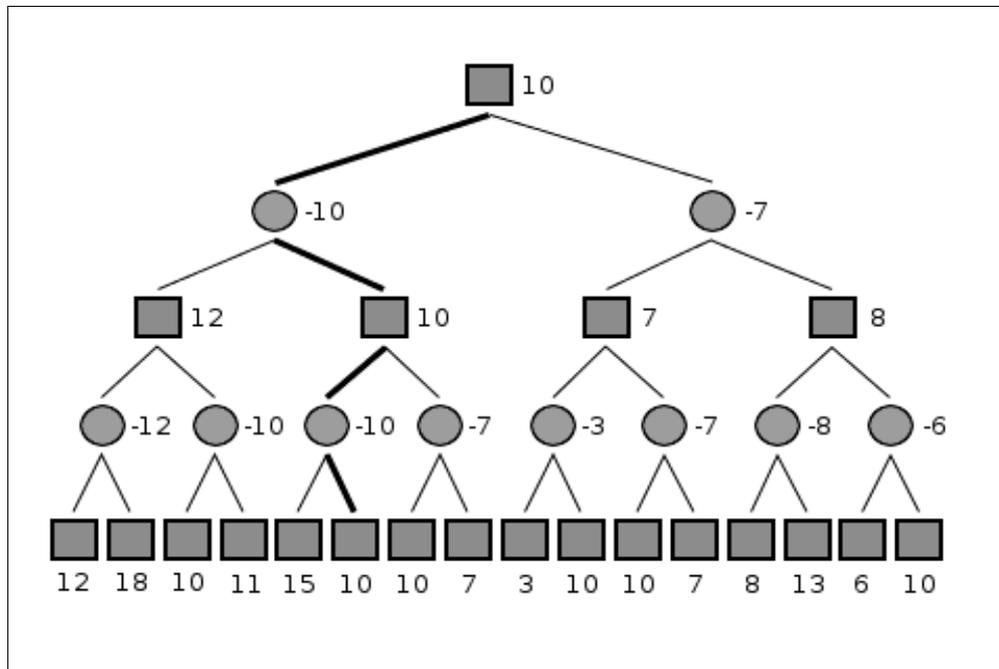
$$\max \{ \min \{x_1, x_2, \dots\}, \min \{y_1, y_2, \dots\}, \dots \} = \max \{ -\max \{-x_1, -x_2, \dots\}, -\max \{y_1, y_2, \dots\}, \dots \}$$

Este algoritmo aportó la posibilidad de simplificar el algoritmo Minimax, de forma que ya no era necesario implementar dos subprocesos, uno minimizador para calcular el Min, y otro maximizador para calcular el Max, sino que con cambiar el signo de los valores de los nodos en cada turno generaba automáticamente una alternancia que permitía al algoritmo trabajar solamente con un proceso maximizador, tanto en los turnos del jugador como durante los del contrincante.

De esta manera las reglas para el cálculo de los valores de los nodos se simplifican a una única regla:

$$\text{valor}(P) = \max_{i=1,n} \{ -\text{valor}(H_i), \forall H_i \in \text{Hijos}(P) \}$$

En la siguiente figura tenemos un ejemplo de aplicación del Negamax a un árbol de búsqueda:



Árbol 3: Algoritmos de búsqueda. Negamax.

Hacer notar, que mientras en el Minimax la función de evaluación calcula los valores para los nodos terminales siempre en relación con el jugador que está realizando la búsqueda con el algoritmo de búsqueda, independientemente de quien sea el turno, en el Negamax la función de evaluación tendrá que evaluar los nodos teniendo en cuenta de a quien le toca mover en el nivel/turno de los nodos evaluados.

El Negamax no aporta, respecto al Minimax, una mayor velocidad de proceso o un mayor potencial de cálculo, pero si ayuda a que el código sea más simple, corto y elegante que el del Minimax original.

En los algoritmos descritos en los siguientes apartados, utilizaremos su variante adoptando este concepto del Negamax, en lugar de utilizar sus variantes tradicionales. Es curioso como el Negamax más que un algoritmo se podría considerar como la

aplicación de un concepto a cualquier algoritmo. De hecho, es fácil ver, que lo que llamamos algoritmo Negamax no es más que un Minimax que ha adoptado el concepto planteado por Baudet [Baudet, 78].

En el apartado [Apéndice A.2. Pseudocódigo del Negamax] de este mismo documento, se presenta el pseudocódigo del Negamax. Como se ve la forma más sencilla de expandir un árbol de búsqueda es implementarlo como un algoritmo recursivo. En el pseudocódigo podemos ver como la introducción del concepto del negamax simplifica la implementación del algoritmo.

2.7.4. Algoritmos de búsqueda : Alfabeta.

El Alfabeta, aún basándose y evolucionando el Minimax, aporta una serie de conceptos a las búsquedas tan interesantes que veremos que se convierte en un punto de referencia y evolución para otros muchos algoritmos posteriores.

El Alfabeta introduce el concepto de **ventana de búsqueda**. La ventana es un rango de valores delimitado por los valores (α, β) . De ahí es de donde al Alfabeta adquiere su nombre.

Junto a la ventana de búsqueda, el Alfabeta, incorpora otro concepto nuevo conocido como **poda** Alfabeta. El algoritmo utiliza el intervalo (α, β) para desestimar (podar) aquellas soluciones que no se adaptan a la ventana de búsqueda entendiendo que no contribuirían a una mejor solución que la que ya tiene.

El límite **alfa** (α) guarda la cota inferior que puede aplicarse a un nodo Max, es decir, guarda el mayor valor para un nodo Max que se ha propagado hacia el padre de la rama que estamos analizando hasta ese momento. El límite **beta** (β) representa la cuota superior del valor que puede asignarse a un nodo Min, es decir, guarda el valor Min más pequeño encontrado. Si el valor **beta** (β) de alguno de los nodos descendientes es menor que el valor **alfa** (α) propagado hasta el momento sabemos que el contrincante eligirá

ese nodo como valor beta (o alguno peor), así que el resto de nodos descendientes que sean mejores para nosotros nunca serán elegidos por el contrincante por lo que es innecesario analizarlos y podemos realizar una “poda”.

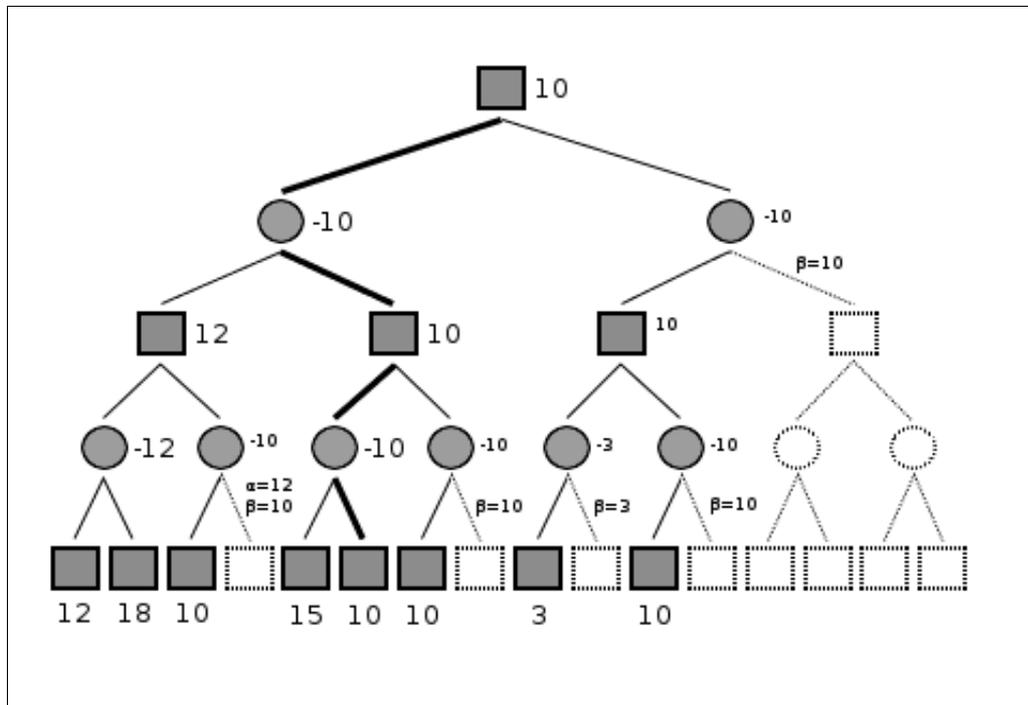
Podemos aplicar la poda alfabeta tradicionalmente sobre el Minimax, o podemos adoptarla directamente sobre el esquema Negamax. Aplicandola sobre el Negamax quedaría de la siguiente manera:

- Inicialmente α y β toman los valores $-\infty$ y $+\infty$ respectivamente.
- Si un nodo P es terminal, su valor es el de la función de evaluación para el jugador para el que se resuelve el árbol de búsqueda, para el jugador del nodo raíz.
- Si el nodo P no es terminal, su valor será:

$$\text{valor}(P) = \max_{i=1..n} \{ -\text{valor}(H_i), \sqrt[n]{\prod_{i=1..n} \text{valor}(H_i)} \}$$

$$\alpha = \max \{ \alpha, \text{valor}(P) \}$$
- En cada nodo, se evalúa la condición de poda ($\alpha \geq \beta$) y si se cumple no se exploran mas hijos y se devuelve el valor almacenado hasta el momento.
- A los hijos que se generan a partir de P se les asigna como valor alfa, $-\beta$, y como valor beta, $-\alpha$. Es decir se se asigna la ventana $(-\beta, -\alpha)$.
- El algoritmo finaliza cuando los valores terminan de propagarse hasta el nodo raíz, siendo la mejor jugada la que devolvió el valor asignado al nodo raíz.

En la siguiente figura analizaremos un ejemplo de Negamax con poda alfabeta, donde se pueden ver varias podas, representadas por líneas discontinuas:



Árbol 4: Algoritmos de Búsqueda. Alfabeta (negamax).

Como ejemplo podemos analizar la poda realizada en el nodo en el primer nodo en trazo discontinuo, el más a la izquierda. Del hermano de su padre se ha propagado hacia arriba un valor de 12. Al tener el un hermano con valor 10, sabemos que su padre, que es nodo del contrincante, no elegirá a nuestro nodo de trazo discontinuo aunque mejorara nuestra puntuación.

Eligiría a su hermano de valor 10, o a algún otro hermano que nos diera peor puntuación ya que el contrincante elegiría las jugadas que peor puntuación nos den a nosotros.

Para resumirlo, cuando durante el turno de nuestro contrincante se encuentre una jugada peor que la encontrada por otras ramas, podemos dejar de analizar el resto de las jugadas del contrincante en esa rama (poda) pues sabemos que si el resto de jugadas son

mejores el contrincante siempre tendrá opción de desestimarlas en favor de la que hemos encontrado que es peor.

De esta forma nos ahorraremos analizar todas esas jugadas que sabemos que el contrincante nunca eligirá habiendo una peor.

Aunque la eficiencia de la poda depende de la ordenación de los nodos, en promedio, el Alfabeta mejora el rendimiento de un Minimax normal en un 33% pudiendo analizar más posiciones con los mismos recursos y tiempo.

En el apartado [*Apéndice A.3. Pseudocódigo del Alfabeta*] de este mismo documento, se presenta el pseudocódigo del Alfabeta. Como se ve la forma más sencilla de expandir un árbol de búsqueda es implementarlo como un algoritmo recursivo. La introducción de la poda alpha-beta es bastante sencilla y reduce el número de nodos analizados.

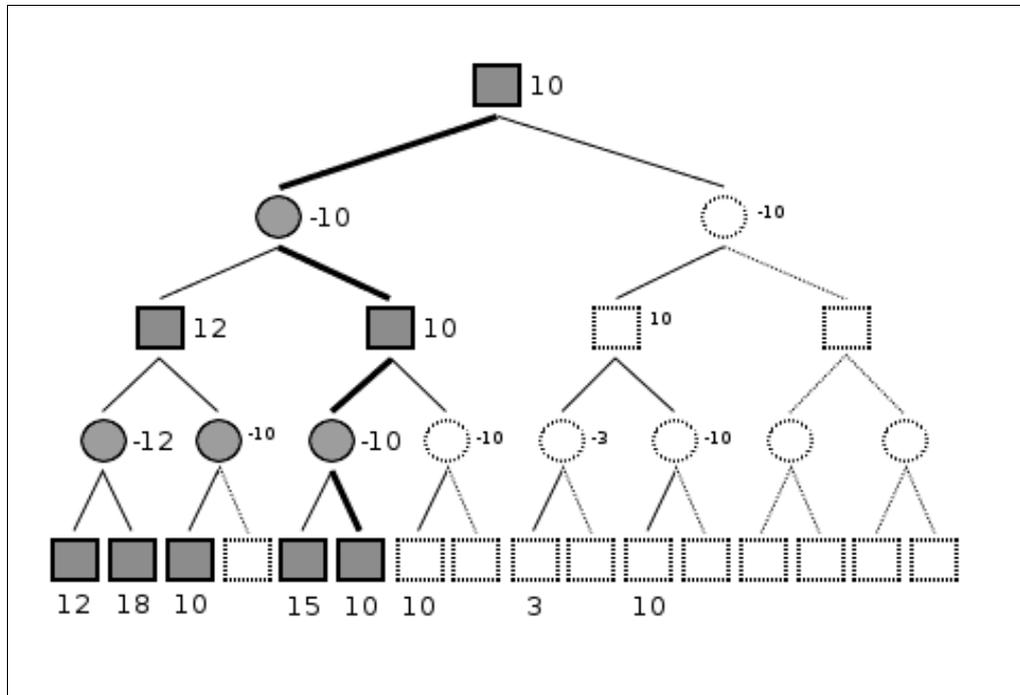
2.7.5. Algoritmos de búsqueda : Scout.

El Scout [*Campbel et al., 83*] es una generalización del Palfa-beta. Los algoritmos basados en el Alfabeta se basan en la modificación de la ventana de búsqueda inicial, que en el Alfabeta es $(-\infty, +\infty)$, por una ventana de búsqueda más pequeña que aún siga acotando la solución al problema.

El Scout en lugar de hacer primero una llamada a un algoritmo de búsqueda con una ventana mínima, se realiza una llamada a un procedimiento booleano Test (mucho más rápido que la llamada con ventana mínima) que comprueba si ocurre un fallo superior. Si es así, realiza una nueva llamada a Scout.

El rendimiento del Scout, gracias a la velocidad de Test ha convertido a este algoritmo en uno de los más populares entre los investigadores de IA. Al no utilizar los límites, los valores devueltos por el Scout o su función auxiliar el Test no pueden ser usados para reducir el tamaño de la ventana de búsqueda al re-explorar un subárbol (es lo que se conoce como algoritmos no direccionales).

En la siguiente figura podemos ver el resultado de la aplicación del Scout a un árbol de búsqueda :



Árbol 5: Algoritmos de Búsqueda. Scout (negamax).

Como se puede ver en la figura, se realizan más podas gracias al análisis preliminar de las ramas por parte del Test.

Los nodos en líneas discontinuas pero con valor son los inspeccionados por el Test, están en discontinua porque no son evaluados propiamente dicho por el Scout, y a su vez con una sobrecarga mínima gracias a la rapidez del Test, vemos como nos permite podar practicamente entera la parte derecha del árbol con sólo un par de evaluaciones del Test en los nodos terminales de la parte derecha del árbol.

En el apartado [Apéndice A.4. Pseudocódigo del Scout] de este mismo documento, se presenta el pseudocódigo del Scout. La forma más sencilla de expandir un árbol de búsqueda es implementarlo como un algoritmo recursivo. Se puede apreciar la utilización de la función test() a modo de sonda para evitar analizar nodos innecesarios.

Capítulo 3

Objetivos

3.Objetivos.

Los objetivos de este proyecto son diseñar e implementar un tablero de Omweso, que permita jugar a dos jugadores, siguiendo las reglas adoptadas y descritas en la sección [2.5.2. Reglas de Omweso]. La principal finalidad será poder enfrentar a dos agentes jugando con las reglas de Omweso, cada uno con sus propios algoritmos de IA, y hacer un estudio comparativo de los resultados de los enfrentamientos.

1. Deberemos diseñar varios agentes que jueguen sobre el tablero, uno de los agentes deberá jugar siguiendo los movimientos indicados por un jugador humano, mientras que el resto de agentes se diseñarán para jugar de forma autónoma usando algoritmos de búsqueda clásicos de IA.
2. En el caso de que uno de los agentes que intervienen en la partida sea el agente controlado por un humano, el tablero deberá proporcionar un *interfaz* de inserción de comandos, y visualizado de resultados, agradable y cómodo de usar por un ser humano, y de ser posible, bajo un entorno gráfico.
3. En el caso de que los dos agentes que participan sean autónomos se dará la opción usar el interfaz visual del tablero, o de ejecutar sin visualización gráfica acelerando de esta forma su ejecución. De igual forma se deberá permitir la creación de lotes de partidas entre agentes autónomos para poder obtener resultados comparativos tras un número significativo de enfrentamientos.
4. En cualquier caso se dará la posibilidad de guardar la actividad y resultados de una partida en un log para su futuro estudio comparativo. En el log se guardarán datos como tipo de búsqueda que usaban los agentes, duración en tiempo en media de los movimientos y las partidas, número de partidas ganadas o perdidas por cada agente, o número de turnos necesarios para la victoria/derrota, etc.

5. Como la principal finalidad es el estudio comparativo de varios algoritmos de búsqueda, y su rendimiento en la resolución de una partida de Omweso, se deberá permitir seleccionar entre varios algoritmos de búsqueda, pudiendo ser diferentes para cada uno de los agentes que participan en la partida.
6. Para los agentes controlados por IA se buscarán y estudiarán posibles funciones de evaluación que les permitan jugar más eficientemente y con más posibilidades de ganar la partida. La forma de evaluar el rendimiento de las funciones de evaluación encontradas será enfrenar a dos jugadores agentes de IA usando diferentes funciones de evaluación durante una serie de partidas y comprobar los resultados.

Capítulo 4

Desarrollo

4. Desarrollo.

Para el análisis y diseño del proyecto utilizaremos UML⁴. Como vista general de lo que ofrece UML, decir que UML se ha convertido en el estándar del mercado, su utilización es independiente del lenguaje de programación y de la complejidad o naturaleza de los proyectos, ya que UML ha sido diseñado para modelar cualquier tipo de sistema o proyecto tanto informático como de cualquier otra disciplina.

Con UML se supera la tradicional visión estática que nos darían metodologías más tradicionales, ampliándola a una visión más dinámica. Es decir gracias a UML no sólo tendremos representados todos los elementos del sistema, sino que podremos representar todo lo que ocurre cuando el sistema está en movimiento, dándonos de esta forma una visión global mucho más completa del sistema. De esta forma todos los problemas derivados del comportamiento del sistema que no eran descubiertos hasta la fase de implantación podrán ahora ser identificados y resueltos en la temprana fase de diseño.

En UML existen bastantes diagramas, varios de ellos indicados para la representación estática del sistema como el diagrama de clases, el diagrama de objetos, diagrama de componentes, diagrama de despliegue o el diagrama de casos de uso, y otros que permiten la representación dinámica del sistema como son el diagrama de secuencia, el diagrama de estados o el diagrama de actividades.

Como se puede ver UML es tan potente que ofrece tantos diagramas que en muchos casos pueden llegar a ser excesivos. Por eso UML permite definir solamente los necesarios, ya que no todos los diagramas son necesarios para todos los proyectos. Algunos proyectos sencillos se podrán representar con sólo un par de diagramas, mientras que algunos proyectos grandes y complejos necesitaremos definir más diagramas para poder representarlos en toda su magnitud.

4 *UML: Unified Modeling Language*

En nuestro caso definiremos los siguientes diagramas:

- a) **Diagrama de casos de uso:** Nos permitirá representar los casos de uso, los actores y sus relaciones, mostrando que pueden hacer los actores y las relaciones que existen entre las acciones (casos de uso). Este diagrama nos permitirá modelar el comportamiento del sistema.

- b) **Diagrama de clases:** Podremos definir la vista estática del sistema, las clases, los interfaces, colaboraciones y sus relaciones.

4.1. Diagrama de casos de uso.

El diagrama de casos de uso nos permitirá representar el comportamiento del sistema. Este diagrama es una buena herramienta para la representación de los requisitos del sistema, pues representa el comportamiento del sistema en el mundo real, alejándose del obscuro mundo de la implementación, por lo que suele ser una buena herramienta mediadora en la toma de requisitos.

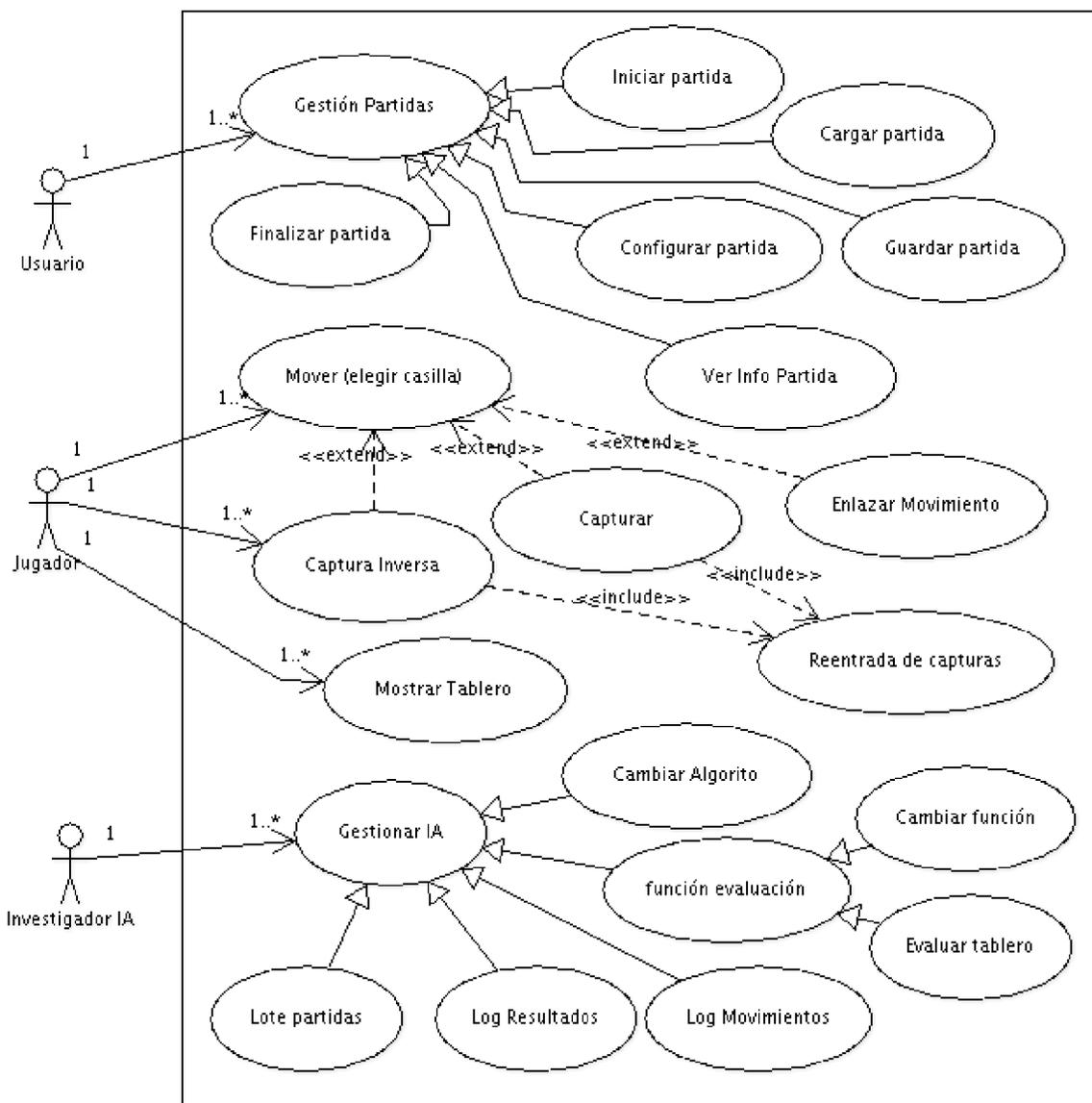


Diagrama 1: Diagrama de casos de uso

En los siguientes apartados de este capítulo intentaremos describir cada uno de los casos de uso, describiendo los actores participantes, las precondiciones que se han de dar, el escenario y las postcondiciones.

4.1.1. Gestión de partidas.

En este capítulo describiremos los casos de uso relacionados con la gestión de partidas. La gestión de partidas se puede dividir en estos casos de uso: Iniciar partida, cargar partida, guardar partida.

- **Caso de uso [01]: Iniciar partida.**

Descripción: El sistema iniciar una partida nueva, creando los jugadores, el tablero y todos los elementos de la partida de acuerdo a unos valores por defecto, o a los valores proporcionados por el usuario.

Actores: Usuario

Precondiciones:

- La partida aún no ha sido creada.

Escenario básico:

- Se piden los nombres de los dos jugadores.
- Se configura la IA de los jugadores, algoritmo de búsqueda y función de evaluación.
- Se solicita el número de columnas del tablero. Entre 3 y 8 columnas.
- Se crea el tablero con las semillas distribuidas según la configuración por defecto.

Postcondiciones:

- Tablero creado, casillas creadas y semillas distribuidas.
- Dos jugadores creados de acuerdo a las configuraciones.

- **Caso de uso [02]: Cargar partida.**

Descripción: El sistema cargará desde el disco una partida previamente guardada, restaurando el estado del tablero, jugadores, etc. al mismo estado en el que estaban cuando se guardó la partida. El formato del archivo debe ser legible por un humano.

Actores: Usuario

Precondiciones:

- Existe una partida previamente guardada.

Escenario básico:

- Si ya está iniciada la partida, avisar de que se perderá el estado actual.
- Seleccionar el fichero que contiene la partida guardada.
- Cargar los datos generales de la partida, el tablero y los jugadores.
- Establecer el turno al jugador que lo tenía cuando se grabó la partida, continuar juego.

Postcondiciones:

- Nuevo estado de todos los objetos de la partida establecido.
- Turno establecido.

- **Caso de uso [03]: Guardar partida.**

Descripción: El sistema guardará todos los datos y estados de todos los componentes del sistema, tablero, jugadores, distribución de las semillas, y se guardarán en un fichero que nos permitirá en un futuro recuperar la partida en el punto que se guardó. El formato del archivo debe ser legible por un humano.

Actores: Usuario

Precondiciones:

- Existe una partida en curso.

Escenario básico:

- Solicitar el nombre de la partida a guardar.
- Guardar distribución de las semillas en el tablero.
- Guardar configuración de los jugadores.
- Guardar información de la partida: turno, etc.

Postcondiciones:

- Partida guardada.
- Se puede continuar con la partida si se desea.

- **Caso de uso [04]: Configurar partida.**

Descripción: Permitirá definir los contenidos de las celdas definiendo la situación de las semillas en el tablero.

Actores: Usuario

Precondiciones:

Existe una partida recién creada o una partida en curso.

Escenario básico:

- Muestra un tablero con el estado actual en el caso de partida en curso.
- Muestra un tablero con las semillas en las posiciones predeterminadas si la partida es nueva.
- Se permite cambiar la ubicación de las semillas.

Postcondiciones:

- El tablero queda configurado con la nueva situación de las semillas.
- La partida puede continuar normalmente.

- **Caso de uso [05]: Finalizar partida.**

Descripción: Permitirá acabar manualmente una partida.

Actores: Usuario

Precondiciones:

- Existe una partida en curso.

Escenario básico:

- Solicitar confirmación de finalización de partida, alertando de la pérdida de datos.

Postcondiciones:

- No existe partida inicializada en curso.
- Nos permite acceder a las funciones que requieres que no haya partidas iniciadas.

4.1.2.Caso de uso : Mover. Seleccionar celda.

Caso de uso [06]: Mover. Seleccionar celda.

Descripción: El jugador seleccionará una casilla para su próximo movimiento. El sistema procederá a realizar el movimiento de siembra originado en la casilla elegida por el jugador.

Actores: Jugador.

Precondiciones:

- Es el turno del jugador.
- La casilla seleccionada pertenece al lado del tablero del jugador.
- La celda seleccionada no está vacía, y tiene más de una semilla.

Escenario básico:

- Empieza el turno del jugador.
- El jugador elige una casilla para su siguiente movimiento.
- El sistema realiza el movimiento iniciado por jugador.
- Se trata de un movimiento de siembra en sentido contrario a las agujas del reloj.
- El sistema realiza cualquier movimiento encadenado o captura que se genere.

- El sistema comprueba la condición de victoria.
- Cambiar el turno al otro jugador.

Postcondiciones:

- Ha cambiado el contenido de varias celdas del tablero.
- Comienza el turno del contrario.

4.1.3.Caso de uso: Enlazar movimiento.**Caso de uso [07]: Enlazar movimiento.**

Descripción: Un movimiento genera unas condiciones que permiten continuar el turno del jugador realizando otro movimiento partiendo de las condiciones que ha dejado el movimiento anterior.

Actores: Sistema

Precondiciones:

- Se acaba de realizar un movimiento del jugador.
- La celda en la que terminó dicho movimiento no estaba vacía.

Escenario básico:

- Se cogen todas las semillas que hay en la casilla en la que acabó el movimiento precedente.
- En el movimiento también se incluye la última semilla que finalizó el movimiento precedente.
- El sistema genera otro movimiento originado en dicha casilla.
- El sistema realiza cualquier movimiento encadenado o captura que se genere.
- Cambiar el turno al otro jugador.

Postcondiciones:

- Ha cambiado el contenido de varias celdas del tablero.
- Comienza el turno del contrario.

4.1.4.Caso de uso: Capturar fichas.

Caso de uso [08]: Capturar fichas.

Descripción: Un movimiento del jugador genera una captura de semillas, dichas semillas pasan al lado del tablero del jugador desde el lado del tablero del contrario.

Actores: Jugador.

Precondiciones:

- Es el turno del jugador.
- Se acaba de realizar un movimiento del jugador.
- El movimiento acaba en una celda no vacía.
- Las dos celdas del contrario que se encuentran justo enfrente están ocupadas.

Escenario básico:

- La celda donde empezó el movimiento que genera la captura se marca como inicio para la futura reentrada de semillas.
- Se vacían las dos celdas del contrario que están enfrente de donde acaba el movimiento.
- Se genera un caso de uso de reentrada de capturas.

Postcondiciones:

- Se han vaciado dos celdas del lado del tablero del Contrario.

4.1.5.Caso de uso: Reentrada de capturas.

Caso de uso [09]: Reentrada de capturas.

Descripción: Tras generarse una captura, las semillas eliminadas de las celdas del lado de tablero del contrario, no se eliminan sino que entran de nuevo en el tablero, en el lado del jugador que ha realizado la captura.

Actores: Sistema

Precondiciones:

- El jugador que conserva el turno ha generado una captura.
- El sistema ha llevado a cabo la captura y vaciado las celdas del contrario implicadas.
- Se ha marcado como celda de inicio de la reentrada la casilla donde comenzó el movimiento que originó la captura.

Escenario Básico:

- El sistema realiza un sistema de siembra usando las semillas capturadas del contrario.
- La primera semilla se siembra en la celda marcada como inicio de la reentrada de capturas.
- Se comprueba si la siembra de reentrada genera un movimiento encadenado u otra captura.
- Se ejecuta cualquier movimiento encadenado o captura que se pueda generar.
- Se cambia el turno al contrario.

Postcondiciones:

- Ha cambiado el contenido de varias celdas del tablero.
- Se ha cambiado el turno.

4.1.6.Caso de uso: Captura inversa.**Caso de uso [10]: Captura inversa.**

Descripción: En el caso de unas condiciones especiales el jugador durante su turno puede elegir hacer un movimiento en sentido contrario al habitual, con la condición de que genere una captura.

Actores: Jugador.

Precondiciones:

- Es el turno del jugador.
- El jugador inicia un movimiento desde las cuatro celdas más a la izquierda de su lado del tablero.
- Desde esa celda, un movimiento de siembra en sentido de las agujas del reloj genera una captura.

Escenario Básico:

- El jugador elige una celda para iniciar el movimiento durante su turno.
- Si existe la posibilidad de captura inversa, el sistema la ejecuta.
- La captura inversa genera una reentrada de capturas, y puede que otros movimientos o capturas.

Postcondiciones:

- Varias celdas del tablero han cambiado su contenido.
- Las celdas del Contrario implicadas en la captura han sido vaciadas.

4.1.7.Caso de uso: Mostrar tablero.**Caso de uso [11]: Mostrar tablero.**

Descripción: Se podrá solicitar mostrar por pantalla gráficamente (aunque en modo texto) el estado actual del tablero. También se mostrará información de la partida.

Actores: Jugador.

Precondiciones:

- Existe una partida en curso.
- Es el turno del jugador.
- El jugador solicita visualizar el estado del tablero.

Escenario Básico:

- El jugador durante su turno solicita visualizar el tablero.
- Se dibuja el tablero (en modo texto), representando la situación de las semillas en las casillas.
- Se muestra información adicional, sobre los jugadores y el turno.

Postcondiciones:

- Se ha mostrado la información sobre el tablero, la partida, los jugadores y los turnos.
- La partida continua normalmente.

4.1.8. Gestionar IA.

Durante este capítulo describiremos los casos de uso relacionados con la gestión de la Inteligencia Artificial del sistema. Principalmente serán servicios utilizados por el Investigador de IA.

(a) Caso de uso [12]: Cambiar algoritmo.

Descripción: Permitirá cambiar el algoritmo de búsqueda usado por un jugador.

Actores: Investigador IA

Precondiciones:

- Partida iniciada y jugadores creados.

Escenario básico:

- Se selecciona uno de los jugadores.
- Se selecciona un algoritmo nuevo.
- Al asignarle un algoritmo un jugador humano se convertirá en jugador autónomo con IA.
- Si a un jugador IA se le desvincula de un algoritmo, se convertirá en un jugador humano.

Postcondiciones:

- El jugador quedará vinculado al nuevo algoritmo de búsqueda.
- La partida continuará normalmente.

(b) Caso de uso [13]: Cambiar función de evaluación.

Descripción: Permitirá cambiar la función de evaluación, seleccionándola entre unas dadas.

Actores: Investigador IA

Precondiciones:

- Al menos uno de los jugadores es un jugador IA, utiliza un algoritmo de búsqueda.

Escenario básico:

- Se selecciona un jugador de IA
- Se selecciona la nueva función de evaluación a usar.

Postcondiciones:

- El jugador IA en evaluaciones futuras utilizará la nueva función de evaluación.
- La partida continuará normalmente.

(c) Caso de uso [14]: Evaluar tablero.

Descripción: Nos permitirá evaluar el tablero con cualquiera de las funciones de evaluación disponibles.

Actores: Investigador IA.

Precondiciones:

- Existe una partida en curso.

Escenario básico:

- Se selecciona una función de evaluación.
- Se evalúa el tablero indicando la evaluación para cada uno de los jugadores.

Postcondiciones:

- No ha cambiado ninguna condición, ni estado de ninguno de los componentes.
- La partida continua normalmente.

(d) Caso de uso [15]: Lote partidas.

Descripción: Permitirá configurar y ejecutar un lote de varias partidas automatizadas.

Actores: Investigador IA.

Precondiciones:

- Existe una partida definida (empezada o no).
- Los jugadores deben ser jugadores autónomos de IA.

Escenario básico:

- Se solicita el número de partidas secuenciales que se quieren jugar.

Postcondiciones:

- Se ejecutarán el número de partidas especificadas, usando la configuración de la partida actual.
- Si se ha activado log de resultados, se habrá generado un archivo de log.

(e) Caso de uso [16]: Log Resultados.

Descripción: Permite activar un log en el que se guardarán los resultados de cantidad de movimientos, derrotas, victorias, etc., de un lote de varias partidas, o de una partida individual.

Actores: Investigador IA

Precondiciones: No existen.

Escenario básico:

- Se solicita el nombre del archivo de log.
- Se puede activar o desactivar la función de grabar al log.
- Al finalizar la partida se guardan las estadísticas de la partida.

Postcondiciones:

- Se ha generado un archivo de log con los resultados de las partidas.

(f) Caso de uso [17]: Log Movimientos.

Descripción: Permitirá visualizar los movimientos realizados durante la partida, así como volver al movimiento anterior, o pasar al movimiento siguiente en caso de que existan. El log de movimientos se guardará cuando se guarde una partida como parte del archivo de guardado de la partida.

Actores: Investigador IA.

Precondiciones: Para ver el log, la partida debe haber comenzado.

Escenario básico:

- La partida ha comenzado.
- Al menos uno de los jugadores realiza un movimiento.
- Al solicitar el log se mostrará la lista de movimientos realizados.
- Se destacará el movimiento actual.
- Podremos volver al movimiento anterior, o ir al movimiento siguiente.

Postcondiciones:

- Se ha mostrado el listado de movimientos, marcando el actual.
- Se ha ido al movimiento anterior o siguiente en caso de solicitarlo.

4.2. Diagrama de clases.

El diagrama de clases nos permitirá representar las clases del sistema con sus relaciones estructurales y de herencia. Las definiciones de clases incluye definiciones para atributos y operaciones, pero para reducir la representación gráfica del sistema nos limitaremos a representar sólo las clases y los atributos que participan en las relaciones.

Así nos podría quedar una representación del sistema como la siguiente:

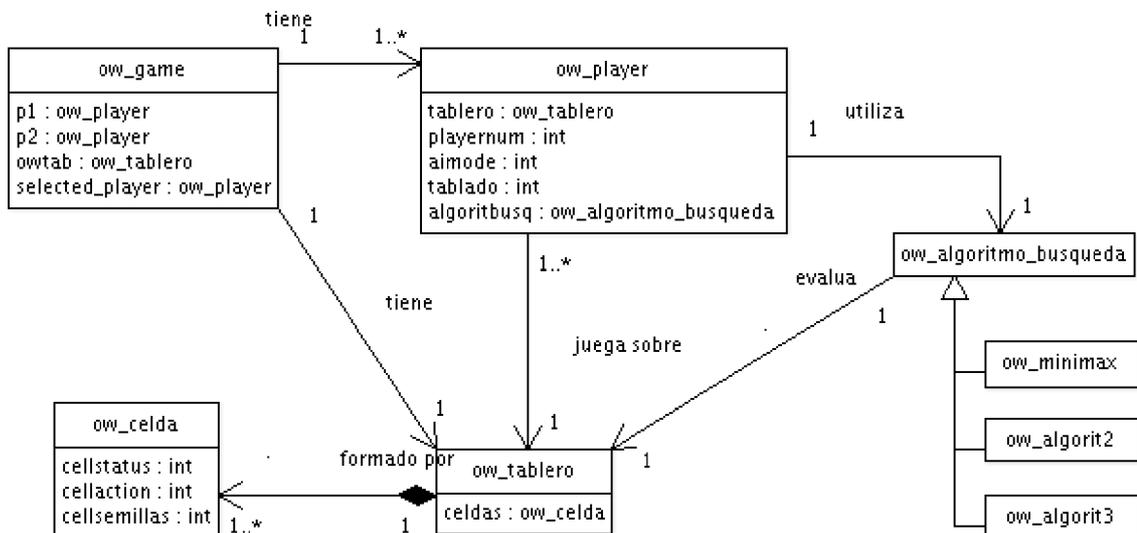


Diagrama 2: Diagrama de clases (simplificado)

Como se puede ver en el diagrama el sistema se puede representar con cinco clases principales: *ow_game*, *ow_player*, *ow_tablero*, *ow_celda* y *ow_algoritmo_búsqueda*.

Pasaremos a describir cada una de ellas:

Clase *ow_game*: Esta clase mantendrá la información general sobre la partida y gestionará los aspectos propios de la misma, como puntuaciones, jugadores que participan, tablero sobre el que juegan, etc. Por lo tanto vemos que tiene relaciones con las clases *ow_player* y *ow_tablero*, teniendo de esta forma acceso a toda la información de la partida.

Clase *ow_tablero*: Como podemos ver, la clase *ow_tablero* podría considerarse principal. Todas las demás clases tienen relaciones con ella. Por supuesto la clase *tablero* almacenará la información de estado de las celdas lo cual se representa por la relación de composición “formado por”, en la que se ve que el *tablero* contiene las 32 celdas. Además podemos ver que el resto de las clases no tienen relaciones directas con las celdas (*ow_celda*) sino que se relacionan indirectamente a través de la clase *ow_tablero*. La clase *ow_tablero* deberá conocer e implementar las reglas del juego: movimientos, capturas, movimientos no permitidos, etc.

Clase *ow_celda*: Guardan la información de estado de cada una de las casillas del teclado, y ofrecen operaciones básicas de manipulación de los contenidos de las casillas. Como hemos dicho esta clase no es accesible directamente sino que deberá ser accedida a través de la clase *ow_tablero* con la que tiene una relación de composición, y que proporcionará operaciones al respecto.

Clase *ow_player*: Esta clase representa a los jugadores de la partida. Como se puede ver representado la clase *ow_player* tiene relaciones con la clase *ow_tablero* y la clase *ow_algoritmo_búsqueda*. La clase utilizará su relación con la clase *ow_algoritmo_búsqueda* para decidir las acciones, movimiento etc. más convenientes durante su turno en la partida, y llevará a cabo dichas decisiones sobre el *tablero* utilizando su relación con la clase *ow_tablero*, y las operaciones proporcionadas por ésta.

Clase *ow_aisearch*: Esta clase será utilizada por la clase *ow_player* para tomar decisiones sobre las acciones más adecuadas durante la partida. Como se puede ver cada jugador de clase *ow_player* solamente utilizará un algoritmo de búsqueda de entre los múltiples que dan origen a las clases de especialización marcadas como *ow_minimax*, *ow_algorit2*, *ow_algorit3*. Durante la implementación del sistema se implementarán varios algoritmos, entre ellos alguna versión adaptada de minimax (negamax, Alfabetá, scout).

Clase *ow_aisearch_simple*: Esta clase extiende la clase *ow_aisearch* implementando un algoritmo de búsqueda simple. El algoritmo evaluará los posibles movimientos desde la posición actual y elegirá el mejor de acuerdo con la función de evaluación que se establezca. Este algoritmo se ha denominado como simple porque la búsqueda no se realiza en profundidad.

Clase *ow_aisearch_negamax*: Esta clase extiende la clase *ow_aisearch* implementando un algoritmo de búsqueda de tipo negamax. Permite la configuración de distintas funciones de evaluación, así como la selección del nivel máximo de profundidad de búsqueda deseado.

Clase *ow_aisearch_alphabeta*: Esta clase extiende la clase *ow_aisearch* implementando una poda alfabeta sobre un algoritmo de tipo negamax, reduciendo de esta manera el número de nodos del árbol de búsqueda y por tanto el consumo de recursos siendo el principal el tiempo de búsqueda. Permite configurar la profundidad de búsqueda, y elegir entre diferentes funciones de evaluación.

Clase *ow_aisearch_scout*: Esta clase extiende la clase *ow_aisearch* implementando un algoritmo de tipo scout, implementando una sonda de búsqueda en profundidad sobre el primer nodo antes de empezar con la búsqueda Alfabetica, basada en un negamax. Permite la configuración de la profundidad de búsqueda y de la función de evaluación utilizada.

4.4. Búsqueda de la función de evaluación.

Los algoritmos de búsqueda utilizan lo que se conoce como función de evaluación para decidir si una jugada es buena o mala, y en consecuencia elegir las jugadas buenas que llevan a la victoria.

Función de evaluación: Función que evalúa una posición estática del tablero de juego, evaluando lo que encuentra (contenidos de las celdas del tablero, y posiblemente otros aspectos) y calculando un valor que representa lo beneficiosa o perjudicial que es dicha posición del tablero para un jugador concreto.

Por supuesto es fácilmente comprensible, que no existen funciones de evaluación predeterminadas, y que para cada juego unas funciones de evaluación pueden representar el estado del juego mejor o peor.

Así pues, el reto consiste en buscar la mejor función de evaluación que seamos capaces de encontrar, que permita al algoritmo de búsqueda seleccionar las mejores jugadas y de esta manera ganar la partida, gracias al uso de un algoritmo de búsqueda.

Generalmente el proceso consiste en encontrar una serie de **características**, como pueden ser la cantidad de fichas, situaciones de las fichas, concentración de las fichas en una sola casilla o dispersas, y cualquier otra característica que se nos ocurra, valorarlas de forma numérica según sea mejor estratégicamente para ganar la partida y combinarlas para obtener una única valoración, una única evaluación, que nos permita determinar si la posición actual es favorable o no.

Expresado matemáticamente, las funciones de evaluación lineales son de la siguiente forma:

$$F(x) = \sum_{i=1}^k \omega_i f_i = \omega_1 f_1 + \omega_2 f_2 + \dots + \omega_k f_k$$

Donde f_i son los valores de las características que hemos tenido en cuenta y ω_i simplemente son unos coeficientes/pesos que hemos dado a cada una de las características para expresar que unas características tienen más importancia o más influencia en el resultado final.

El único límite que existe es nuestra imaginación, y pericia, en encontrar y combinar características que nos permitan encontrar la mejor función de evaluación, en definitiva que nos permita seleccionar las mejores jugadas para ganar el juego.

Solamente decir que suele ser habitual otro límite en la elección de una función de evaluación más allá de nuestra imaginación, que son los recursos limitados. En muchas ocasiones no tenemos memoria suficiente, o no queremos que la toma de decisión de qué jugada elegir nos lleve una cantidad de tiempo poco práctica.

De esta manera en la mayoría de los casos tendremos que elegir entre una solución de compromiso, entre una función de evaluación muy precisa y que el consumo de recursos no sea abusivo.

4.3.1. Evaluación de la diferencia de material.

Una de las primeras características que se nos pueden ocurrir para evaluar la situación de una partida suele ser la diferencia de material.

La diferencia de material, es algo tan sencillo como la diferencia entre la cantidad de fichas del juego que posee uno de los jugadores y el otro. Generalmente el jugador en posesión de un mayor número de fichas, el jugador con más material en un momento de la partida, se supone que tiene una situación más favorable en ese momento del juego.

De esta manera podríamos definir la diferencia de material de la siguiente manera:

$$F(x) = \frac{(\zeta_1 - \zeta_2)}{(1 + \zeta_1 + \zeta_2)}$$

Siendo f_i la función que evalúa la diferencia de material, y siendo ζ_1 y ζ_2 , el número de fichas que poseen el jugador 1 y el jugador 2 respectivamente en ese momento de la partida.

Como podemos ver el numerador simplemente nos calcula la diferencia aritmética entre el número de fichas de uno y otro jugador. Pero obviamente no es lo mismo una diferencia de 1 ficha cuando las cantidades son pequeñas que cuando son grandes.

Por ejemplo, para valores de ζ_1 y ζ_2 de 5 y 10 respectivamente, la diferencia es la misma que para valores 105 y 110, por ejemplo, pero en el primer caso ζ_2 es el doble que ζ_1 al tratarse de números muy pequeños, pero el numerador con la diferencia aritmética simple nos los evaluaría como iguales, cuando en realidad el primer caso es mucho más favorable para el jugador 2.

Para compensar este efecto utilizaremos como denominador $1 + \zeta_1 + \zeta_2$ lo que nos permitirá obtener un resultado válido en ambos casos (el 1 que aparece en el denominador evitará los casos de división por cero).

4.3.2. Evaluación de la dispersión del material.

Como hemos comentado anteriormente, una función de evaluación que solamente evalúa la diferencia de material, no sería capaz de tomar ninguna decisión sobre que jugada escoger en el caso de que ninguna de las jugadas disponibles implique una captura y por tanto un cambio en la posesión del material.

Posiblemente pues, la mejor opción sería añadir alguna nueva característica a nuestra función de evaluación $F(x)$ que combinada con la característica de la diferencia de material sea capaz de tomar decisiones en esas situaciones en las que no existen capturas.

La dispersión del material en el tablero puede ser una buena característica a tener en cuenta. Como comentábamos la búsqueda de nuevas características interesantes puede ser más cosa de creatividad y sentido común que de un proceso matemático, así pues podemos justificar el uso de la dispersión en nuestra función de evaluación haciendo un par de razonamientos de sentido común:

- (a) Cuanto más disperso este el material, más jugadas tendremos disponibles.
- (b) Cuanto más concentrado, más material nos pueden capturar de una sola vez.

La dispersión la mediremos por columna, en lugar de por celda, porque en Omweso las capturas capturan todas las fichas de la columna del contrario, y la posibilidad de capturas es lo que queremos evaluar.

Una vez justificada la introducción de la dispersión en nuestra $F(x)$ pasemos a su interpretación matemática.

Para calcular la dispersión del material utilizaremos la varianza muestral, que está relacionada con la desviación típica o estándar (σ). La varianza muestral nos medirá, sabiendo cual es la cantidad media de material por columna del tablero, cuanto se separa

de esta media los contenidos que tenemos en las columnas en el momento determinado del juego que estamos evaluando.

$$\sigma_i = \sqrt{\frac{1}{n}(c_i - \mu)^2} \quad \sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (c_i - \mu)^2}$$

Donde c_i es la cantidad de material en una columna del tablero, y n es la cantidad de columnas de dos celdas que tiene el tablero de Omweso. A su vez μ es el promedio o media aritmética de la cantidad de material en las columnas, calculada de la siguiente manera:

$$\mu = \frac{1}{n} \sum_{i=1}^n c_i$$

Donde, como ya hemos descrito, n es el número de columnas del tablero, y c_i es la cantidad de material que hay en cada una de las columnas del tablero.

Vamos a definir nuestra f_i que nos evaluará el tablero atendiendo a la dispersión de la siguiente manera:

$$f_i = \frac{(\zeta_1 - \zeta_2)}{(1 + \zeta_1 + \zeta_2)} \quad \zeta_{k=1,2} = \sigma_{max} - \sigma_{k=1,2}$$

Es decir, como podemos ver, calcularemos ζ_i para cada jugador como la diferencia entre la dispersión máxima y la dispersión de las fichas en poder del jugador. Una vez calculadas, compararemos ζ_1 y ζ_2 usando una expresión equivalente a la definida en el capítulo [4.3.1. Evaluación de la diferencia de material] para compararlas de una forma eficiente compensando la peculiaridades matemáticas descritas en dicho capítulo.

Ya hemos visto como calcular la dispersión del material para una posición concreta del tablero, pasaremos pues a calcular cuando valdría nuestra σ_{\max} para poder incluirla en el cálculo de nuestra función de evaluación.

$$\sigma_{\max} = \sigma_{(0,0,\dots,0,8n)}$$

Lo que es lo mismo, la σ_{\max} se dará cuando todas las fichas del tablero esten concentradas en una sola columna, tanto las del jugador como las del contrincante ($8n$ fichas totales).

En una partida de Omweso, la distribución de fichas al comienzo de la partida es de 4 fichas por cada columna que tenga el tablero, o lo que es lo mismo, 32 semillas (fichas) cada jugador, para un tablero estándar de 8 columnas.

Por tanto para un número de columnas variable n , cada jugador tendrá $4n$ fichas, por lo que el número total de fichas en el tablero será de $8n$, al sumar las de los dos jugadores.

Justificada la concentración de las $8n$ semillas todas en la misma columna del tablero, pasemos a calcular la σ_{\max} para esta distribución extrema.

$$\sigma_{\max} = \sigma_{(0,0,\dots,0,8n)} = \sqrt{\frac{1}{n} [(n-1)(0-\mu)^2 + (1)(8n-\mu)^2]}$$

Como se puede ver, simplemente hemos expresado que si n es el número de columnas, $(n-1)$ veces tendremos 0 fichas en la columna y (1) vez tendremos $8n$ fichas en la columna que soporta toda la concentración de fichas.

Calcularemos el promedio μ para intentar simplificar un poco más la expresión de σ_{\max} quedando las siguientes expresiones:

$$\mu = \frac{1}{n} \sum_{i=1}^n c_i = \frac{1}{n} (8n) = 8$$

$$\sigma_{(0,0,\dots,0,8n)} = \sqrt{\frac{1}{n} [(n-1)(0-\mu)^2 + (1)(8n-\mu)^2]} = \sqrt{\frac{1}{n} [64(n-1) + 64(n-1)^2]}$$

Simplifiquemos un poco más la expresión, sacando factor común y eliminando los factores redundantes:

$$\sigma_{(0,0,\dots,0,8n)} = \sqrt{\frac{1}{n} [64(n-1) + 64(n-1)^2]} = \sqrt{\frac{1}{n} 64(n-1)(1+(n-1))} = \sqrt{64(n-1)}$$

Con lo que llegamos a una expresión bastante sencilla para la σ_{\max} que es la siguiente:

$$\sigma_{\max} = \sigma_{(0,0,\dots,0,8n)} = 8\sqrt{(n-1)}$$

Después de todos estos cálculos, solamente recordar que todo esto lo estábamos haciendo para insertar una componente de la dispersión en nuestra función de evaluación, así que es buen momento para ver como quedaría nuestra f_i tras incorporar la nueva expresión que hemos encontrado.

$$f_i = \frac{(\zeta_1 - \zeta_2)}{(1 + \zeta_1 + \zeta_2)}$$

$$\zeta_{k=1,2} = \sigma_{\max} - \sigma_{k=1,2} = 8\sqrt{(n-1)} - \sigma_{k=1,2}$$

4.3.3. Evaluación de la movilidad.

Entendiendo por movilidad la evaluación de la cantidad de movimientos disponibles para el jugador durante su turno, y teniendo en cuenta que el juego del Omweso termina cuando uno de los dos jugadores no puede realizar ningún movimiento legal, parece interesante introducir alguna componente que mida la movilidad en nuestra función de evaluación.

Parece obvio que nuestra función de evaluación deberá devolver un valor que indique la victoria o derrota cuando uno de los jugadores ya no tiene ningún movimiento legal disponible para elegir, pero adicionalmente a esto, estudiemos si el tener más movimientos disponibles es indicativo de una situación favorable durante el juego.

Sin nada más que una reflexión lógica, y a falta de comprobarlo empíricamente, parece lógico que si tenemos más jugadas disponibles para elegir será más probable que encontremos jugadas favorables, posiblemente que impliquen captura de fichas enemigas, y a su vez reducir el número de jugadas disponibles para el jugador enemigo, y con un poco de suerte las jugadas favorables para él.

Así pues podemos definir la diferencia de movilidad de la forma ya habitual descrita anteriormente para otras características:

$$f_i = \frac{(\zeta_1 - \zeta_2)}{(1 + \zeta_1 + \zeta_2)}$$

Calculando con f_i , la diferencia de movilidad entre ambos jugadores del juego, y siendo por tanto, en este caso, ζ_1 y ζ_2 las variables que miden la cantidad de jugadas disponibles que tienen el jugador 1 y el jugador 2, respectivamente, en un momento del juego.

4.3.4. Evaluación del alcance.

Introducimos una nueva característica. La característica la denominaremos alcance, y proviene de la intuición de un jugador humano que tendería a colocarse las fichas de forma que estén dispuestas para con un movimiento caer en casillas con susceptibilidad de captura.

Es una característica con poco trasfondo estadístico pero nos gustaría ver si esa corazonada o intuición, o incluso se podría considerar instinto del jugador humano pudiera dar algún resultado interesante.

Definamos pues como alcance como, sabiendo que para que una captura sea posible el movimiento de siembra deberá acabar en una celda perteneciente a la fila más interior del tablero del lado del jugador como, la cantidad de celdas que tienen una cantidad de semillas tal que si se iniciará un movimiento en dicha casilla el movimiento de siembra acabaría depositando la última semilla que finaliza la siembra en una casilla perteneciente a esa fila más interior.

sólo nos quedará, como ya es habitual con todas las características, comparar evaluaciones de ambos jugadores con la función habitual:

$$f_i = \frac{(\zeta_1 - \zeta_2)}{(1 + \zeta_1 + \zeta_2)}$$

Siendo, en este caso, ζ_1 y ζ_2 las variables que miden el alcance que tienen el jugador1 y el jugador2, respectivamente, en un momento del juego.

Más adelante en el capítulo de resultados, experimentaremos con esta característica viendo tanto su rendimiento individual, como su aportación al usarla en combinación con otras en una función de evaluación más compleja.

4.4.5. Cálculo de los pesos de cada una de las características.

Una vez decididas las características que tendremos en cuenta en nuestra función de evaluación (material, dispersión y movilidad) ya sólo nos queda decidir que peso/importancia daremos a cada una en nuestra función de evaluación.

Por desgracia, decidir qué peso deben tener cada una de las características que influyen en el resultado de una función de evaluación no sólo no es tarea fácil, sino que no existe ningún método de cálculo para elegir los pesos.

$$F(x) = \sum_{i=1}^k \omega_i f_i = \omega_1 f_1 + \omega_2 f_2 + \dots + \omega_k f_k$$

Se trata pues, de un trabajo empírico, durante el que tendremos que ir cambiando los valores de los pesos ω_i , y viendo si los resultados mejoran o empeoran, es decir, ver si nuestro jugador es capaz de ganar más juegos o al menos con menos esfuerzo usando unos determinados pesos para las características usadas en la función de evaluación.

Durante el capítulo de resultados [5.5. *Combinación lineal de las características para la función de evaluación*] calcularemos diferentes pesos para cada una de las características y veremos como dichos pesos influyen en el rendimiento de las búsquedas.

4.4. Entorno de desarrollo.

En este capítulo intentaremos describir el entorno de trabajo que se ha escogido para la realización del proyecto haciendo especial referencia a las decisiones que se han tomado para elegir unas herramientas, lenguajes, sistemas operativos u otros.

Veremos pues que el entorno de desarrollo elegido será : Linux (distribución Ubuntu) como sistema operativo, el lenguaje de programación C/C++, para el análisis y diseño se ha elegido UML usando la herramienta ArgoUML para el modelado, y veremos algunas otras herramientas de apoyo utilizadas como por ejemplo Gimp para la creación y edición de los gráficos usados, o el entorno de desarrollo Kdevelop para la edición del código fuente.

4.4.1. Sistema Operativo: Linux/Ubuntu.

Hoy día podemos elegir básicamente entre tres familias de sistemas operativos. La familia Microsoft Windows, la familia Unix y la familia MacOS. Mientras que MacOS queda relegado a un uso más minoritario, Microsoft Windows y Linux/Unix están ambos bastante extendidos, y utilizados prácticamente por igual dentro del area de servidores, mientras que Windows aún a día de hoy tiene un calado más profundo en el mercado de los ordenadores de uso personal.

Dado que en el entorno universitario, al menos en el área de inteligencia artificial parece usar más habitual Linux, eligiremos este como sistema operativo para desarrollar el proyecto.

Veremos que Linux tiene una buena batería de herramientas para el desarrollador de aplicaciones, y que incluso algunas herramientas tradicionalmente más habituales de Windows, como las del diseño UML podremos encontrar opciones muy buenas en Linux.

Además veremos tanto el lenguaje de programación escogido, como los formatos de la documentación, nos permitira fácilmente portar o ejecutar nuestros programas en varios sistemas operativos, entre ellos Linux y Windows, o leer la documentación en formatos portables, por lo que de cara al usuario final, que pueda encontrar alguna utilidad a este proyecto, no tendra problema de trabajar en otros sistemas operativos.

4.4.2. Análisis y diseño UML: ArgoUML.

Para el análisis y diseño del proyecto utilizaremos UML⁵. Veremos que UML será la metodología idónea para diseñar un sistema orientado a objetos, que nos hará el proyecto más cómodo de modelar y que será comodamente implementado en cualquier lenguaje orientado a objetos.

En Windows existen multitud de herramientas para modelado UML. Para Linux no existen demasiadas pero existen algunas bastante completas y que nos permitirán el modelado de una manera sencilla, y entre ellas hemos escogido ArgoUML⁶.

ArgoUML es opensource, está completamente implementado en Java e incluye soporte para los 9 esquemas especificados en el estándar de UML 1.4. Al estar implementado en Java, puede ejecutarse virtualmente en casi cualquier plataforma que soporte Java 1.4 o superior. Además nos permitirá exportar los diagramas en GIF, PNG, PostScript, PS Encapsulado, PGML, SVG, etc.

ArgoUML ofrece otras muchas funcionalidades como generación de código para Java, C++, C#, PHP4 and PHP5, o incluso un módulo de ingeniería inversa.

⁵ UML: *Unified Modeling Language*

⁶ ArgoUML : <http://argouml.tigris.org/>

4.4.3. Lenguaje de programación: C++/SDL.

El lenguaje elegido para desarrollar el proyecto es C++. Los elementos del modelado UML orientado a objetos podrá ser comodamente implementado en C++ usando sus clases y características. Además, C++ es un lenguaje bastante extendido y del que existen compiladores tanto de pago como gratuitos para muchas plataformas y sistemas operativos, entre los que se encuentran Linux y Windows, por lo que si en algún momento se desea portar el código a otras plataformas tendremos la puerta abierta pues encontraremos las herramientas para hacerlo.

En cuando a SDL, se trata de una librería gráfica que utilizaremos para la representación gráfica del tablero y el resto de los elementos en pantalla. SDL proporciona funciones para manipular graficos, dibujarlos en pantalla, y también proporciona otras muchas funciones multimedia. SDL es una librería opensource, para la que existen versiones para multitud de plataformas y sistemas operativos, lo que junto con C++ nos dejará la puerta abierta a compilar nuestro proyecto en otros entornos de hardware y sistema operativos diferentes.

4.1.4.Herramientas secundarias: Gimp, Kdevelop, doxygen.

Describimos a continuación las herramientas que utilizaremos para la implementación del proyecto:

Gimp: El editor de imágenes más extendido hoy dia para sistema operativo Linux y entorno X. Proporciona una entorno de trabajo muy completo y bastante potente, proporcionando desde las funcionalidades más básicas hasta trabajo con capas, transformación de fotografías, etc. Para el proyecto sólo utilizaremos las funcionalidades más básicas, para crear los gráficos de representación de tablero y resto del juego. También existe versión para Windows.

Kdevelop: Es el entorno de desarrollo para Linux proporcionado por KDE. Es un entorno de desarrollo muy completo, que da soporte a múltiples lenguajes, entre ellos C++. Inicialmente estaba pensado para desarrollar aplicaciones KDE, incluyendo diseño e implementación de interfaces gráficas con los controles estándar de KDE, pero también permite desarrollar aplicaciones gráficas fuera de KDE o incluso aplicaciones en modo texto. De todas las funciones que proporciona el entorno Kdevelop se usarán: sintaxis coloreada, árbol de clases y depuración de aplicaciones integrada.

Doxygen: Es un sistema de generación de documentación. Es capaz de generar documentación sobre las clases y funciones/métodos automáticamente a partir de los comentarios insertados en el propio código. Para ello los comentarios deberán tener un formato reconocido por doxygen. Doxygen es capaz de generar documentación a partir de dichos comentarios en formato LaTeX, RTF (MS-Word), PostScript, PDF con hipervínculos, HTML, o páginas de manual para la herramienta 'man' de Unix.

LeakTracer: Leak Tracer es una herramienta de línea de comandos que permite buscar puntos de nuestro programa donde la memoria utilizada no se esta liberando correctamente. Aunque con los ordenadores que existen hoy dia y con la más que suficiente cantidad de memoria RAM de la que suelen disponer puede parecer algo innecesario, dada la naturaleza de los algoritmos de búsqueda (recursivos y se ejecutan millones de veces) resulta especialmente importante, e incluso crucial, no tener perdidas de áreas de memoria durante las búsquedas. Leak Tracer es una estupenda herramienta, sencilla de utilizar, y rápida que nos permite encontrar estos puntos de nuestro código que no liberan la memoria correctamente.

4.5.Manual de Usuario.

Durante este manual de usuario intentaremos describir todo lo necesario para trabajar con Omweso. Desde los requisitos del sistema, las librerías necesarias y como instalarlas, hasta como ejecutar Omweso, como gestionar partidas, como trabajar con el interfaz de texto y con el interfaz gráfico o como gestionar la IA.

Veremos que Omweso no solamente está desarrollado con la finalidad de jugar partidas, sino que está creado como una herramienta para investigar en aspectos relacionados con técnicas de IA, por lo que nos proporcionará una serie de herramientas adicionales a las funcionalidades de juego básico que nos permitirá utilizar Omweso como una herramienta de investigación de IA.

4.5.1.Requisitos de sistema.

La aplicación Omweso no tiene unos requisitos de hardware específicos, aunque un hardware relativamente moderno ayudará a una ejecución más fluida. Podríamos definir algunos requisitos básicos:

Sistema Operativo: La aplicación Omweso ha sido desarrollada íntegramente bajo sistema operativo *Linux*, concretamente bajo la distribución *Ubuntu*, por lo que todas las pruebas y ejemplos que aparecen en este manual se limitan a dicha distribución *Linux*. No debería ser complicado trabajar de igual manera en cualquier otra distribución *Linux*, y por supuesto más concretamente en cualquier distribución *Debian* gracias a que *Ubuntu* mantiene gran cantidad de cosas en común con cualquier distribución *Debian*. Aunque no se ha probado, debería ser factible compilar y ejecutar Omweso bajo *Microsoft Windows* a partir del código fuente, ya que todas las herramientas de compilación y librerías también se encuentran disponibles para *Microsoft Windows*.

Entorno de ejecución: Omweso, bajo *Linux*, requiere de un sistema con capacidades gráficas corriendo *X11* y con acceso a un ratón, y teclado. Así mismo, Omweso deberá ser ejecutado desde un terminal de texto que será utilizado como interfaz primario de la aplicación, adicionalmente usará un interfaz gráfico *X11* controlado por el ratón.

Librerías de ejecución: Para su correcta ejecución, Omweso requiere que en el sistema existan instaladas las siguientes librerías en su versión de tiempo de ejecución (*runtime*): *libSDL1.2*, *libSDL-image1.2*, *libSDL-ttf2.0* y *libreadline5*. Más adelante, en las secciones [4.5.2.1. *Instalación de la librería GNU Readline*] y [4.5.2.2. *Instalación de las librerías SDL, SDL_image, SDL_ttf*], explicará el proceso de instalación de estas librerías en el sistema.

4.5.2. Instalación del programa.

Durante este capítulo del manual veremos como instalar Omweso. No solamente como instalar la aplicación, sino también como instalar las librerías de ejecución necesarias para que Omweso funcione correctamente.

4.5.2.1. Instalación de la librería GNU Readline.

La aplicación Omweso utiliza, para toda la parte del interfaz a través de terminal de texto, las librerías *Readline* y *History*, ambas incluidas en el paquete *Debian readline5*.

- **Librería *Readline*:** La librería *Readline* permite el desarrollo de aplicaciones con interfaz de línea de comandos. Proporciona funcionalidades para capturar comandos introducidos por el usuario, así como una funcionalidad adicional muy interesante, conocida como completado de comandos (*completion*) muy popular en aplicaciones de línea de comandos como *bash*.

- **Librería *History*:** Librería que suele ir incluida y vinculada inseparablemente a la librería *Readline*, aunque podemos no utilizar sus funcionalidades. Nos permite dotar a la aplicación de un historial de comandos a los que el usuario podrá acceder fácilmente para poder retocarlos y/o volver a ejecutarlos.

En el momento de revisión de este manual la versión de la librería es *libreadline5* pero cualquier versión más actual o posterior debería ser totalmente compatible.

Como comentábamos, Omweso requiere de la librería *Readline*, por lo que procederemos a instalarla en nuestro sistema *Debian/Ubuntu* de la forma habitual utilizando el comando *apt-get*.

```
dtorres@Aiken:~$ sudo apt-get install libreadline5
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
libreadline5 ya está en su versión más reciente.
0 actualizados, 0 se instalarán, 0 para eliminar y 0 no actualizados.
dtorres@Aiken:~$ █
```

Instalamos el paquete *Debian* llamado *libreadline5*, utilizando el comando *apt-get*. El comando *sudo* en sistemas *Ubuntu* nos dará permisos temporales de administrador (*root*) para poder instalar las librerías.

Si lo que deseamos es compilar a partir de los fuentes nuestros propios binarios de Omweso, necesitaremos las versiones de desarrollo de *Readline*, concretamente para *Ubuntu*, necesitaremos el paquete *libreadline5-dev*. Lo instalaremos de igual manera:

```
dtorres@Aiken:~$ sudo apt-get install libreadline5-dev
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
libreadline5-dev ya está en su versión más reciente.
0 actualizados, 0 se instalarán, 0 para eliminar y 0 no actualizados.
dtorres@Aiken:~$ █
```

Adicionalmente, para compilar Omweso a partir del código fuente necesitaremos las versiones de desarrollo de las librerías SDL (se explicará en la siguiente sección como instalarlas) y las herramientas básicas de desarrollo instaladas en el sistema (*sudo apt-get install build-essential*).

4.5.2.2. Instalación de las librerías SDL, SDL_image, SDL_ttf.

La aplicación Omweso utiliza, para toda la parte de representación gráfica (*gfxboard*), las librerías *SDL*⁷ y las extensiones *SDL_image* y *SDL_ttf*.

- **Librería SDL:** La librería *SDL* (*Simple DirectMedia Layer*) es una librería multimedia, multiplataforma escrita en C y *opensource*, que crea una capa abstracta que permite trabajar con gráficos, sonidos, dispositivos de entrada (ratón, teclados) etc. de una forma transparente y abstracta en diferentes plataformas y sistemas operativos.
- **Librería *SDL-image*:** Extensión para la librería *SDL* que permite cargar gráficos para ser utilizados en la aplicación desarrollada con *SDL*, desde formatos de archivos gráficos como *PNG*, *JPEG*, *GIF*⁸ y otros formatos populares.
- **Librería *SDL-ttf*:** Extensión a la librería *SDL* que permite utilizar archivos de fuentes *TTF*⁹ para escribir texto dentro de la aplicación desarrollada con *SDL*.

Si sólo queremos utilizar Omweso solamente necesitaremos tener instaladas las librerías de ejecución (*runtime*) de las librerías especificadas más arriba.

En el momento en que se escribió este manual, las versiones de las librerías *SDL* más recientes son la 1.2, pero cualquier versión posterior superior debería ser totalmente compatible.

7 SDL: Simple DirectMedia Layer.

8 PNG: Portable Network Graphic; JPEG: Joint Photo Experts Group; GIF: CompuServe graphic;

9 TTF: True Type Font

Los nombres de los paquetes *Debian/Ubuntu* de estas librerías son: *libsdl1.2debian*, *libsdl-image1.2* y *libsdl-ttf2.0-0*.

Podremos instalarlas fácilmente, en un sistema *Debian/Ubuntu*, de la siguiente manera. En otros sistemas deberemos buscar el sistema de instalación que utilicen, pero no debería ser muy diferente.

```
dtorres@Aiken:~$ sudo apt-get install libsdl1.2debian libsdl-image1.2 libsdl-ttf2.0-0
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
libsdl1.2debian ya está en su versión más reciente.
libsdl-image1.2 ya está en su versión más reciente.
libsdl-ttf2.0-0 ya está en su versión más reciente.
0 actualizados, 0 se instalarán, 0 para eliminar y 0 no actualizados.
dtorres@Aiken:~$
```

Como vemos utilizaremos el comando *apt-get* (gestor de paquetes *Debian/Ubuntu*) para instalar en un solo comando a la vez las tres librerías requeridas. El comando *sudo* que aparece como prefijo en el comando del ejemplo se utiliza típicamente en *Ubuntu* para dar temporalmente permisos de administrador al usuario y que así tenga permisos para instalar las librerías. En otros sistemas *Debian* usaremos el comando *apt-get* sin *sudo*, pero necesitaremos identificarnos como administrador (*root*) previamente.

Si por el contrario lo que queremos es compilar a partir de los fuentes nuestra propia versión de los binarios de Omweso, necesitaremos tener en el sistema, además de los paquetes básicos de desarrollo (*sudo apt-get install build-essential*), las versiones de desarrollo de las librerías *SDL*.

Así pues instalaremos los paquetes *libsdl1.2-dev*, *libsdl-image1.2-dev* y *libsdl-ttf2.0-dev*, de la misma manera que hemos hecho anteriormente con los paquetes de las librerías de ejecución.

```
dtorres@Aiken:~$ sudo apt-get install libsdl1.2-dev libsdl-image1.2-dev libsdl-ttf2.0-dev
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
libsdl1.2-dev ya está en su versión más reciente.
libsdl-image1.2-dev ya está en su versión más reciente.
libsdl-ttf2.0-dev ya está en su versión más reciente.
0 actualizados, 0 se instalarán, 0 para eliminar y 0 no actualizados.
dtorres@Aiken:~$ █
```

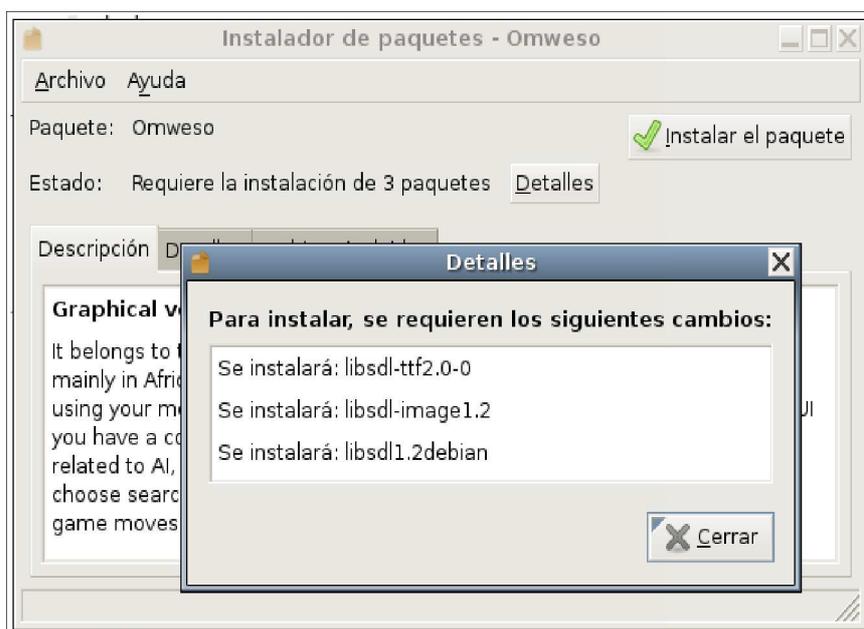
De igual manera que antes, utilizaremos el comando *sudo* sólo en sistemas Ubuntu para obtener temporalmente permisos de administrador. Hacer notar que la primera vez el comando *sudo* nos pedirá nuestra contraseña, pero no nos la pedirá en sucesivas ejecuciones. También podemos ver en los ejemplos que el comando *apt-get* nos avisa si ya tenemos instaladas los paquetes, o incluso se encarga de instalar las librerías adicionales requeridas automáticamente sin que nos tengamos que preocupar por ello.

4.5.2.3. Instalación del programa Omweso.

Para su instalación, la distribución binaria de Omweso se proporciona empaquetada como un paquete *Debian*. Podremos instalarlo fácilmente con cualquiera de las herramientas existentes en sistemas *Debian* destinadas a la gestión de paquetes.

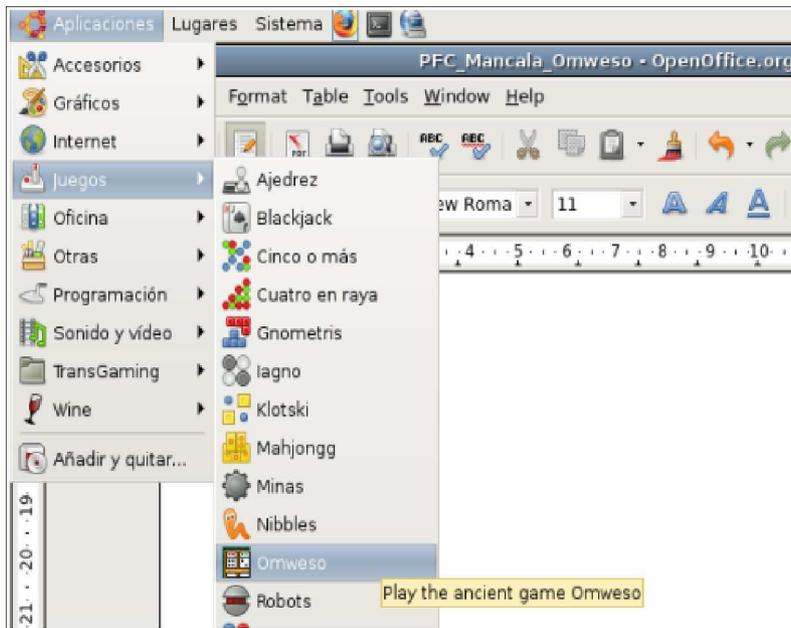
```
dtorres@Aiken:~/PFC$ sudo dpkg -i Omweso_20080622_bin.deb
[sudo] password for dtorres:
Seleccionando el paquete omweso previamente no seleccionado.
(Leyendo la base de datos ...
129020 ficheros y directorios instalados actualmente.)
Desempaquetando omweso (de Omweso_20080622_bin.deb) ...
dpkg: problemas de dependencias impiden la configuración de omweso:
omweso depende de libsdl1.2debian; sin embargo:
 El paquete `libsdl1.2debian' no está instalado.
omweso depende de libsdl-image1.2; sin embargo:
 El paquete `libsdl-image1.2' no está instalado.
omweso depende de libsdl-ttf2.0-0; sin embargo:
 El paquete `libsdl-ttf2.0-0' no está instalado.
dpkg: error al procesar omweso (--install):
problemas de dependencias - se deja sin configurar
Se encontraron errores al procesar:
 omweso
dtorres@Aiken:~/PFC$ █
```

Por ejemplo, podremos instalarlo utilizando por ejemplo el comando `[dpkg -i Omweso_20080622_bin.deb]`, aunque veremos que en el caso de no haber instalado previamente las librerías definidas en el apartado [4.5.2.2.Instalación de las librerías *SDL*, *SDL_image*, *SDL_ttf*.] de este manual recibiremos un error de dependencias, que deberemos solucionar instalando las librerías necesarias según se define en dicho apartado.



Si por el contrario no nos gusta trabajar en la línea de comandos, existen algunos gestores de paquetes con interfaz gráfico, como Synaptic en sistemas *Debian/Ubuntu*, que nos permitirán instalar Omweso de una forma sencilla e intuitiva.

En sistemas Ubuntu, por ejemplo, con sólo hacer doble click sobre el archivo `[Omweso_20080622_bin.deb]`, automáticamente nos abrirá el instalador de paquetes y nos guiará intuitivamente por el proceso de instalación. En el caso de que no hallamos instalado previamente alguna de las librerías requeridas por Omweso, el propio instalador se encargará de instalarlas por nosotros.



Tras finalizar la instalación de forma satisfactoria podremos ver que Omneso está disponible desde el menú Aplicaciones/Juegos de nuestro sistema de escritorio.

4.5.3. Ejecución de Omneso.

Para empezar a utilizar Omneso, simplemente tendremos que, bien desde el directorio donde se instaló, o bien desde cualquier lugar si se instaló en alguna ruta del *path* del sistema, ejecutar desde la línea de comandos, el comando `[omneso]` sin ningún parámetro. En el caso de haberlo instalado en forma de paquete *Debian*, también podremos lanzar Omneso desde el menú Aplicaciones/Juegos de nuestro gestor de ventanas.

```
Omneso v-0.1 Shell Copyright(C) November 2007 by David Torres
Type 'h' at any moment for help.
omneso: /> █
```

Si Omneso se ejecutó correctamente obtendremos el *prompt* parpadeante de la *shell* (interfaz en modo texto) de la aplicación Omneso. Desde esta *shell* podremos manejar en modo texto todas opciones y funcionalidades o incluso jugar a Omneso.

Cabe destacar que podremos conseguir ayuda contextual en cualquier momento, en cualquier menú de la aplicación, con sólo ejecutar el comando *[help]*, o simplemente *[h]* para abreviar.

4.5.4. Gestión de partidas.

Desde la pantalla de inicio de la aplicación, con sólo solicitar la ayuda contextual mediante el comando *[h]*, tendremos acceso al menú principal de la aplicación. Desde aquí podremos gestionar las partidas, así como movernos a los demás menús de gestión de Omweso, como son el menú de jugar (*play*), el menú gestionar la IA (*manageai*), y opciones como salir de la aplicación *[quit]*.

```
APPLICATION MAIN MENU
newgame [new]   create a new game.
loadgame [load] load an existing game.
savegame [save] save the current game.
setup    [setup] edit the cell contents of the current board.
endgame  [end]   end the current game.
play     [p]    continue playing the current game.
manageai [ai]   manage artificial intelligence.
version  [ver]  show program version and build.
help     [h]    show this help screen.
quit     [q]    exit from omweso.

omweso: /> █
```

Vemos que tendremos acceso a cada una de las opciones simplemente escribiendo su nombre en la línea de comandos. También tendremos un acceso más corto especificado en la segunda columna de la ayuda.

Pasemos a describir cada una de la opciones a las que tenemos acceso desde el menú principal de la aplicación:

newgame [new] : Esta opción nos permitirá crear una nueva partida, y nos llevará al submenú de configuración de la partida donde podremos especificar el número de columnas del tablero, nombre de los jugadores, turnos, etc. Las opciones del menú *newgame* se explicarán mas adelante en la sección *[4.5.4.1. Crear una partida nueva]*.

loadgame [load] : Este comando nos permitirá cargar una partida que se guardó previamente y que estaba en mitad de su trascurso. Al cargarse se recuperará la partida en el mismo estado que estaba, con los mismos jugadores, el mismo tablero, mismos turnos, etc. Para cargar una partida simplemente escribiremos en la línea de comandos el comando '*loadgame <savegamefile>*' donde sustituiremos *<savegamefile>* por el nombre del archivo donde previamente se guardó la partida.

savegame [save] : Nos permitirá guardar una partida iniciada, guardando el estado actual del tablero, los jugadores, los turnos, etc., para poder recuperarla en un futuro con el comando [*loadgame*]. Para guardar una partida escribiremos en la línea de comandos el comando '*savegame <savegamefile>*' donde sustituiremos *<savegamefile>* por el nombre del archivo donde deseamos guardar el estado de la partida. Para su correcto funcionamiento los archivos deberán guardarse con la extensión *.sav*, de lo contrario no podrán ser cargados por el comando [*loadgame*].

setup [setup] : Este comando nos permitirá, sobre una partida ya iniciada, mover semillas de una celda a otra del tablero, permitiéndonos de esta manera recrear situaciones y estados del tablero que deseamos estudiar de forma especial. Las opciones del menú *setup* se explicarán más adelante en la sección [4.5.4.4. *Editar los contenidos del tablero*].

endgame [end] : En ningún momento podrán existir dos partidas iniciadas simultáneamente, por lo que siempre que queramos iniciar una nueva partida deberemos antes usar este comando para finalizar la partida actual.

play [p] : Siempre que necesitemos salir del modo juego para realizar alguna función de gestión de la partida, necesitaremos volver al modo de juego usando este comando. El menú [*play*] se explicará más adelante en la sección [4.5.5. *Jugar una partida de Omweso*].

manageai [ai] : Este comando nos permitirá ir al menú de gestión de la IA. Las opciones del menú de IA se explicarán más adelante en la sección [4.5.6. *Herramientas para el investigador de IA*].

version [ver] : Este comando nos mostrará información de la versión de Omweso que estamos ejecutando, fecha de compilación, etc.

help [h] : El comando [help] estará disponible desde cualquier menú de la aplicación y nos dará ayuda contextual del menú en que nos encontremos.

quit [q] : El comando [quit], finalizará la aplicación Omweso regresando al sistema operativo, finalizando cualquier partida que exista en curso. La partida no se grabará automáticamente, así que si deseamos conservarla deberemos usar el comando [savegame] antes de finalizar la aplicación con el comando [quit].

4.5.4.1. Crear una partida nueva.

Para crear una partida nueva, desde el menú principal usaremos el comando [newgame]. Antes de iniciar una partida nueva deberemos confirmar que no existe ninguna partida activa, pues en todo momento sólo podrá existir una única partida iniciada. En caso de existir una partida activa deberemos finalizarla primero usando el comando [endgame], antes de crear la nueva partida [4.5.4.6. *Finalizar una partida prematuramente*].

```
MAINMENU::NEWGAME MENU

showcfg [cfg] show game current configuration.
set      [SET] set/change configuration value.
create  [c]  create new game with current configuration.
help    [h]  show this help screen.
quit    [q]  cancel creating new game, back to mainmenu.

omweso:/newgame> █
```

Describamos las opciones de las que disponemos en este punto:

showcfg [cfg] : Este comando nos mostrará la configuración de la partida que vamos a crear, podremos cambiar las opciones de nuestra partida usando el comando *[set]*.

set [SET] : Podremos cambiar las opciones de nuestra partida antes de crearla con este comando. Nos permitirá cambiar los valores de las variables que gestionan el comportamiento de la partida. Se puede ver una lista de las variables disponibles usando el comando *[showcfg]*. Las opciones de una partida se explicarán más adelante en la sección *[4.5.4.2. Configurar las opciones de la partida]*.

create [c] : Crea una partida nueva usando las opciones establecidas y que nos muestra el comando *[showcfg]*.

help [h] : Muestra la ayuda contextual del menú *[newgame]*.

quit [q] : Si por alguna razón vemos que nos hemos equivocado y/o no queremos crear una partida en este momento, podemos volver al menú principal sin crear ninguna partida usando este comando *quit*.

4.5.4.2. Configurar las opciones de la partida.

Lo más habitual es que la configuración por defecto de una partida no sea óptima para nuestras necesidades. Veremos como podemos configurar las opciones de la partida.

El lugar más habitual donde configurar la partida es en el momento de crearla. De esta manera, desde el menú principal usaremos el comando *[newgame]*. Desde ahí, justo antes de crear la nueva partida, tendremos acceso a la configuración actual con el comando *[showcfg]*, y podremos cambiar la configuración de cada una de las variables de control mediante el comando *[set]*.

Veamos la descripción más detallada de cada una de las variables, y los valores que podremos darlas con el comando *[set]*.

```
omweso:/newgame> cfg
Game configuration:
P1NAME:          player1
P1AI(P1DEPTH):  HUMAN(1)
P1EVAL:          HUMAN
P2NAME:          player2
P2AI(P2DEPTH):  SIMPLE(1)
P2EVAL:          RANDOM
NUMCOLS:        8
FIRSTTURN:      P1
GFXBOARD:       OFF
```

P1NAME : Es la variable que guarda el nombre que utilizará el jugador 1 durante la partida. El valor por defecto es '*player1*'.

P1AI : Es el algoritmo de IA que utilizará el jugador1. Este parámetro trabaja en combinación con *PIEVAL* para definir el comportamiento del jugador durante la partida. El valor por defecto de *P1AI* es '*NEGAMAX*'. Ver más adelante en la sección *[4.5.4.3.Configuración de opciones de IA de los jugadores]* los posibles valores de *P1AI* así como las posibles combinaciones con *PIEVAL*.

P1DEPTH : Es la profundidad de búsqueda usada por el algoritmo de búsqueda definido por *P1AI*. El valor por defecto es 3, un valor aceptable para *NEGAMAX*. No todos los algoritmos aceptan todas las profundidades, ver más adelante en la sección *[4.5.4.3.Configuración de opciones de IA de los jugadores]*.

PIEVAL : Esta variable define la función de evaluación que será utilizada por el algoritmo de IA para evaluar el tablero durante las búsquedas. El valor por defecto de *PIEVAL* es '*MATERIAL*'. Ver más adelante en la sección *[4.5.4.3.Configuración de opciones de IA de los jugadores]* los valores posibles y sus combinaciones.

P2NAME : Es el nombre utilizado por el jugador 2 durante la partida. El valor por defecto es '*player2*'.

P2AI : Es el algoritmo de IA que regirá el comportamiento del jugador 2 durante la partida. El valor por defecto es '*SIMPLE*'. Ver más adelante en la sección [4.5.4.3. *Configuración de opciones de IA de los jugadores*] los posibles valores de *P2AI* así como las posibles combinaciones con *P2EVAL*.

P2DEPTH : Es la profundidad de búsqueda usada por el algoritmo de búsqueda definido por *P2AI*. El valor por defecto es 1, que es el único valor aceptado por '*SIMPLE*'. No todos los algoritmos aceptan todas las profundidades, ver más adelante en la sección [4.5.4.3. *Configuración de opciones de IA de los jugadores*].

P2EVAL : Es la función de evaluación utilizada por el algoritmo de IA durante la búsqueda de movimientos. El valor por defecto de *P2EVAL* es '*RANDOM*'. Ver más adelante en la sección [4.5.4.3. *Configuración de opciones de IA de los jugadores*] los valores posibles y sus combinaciones con *P2AI*.

NUMCOLS : Define el número de columnas que tendrá el tablero para la partida. Podemos seleccionar entre 3-8 columnas. El valor por defecto es 8, que es el número de columnas típicas de un juego de Omweso. El número de filas del tablero es fijo y no se podrá cambiar. El número de filas del tablero es de 4 filas, dos de ellas pertenecen al jugador de la zona norte del tablero y otras dos pertenecen al jugador que juega en la zona sur del tablero.

FIRSTTURN: Establece que jugador moverá primero al comenzar la partida. El valor por defecto es '*P1*', y los valores posibles son '*P1*' y '*P2*'.

GFXBOARD: Define si el modo gráfico del tablero se activará automáticamente al iniciarse la partida. El modo gráfico podrá activarlo/desactivarlo cualquiera de los jugadores durante el transcurso de la partida mediante el comando [*gfxboard*] del menú [*play*]. El valor por defecto es '*ON*'. Los valores posibles son '*ON*' y '*OFF*'.

4.5.4.3. Configuración de opciones de IA de los jugadores.

Mediante la combinación de las variables *PIAI* y *PIEVAL* controlaremos el comportamiento del jugador 1 durante la partida. De igual manera las variables *P2AI* y *P2EVAL* controlaran el comportamiento del jugador 2.

Los valores que pueden tomar las variables del jugador 1 y jugador 2 son los mismas por lo que las definiremos a continuación de forma genérica como *PxAI* y *PxEVAL*.

Veremos que los valores de *PxAI* y *PxEVAL* se pueden combinar como queramos, de un modo muy flexible, salvo pequeñas excepciones, para definir el comportamiento que deseemos de nuestro jugador.

Valores de *PxAI* : Definen el algoritmo de IA usado por el jugador. Podemos ver una descripción detallada de cada uno de los algoritmos de búsqueda en la sección [2.7. Algoritmos de búsqueda].

- ***HUMAN*** : El jugador no tendrá IA, será manualmente controlado por un jugador humano.
- ***SIMPLE***: Algoritmo de IA, de profundidad 1, evalúa cada celda del tablero para cada uno de los movimientos que puede realizar el jugador en el turno actual.
- ***NEGAMAX***: Algoritmo de IA que admite profundidades de búsqueda mayores a 1. A mayor profundidad mayor precisión pero más consumo de tiempo y recursos en general.
- ***ALPHABETA***: Algoritmo de IA, permite profundidades mayores a 1, aumentando la precisión y también el tiempo de búsqueda según aumenta la profundidad. Realiza una poda Alfabeta por lo que reduce de forma importante las posiciones analizadas y los nodos del árbol de búsqueda, y por tanto el tiempo necesario.

- **SCOUT:** Algoritmo de AI, que permite profundidades mayores a 1. Introduce el concepto de sonda, lanza una prueba antes de realizar la búsqueda real.

Valores de Px EVAL: Define las funciones de evaluación usadas por la IA. Podemos

- **HUMAN:** No se utilizará ninguna función de evaluación. El jugador es humano.
- **RANDOM:** La evaluación del tablero consiste en número aleatorio, si tener en cuenta estado del tablero o contenido de las celdas.
- **MATERIAL:** Maximiza el número de semillas que tiene el jugador en posesión en su lado del tablero. Podemos encontrar una descripción más detallada en la sección [4.3.1. *Evaluación de la diferencia de material*].
- **DISPERSION:** Maximiza la dispersión de las semillas en el lado del tablero del jugador, de esta manera aumenta el número de jugadas disponibles en el tablero, a la vez que reduce el número de semillas en peligro de captura al no estar concentradas las semillas en las casillas del tablero. Podemos encontrar una descripción más detallada en la sección [4.3.2. *Evaluación de la dispersión del material*].
- **MOBILITY:** Maximiza la movilidad propiciando que existan más casillas desde las que el jugador pueda iniciar un movimiento. Teniendo en cuenta que el principal objetivo de Omweso para ganar la partida es dejar al jugador contrario sin movimientos/jugadas disponibles esta parece una interesante evaluación. Podemos encontrar una descripción más detallada en la sección [4.3.1. *Evaluación de la movilidad*].

- **REACH:** Se dice que una casilla y su contenido en semillas está en '*alcance*' si iniciando un movimiento desde dicha casilla finalizaría en una casilla de la fila interior del tablero. Las casillas de la fila interior del tablero son las únicas en las que se puede realizar una captura. Podemos encontrar una descripción más detallada en la sección [4.3.4. *Evaluación del alcance*].

Los valores '*HUMAN*' son utilizados para definir jugadores humanos y la única combinación posible de estos valores para '*PxAI/PxEVAL*' es '*HUMAN/HUMAN*' definiendo un jugador puramente humano.

Los jugadores autónomos, con IA controlada por el ordenador pueden utilizar cualquier combinación de *PxAI* y *PxEVAL*, siempre que no intervenga ningún valor '*HUMAN*'.

La aplicación controlará automáticamente las combinaciones posibles, no permitiendo realizar combinaciones ilegales de *PxAI* y *PxEVAL*, asignando los valores combinatorios por defecto en caso de encontrar alguna incompatibilidad en la combinación.

Valores de *PxDEPTH*: Define la profundidad máxima a la que realiza la búsqueda el algoritmo de IA. Los valores posibles son desde 1 hasta cualquier valor, siempre teniendo en cuenta que a mayor profundidad mayor consumo de recursos (el principal el tiempo), por lo que si utilizamos una profundidad excesivamente grande podremos llegar a agotar los recursos del equipo, o simplemente tardarán demasiado en realizarse las búsquedas. Valores recomendados, serían profundidades menores de 10, sobre todo para algoritmos de IA de los conocidos como de '*fuerza bruta*'.

Para jugadores *HUMAN* no tiene sentido definir una profundidad pues el sistema no utilizará este parámetro.

Veremos que no tiene sentido utilizar profundidades mayores de 1 para algoritmos que utilicen valores de *PxEVAL* de tipo *RANDOM*, por lo que esta combinación no es permitida.

4.5.4.4. Editar los contenidos del tablero.

En cualquier momento de la partida podremos editar los contenidos de casillas del tablero entrando en el menú *[setup]*, disponible desde el menú principal. La principal finalidad es la de que, al principio de la partida, cada jugador coloque sus 32 semillas en las casillas de su lado del tablero de acuerdo a la estrategia que desee seguir durante la partida.

Por otro lado, la edición del tablero no se ha limitado de ninguna forma. Por ejemplo no se comprueba que cada jugador tenga la mitad de las semillas del tablero. De esta forma, podremos utilizar esta opción de editar tablero no solamente para colocar las semillas al principio de la partida sino que podremos, en cualquier momento de una partida ya iniciada, modificar el tablero para recrear una situación que queramos estudiar.

Solamente existe una única restricción, y es que, antes de poder finalizar y confirmar los cambios realizados durante la edición del tablero, el sistema comprueba que las 64 semillas totales se hallan distribuido por el tablero.

Podremos realizar la edición del tablero desde el interfaz de línea de comandos usando las opciones disponibles:

```
omweso: /> setup
MAINMENU::SETUPBOARD
board      [b]      show the current board status.
gfxboard   [gfx]    show/hidde the graphic board.
put        [p]      put seeds in the board.
take       [t]      take seeds from the board.
confirm    [ok]    confirms setup and save it.
cancel     [main]  cancels setup and back to app mainmenu.
help       [h]      show this help screen.

Setting up the board will set your setup as initial position of the board
and will reset game deleting all history.
omweso: /setup> █
```

Pasamos a describir los comandos disponibles en el menú de edición [*setup*]:

board [b] : Este comando nos mostrará una representación en modo texto del tablero, y de su estado. Ver apartado 4.5.5.1 más adelante en este manual.

gfxboard [gfx] : Activará (o desactivará, si está activo) el modo gráfico del tablero, permitiéndonos no sólo contemplar es estado, sino jugar gráficamente.

take [t] : Toma semillas de una casilla del tablero mientras estamos en el modo de edición. Debemos especificar la casilla que se verá afectada y la cantidad de semillas que queremos tomar, que puede coincidir con el total de semillas existentes en la casilla o cualquier número inferior. El formato sería: [*take <nombre celda> <num semillas>*].

put [p] : Pone un número determinado de semillas en la celda especificada. Para poder poner semillas antes deberemos haberlas tomado de otra casilla mediante el comando [*take*]. El formato del comando sería: [*put <nombre celda> <num semillas>*] donde el número de semillas a poner deberá ser menor o igual al número de semillas tomadas anteriormente. Aunque las semillas hayan sido tomadas todas con un solo comando [*take*] podremos repartirlas entre dos casillas con dos comandos [*put*].

confirm [ok] : Confirma los cambios realizados en el tablero aplicándolos y posteriormente sale al menú principal, desde podremos volver a la partida y seguir jugando con el nuevo estado del tablero.

cancel [main] : Cancela la edición del tablero restableciendo los valores anteriores a la edición y después sale al menú principal.

help [h] : Muestra la ayuda contextual del menú [*setup*].

4.5.4.5. Guardar una partida a disco.

En cualquier momento de la partida, durante el turno de un jugador humano, podremos guardar el estado de la partida a un archivo, con lo que podremos continuar la partida en cualquier otro momento.

El comando que nos permitirá guardar la partida es el comando `savegame`. Tendremos acceso a él desde el menú principal y podremos guardar una partida simplemente ejecutando `savegame <nombre de archivo>`. Por convenio, se guardará con extensión `.sav`.

```
omweso: /> savegame
prueba_eval.sav  savegame2.sav  savegame4.sav
savegame1.sav   savegame3.sav
omweso: /> savegame prueba_savegame.sav
Game saved sucessfully.
omweso: /> █
```

Gracias a que la aplicación utiliza la librería `readline` para la gestión de la línea de comandos, si solamente escribimos `'savegame'` y damos dos veces la tecla tabulador para invocar el completador de `readline`, automáticamente veremos un listado de todos los archivos `.sav` ya existentes. Debemos tener cuidado de no sobrescribir ninguna de las partidas ya guardadas, salvo que expresamente queramos sobrescribirla en cuyo caso sólo tendremos que grabarla con el mismo nombre que ya existe.

Datos del histórico (log) de movimientos: En el caso de salvarse la partida una vez iniciada y tras haber realizado varios movimientos (caso más habitual), se añadirá al fichero de partida guardado, el listado de movimientos realizados hasta el momento de guardar la partida, lo que nos permitirá en un futuro tras cargar la partida utilizar las funcionalidades del log de movimientos para retroceder a movimientos anteriores para analizarlos. Podemos ver información más detallada al respecto en la sección [4.5.6.3. *Registrar los movimientos de las partidas en un fichero*].

4.5.4.6. Cargar una partida desde disco.

La aplicación nos permitirá, además de iniciar una partida configurandola manualmente, inciar una partida cargándola desde un archivo '.sav' en la que fue guardada previamente una partida ya iniciada. La partida creada a partir del archivo se creará en el mismo punto donde se dejó, mismo turno, mismo estado del tablero, de los jugadores, etc.

```
omweso: /> loadgame
prueba_eval.sav  savegame2.sav  savegame4.sav
savegame1.sav   savegame3.sav
omweso: /> loadgame savegame1.sav █
```

El comando para cargar partidas es el comando *loadgame*, es accesible desde el menú principal de la aplicación. Para cargar una partida simplemente ejecutaremos el comando *loadgame <nombre de archivo>*.

Por convenio, los archivos de partidas guardadas se guardaran con la extensión '.sav', cualquier otra extensión no será manejada por la aplicación.

Gracias a que la aplicación utiliza la librería *readline* para la gestión de la línea de comandos, si solamente escribimos '*loadgame*' y damos dos veces la tecla tabulador para invocar el completador de *readline*, automáticamente veremos un listado de todos los archivos .sav disponibles.

Formato del archivo de partida guardada: Los archivos de partida guardada tienen formato texto fácilmente interpretable y modificable manualmente con cualquier editor de texto. Así que aunque lo más común es que sean generados por el comando *savegame*, la realidad es que podríamos crearnos nuestros propios archivos de guardado manualmente, bien desde cero, o bien editando alguno existente.

Un ejemplo sencillo, que podríamos utilizar como base para crear nuestros propios archivos de guardado sería el siguiente:

```
# Omweso Savegame
P1NAME: player1
P1AI: ALPHABETA
P1DEPTH: 3
P1EVAL: 90ATT10BKDEF
P2NAME: player2
P2AI: ALPHABETA
P2DEPTH: 1
P2EVAL: RANDOM
NUMCOLS: 8
GFXBOARD: OFF
FIRSTTURN: P1
TURN: 24
PLAYS: P1
BRD_INIDATA:
04 00 00 00 00 00 00 00
00 04 04 04 04 04 04 04
04 04 04 04 04 04 04 00
00 00 00 00 00 00 00 04
LOGMOVES:
move h1
move C0
rcapture b0
move E0
```

Simplemente con copiar y pegar estas líneas en un archivo con extensión `.sav` y ponerlo en la misma carpeta en la que Omweso está instalado ya podremos cargarlo con el comando `loadgame` y nos creará una partida nueva a partir de él.

Todas las variables que aparecen en los archivos de guardado, y todos los valores que pueden tomar son los mismos descritos en los apartados [\[4.5.4.2. Configurar las opciones de la partida\]](#). Y [\[4.5.4.3. Configuración de las opciones de IA de los jugadores\]](#), de este mismo documento.

La variable `BRD_INIDATA` define los contenidos de cada celda del tablero en el momento en el que se creó la partida. Como se puede ver en el ejemplo utiliza cuatro líneas de valores para definir los contenidos del tablero. Los datos deben tener un formato numérico de dos dígitos, rellenando con ceros a la izquierda si es necesario, están separados por espacios o tabuladores, y en cada fila deben existir tantos datos como columnas defina la variable `NUMCOLS` previamente en el fichero de guardado.

Hay otras dos variables, que son: **TURN**, que define en que turno se encontraba la partida cuando se guardó, y **PLAYS**, que indica que jugador tenía que mover.

El otro apartado del archivo de partida guardada se almacenará la información del log de movimientos, vendrá encabezado por la variable **LOGMOVES**, y tendrá el siguiente formato:

```
PLAYS: P1
BRD_INIDATA:
04 00 00 00 00 00 00 00
00 04 04 04 04 04 04 04
01 04 04 04 04 04 04 03
00 00 00 00 00 00 00 04
LOGMOVES:
move b0
move G0
move e0
move D0
move d0
move C1
# EOF
```

Como vemos se guardarán la lista de movimientos, en formato de texto legible fácilmente.

Al cargar una partida, se cargará automáticamente la lista de movimientos, y la partida continuará en el último movimiento, en el mismo punto donde se encontraba en el momento de guardar la partida.

Por último, hacer notar que si la línea comienza con almohadilla (#), esa línea es ignorada en el momento de la carga del archivo, y que las variables que no sean definidas en el archivo de guardado, o que estén en una línea comentada (#), tomarán el valor por defecto definido en los apartados de configuración previamente mencionados en este mismo apartado.

4.5.4.7. Finalizar una partida prematuramente.

Si en cualquier momento durante una partida necesitamos finalizar la partida en curso podemos hacerlo con el comando *[endgame]*. Tendremos acceso al comando *endgame* desde el menú principal de la aplicación.

```
omweso: /> endgame
Game ended successfully.
omweso: /> █
```

Una partida sólo podrá finalizarse desde el turno de un jugador humano, con lo que tendremos que tener cuidado en partidas de dos jugadores de IA.

Además de por voluntad propia, habrá algunos casos en los que necesitaremos finalizar la partida, por ejemplo, será necesario finalizar la partida actual con el comando *[endgame]*, antes de poder crear una partida nueva, o incluso antes de cualquier acción que implique la creación indirecta de una partida nueva, como es el caso de que necesitaremos finalizar la partida actual antes de cargar una nueva partida con el comando *[loadgame]*.

4.5.5. Jugar una partida de Omweso.

Para jugar una partida de Omweso, antes de nada deberemos crear una partida nueva según se explica en la sección *[4.5.4.1 Crear una partida nueva]* de este mismo documento.

Una vez creada la partida, si al menos uno de los dos jugadores son controlados por un humano, tendremos acceso al menú de juego (menú *[play]*).

```
MAINMENU::PLAY MENU

board [b] show the current board status.
gfxboard [gfx] show/hidde the graphic board.
move [mv] make movement during your turn.
rcapture [rc] make a reverse capture movement.
undo [ud] Returns game to the previous turn of the current player.
mainmenu [main] go to app main menu.
manageai [ai] manage artificial intelligence.
help [h] show this help screen.

omweso[S1]:/player1> █
```

Si mientras estamos en el modo de juego, ejecutamos el comando *[help]*, nos mostrará la ayuda contextual del menú *[play]*, mostrándonos todos los comandos disponibles:

board [b] : Este comando nos mostrará una representación en modo texto del tablero, y de su estado. Ver apartado 4.5.5.1 más adelante en este manual.

gfxboard [gfx] : Activará (o desactivará, si está activo) el modo gráfico del tablero, permitiéndonos no sólo contemplar su estado, sino jugar gráficamente.

move [m] : Ejecuta un movimiento típico de siembra, iniciado desde la celda del tablero indicada como parámetro del comando.

rcapture [rc] : Mientrás que las capturas directas se ejecutarán automáticamente (si se dan las condiciones de captura) tras un movimiento de siembra, las capturas inversas necesitan que se inicie un movimiento inverso con el comando *[rcapture]*. El movimiento inverso se iniciará en la celda del tablero indicada como parámetro.

undo [ud] : Deshace el último turno completo volviendo la partida al mismo estado que había en el turno anterior del jugador que ejecuta este comando.

mainmenu [main] : Vuelve al menú principal, donde podrá realizar las acciones descritas en la sección *[4.5.4.Gestión de partidas]*.

manageai [ai] : Entrá en el menú de IA, donde nos permitirá configurar las opciones de IA de los jugadores, crear lotes de partidas o guardar los resultados estadísticos de la partida en un archivo de log.

4.5.5.1. Visualizar el tablero.

Desde el menú *[play]*, el comando *[board]* nos permitirá obtener una representación en modo texto del tablero y su estado.

```
omweso[S1]:/player1> b
[N]  H G F E D C B A [N]
  0-----0
  1 | 04 00 00 00 00 00 00 00 | 1
  0 | 00 04 04 04 04 04 04 04 | 0
  0-----0
  0 | 04 04 04 04 04 04 04 00 | 0
  1 | 00 00 00 00 00 00 00 04 | 1
  0-----0
[S]  a b c d e f g h [S]
omweso[S1]:/player1> █
```

Vemos que básicamente nos define las dos áreas del tablero (norte y sur), pertenecientes a cada uno de los jugadores. Y define 4 filas de números que representan los contenidos de las celdas del tablero.

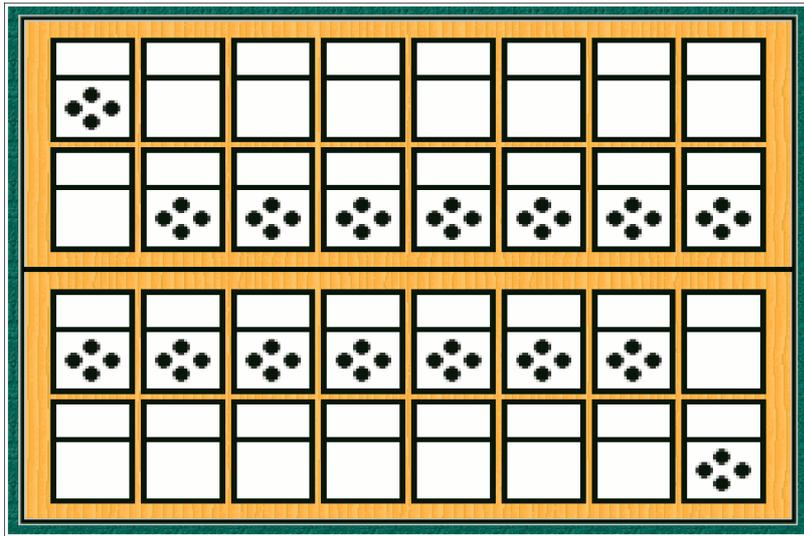
También vemos que se define una nomenclatura para las filas y columnas del tablero, así tendremos las filas 0-1 tanto para la zona norte como sur, y las columnas a-h para la zona sur, y columnas A-H para la zona norte.

Para realizar cualquier acción sobre una celda, deberemos nombrarla como la unión del nombre de la fila y columna a las que pertenece, así por ejemplo, en el tablero mostrado más arriba la primera celda (arriba, izquierda) se denomina *H1* y contiene 4 semillas.

Para iniciar, por ejemplo, un movimiento de siembra, ejecutaremos el comando *[move H1]* durante el turno del jugador que juega en la zona norte del tablero.

4.5.5.2. Visualizar el tablero gráfico.

El modo gráfico del tablero nos permitirá realizar las opciones del modo de juego (mover, captura inversa) sin necesidad de escribir en la consola, simplemente usando el ratón sobre el tablero gráfico y pulsando con el ratón sobre la casilla deseada.



Abriremos (o cerraremos si estaba abierto) el tablero gráfico, usando el comando `[gfxboard]` desde el menú de juego (menú `[play]`).

Mientras tengamos el tablero gráfico abierto, podremos continuar ordenando acciones (`[move]`, `[rcapture]`) desde la consola de texto, y veremos reflejado la ejecución en el tablero gráfico, o también podemos jugar directamente usando el ratón sobre el propio tablero gráfico.

4.5.5.3. Realizar un movimiento durante un turno.

En el caso de que al menos uno de los jugadores de la partida sea definido como humano, podremos controlar sus acciones mediante la consola de texto o por el tablero gráfico.

El movimiento lo ejecutaremos con el comando *[move]*, desde el menú de juego indicándole como parámetro la celda desde la que queremos iniciar el movimiento según se describe en la sección *[4.5.5.1. Jugar una partida de Omweso]*.

Para realizar un movimiento usando el tablero gráfico, simplemente haremos click con el botón izquierdo del ratón sobre la casilla deseada. Si deseamos iniciar una captura inversa realizaremos el click sobre la casilla con el botón derecho del ratón.

4.5.5.4. Realizar una captura inversa.

En el caso de darse las condiciones de captura inversa (que exista la posibilidad de captura mediante un movimiento en sentido de las agujas del reloj iniciándose sólo desde las 4 celdas más a la izquierda de las zonas norte y sur), la aplicación nos permitirá ejecutar una captura inversa.

Podremos ejecutarla desde la consola de texto, usando el comando *[rcapture]*, pasándole como parámetro el nombre de la celda deseada, o desde el tablero gráfico haciendo click sobre la celda con el botón derecho del ratón.

4.5.6. Herramientas para el investigador de IA.

La aplicación Omweso no solamente permite jugar partidas a dos jugadores, ya sean humanos o autómatas, sino que proporciona una serie de herramientas orientadas al estudio del comportamiento de la IA, como son la creación de lotes de partidas, almacenado de resultados en logs que posteriormente pueden ser analizados, o diferentes configuraciones de la IA de los jugadores no humanos.

```
omweso:/> ai
MAINMENU:: MANAGEAI MENU
showcfg [cfg] show game current configuration.
logmoves [logm] show the list of moves executed.
undom [udm] UnDo of the last movement.
redom [rdm] ReDo the last 'undone' movement.
batch [bat] run various games with the current config.
logresult [logr] Save games results to a log file.
play [p] continue playing the current game.
mainmenu [main] go to app main menu.
help [h] show this help screen.
```

Desde el menú principal, podremos acceder al menú de herramientas de IA mediante el comando *manageai [ai]*. Una vez en el menú de IA el comando *help [h]* nos permitirá ver las opciones disponibles como es habitual.

Aparte de las herramientas comentadas tendremos la posibilidad de retroceder [*undom*] o avanzar [*redom*] movimientos/jugadas dentro de la partida.

Analizaremos más en profundidad todas las opciones disponibles en las secciones que siguen a continuación.

4.5.6.1. Crear un lote de partidas automatizadas.

Esta opción nos permitirá crear un lote de varias partidas, en principio un número ilimitado para su ejecución de forma secuencial. La principal funcionalidad de esta opción es la de ejecutar partidas de dos jugadores no humanos y analizar sus resultados.

Como en un número pequeño de partidas la aleatoriedad puede afectar a los resultados, la forma estadísticamente correcta de analizar los resultados es ejecutar un gran número de partidas con la mismas configuraciones.

Podremos crear un lote de partidas mediante el comando *batch [bat]*, que nos permitirá definir el número de partidas del que se compondrá el lote.

```
omweso:/ai> bat
Incorrect syntax, try: batch <NUMBER_OF_GAMES>
omweso:/ai> bat 50
The batch queue has been set to [50] game(s).
The next game you create/play will run [50] times, sequentially.
omweso:/ai> █
```

Todas las partidas se ejecutarán con la misma configuración. Si ya habíamos creado una partida, todas las partidas del lote utilizarán esa misma configuración. Si aún no habíamos creado ninguna partida, la siguiente partida que creemos se utilizará como plantilla y como primera partida del lote.

Si en cualquier momento queremos anular un lote de partidas solamente tendremos que ejecutar un comando tal como `[bat 0]` o `[bat 1]` y una vez que termine la partida actual el lote acabará automáticamente.

Si ninguno de los dos jugadores es humano, sino que son dos jugadores autómatas con IA controlada por el ordenador, la aplicación ejecutará el lote completo de partidas sin devolver el control a los usuarios en ningún momento por lo que no se podrá interactuar con el programa durante la ejecución del lote, y por tanto tampoco se podrá detener el lote, salvo finalizando la aplicación Omweso por la fuerza desde el sistema operativo, pero en este caso se perderá el estado y progreso de la aplicación hasta ese momento, salvo las cosas que haya quedando guardadas en logs, como los resultados de las partidas, etc.

4.5.6.2.Registrar los resultados de las partidas en un fichero.

Para cualquier partida podremos guardar los resultados del juego automáticamente en un log. Esta opción resulta aún más interesante en el caso de los lotes automáticos de partidas, en el que dos jugadores no humanos juegan a gran velocidad un gran número de partidas. Sería imposible, o al menos muy complicado, poder ni siquiera seguir el juego de forma individual y manual por lo que la forma más cómoda es guardar los resultados de todo el lote de partidas en un log de forma que podamos analizarlo posteriormente.

```

omweso:/ai> logr
Logresults is ON, writing results to file: ow_results.log
To change config use: logr [ON|OFF [<LOGRESULTS_FILENAME>]]
omweso:/ai> logr ON ow_results.log
Logresults is ON, writing results to file: ow_results.log
omweso:/ai> _

```

Podremos activar o desactivar el log de resultados, así como cambiar el nombre del archivo log mediante el comando *logresults [logr]*. El log de resultados está por defecto desactivado, y en el caso de estar activado, salvo que se cambie el nombre, el archivo log por defecto será *ow_results.log*. Si deseamos cambiar el nombre del log podremos hacerlo ejecutando el comando *[logr ON <nombrelogresultados>]*.

En el caso de no existir el log la aplicación lo creará automáticamente con el nombre especificado. En caso de existir con anterioridad los resultados se añadirán a los ya existentes y la aplicación en ningún momento vaciará o borrará el log, por lo que si deseamos vaciarlo deberemos eliminar el archivo del log manualmente, y volverá a ser regenerado vacío automáticamente.

El formato con el que se guardarán los resultados de las partidas en el log de resultados será el mostrado en el siguiente recuadro:

```

P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 21, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 21, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 21, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 21, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 21, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 35, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 29, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 79, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 51, P1
P1, ALPHABETA, 3, 90ATT10BKDEF, P2, ALPHABETA, 1, RANDOM, 8, P1, 31, P1

```

Detallando el formato del log, vemos que será *[P1, P1AIMODE, P1AIDDEPTH, P1AIEVAL, P2, P2AIMODE, P2AIDDEPTH, P2AIEVAL, NUMCOLS, FIRSTTURN, MOVES2WIN, WINNER]*. Básicamente representan la configuración de IA que tenía cada uno de los jugadores (algoritmo de búsqueda, profundidad, función de evaluación), por supuesto quien fue el ganador de cada juego, y resultados interesantes como son el número de movimientos necesarios para ganar la partida. Los valores que puede tomar

las variables relacionadas con la IA de los jugadores son los mismos definidos en la sección [4.5.4.3. *Configuración de las opciones de IA de los jugadores*]. Adicionalmente, en el log aparecen [MOVES2WIN] y [WINNER], que indican el número de movimientos que fueron necesarios antes de acabar la partida, y el ganador de la misma, respectivamente.

No existe ninguna herramienta integrada en la aplicación Omweso para analizar estos resultados pero dado su formato de valores separados por comas es bastante sencillo importarlo en cualquier herramienta de análisis o incluso en una simple hoja de cálculo para su posterior estudio y análisis.

4.5.6.3.Registrar los movimientos de las partidas en un fichero.

Desde el menú de gestión de la IA, al que se accede con el comando *manageai [ai]* desde el menú principal, tendremos la opción de visualizar todas las jugadas que se han realizado durante la partida de Omweso.

```
omweso:/ai> logmoves
List of game moves:
P1[S1]: move h1
P2[N1]: move H1
P1[S2]: move g0
P2[N2]: move G0
P1[S3]: move a1
P2[N3]: move A1
P1[S4]: move d1
*P2[N4]: Waiting input
omweso:/ai> □
```

El comando *logmoves [logm]* nos permitirá listar todos los movimientos listados hasta el momento. Como vemos nos especifica que jugador realiza la el movimiento (P1,P2), el movimiento/turno (N1, S1, N2, S2, ...) y la jugada realizada (*move/rcapture*), así como la casilla sobre la que se inicia el movimiento. Además nos indica como ultima entrada que está esperando la entrada del movimiento del siguiente jugador.

El comando *undom [udm]* nos permitirá deshacer movimientos/jugadas de cualquiera de los dos jugadores de la partida.

Podremos deshacer movimientos hasta llegar al estado inicial de la partida antes del primer movimiento, o detenernos en cualquiera de los movimientos intermedios para estudiar la situación de la partida en ese punto.

Si visualizamos de nuevo el log de movimientos con el comando *logmoves* [*logm*] podremos ver que nos señala que hemos retrocedido a un estado anterior de la partida.

```
omweso:/ai> logmoves
List of game moves:
  P1[S1]: move h1
  P2[N1]: move H1
  P1[S2]: move g0
  P2[N2]: move G0
  P1[S3]: move a1
  *P2[N3]: move A1
  P1[S4]: move d1
omweso:/ai> □
```

Podremos usar el comando *redom* [*rdm*] para avanzar en el log de movimiento y volver a adelante y rehacer movimientos que antes habíamos deshecho con el comando *undom* [*udm*], hasta llegar al último movimiento que habíamos realizado.

```
omweso:/ai> logm
List of game moves:
  P1[S1]: move h1
  P2[N1]: move H1
  P1[S2]: move g0
  P2[N2]: move G0
  P1[S3]: move a1
  P2[N3]: move A1
  P1[S4]: move d1
  *P1[S4]: Waiting input
omweso:/ai> redom
Cannot ReDo more moves from the undo queue.
omweso:/ai> □
```

Como posibilidad interesante podemos destacar que podemos retroceder jugadas/movimientos usando el comando *undom* [*udm*], y podemos reanudar el juego desde ese punto tomando decisiones diferentes y realizando jugadas o movimientos distintos, llegando a un desenlace diferente, sin ser obligatorio volver a avanzar en el log de movimientos con el comando *redom* [*rd*] en ningún caso.

Mediante esta opción podremos investigar qué consecuencias tiene tomar decisiones diferentes partiendo de un mismo punto y estado del juego, y ver cómo afectan esas decisiones y jugadas diferentes al desenlace de la partida, por lo que resulta una herramienta de estudio de la IA muy interesante.

4.5.6.4. Cambiar el algoritmo de búsqueda.

La aplicación Omweso nos permitirá configurar la IA de los jugadores no humanos de forma que podremos seleccionar el algoritmo de búsqueda utilizado durante la toma de decisiones.

```
omweso:/newgame> cfg
Game configuration:
P1NAME:          player1
P1AI(P1DEPTH):   NEGAMAX(3)
P1EVAL:          90ATT10BKDEF
P2NAME:          player2
P2AI(P2DEPTH):   ALPHABETA(1)
P2EVAL:          RANDOM
NUMCOLS:         8
FIRSTTURN:       P1
GFXBOARD:        OFF
omweso:/newgame> set P1AI NEGAMAX
```

El algoritmo de búsqueda utilizado por los jugadores controlados por el ordenador no se puede cambiar en cualquier momento de la partida, solamente podrán ser configurados durante el proceso de creación de una partida nueva, y los jugadores utilizarán esa configuración a lo largo de toda la partida.

Para una explicación más detallada de los algoritmos de búsqueda disponibles y su utilización remitimos a la sección [4.5.4.3. *Configuración de opciones de IA de los jugadores*].

4.5.6.5. La función de evaluación.

La aplicación Omweso nos permitirá seleccionar entre varias funciones de evaluación para usar junto con los algoritmos de búsqueda disponibles. Mientras que el algoritmo de búsqueda determinará la velocidad de la búsqueda, número de nodos analizados, etc., de la función de evaluación dependerá discernir cuales las jugadas que nos acercarán más a la victoria.

En nuestra implementación de Omweso hemos incluido cuatro funciones de evaluación : *MATERIAL*, *DISPERSION*, *MOBILITY* y *REACH* . Podemos ver una descripción más detallada de estas características en la sección [4.3. Búsqueda de la función de evaluación óptima].

```
Game configuration:
P1NAME:      player1
P1AI(P1DEPTH):  NEGAMAX(3)
P1EVAL:      DISPERSION
P2NAME:      player2
P2AI(P2DEPTH):  SIMPLE(1)
P2EVAL:      RANDOM
NUMCOLS:     8
FIRSTTURN:    P1
GFXBOARD:    ON

omweso:/newgame> set p2eval MOBILITY
```

También se han implementado dos funciones más complejas *F1COMB* y *F2COMB* que son el resultado de una combinación lineal de las características individuales. Tenemos más detalles sobre la combinación lineal en las secciones [5.5. Combinación lineal de las características para la función de evaluación] y [5.6. Eliminación de algunas características de la función de evaluación].

La configuración de la función de evaluación utilizada por un jugador solamente podrá ser seleccionada en el momento de la creación de la partida, no una vez iniciada la misma.

Capítulo 5

Resultados

5.Resultados.

Durante este capítulo analizaremos el comportamiento de nuestro sistema. Por un lado, se comparará el comportamiento de cada uno de los algoritmos de búsqueda (recursos utilizados, nodos analizados, mejoras que suponen unos respecto a otros). Por otro lado, analizaremos las características utilizadas en nuestras funciones de evaluación, y veremos con resultados como influyen cada una de las características.

5.1.Rendimiento de los algoritmos de búsqueda.

Lo primero que intentaremos comprobar empíricamente es el rendimiento de cada uno de los algoritmos de búsqueda que hemos implementado.

Supuestamente, el Negamax debería ser el que de un peor rendimiento, analizando más nodos del árbol de búsqueda y por tanto utilizando más tiempo y recursos para tomar cada una de las decisiones de que jugada utilizar en cada turno. Por otro lado, Alfabetá y Scout evolucionan mejorando ese rendimiento, siendo el Scout el que debiera tener el mejor rendimiento.

Veamos unos datos empíricos y si la teoría se cumple:

Negamax			Alfabetá			Scout		
	nodos	time		nodos	time		nodos	time
1	466	0.0162	1	254	0.0064	1	42	0.0136
2	255	0.0093	2	195	0.0067	2	41	0.0425
3	383	0.0265	3	99	0.0030	3	51	0.0291
4	381	0.0329	4	138	0.0043	4	26	0.0345
5	284	0.0282	5	85	0.0028	5	46	0.0097
6	174	0.0223	6	54	0.0021	6	36	0.0358
7	76	0.0029	7	29	0.0013	7	23	0.0076
8	104	0.0058	8	20	0.0008	8	19	0.0200
9	160	0.0057	9	22	0.0008	9	11	0.0040

TablaResultados 1: Nodos analizados y tiempo consumido.

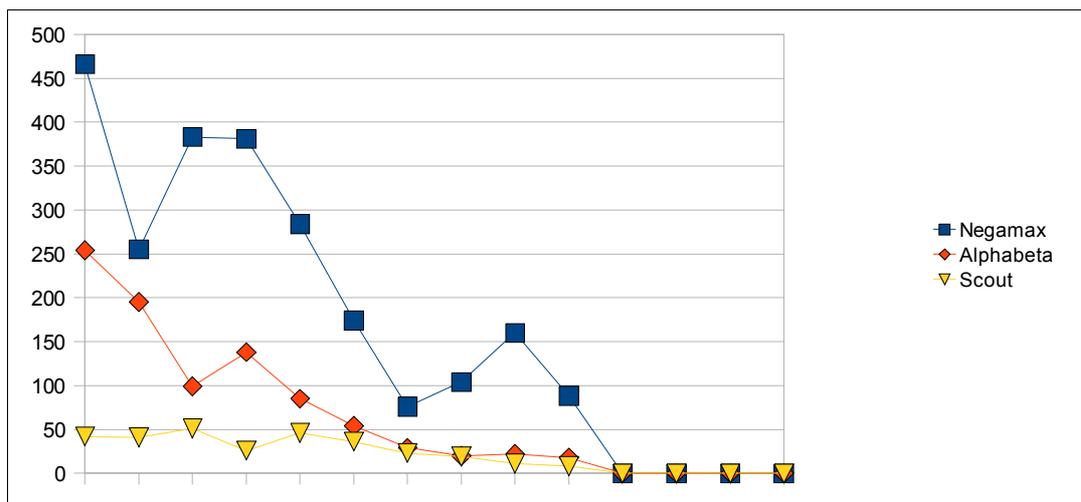
Se han recogido los datos de los tres algoritmos de búsqueda implementados, cogiendo tres partidas aleatorias, con una profundidad igual a 3, y se ha limitado el

estudio a los diez primeros movimientos de la partida. En dos casos la partida terminó en esos diez movimientos, en el tercer caso la partida duró un par de movimientos más, pero no eran relevantes para el estudio así que se optó por limitar el estudio en los tres casos a diez movimientos.

En una columna tendremos la cantidad de nodos que tuvo que recorrer y analizar el algoritmo de búsqueda para poder tomar una decisión, en la otra columna además hemos añadido el tiempo total que tardo en tomar la decisión, incluyendo el tiempo necesario para analizar todos los nodos especificados por la columna anterior.

A primera vista podemos ver que nuestras previsiones se cumplen según lo esperado, mejorando cada algoritmo al anterior tanto reduciendo el número de nodos como el tiempo consumido.

Decir que en los tres casos, como era de esperar, los tres algoritmos finalmente tomaron las mismas decisiones, por lo que es bastante justo comparar los resultados obtenidos pues los tres algoritmos tuvieron rendimientos diferentes para tomar exactamente las mismas decisiones.



Gráfica 1: Nodos analizados por cada algoritmo de búsqueda

En la *gráfica 1*, vemos claramente que el rendimiento del Negamax es mucho peor, analizando muchos más nodos, y vemos como efectivamente el Alfabetá y el Scout están en todo momento por debajo analizando un número menor de nodos, obteniendo portanto un mejor rendimiento, siendo el Scout el que mejor rendimiento consigue de los tres algoritmos.

Vemos que el Alfabetá y el Scout en algunos momentos rinden de forma similar, o muy próxima, aunque el Scout, en general, viendo en global la partida, es el que mejor rendimiento está consiguiendo.

Solamente hacer notar que un mismo algoritmo analiza muchos más nodos en una jugada que en la anterior o posterior en algunas ocasiones. Sería de esperar que siguiera alguna progresión, posiblemente analizando más nodos al principio de la partida y menos nodos al final de la partida, cuando el juego ya está casi sentenciado.

Pero veremos que el Omweso es muy caprichoso en este aspecto, y que debido a la captura y reentrada de semillas, y por supuesto al movimiento de siembra, el estado del tablero se ve afectado de gran manera tras cada jugada.

Al cambiar tanto el tablero tras cada jugada, veremos que también puede verse afectada la cantidad de nodos analizados para decidir cada jugada, generando esa irregularidad y no progresión en los datos recogidos.

5.2.Posesión del material durante una partida.

Una de las primeras características a medir e incorporar a nuestra función de evaluación ha sido la cantidad de material (fichas, semillas) que tiene cada jugador en un momento del juego. La idea, es que el jugador que acumula más material en su poder encuentra en una situación positiva para convertirse en el ganador.

Una primera comprobación que haremos pues será comprobar a lo largo de una partida ver qué cantidad de material tiene cada jugador tras cada movimiento, y ver si ciertamente, y de acuerdo con nuestra suposición, el jugador que va acumulando más material es el que finalmente se hace con la victoria.

Mov.#	MAT(N)	MAT(S)	Mov.#	MAT(N)	MAT(S)	Mov.#	MAT(N)	MAT(S)
0	32	32	12	13	51	24	58	6
1	32	32	13	10	54	25	58	6
2	38	26	14	19	45	26	58	6
3	32	32	15	9	55	27	58	6
4	32	32	16	9	55	28	58	6
5	24	40	17	9	55	29	58	6
6	24	40	18	22	42	30	58	6
7	17	47	19	15	49	31	58	6
8	17	47	20	25	39	32	58	6
9	15	49	21	23	41	33	58	6
10	15	49	22	52	12			
11	13	51	23	52	12			

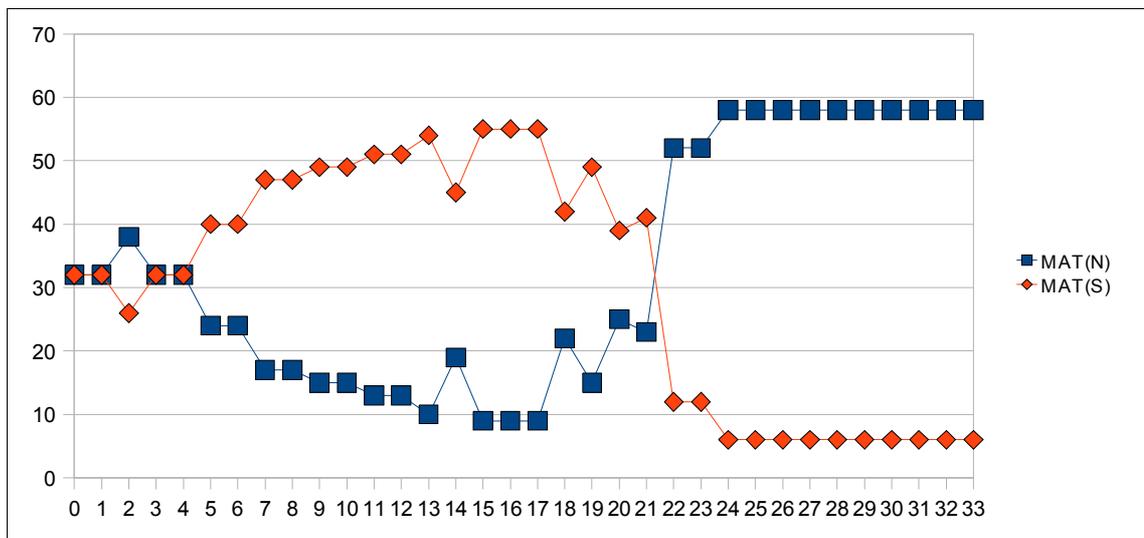
TablaResultados 2: Distribución del material durante una partida.

En la *tabla de resultados 2*, podemos ver los datos de material durante el juego. Simplemente hemos recogido los datos en tres columnas. La primera columna simplemente marca secuencialmente los movimientos alternados de los dos jugadores. Las otras dos columnas, reflejan el material del jugador que juega en el lado norte del tablero [*MAT(N)*] y el material del jugador que juega en el lado sur [*MAT(S)*], respectivamente. La partida se jugo sobre un tablero de Omweso estándar de ocho columnas por dos filas y 32 semillas, cada jugador. Los dos jugadores utilizan una lógica que intenta maximizar el material con la finalidad de reducir la movilidad del contrario, y una profundidad igual a 3.

Viendo la situación en el último movimiento, el jugador que juega al norte acumulaba la inmensa mayoría del material en su poder (58 fichas contra 6). Aún sin saberlo explícitamente, podemos asegurar, teniendo en cuenta las reglas de Omweso, que el jugador del norte, fue el jugador ganador.

Según las reglas de Omweso, el jugador perdedor será el que se quede sin movimientos legales. Con 58 semillas el jugador norte, no existe ninguna distribución posible de esas semillas en su lado del tablero para que sea posible que el jugador no tenga movimientos legales. Hay que recordar que siempre que tengamos una casilla con 2 o más semillas, tendremos un movimiento válido disponible. Es difícil, si no casi imposible, quedarse sin movimientos si tenemos bastante material.

Pero veremos por experimentación, que lo único que podemos afirmar con seguridad, es que al finalizar la partida el jugador que tiene más material coincide con el jugador ganador. Pero, sin embargo, veremos que no podemos asegurar que el jugador que finalmente se alza con la victoria ha tenido superioridad de material durante todo el desarrollo de la partida.



Gráfica 2: Distribución del material durante una partida.

En la *gráfica 2*, vemos una representación gráfica de los mismos datos que teníamos en la *tabla de resultados 2*. En azul tenemos representados los datos del jugador que juega en el lado norte del tablero, y en rojo tenemos representados los datos la cantidad de material que tiene el jugador que juega en el lado sur del tablero.

Lo primero que nos llama la atención es que la evolución del material acumulado por los dos jugadores es estrictamente asimétrica. Esto se debe a que Omweso es un juego de suma nula, es decir, el material ganado por uno de los jugadores en todo momento proviene del material poseído por el jugador contrario, manteniéndose en todo momento constante la cantidad total de material en juego.

Pero hay otra cosa en la *gráfica 1*, que nos llama aún más la atención, y que sin duda es más importante, y que es que, mientras el jugador sur (representado en rojo) tuvo en posesión más material durante los primeros veinte movimientos del juego, en el movimiento 21 se ha invertido la situación completamente, acabando finalmente la partida con la victoria del jugador norte (azul).

Veamos como algunas de las reglas de Omweso hacen que la distribución del material en el tablero afecten de una forma curiosa a la evolución del juego durante una partida:

Restricciones de movimiento: *[Las casillas con una sola semilla, y obviamente las casillas sin semillas, no pueden iniciar un movimiento.]* La acumulación de material, es decir cuanto más fichas tenga el jugador más probable es que tengamos movimientos disponibles.

Fin del turno: *[Si la última semilla del movimiento cae en una casilla vacía, finalizará el movimiento pasando el turno al jugador contrario, sino se continuará con un movimiento encadenado.]* Es obvio que si el jugador tiene más material es más probable que la casilla donde finaliza un movimiento no este vacía y continúe con un movimiento encadenado de forma que no acabaría su turno y sigue teniendo oportunidad de encontrar jugadas ganadoras.

Objetivo del juego: *[Capturar las semillas del contrincante, hasta que este no pueda realizar ningún movimiento legal durante su turno.]* Incluso el objetivo del juego parece indicar que la acumulación de material es parte esencial de la victoria.

Sin embargo, hay dos reglas de Omweso que complican la claridad que teníamos hasta este momento de que la acumulación de material, al menos la acumulación indiscriminada, es positiva para acercarnos a la victoria.

Captura: *[Se producirá una captura si se cumplen que el movimiento cae en una casilla de la filas interiores del tablero, no vacía (servira de marcador de captura), y que las dos casillas del jugador contrario opuestas de las misma columna están ocupadas.]*

Reentrada de semillas: *[Las semillas capturadas reentraran en el lado del jugador que ha realizado la captura mediante una siembra estándar partiendo de la misma casilla que partió el movimiento anterior que originó la captura.]*

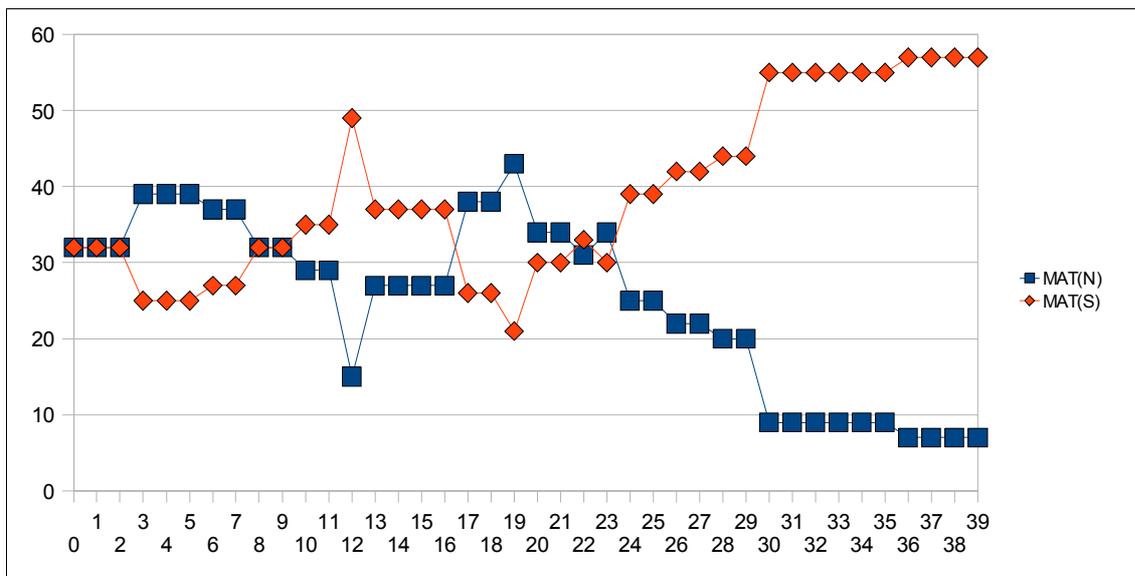
Esta claro que la única manera de aumentar nuestro material es capturarlo de las fichas del jugador enemigo. Tras capturarlo el material reentrará en nuestra zona del tablero y pasara a aumentar nuestro material en el juego. Para poder realizar la captura, necesitamos que el jugador contrario tenga material en dos casillas simultaneamente, situadas de una forma específica.

La situación es, que si solamente nos limitamos a acumular material de forma indiscriminada sin cuidar su distribución en el tablero, el jugador con más material, que hasta ahora parecía tener una posición favorable en el juego, tendrá una probabilidad más alta de tener material desprotegido susceptible de captura.

Y curiosamente el jugador con menos material, estadísticamente y sin tener que cuidar su distribución, al tener menor concentración de fichas en sus celdas es menos probable que se de la situación de que las semillas esten dispuestas de forma que sean susceptibles de ser capturadas.

Tampoco busquemos una situación con poco material pues también necesitamos tener material (*[el movimiento cae en una casilla de la filas interiores del tablero no vacía]*) para poder realizar capturas y de esa forma reducir la movilidad del contrario.

Resumiendo, la finalidad del juego es dejar al jugador contrario sin movimientos, para ello lo más lógico es reducir su material capturándolo. Pero la acumulación de material nos dejará más vulnerables a futuras capturas. Tendremos pues que acumular material pero cuidando su distribución para evitar ser vulnerables.



Gráfica 3: Efecto de la captura y reentrada de material

A continuación se muestra otra gráfica de la evolución del material a lo largo de una partida. Ambos jugadores utilizaron una política de acumular material con la esperanza de reducir la movilidad del contrario.

En la *gráfica 3*, se ve claramente el efecto que describíamos en párrafos anteriores. Al acumular material un jugador, si no lo hace con cuidado se pone en peligro para siguientes movimientos, por lo que como se ve claramente en la gráfica el material va pasando de un jugador a otro de forma alternada.

Tratándose de dos jugadores usando la política de maximizar el material, el material continuará pasando de un lado a otro del tablero, y el juego sólo se resolverá cuando aleatoriamente uno de los dos jugadores encuentre una jugada que reduzca tanto la movilidad del contrario que se reduzcan sus posibilidad de capturas futuras.

De donde deducimos que lo que deberemos buscar es la reducción de movilidad del contrario, pero sin sacrificar nuestra vulnerabilidad, eligiendo no las capturas que más material nos proporcionan sino las que más reducen la movilidad del contrario, sin olvidar tener cuidado con la distribución del material acumulado en nuestra parte del tablero.

Para ello, a continuación experimentaremos con conceptos como la dispersión del material por el tablero para intentar evitar esa vulnerabilidad ante la concentración de material, o conceptos como intentar reducir la movilidad sin pasar por una evaluación de la cantidad de material.

5.3.Duración de una partida. Cantidad de turnos y jugadas.

Tras jugar varias partidas de Omweso tenemos que, mientras unas partidas acaban con bastante premura, otras se alargan bastante más.

De hecho es bastante habitual que encontrándose en una posición aparentemente ganadora para uno de los jugadores, una sola jugada trastorne completamente el juego, volviéndolo a una situación neutral similar al comienzo de la partida, o incluso volcando la partida completamente y dejando una situación beneficiosa para el jugador que iba perdiendo hasta ese momento.

Esto es debido a la característica de reentrada de fichas capturadas de Omweso, que hace que una sola jugada transforme completamente el estado del tablero.

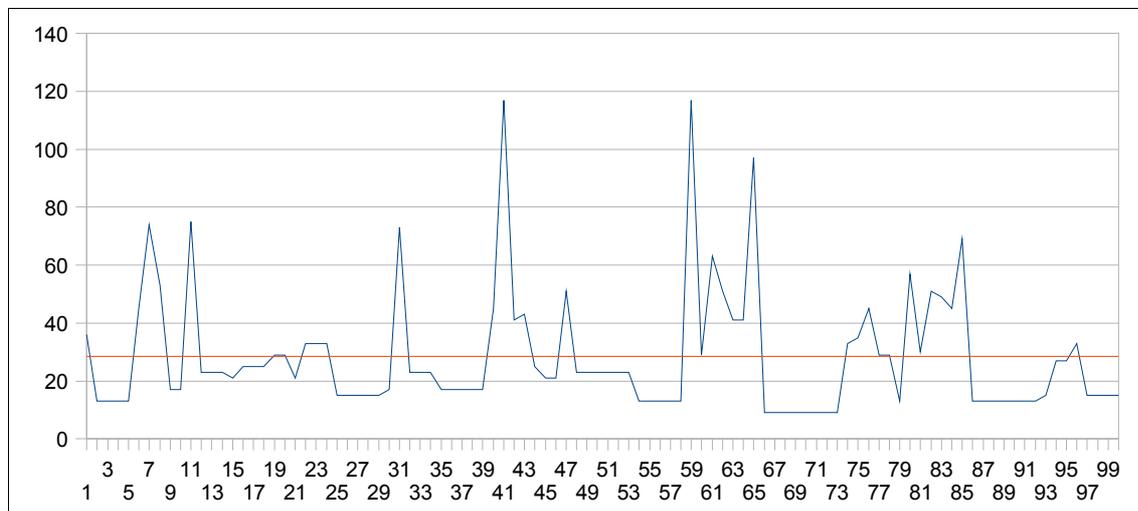
Este efecto lo hemos visto anteriormente, por ejemplo en el análisis del reparto de material a lo largo de una partida [5.2.Posesión del material durante una partida de *Omweso*].

Parece lógico pensar que esta variabilidad tan grande del tablero de una jugada a la siguiente pueda ocasionar situaciones dispares, como partidas que se desencadenan muy rápidamente, en muy pocas jugadas, o partidas que se alargan bastante en el caso de que alguno de los jugadores encuentre una jugada capaz de volcar completamente la situación con un solo movimiento.

Hagamos algún experimento, realizando un lote de varias partidas para analizar la variabilidad de la longitud de las partidas.

Se jugaron cien partidas de Omweso, entre dos jugadores de IA, uno de ellos era el jugador aleatorio, y el otro un jugador que utilizaba la función de evaluación que trata de maximizar el material. Se utilizó como algoritmo de búsqueda el Negamax, utilizando una profundidad de búsqueda igual a 3.

A continuación mostramos la gráfica que representa los resultados obtenidos.



Gráfica 4: Cantidad de turnos/movimientos de una partida.

La línea roja representa la media de movimientos durante esas cien partidas, que fue de 28,47 movimientos. La línea azul representa la cantidad de jugadas o movimientos que se realizaron en cada una de las partidas antes de acabar.

Destacar que en ningún momento estamos distinguiendo cual de los dos jugadores consiguió la victoria, y que la IA que utilizan cada uno de los jugadores no afecta demasiado a los resultados mostrados, pues se realizaron otros lotes de partidas de prueba y los resultados fueron similares.

Podemos apreciar claramente en la gráfica que la duración de las partidas es muy variable, de hecho pocas partidas están próximas a la media de jugadas, existiendo bastantes partidas que duraron menos de 20 jugadas, y otras partidas que duraron casi 120 jugadas, muy por encima de las 28,47 jugadas de media.

Las partidas se alargan o acortan en duración, debido a que algunas jugadas dan un vuelco completo a la partida en el momento menos esperado. De la dispersión de los resultados respecto de la media podemos deducir que existen un buen número de jugadas durante una partida de Omweso que cambian radicalmente el estado del tablero y de la partida.

5.4.Análisis de la función de evaluación.

Durante esta sección se analizará el rendimiento individual de las características y la forma de combinarlas linealmente para obtener una función de evaluación.

5.4.1.Rendimiento individual de las características.

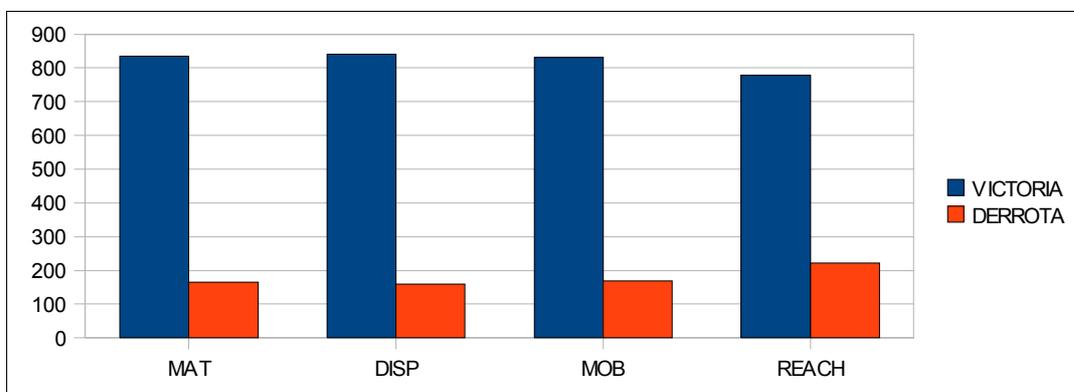
Como ya hemos comentado, una de las decisiones más complicadas es decir que peso le daremos a cada una de las características cuando pasen a formar parte de la función de evaluación. Unas características deberían tener más influencia que otras en la selección de jugadas durante el juego, unas deberán tener más influencia pues sobre el valor devuelto por la función de evaluación.

Una forma de elegir esos valores es haciendo múltiples pruebas, utilizando diferentes pesos que iríamos variando de manera que veamos si el hecho de dar más peso a una u otra característica hace que se alcance más o menos veces la victoria.

Haciendo las combinaciones de las tres características simultáneamente, y realizando variaciones en los pesos podría ser la solución, pero la verdad es que trabajar con las tres características hace que no podamos ver tan claro si la combinación esta funcionando peor por la variación de una u otra de las características.

En su lugar, si creamos cuatro funciones de evaluación, en la que cada una de ellas sólo se ve influenciada por una de las características, y a cada una de esas funciones las enfrentamos contra el mismo oponente, posiblemente la mejor opción sea un jugador aleatorio, podremos ver si cada característica por separado tiene un rendimiento mejor o peor en cuanto a conseguir victorias se refiere.

Y si una característica por si sola consigue vencer más veces a un jugador aleatorio que otra característica, parece al menos lógico, que en una función de evaluación en la que combinemos todas las características, podría ser interesante dar más peso a las características que funcionaron mejor por si solas.



Gráfica 5: Rendimiento individual contra un jugador aleatorio

En la *gráfica 4*, vemos los datos representados para analizarlos más fácilmente de una forma visual. A continuación presentamos de forma tabular los datos en la *tabla de resultados 3*.

	MAT	DISP	MOB	REACH
VICTORIA	835	840	832	779
DERROTA	165	160	168	221

TablaResultados 3: Resultados individuales

En la *gráfica 4* podemos ver los resultados del experimento que hemos definido anteriormente en esta misma sección. Se enfrentó a cuatro jugadores que usaban como función de evaluación por separado cada una de las características, contra 4 jugadores aleatorios. Los enfrentamientos fueron a 1000 partidas por cada característica, haciendo un total de 4000 partidas.

Destacar que cualquiera de las cuatro características individualmente consiguen un rendimiento positivo obteniendo un buen número de capturas contra un jugador aleatorio. Quizas la última característica en ser testeada, el alcance, esta obteniendo un rendimiento ligeramente inferior. Aunque las diferencias de resultados entre características sean muy similares cualquier pequeña diferencia de rendimiento se acentuará cuando juguemos contra un jugador con una inteligencia más elaborada que un jugador aleatorio.

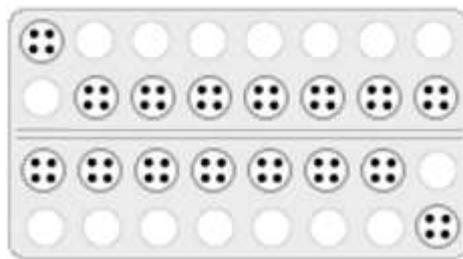
5.4.2. Combinación lineal de las características.

Esta claro, en nuestro caso, que tanto el material como la dispersión o la movilidad por si solas formarían una función de evaluación interesante, pero no desestimemos reforzar la función de evaluación combinándolas todas de forma que la fortaleza de unas compense la debilidad de las otras.

Para entender como la combinación de características puede reforzar nuestra función de evaluación veamos los dos casos extremos: la evaluación de la diferencia de material que fue la que mejores resultados individuales consiguió, y la evaluación del alcance que obtuvo menos menos victorias.

Parece lógico pensar, que si en una función de evaluación constituída solamente por la evaluación del material, que tiene un buen rendimiento, añadimos, aunque sea con un peso infimo, una componente que evalúa el alcance, contagiara ese mal rendimiento al conjunto global, pues por poco peso que tenga estará restando peso a la característica que tiene un mejor rendimiento individual.

Nada más lejos de la realidad. Veamos un ejemplo, en el que veremos claramente que el alcance puede complementar y reforzar los resultados obtenidos por el material individualmente:



Tablero 1: Posición neutral

En el *tablero 1* podemos ver una posición típica de comienzo de partida en Omweso. Es una posición neutral, ambos jugadores tienen la misma cantidad de material, y lo que es más interesante para nuestro ejemplo, ninguno de los movimientos disponibles genera ninguna captura. Imaginemos pues que tenemos la posición descrita por el *tablero 1*, y que estamos utilizando una función de evaluación que utiliza como única característica la evaluación de la diferencia de material.

Como hemos dicho, ninguna de las jugadas disponibles genera ninguna captura (sin analizar turnos en profundidad). Si no se genera ninguna captura no existirá ningún cambio en la diferencia de material, el material cambiará de posición pero cada jugador seguirá teniendo la misma cantidad de material.

Lo que ocurrirá con nuestra función de evaluación es que, si usa solamente la diferencia de material, encontrará que todas las jugadas conseguirán la misma puntuación, por lo que no será capaz de decidir si alguna de las jugadas es mejor.

En estos casos en los que la diferencia de material no nos sirve para discernir entre jugadas, sería interesante evaluar alguna otra característica que nos ayudara a decidirnos por unas jugadas u otras.

En este caso, hemos analizado una deficiencia de la evaluación de la diferencia material, pero podemos tener por seguro, que cualquiera de las características utilizadas individualmente tiene sus deficiencias (jugadas que no pueden identificar como buenas o malas) y será pues en una combinación de varias características en una misma función de evaluación donde encontraremos un funcionamiento compensado, y posiblemente mejorado:

$$F(x) = \sum_{i=1}^k \omega_i f_i = \omega_1 f_1 + \omega_2 f_2 + \dots + \omega_k f_k$$

Justificado el beneficio de combinar varias características en nuestra función de evaluación, sólo nos quedará, decidir qué pesos daremos a cada una de nuestras características en nuestra función de evaluación.

Volviendo a nuestro experimento de enfrentar cada característica individualmente a un jugador aleatorio y comparar el número de victorias, podríamos utilizar esos resultados para decidir un peso para cada una de nuestras características.

Asignar los pesos de cada una de las características proporcionalmente a las victorias obtenidas de enfrentar cada una de ellas individualmente contra un jugador aleatorio puede ser un buen comienzo, pero la realidad es que un jugador aleatorio ofrece una oposición tan poco consistente que los resultados obtenidos de este experimento no arrojan ningún resultado demasiado homogéneo.

Así pues realizaremos otro experimento. Enfrentaremos a cada una de las características individualmente contra cada una de las otras características individuales, observaremos los resultados, e intentaremos sacar alguna conclusión, que probablemente sí nos dará una solución mejorada al problema de los pesos de las características en nuestra combinación lineal.

Se ha realizado un experimento consistente en enfrentar jugadores utilizando cada una de las características de forma independiente contra otros jugadores que utilizaban el resto de las características. Cada jugador utilizaba como función de evaluación una y sólo una de las características con todo el peso en su función de evaluación.

El experimento se ha realizado en dos ocasiones, representado los resultados cada una de las dos tablas siguientes:

	<i>MAT</i>	<i>DISP</i>	<i>MOB</i>	<i>REACH</i>	<i>SCORE</i>
<i>MAT</i>	-	430	480	647	1557/3000
<i>DISP</i>	570	-	531	651	1752/3000
<i>MOB</i>	520	469	-	685	1674/3000
<i>REACH</i>	353	349	315	-	1017/3000

TablaResultados 4: Enfrentamiento individual de las características I

	<i>MAT</i>	<i>DISP</i>	<i>MOB</i>	<i>REACH</i>	<i>SCORE</i>
<i>MAT</i>	-	451	506	697	1654/3000
<i>DISP</i>	549	-	496	629	1674/3000
<i>MOB</i>	494	504	-	632	1630/3000
<i>REACH</i>	303	371	368	-	1042/3000

TablaResultados 5: Enfrentamiento individual de las características II

Los resultados de las celdas representan el número de victorias del jugador que utiliza la característica de la fila como función de evaluación contra el jugador que usa la característica de la columna. Cada casilla representa el número de victorias sobre un bloque de 1000 partidas consecutivas con la misma configuración.

Para intentar trabajar con unos resultados más precisos, estadísticamente hablando, calcularemos una tercera tabla a partir de las dos primeras, en la que los resultados serán la media aritmética de los valores encontrados en cada celda de las dos tablas precedentes, y esta nueva tabla contendrá los datos con los que realizaremos nuestros cálculos.

	<i>MAT</i>	<i>DISP</i>	<i>MOB</i>	<i>REACH</i>	<i>SCORE</i>
<i>MAT</i>	-	440	493	672	1605/3000
<i>DISP</i>	559	-	513	640	1712/3000
<i>MOB</i>	507	486	-	658	1651/3000
<i>REACH</i>	328	360	341	-	1029/3000

TablaResultados 6: Resultado medio de enfrentar las características

Los nuevos resultados, una vez más al igual que en experimentos anteriores, denotan que la dispersión es la característica que mejor rendimiento proporciona individualmente, mientras que el alcance fue de nuevo la que obtuvo peores resultados.

De lo que se trata es de buscar es el peso relativo (ω_i) que tendrá cada una de las características comparada con las otras. Asumiremos que la proporción que existe entre la cantidad de victorias de cada característica individual será equivalente a la proporción que debería existir entre los pesos, es decir, los pesos (ω_i) deberían mantener entre ellos la misma proporción que mantienen el número de victorias.

Partiendo de esas premisa de que respetarán la misma proporción, lo único que haremos para calcular nuestros pesos (ω_i) será normalizarlos, dividiéndolos por el valor más pequeño de su columna, obteniendo de esta forma los coeficientes que se muestran a continuación:

$$\omega_{mat} = \frac{C_{mat,disp}}{(\min(C_{i,disp}))} = \frac{440}{360} = 1,22$$

$$\omega_{mat} = \frac{C_{mat,mob}}{(\min(C_{i,mob}))} = \frac{493}{341} = 1,44$$

No se ha utilizado la columna de la característica del alcance (*reach*) para obtener un tercer valor de ω_{mat} , porque al tener esta característica un rendimiento bastante peor desvirtuaría los cálculos para el efecto que queremos conseguir.

Pero ahora que tenemos dos valores diferentes para ω_{mat} , tendremos que decidir que hacemos con dichos valores pues necesitaremos un solo valor para el coeficiente de peso. Lo que podremos hacer es asumir que ω_{mat} pertenecerá al intervalo marcado por los dos valores obtenidos, y que una buena aproximación podría ser el punto medio de dicho intervalo.

$$\omega_{mat} \in [1,22 ; 1,44]$$

$$\omega_{mat} = \frac{1}{2}(1,22 + 1,44) = 1,33$$

Ya tenemos nuestro valor estimado medio para el peso ω_{mat} de la característica del material en nuestra función de evaluación.

Realizaremos el mismo proceso para calcular el peso del resto de características y obtendremos así todos los coeficientes que necesitamos para nuestra función de evaluación.

$$\omega_{disp} = \frac{C_{disp,mat}}{(\min(C_{i,mat}))} = \frac{559}{328} = 1,70$$

$$\omega_{disp} = \frac{C_{disp,mob}}{(\min(C_{i,mob}))} = \frac{513}{341} = 1,50$$

$$\omega_{mob} = \frac{C_{mob,mat}}{\min(C_{i,mat})} = \frac{507}{328} = 1,54$$

$$\omega_{mob} = \frac{C_{mob,disp}}{\min(C_{i,disp})} = \frac{486}{360} = 1,35$$

De donde procederemos de la misma manera, definiendo los intervalos a los que pertenece cada uno de los pesos (ω_i), y calculando el punto medio del intervalo como valor del coeficiente que utilizaremos para cada característica en nuestra función de evaluación.

$$\omega_{disp} \in [1,50; 1,70]$$

$$\omega_{disp} = \frac{1}{2}(1,50 + 1,70) = 1,60$$

$$\omega_{mob} \in [1,35; 1,54]$$

$$\omega_{mob} = \frac{1}{2}(1,35 + 1,54) = 1,44$$

Ahora que ya tenemos nuestros valores para ω_i , sólo nos quedará montar nuestra función de evaluación utilizando esos valores y hacer algunos experimentos para comprobar los resultados.

$$\omega_{mat} = 1,33; \omega_{disp} = 1,60; \omega_{mob} = 1,44; \omega_{reach} = 1$$

Montemos pues nuestra función de evaluación resultante de la combinación lineal de las características utilizando como coeficientes cada uno de los pesos calculados.

$$F_1 = \sum_{i=1}^k \omega_i f_i = (1,33 f_{mat}) + (1,60 f_{disp}) + (1,44 f_{mob}) + (1,00 f_{reach})$$

5.4.3.Eliminación de algunas características.

Hemos visto durante la experimentación que una de las características propuestas, el alcance, obtuvo en todos los casos un peor rendimiento, y por tanto menos victorias.

Vemos que debido a esto nuestro razonamiento para obtener una combinación lineal de las características otorgaba al alcance el menor de los pesos en la función de evaluación.

Proponemos el siguiente razonamiento: si el alcance tiene un mal rendimiento, quizás no sea realmente beneficioso incorporarlo a nuestra función de evaluación, ni siquiera aunque sea con un peso muy bajo. Nos gustaría pues hacer un experimento, eliminando al alcance de nuestra función de evaluación.

	<i>MAT</i>	<i>DISP</i>	<i>MOB</i>
<i>MAT</i>	-	440	493
<i>DISP</i>	559	-	513
<i>MOB</i>	507	486	-

TablaResultados 7: Resultados tras eliminar la característica del alcance

En la *tabla de resultados 7* vemos los datos obtenidos de enfrentar cada una de las características 1000 veces con profundidad igual a 3, pero eliminando la característica del alcance. Obviamente los resultados de los enfrentamientos son los mismos, salvo por el detalle de que han desaparecido la fila y la columna de la característica eliminada.

Sin embargo veremos que al eliminar los datos relativos a dicha característica los cálculos a los que nos llevará el razonamiento para el cálculo de los pesos cambiarán sustancialmente, pues esos valores eliminados al ser los valores más bajos que existían entre los resultados nos estaban condicionando el resultado de nuestros cálculos.

Procederemos a calcular los pesos relativos de la misma forma que lo hicimos anteriormente, simplemente normalizando los valores, dividiendolos por el menor de los valores de su columna para de esta forma obtener valores que mantendrán la misma proporción y lo más simple posibles.

$$\omega_{mat} = \frac{C_{mat,disp}}{(\min(C_{i,disp}))} = \frac{440}{440} = 1,0$$

$$\omega_{mat} = \frac{C_{mat,disp}}{(\min(C_{i,disp}))} = \frac{493}{493} = 1,0$$

Vemos que, al eliminar la característica del alcance, la característica que ha pasado a ser la que peor rendimiento tiene es la evaluación del material, y vemos a su vez como en este caso nuestro razonamiento le otorga un peso $\omega_{mat}=1$.

Calculemos el resto de los pesos de acuerdo al mismo método que hemos utilizado antes completando así todos los coeficientes que necesitaremos para nuestra nueva combinación lineal.

$$\omega_{disp} = \frac{C_{disp,mat}}{(\min(C_{i,mat}))} = \frac{559}{507} = 1,10$$

$$\omega_{disp} = \frac{C_{disp,mat}}{(\min(C_{i,mat}))} = \frac{513}{493} = 1,05$$

$$\omega_{mob} = \frac{C_{mob,mat}}{(\min(C_{i,mat}))} = \frac{507}{507} = 1,00$$

$$\omega_{mob} = \frac{C_{mob,disp}}{(\min(C_{i,disp}))} = \frac{486}{440} = 1,10$$

Veamos los intervalos en los que se mueven nuestros coeficientes, y utilizemos el valor medio, punto medio del intervalo como valor más representativo para nuestro coeficiente.

$$\omega_{mat} \in [1,00; 1,00]$$

$$\omega_{mat} = 1$$

$$\omega_{disp} \in [1,05; 1,10]$$

$$\omega_{disp} = \frac{1}{2}(1,05 + 1,10) = 1,07$$

$$\omega_{mob} \in [1,00; 1,10]$$

$$\omega_{mob} = \frac{1}{2}(1,00 + 1,10) = 1,05$$

Montemos pues nuestra función de evaluación resultante de la combinación lineal de las características utilizando como coeficientes cada uno de los pesos calculados.

$$F_2 = \sum_{i=1}^k \omega_i f_i = (1,00 f_{mat}) + (1,07 f_{disp}) + (1,05 f_{mob})$$

Vemos que en esta nueva función de evaluación F_2 , los valores de los coeficientes son todos mucho más próximos a 1. Esto lo único que significa es que la característica eliminada era la única más alejada, en cuanto a rendimiento, lo que hacía las características con mejores rendimientos se distanciarán con coeficientes bastante más grandes que la unidad.

También debido a esto, da la sensación de que los coeficientes son mucho más parecidos al ser todos más pequeños, pero en realidad las proporciones entre ellos eran las mismas o muy similares en F_1 .

Pasemos a comprobar si la eliminación de una de las características supone una mejora en el rendimiento de la función F_2 frente a la función inicial F_1 . Para ello enfrentaremos a F_2 contra las características individuales igual que hicimos con F_1 en su momento, y además añadiremos un enfrentamiento directo entre F_2 y F_1 .

Tras realizar todos los experimentos, los resultados obtenidos de los enfrentamientos fueron los siguientes:

	<i>F1</i>	<i>F2</i>	<i>RAND</i>	<i>MAT</i>	<i>DISP</i>	<i>MOB</i>	<i>REACH</i>
<i>F1</i>	-	503	895	490	443	514	681
<i>F2</i>	491	-	892	499	525	523	690

TablaResultados 8: Resultados comparativos de F_1 y F_2

Como en las ocasiones anteriores, los experimentos se realizaron utilizando una profundidad de búsqueda igual a 3, y el número reflejado en cada celda de la tabla representa el número de victorias de la función que se encuentra en la fila, contra cada una de las funciones expuestas en las columnas, en enfrentamientos de 1000 partidas consecutivas.

De los resultados originados podemos deducir varias cosas. En primer lugar, vemos que los enfrentamientos directos entre F_1 y F_2 obtienen valores muy próximos a 500 victorias sobre 1000, es decir, ambas funciones son prácticamente igual de fuertes al enfrentarlas entre ellas. A pesar de que en la tabla de resultados 8 la función F_2 pierde más veces contra F_1 , lo cierto es que es que las victorias de F_1 sólo son ligeramente superiores. Ambas funciones F_1 y F_2 , tienen una pequeña componente aleatoria en su evaluación, es muy complejo tener una función de evaluación completamente precisa por lo que en mayor o menor medida siempre tienen una componente aleatoria, lo que podría justificar que en otro experimento similar esa pequeña diferencia fuese aleatoriamente en favor de F_2 , por lo que no podremos sino calificar esa pequeña diferencia de no relevante.

En segundo lugar vemos, que los resultados de enfrentar a F_2 contra cada una de las características individuales, son ligeramente superiores, en casi todos los casos, que los obtenidos por F_1 . Aún en el caso de que no mejorara absolutamente nada los resultados, siempre que no los empeorara, la eliminación del alcance en F_2 redundará, cuando menos, en una mejora en el consumo tiempo u otros recursos utilizados por la búsqueda, de modo que podría utilizarse con un algoritmo de búsqueda a una profundidad mayor.

Por esta razón recomendamos la eliminación de la característica del alcance de nuestra función de evaluación, convirtiéndose F_2 en nuestra función preferida.

5.5.Resultados contra jugadores humanos.

Pasaremos a realizar algunos experimentos de juego en los que enfrentaremos a nuestra función F_2 contra jugadores humanos. Obviamente, los jugadores humanos no están atados a ningún parámetro ni estrategia prefijada por lo que estos experimentos pueden dar unos resultados muy diferentes dependiendo de la habilidad de los humanos que tomen parte en el experimento, pero tomaremos como muestra los siguientes resultados:

	F_2 PROF3	F_2 PROF4
HUMANO1	5/10	1/10
HUMANO2	4/10	2/10

TablaResultados 9: Resultados de F_2 contra jugadores humanos

El jugador controlado por el ordenador tenía como configuración de IA el algoritmo de búsqueda Alfabeta, la función de evaluación F_2 , y como profundidad 3 y 4, en el primer y segundo experimento, respectivamente.

Como podemos ver en la tabla de resultados 9, en todo momento los jugadores humanos obtienen un número de victorias por debajo de la mitad de las 10 partidas jugadas. En el caso de utilizar una profundidad de búsqueda de 4, el número de victorias de los jugadores se reduce a 2 victorias de cada 10 partidas jugadas. Adicionalmente, se jugaron varias partidas a profundidad 6, a modo de prueba, y los resultados fueron que los jugadores humanos perdieron en todos los casos, y además en un número reducido de turnos.

Por lo que podemos concluir, que si bien los resultados de jugadores humanos pueden mejorar si los humanos que participan tienen una mayor habilidad o experiencia jugando a Omweso, en los experimentos se ha utilizado una profundidad no demasiado alta que podría ser incrementada sin afectar negativamente al rendimiento del sistema, y que produciría un más que digno adversario contra jugadores humanos. El sistema puede trabajar con profundidades iguales a 6 sin una caída drástica del rendimiento.

Capítulo 6

Conclusiones

6.Conclusiones.

Durante este capítulo intentaremos sacar algunas conclusiones de todo lo planteado en capítulos anteriores con la finalidad de tener una visión global del trabajo realizado.

6.1.Repaso de los objetivos.

Revisando los objetivos propuestos en la sección [3.*Objetivos*] podemos concluir que los objetivos se han cumplido, incluso en algunos casos se han desarrollado soluciones que van más allá de los objetivos planteados.

6.1.1.El tablero.

Se ha diseñado e implementado un software capaz de reproducir un tablero de Omweso permitiéndonos representar tanto el tablero, como las semillas contenidas en cada celda, así como permitirnos realizar movimientos con las mismas.

El tablero no solamente representa el estado del juego y permite realizar movimientos, sino que el propio tablero implementa el conjunto de reglas de Omweso gestionando automáticamente tanto el movimiento de siembra, como la legalidad de los movimientos, ejecución de las capturas y reentrada de semillas, así como las condiciones de victoria, dejando a los jugadores únicamente la tarea de tomar la decisión de qué jugadas realizar durante su turno.

6.1.2.El interfaz.

Como se planteó en los objetivos se ha implementado un interfaz para poder interactuar con el sistema. Se trata de un interfaz de línea de comandos, guiado por menus contextuales que contiene la descripción y sintáxis de utilización de cada uno de los comandos. En el caso de encontrar algún error en el comando introducido el interfaz automáticamente nos proporciona una descripción y la sintáxis correcta mediante mensajes claro y fácilmente interpretables.

Además el sistema mantiene una memoria de los últimos comandos ejecutados, para su reutilización, mediante las funciones de histórico de la librería readline.

El interfaz de línea de comandos presenta comandos tanto para la gestión de la IA, gestión de carga y salvado de partidas, funcionalidades para experimentación y obtención de resultados como lotes de partidas, y por supuesto, comandos para ordenar movimientos y jugadas durante una partida.

Adicionalmente al interfaz de línea de comandos, para la representación del tablero y para ordenar jugadas por parte de un jugador humano, se ha implementado un interfaz gráfico que permite interactuar con el tablero mediante la utilización de un ratón a la vez que muestra gráficamente el estado del juego.

6.1.3.Los diferentes agentes de IA.

Se planteó en los objetivos el diseño e implementación de varios agentes, uno de ellos controlado por un jugador humano y otros agentes autónomos con toma de decisiones controlada por algoritmos de búsqueda clásicos de IA.

Se ha implementado un agente que puede ser controlado por un jugador humano, tanto usando el interfaz de línea de comandos, como utilizando el interfaz gráfico mediante el uso de un ratón.

También se ha implementado un agente de IA configurable capaz de utilizar cualquiera de los algoritmos de IA clásicos implementados (negamax, alfabeta, scout), o bien configurar un agente totalmente aleatorio, o utilizar para la toma de decisiones cualquiera de las funciones de evaluación implementadas entre las que se encuentran algunas basadas en características individuales y otras configuradas en forma de combinación lineal de varias características individuales.

6.1.4.Lotes de partidas y log de resultados.

Entre los objetivos se encontraba la posibilidad de ejecutar un gran número de partidas consecutivas y sin intervención de un humano con la finalidad de realizar experimentación enfrentando varios agentes autónomos.

Se ha implementado la opción de lotes de partidas, mediante la cual podremos definir el número de partidas a ejecutar, la configuración de la partida y los jugadores, además de darnos la posibilidad de grabar los resultados de las múltiples partidas a un log para su posterior análisis.

6.1.5.Objetivos de IA.

Se han implementado agentes de IA configurables, pudiendo configurarse desde el algoritmo de búsqueda utilizado, la profundidad de la búsqueda, así como elegir para la toma de decisiones entre varias funciones de evaluación.

Adicionalmente, y para facilitar el estudio del comportamiento de la IA, se ha implementado un sistema de log de movimientos, que guardará todos las jugadas realizadas hasta el momento durante un juego, y lo que es más importante nos permitirá deshacer movimientos, permitiéndonos por ejemplo ver las consecuencias de realizar diferentes jugadas desde un mismo punto de la partida. Este log de movimiento se guardará durante el guardado de la partida por lo que nos permitirá volver a analizarlo o rejugar la misma partida de nuevo en un futuro para estudiar el comportamiento de los agentes.

6.2.Implementación del sistema.

Veremos los aspectos de la implementación del sistema que destacan especialmente bien sea por su aporte a la funcionalidad del sistema, posible reutilización futura, portabilidad o usabilidad.

6.2.1.Un sistema portable.

El sistema ha sido implementado en C++ como lenguaje de programación, y como librerías adicionales se han utilizado las librerías SDL para el sistema gráfico, y la librería readline para la gestión de la línea de comandos.

En cuanto al lenguaje C++ tenemos compiladores disponibles para casi cualquiera de las arquitecturas hardware y sistemas operativos disponibles hoy día. De la misma manera las librerías SDL y readline han sido portadas a muchas plataformas de desarrollo.

De esta manera, no debería ser demasiado complejo portar este proyecto y tener una versión para otros sistemas operativos, como Ms Windows o MacOS , o incluso para otras arquitecturas pues las librerías han sido portadas a otras arquitecturas como ARM, y posiblemente a RISC.

La documentación del proyecto, incluido este mismo documento, han sido creados con Open Office¹⁰, una herramienta disponible también en los sistemas operativos más populares.

¹⁰ Open Office: The free and open productivity suite. (<http://es.openoffice.org>)

6.2.2.Un sistema ampliable.

El sistema ha sido diseñado e implementado con clases fácilmente extensibles y ampliables. Los dos elementos que serían más susceptibles de extender serían las clases de los algoritmos de búsqueda y añadir más funciones de evaluación.

Así mismo la implementación del tablero, tanto textual como gráfica podría ser modificada fácilmente para trabajar con las reglas de juego de alguno de los otros juegos de la familia Mancala.

6.2.3.Un sistema eficiente y rápido.

Una de las primeras cosas que sorprenderá gratamente a cualquiera es la rapidez en la toma de decisiones del sistema, que incluso a profundidades de búsqueda superiores a 5-6, no necesita más de unos segundos para tomar la decisión.

Este aspecto más que interesante a la hora de jugar una partida, es especialmente interesante en fases de experimentación pues nos permitirá ejecutar lotes de un gran número de partidas desatendidas en un tiempo relativamente corto. De esta manera nos ha permitido realizar lotes de 1000 partidas para obtener los resultados utilizados durante el capítulo de resultados, que de otra manera no habría sido posible, o tendría que haber sido generados con un menor número de partidas lo que les habría dado una menor validez estadística.

6.3.Conclusiones desprendidas de los resultados.

A la vista de los datos obtenidos durante el capítulo de resultados podemos sacar algunas conclusiones interesantes sobre algunos aspectos del proyecto.

6.3.1.Rendimiento de los algoritmos de búsqueda.

Uno de los aspectos bien conocido, pero que queríamos comprobar que efectivamente mantenían dicho comportamiento en el sistema desarrollado es el rendimiento de los diferentes algoritmos de búsqueda implementados.

Así pues podemos concluir que, efectivamente, de los tres algoritmos utilizados (Negamax, Alfabeta y Scout), el Negamax es que el que da un peor rendimiento, analizando más nodos y tardando más tiempo, algo por otro lado lógico pues el negamax no es más que una simplificación sintáctica del minimax, y tanto el Alfabeta como el scout se basaron en el minimax y se crearon como mejoras de rendimiento de este.

Podemos confirmar como el Alfabeta, introduce una mejora sustancial en el número de nodos analizados, y por tanto el tiempo consumido por la búsqueda, gracias a la optimización de la poda, lo cual se agradece especialmente porque nos permite realizar búsquedas a una mayor profundidad en unos tiempos admisibles. El algoritmo Alfabeta ha sido el utilizado durante todos los experimentos realizados para obtener los datos utilizados en el capítulo de resultados.

Y por último, podemos comprobar como, según se esperaba, el Scout reduce aún más el número de nodos analizados gracias a su optimización de lanzamiento de una sonda previa al análisis de las ramas del árbol de búsqueda.

Como caso especial, concluir que si bien sí se han reducido el número de nodos analizados, en nuestro caso concreto, no ha supuesto una reducción en el tiempo consumido, o incluso en algunos momentos ha supuesto un ligero aumento del tiempo necesario.

La optimización del Scout basa su mejora en que los recursos necesarios para lanzar la sonda son mucho menores que los necesarios para analizar los nodos directamente, pero en nuestro caso el análisis de un nodos consume tan pocos recursos que el lanzamiento de la sonda en muchos casos con supone un aumento del rendimiento de la búsqueda.

De esta manera pues podemos decir que el algoritmo de búsqueda más recomendable, de los tres analizados, para la experimentación con esta implementación de Omweso es el Alfabeta, y es el que ha sido utilizado para toda la experimentación llevada a cabo para este proyecto.

6.3.2.Las funciones de evaluación.

Entre los objetivos del proyecto se encontraba la creación y estudio de varias funciones de evaluación para su uso por parte de los algoritmos de búsqueda.

Se han definido varias características simples (material, dispersión, movilidad y alcance) para evaluar el estado del juego, y se ha estudiado su comportamiento individual usándolas como funciones de evaluación sencillas.

La característica del material sería posiblemente la que evaluaría casi incoscientemente cualquier jugador humano sin tener que entrar en demasiada estrategia, y se ha comprobado que individualmente no funciona mal.

La característica de la movilidad, tampoco funciona mal, y además, teniendo en cuenta que la finalidad del juego es dejar sin movilidad al contrario tiene bastante sentido su utilización.

La característica de la dispersión, un poco menos intuitiva de comprender, es por otro lado la que mejores resultados individuales ha tenido. Si nos paramos a pensarlo un poco más en profundidad tiene sentido, pues la dispersión afecta directamente a la movilidad y al material.

Y por fin, tenemos el alcance, una característica con poco fundamento, basada en la regla que indica que todas las capturas se realizan en la fila interior del tablero, por lo que tener posibilidad de movimientos que acaben en esta fila del tablero puede tener sentido. Sin embargo, a pesar de no funcionar mal contra un jugador aleatorio, contra cualquier otra de las características funciona bastante mal.

Finalmente, se han creado varias funciones de evaluación como combinación lineal de las características individuales, usando como coeficientes o pesos lineales de cada característica la proporción de victorias obtenida de enfrentar las características individualmente. Viendo efectivamente, como era de esperar, que una función de evaluación producto de una combinación lineal de las características individuales es ligeramente más eficiente que las funciones simples.

Sin embargo, hemos de notar que todos los resultados obtenidos el número de victorias es bastante próximo al 50%, lo que indica que ninguna de las funciones de evaluación propuestas tienen una precisión suficiente para destacar en el número de victorias. Creemos que esto viene dado por la dificultad de encontrar características que evalúen con precisión el estado de la partida debido a la naturaleza cambiante de un tablero de Omweso, que viene dada por el movimiento de siembra y la reentrada de semillas durante las capturas.

En cualquier caso, las funciones de evaluación propuestas, utilizando profundidades de búsqueda superiores a 3, consiguen ganar claramente a jugadores humanos.

6.3.3.El movimiento de siembra y la reentrada de semillas.

El movimiento de siembra, característico de la familia Mancala, hace que tras cualquier jugada una gran parte del tablero cambie su estado, por lo que se hace complicado establecer una estrategia basada en el estado del tablero. De ahí viene la dificultad para encontrar funciones de evaluación precisas, pero de ahí viene también el interés del estudio de la familia Mancala. Los movimientos encadenados, y la reentrada de semillas capturadas enfatizan aún más este efecto, en el caso de Omweso.

Capítulo 7

Líneas futuras

7.Líneas futuras.

Este proyecto pretende ser un comienzo, y pretende aportar posibilidades y dejar puertas abiertas interesantes para su análisis y futuro estudio. Intentaré presentar algunas ideas con la esperanza de que puedan servir de inspiración o ayuda para alguien que pueda estar interesado en continuar este trabajo.

7.1.Mejoras a la implementación del sistema.

Una de las mejoras interesantes es la posibilidad de añadir funcionalidad online al sistema. La opción de jugar online, no sólo con la intención de que varios jugadores humanos puedan jugar desde dos lugares remotos, sino la posibilidad de que investigadores puedan enfrentar online a dos versiones modificadas, con nuevos algoritmos o nuevas funciones de evaluación, de Omweso.

Esto ya se ha llevado a cabo en otros proyectos de IA con bastante éxito. Un sistema online en el que cada investigador puede enfrentar sus soluciones al problema, manteniendo una especie de ranking convirtiendolo el algo similar a un torneo online. Este esquema fomenta la competitividad, e incentiva a investigadores de todos los niveles a proponer sus propias soluciones al problema, y como hemos dicho es algo que ha funcionado bastante bien en el pasado con otros proyectos.

Aparte de esta aplicación de la funcionalidad online del juego, otra bastante interesante podría ser la distribución de la carga y el consumo de recursos. De esta manera, podríamos tener, por ejemplo, a cada uno de los agentes de IA ejecutandose en máquinas diferentes y separadas del interfaz, repartiéndose de esta forma la carga, o simplemente tener a los agentes en máquinas mucho más potentes pero sin funcionalidades gráficas.

Por otro lado, ya a nivel más de implementación pura, podría ser interesante reprogramar el sistema mediante la utilización de varios hilos (threads) independientes. Utilizando por ejemplo hilos independientes para el sistema gráfico, la línea de comando, y por otro lado los jugadores, y la IA. Esto nos permitiría, por ejemplo, poder seguir interactuando con el sistema mientras la IA esta trabajando en las búsquedas, permitir a los agentes continuar con su búsqueda aún después de haber decidido y ejecutado su jugada, o simplemente independizar partes que por naturaleza no son secuenciales.

No podemos terminar, sin recomendar el que el sistema sea portado a otras plataformas o sistemas operativos, por ejemplo Ms-Windows, lo cual permitiría el acceso a más investigadores y usuarios, o a nuevas arquitecturas, por ejemplo RISC.

7.2.Mejoras a la IA del sistema.

Como ya se ha comentado anteriormente, la IA que se ha implementado, aunque cumple con su cometido, en ningún momento destaca especialmente obteniendo unos resultados muy aplastantes. Los algoritmos y técnicas de IA implementados pertenecen a los considerados como “clásicos”, y existen otras muchas técnicas que se podrían aplicar, entre las que podemos recomendar: nuevos algoritmos, funciones de evaluación no lineales, funciones de evaluación dependientes del turno, libros de aperturas, bases de datos de movimientos, técnicas de aprendizaje, etc.

Capítulo 8

Bibliografía

8.Bibliografía.

- B.Wemham. Omweso: The royal Mancala Game of Uganda. A General Overview of Current Research. 2007
- James S. Coleman, Play in Uganda: Omweso a Game People, UCLA African Studies Center, 1970
- Irving, Donkers & Uiterwijk. Solving Kalah. September 2000
- Romein & Bal. Solving the Game of Awari using Parallel Retrograde Analysis. October 2003
- David & Kendall. An Investigation, using Co-Evolution , to Evolve an Awari Player. 2002
- J.Donkers, J.Uiterwijk & A.Voogt. Mancala Games. Topics in Mathematics and Artificial Intelligence. 2001
- H.J.R. Murray; A History of Board-Games Other Than Chess; Oxford: The Clarendon Press, 1952, Chapter 7, pages 158-159
- Binmore, Ken, 2004. Teoría de juegos. Madrid. Mc Graw Hill.
- Murray S. Campbell and T.A. Marsland. A Comparison of Minimax Tree Search Algorithms. Department of Computer Science, University of Alberta, Edmonton, Alberta T6G 2H1, Canada.
- Korf, Richard E., 1991. Multi-player Alpha-beta pruning. Artificial Intelligence. Vol. 48. 99 – 111.
- Nilsson, Nils J., 2000. Inteligencia Artificial. Una nueva síntesis. Madrid. McGraw-Hill.
- Russell J., Stuart y Norvig, Peter, 2004. Inteligencia Artificial. Un enfoque moderno. 2a edición. Madrid. Pearson Prentice Hall.

Apéndices

Apéndices.

A continuación, se presentan los apéndices referenciados a lo largo de toda la documentación de este proyecto. Concretamente se presentan los pseudocódigos de los algoritmos analizados e implementados en el sistema.

Apéndice A.1. Pseudocódigo del Minimax.

Para una explicación detallada del algoritmo Minimax consultar la sección 2.7.2 de este mismo documento.

```
minimax(p:posicion) {
  i,w: integer;
  m,t: score_t;

  if (terminal(p)) then
    return (funcion_evaluacion(p));

  w=generar(pi);
  m =-∞;

  for (i=1 to w) do {
    t=minimax(pi);
    if ((t>m) and (tipo(pi)==Max) then m=t;
    if ((t<m) and (tipo(pi)==Min) then m=t;
  }
  return (m);
}
```

El minimax es el algoritmo en el que se basan los otros tres algoritmos utilizados para este proyectos por lo que veremos en su pseudocódigo que las similitudes son evidentes.

El resto de los algoritmos tratados veremos que vendrán a traer mejoras y optimizaciones a este esquema básico del Minimax.

Apéndice A.2. Pseudocódigo del Negamax.

Para una descripción más detallada del algoritmo Negamax consultar la sección 2.7.3. de este mismo documento.

```
negamax(p:posicion) {  
  m,i,t,w:integer;  
  if (terminal(p))  
    return (funcion_evaluacion(p));  
  
  w = generar(pi);  
  m = -∞;  
  
  for (i=1 to w) do {  
    t = - negamax(pi);  
    if (t>m) m = t;  
  }  
  return(m);  
}
```

El algoritmo Negamax viene a realizar una mejora simplemente sintáctica sobre el Minimax. Elimina trabajar con dos fases y distinguiendo entre nodos min y nodos max, por una sola tipología de nodos y una negación del valor de la fase anterior simplificando la sintaxis.

Apéndice A.3. Pseudocódigo del Alfabeta.

Para una descripción más detallada del algoritmo Alfabeta consultar la sección 2.7.4. de este mismo documento.

```
alfabeta(p:posicion; alpha,beta: integer) {
  m,i,t,w:integer;
  if (terminal(p))
    return (funcion_evaluacion(p));

  w = generar(p);
  m = -∞;

  for (i=1 to w) do {
    t = - alfabeta(pi, -beta, -m);
    if (t>m) m = t;
    if (m >= beta) return (m);
  }
  return(m);
}
```

El Alpha-beta, con la introducción de la poda y los límites alpha y beta, consigue reducir el número de nodos analizados drásticamente respecto al Minimax o Negamax.

Apéndice A.4. Pseudocódigo del Scout.

Para una descripción más detallada del algoritmo Scout consultar la sección 2.7.5. de este mismo documento.

```
scout(p:posicion) {
  m,i,t,w:integer;
  if (terminal(p))
    return (funcion_evaluacion(p));

  w = generar(p);
  m =-scout(p);

  for (i=2 to w) do {
    if (not test(p, -m))
      m = -scout(p);
  }
  return(m);
}
```

El Scout analiza el primer nodo de la forma habitual, pero para el resto de los nodos en lugar de analizarlos lanza una sonda mediante un procedimiento test() muy ligero y que consume muy poco recursos, y solamente termina analizando aquellos nodos en los que la sonda no ha conseguido desestimar.

```
test(p:posicion; v: integer) {  
  i,w:integer;  
  if (terminal(p))  
    if (funcion_evaluacion(p)>v) return (TRUE);  
    else return (FALSE);  
  
  w = generar(p);  
  
  for (i=1 to w) do {  
    if (not test(p, -m))  
      return (TRUE);  
  }  
  return(FALSE);  
}
```

La función `test(p:posicion; v: integer)` es la denominada como sonda en el algoritmo Scout. En algunas ocasiones en lugar de implementar una función `test`, se utiliza como sonda el propio Alpha-beta pero con una ventana de llamada mínima.

