Pontificia Universidad Católica de Valparaíso

Facultad de Ingeniería

Escuela de Ingeniería Informática

IMPLEMENTACIÓN DE ARIMAA UTILIZANDO OPTIMIZACIÓN POR COLONIA DE HORMIGAS

ÁNGEL ANDRÉ ABSÉ HIDALGO

INFORME FINAL DEL PROYECTO PARA
OPTAR AL TÍTULO PROFESIONAL DE
INGENIERO DE EJECUCIÓN EN INFORMÁTICA

Diciembre 2014

Pontificia Universidad Católica de Valparaíso

Facultad de Ingeniería

Escuela de Ingeniería Informática

IMPLEMENTACIÓN DE ARIMAA UTILIZANDO OPTIMIZACIÓN POR COLONIA DE HORMIGAS

ÁNGEL ANDRÉ ABSÉ HIDALGO

Profesor Guía: **José Rubio León**

Profesor Co-referente: **Ignacio Araya Zamorano**

Carrera: **Ingeniería de Ejecución en Informática**

Diciembre 2014

Dedicatoria

Dedicado especialmente a mi familia, la cual ha sido un pilar muy importante en mi vida. Ellos me han educado desde pequeño, dándome el apoyo necesario y el cariño incondicional.

Agradecimientos

A mis padres, los cuales han sido una fuente de apoyo constante. A mi hermano, quien ha representado una motivación extra. A Dios, por la fortaleza que me ha dado. Y a la Universidad, por las herramientas brindadas.

Índice

Resumen

El presente proyecto tiene como objetivo la implementación del juego Arimaa en un computador. Este juego, el cual fue creado por el ingeniero en computación Omar Syed, aparece como un desafío para los programadores teniendo reglas muy simples para los seres humanos, pero muy difíciles para los computadores.

Para poder implementar este tipo de programa, se debió realizar un estudio del juego, con el fin de conocer su historia, sus reglas y qué tan complicado es programar un buen programa de Arimaa. Se tuvo que estudiar los distintos algoritmos que se han utilizado, entre los cuales están Minimax y Poda Alfa-Beta, además de la Metaheurística Optimización por Colonia de Hormigas y sus variantes, las cuales han podido resolver distintos problemas con grafos. Finalmente se realizó un análisis de la solución basándose en pruebas a diferentes puzles.

Palabras claves: Arimaa, Minimax, Poda Alfa-Beta, Metaheurística, Optimización por Colonia de Hormigas, grafos.

Abstract

This project aims at the implementation of the game Arimma in a computer. This game, which has been created by the computer engineer Syed Omar, is a challenge for programmers having very simple rules for humans but difficult for computers.

To be able to implement this kind of program, it was needed an analysis of the game, in order to know it’s history, it’s rules and how complicated it is to program a good Arimma game. It was needed to study the different algorithms that have been used before, including Minimax and Poda Alfa-Beta; besides the Metaheuristic Ant Colony Optimization and its variants, which have been able to solve different problems with graphs. Finally, an analysis of the solution based on tests performed at different puzzles.

Keywords: Arimaa, Minimax, Poda Alfa-Beta, Metaheuristic, Ant Colony Optimization.

Lista de Figuras

Lista de Tablas

Glosario

Ajedrez: Juego de tablero para 2 jugadores, los cuales disponen de 16 piezas cada uno.

Arimaa: Juego de tablero, similar al ajedrez.

Autómata: Máquina que imita la figura de un ser animado.

Deep Blue: Máquina que en el año 1997 vence al campeón del mundo en Ajedrez Gary Kasparov.

Demonio: Son utilizados para implementar tareas desde una perspectiva global que no pueden llevar a cabo las hormigas.

Gary Kasparov: Ajedrecista ruso, el cual se convirtió en el campeón de ajedrez más joven de la historia en el año 1985.

Heurística: Algoritmo para problemas que no tienen una solución óptima bajo ciertas restricciones.

Metaheurística: Método heurístico para resolver un tipo de problema computacional general.

Minimax: Método de decisión para minimizar la pérdida máxima esperada en juegos con adversario y con información perfecta.

Omar Syed: Ingeniero en computación, el cual creó el juego Arimaa.

Poda Alfa-Beta: Variación del algoritmo Minimax, en el cual se desechan alternativas inferiores.

1 Introducción

Desde sus inicios, el computador ha servido para solucionar distintas problemáticas para el ser humano. Y es que gracias a su nivel de procesamiento, puede realizar cálculos de manera muy rápida. Esta herramienta a medida que han pasado los años y la tecnología va creciendo se va volviendo más importante.

Uno de los campos de la informática que ha sido estudiado por varios años es la inteligencia artificial. Esto debido a que cada vez va apareciendo más problemáticas que requiere imitar la inteligencia humana. Dentro de estas problemáticas están los de optimización combinatoria, perteneciente a la clase de problemas NP-duros, lo que significa que no existe un algoritmo conocido que los resuelva en un tiempo polinomial.

Para poder solucionar estos tipos de problemas existen los algoritmos exactos y los algoritmos aproximados. Los primeros intentan encontrar una solución óptima al problema; sin embargo, muchas veces esta técnica demora mucho en encontrar la solución, por lo cual se hace necesario implementar otro tipo de algoritmo que no necesariamente encuentre la solución óptima, pero si una de buena calidad; estos son los llamados algoritmos aproximados.

Un ejemplo en donde se requiere una búsqueda rápida, debido al poco tiempo límite que hay, es el Ajedrez. El implementar un programa que sea capaz de realizar estrategias por sí mismo no es una tarea sencilla. Esto último sumado a poder ganarle a un jugador profesional resulta una tarea sumamente complicada. Es por esto que es tan destacado el caso del año 1997 cuando la máquina Deep Blue vence a Gary Kasparov.

Es precisamente gracias a este último suceso que el ingeniero en computación Omar Syed decide inventar un nuevo juego, el cual se pueda jugar con la misma cantidad de piezas que el ajedrez y con reglas que puedan ser fáciles de entender para un humano, pero que para un computador sea difícil jugarlo. Este juego fue llamado Arimaa.

En este proyecto se verá la implementación del juego Arimaa en un computador utilizando técnicas de inteligencia artificial. Para esto se realizará un estudio del juego y los distintos algoritmos que se han utilizado tanto en Arimaa como en el Ajedrez. Además se analizará de qué manera implementar la solución.

Finalmente se realizará un análisis comparando los resultados de la implementación de la metaheurística de Optimización por Colonia de Hormigas con el algoritmo Poda Alfa-Beta utilizada en el proyecto de tesis “El Desafío de Arimaa” realizada por Andrés Cerón Lillo [1].

2 Análisis de Objetivos y Metodología

2.1 Objetivos

2.1.1 Objetivo General

Implementar un programa de Arimaa que sea capaz de jugar una partida con un usuario.

2.1.2 Objetivos Específicos

- Estudiar, analizar y explicar el juego Arimaa.
- Estudiar, analizar y explicar el algoritmo de "Optimización por Colonia de Hormigas" y sus variantes.
- Seleccionar e implementar alguna variante del algoritmo "Optimización por Colonia de Hormigas" en el programa Arimaa.
- Probar el funcionamiento del algoritmo utilizado en el juego, usando puzles predeterminados.

2.2 Metodología

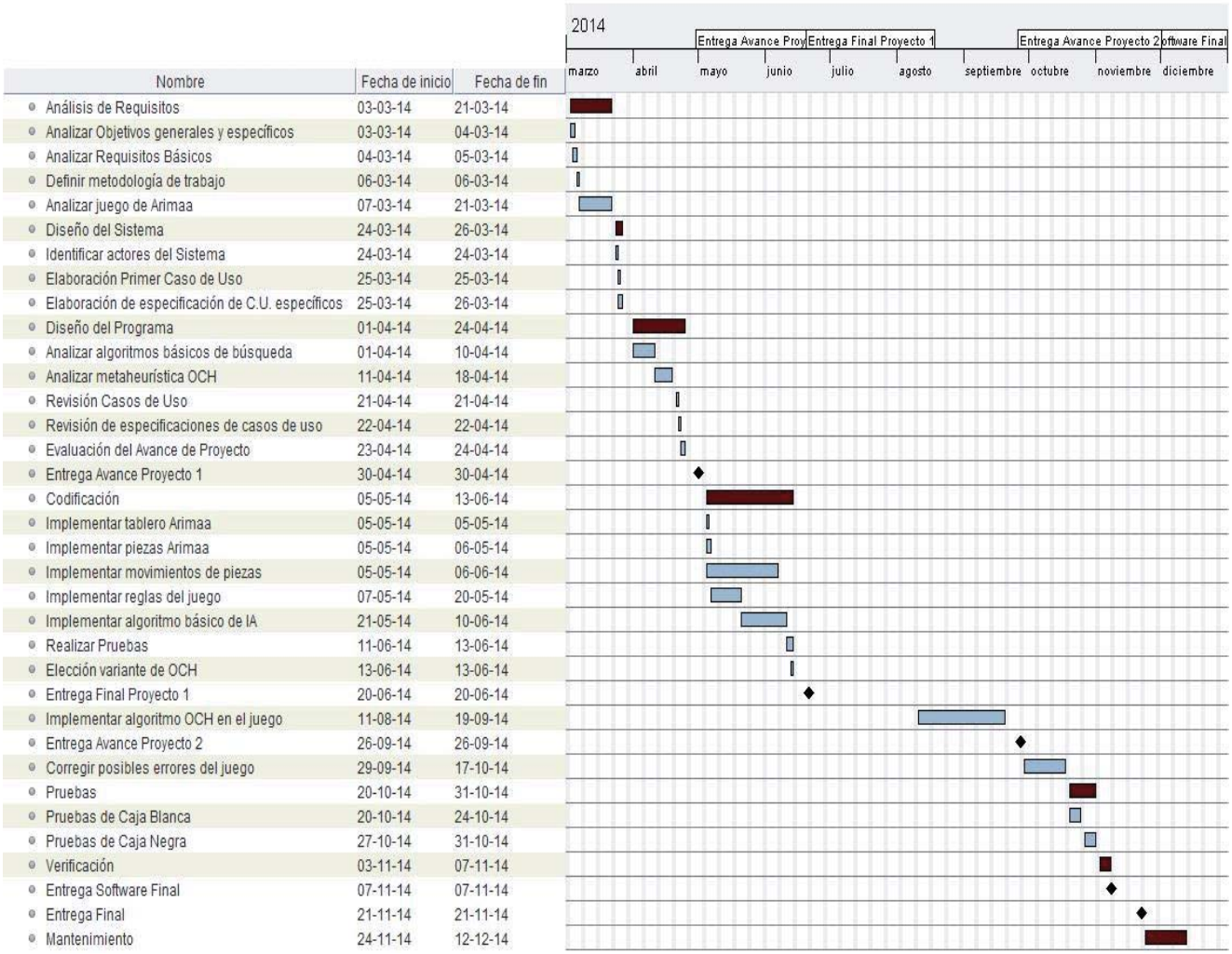
2.2.1 Modelo de Proceso Utilizado

El modelo de proceso que se escogió para el desarrollo del software es el "Modelo en Cascada". Este modelo ordena de forma rigurosa todas las etapas del ciclo de vida del software. La idea es que al final de cada una de estas etapas se pueda llevar a cabo una revisión, con el fin de determinar si se está listo para continuar a la etapa siguiente.

Las etapas del "Modelo en Cascada" son las siguientes:

1. Análisis de Requisitos: Se analizan las necesidades que el software debe cubrir al final del proceso.
2. Diseño del Sistema: En esta etapa se dividen los requerimientos en subsistemas, se establece una arquitectura completa y se identifican y describen las relaciones fundamentales del software.
3. Diseño del Programa: En esta fase es donde se realizan los algoritmos necesarios para el cumplimiento de los requerimientos analizados en la primera etapa. Además se realiza un estudio para ver qué herramientas usar en la etapa de Codificación.
4. Codificación: Es la fase donde es implementado el código fuente. Se hace uso de prototipos, así como pruebas y ensayos para corregir errores.
5. Pruebas: Se comprueba que el sistema funciona de manera correcta y que cumpla todos los requisitos.
6. Verificación: En esta fase el usuario final ejecuta el sistema. Para esto, ya se realizaron las correspondientes pruebas para comprobar que el sistema no falle.
7. Mantenimiento: Esta fase es la más larga de todos los procesos de desarrollo. El sistema es ejecutado, corrigiendo todos los errores no descubiertos en las etapas anteriores. [8]

2.2.2 Planificación



Nombre	Fecha de inicio	Fecha de fin
Análisis de Requisitos	03-03-14	21-03-14
Analizar Objetivos generales y específicos	03-03-14	04-03-14
Analizar Requisitos Básicos	04-03-14	05-03-14
Definir metodología de trabajo	06-03-14	06-03-14
Analizar juego de Arimaa	07-03-14	21-03-14
Diseño del Sistema	24-03-14	26-03-14
Identificar actores del Sistema	24-03-14	24-03-14
Elaboración Primer Caso de Uso	25-03-14	25-03-14
Elaboración de especificación de C.U. específicos	25-03-14	26-03-14
Diseño del Programa	01-04-14	24-04-14
Analizar algoritmos básicos de búsqueda	01-04-14	10-04-14
Analizar metaheurística OCH	11-04-14	18-04-14
Revisión Casos de Uso	21-04-14	21-04-14
Revisión de especificaciones de casos de uso	22-04-14	22-04-14
Evaluación del Avance de Proyecto	23-04-14	24-04-14
Entrega Avance Proyecto 1	30-04-14	30-04-14
Codificación	05-05-14	13-06-14
Implementar tablero Arimaa	05-05-14	05-05-14
Implementar piezas Arimaa	05-05-14	06-05-14
Implementar movimientos de piezas	05-05-14	06-06-14
Implementar reglas del juego	07-05-14	20-05-14
Implementar algoritmo básico de IA	21-05-14	10-06-14
Realizar Pruebas	11-06-14	13-06-14
Elección variante de OCH	13-06-14	13-06-14
Entrega Final Proyecto 1	20-06-14	20-06-14
Implementar algoritmo OCH en el juego	11-08-14	19-09-14
Entrega Avance Proyecto 2	26-09-14	26-09-14
Corregir posibles errores del juego	29-09-14	17-10-14
Pruebas	20-10-14	31-10-14
Pruebas de Caja Blanca	20-10-14	24-10-14
Pruebas de Caja Negra	27-10-14	31-10-14
Verificación	03-11-14	07-11-14
Entrega Software Final	07-11-14	07-11-14
Entrega Final	21-11-14	21-11-14
Mantenimiento	24-11-14	12-12-14

Figura 2.1: Carta Gantt

3 Marco teórico

3.1 Juego Arimaa

Como se sabe, el Arimaa fue creado gracias a otro juego de mesa llamado Ajedrez, en el cual se han implementado diferentes formas para que un computador pueda jugarlo. A continuación se dará a conocer cómo ha ido evolucionando el Ajedrez en la computación;el nacimiento de un nuevo juego llamado Arimaa, indicando cuáles son sus reglas, y la dificultad que conlleva implementar un juego como este en un computador.

3.1.1 Ajedrez en la Computación

A pesar de lo que muchos pueden pensar, el Ajedrez no ha aportado mucho en lo que se refiere a la inteligencia artificial, y es que, los métodos que utiliza se basan, principalmente, en explorar un número elevado de posibles futuros movimientos y aplicarles una función de evaluación al resultado.

Antes de la aparición de la computadora en la década de los 50, ya empezaron a aparecer los primeros autómatas capaces de jugar al Ajedrez. El primero, fue el autómata "El Turco", el cual fue muy conocido, sin embargo, después de un tiempo, se descubrió que este solo era una farsa. Luego, en el año 1912, el español Torres y Quevedo construyó un autómata llamado "El Ajedrecista".

En el año 1950, Claude Shannon publica el primer artículo que describía la programación de una computadora para jugar al Ajedrez. En este, hace notar la existencia las dos principales formas de búsqueda, "Tipo A" y "Tipo B". La primera consiste en una búsqueda basada en la "fuerza bruta", en donde se examinan todas las posibles posiciones. La segunda forma radica en utilizar una especie de "inteligencia artificial estratégica", en donde solo se analizarían las mejores jugadas de cada posición.

Si bien, a simple vista el "Tipo B" pareciera ser mejor que el "Tipo A", este último fue el que más se utilizó hasta finales de los años 90. Y es que, para un computador, es sumamente complicado ver qué posiciones son mejores que otras, sobretodo en un nivel profesional. Un jugador profesional, el cual puede examinar muchas posibles jugadas según la posición de las piezas, va a elegir el mejor movimiento según la experiencia obtenida en partidas anteriores, por lo que, para una computadora, no es tan sencillo simular esto.

En el año 1973, la Universidad de Northwestern crea el programa de Ajedrez "Chess 4.0", el cual utilizaba algoritmos de "Tipo A". Durante 5 años seguidos, este programa resultó ganador del torneo ACM Computer Chess Championships. Además, la influencia que tuvo Chess 4.0 es enorme, pues hasta los programas de ajedrez actuales siguen el paradigma que utilizaba este.

En 1975, empieza el desarrollo de Cray Blitz, que entre 1983-1989 fue el más rápido de los programas y fue el campeón de los programas de ajedrez. En 1983, Cray Blitz era capaz de buscar entre 40.000 y 50.000 posiciones por segundo.

En 1977, Belle fue el primer el primer sistema computacional que utilizó chips personalizados para incrementar su fuerza de juego, aumentando la velocidad de 200 posiciones por segundo a 160.000 (8 movidas de profundidad).

En 1988, fue creada la máquina Deep Thought por un grupo de graduados de la Universidad Carnegie-Mellon. Esta contenía 250 chips, 2 procesadores, y era capaz de analizar 750.000 posiciones por segundo (10 movidas de profundidad). Además, en este mismo año, Deep Thought se convierte en la primera máquina en derrotar a un Gran Maestro en un torneo.

En 1997, la máquina Deep Blue de IBM, la cual contaba con 30 procesadores y 480 chips, pudiendo evaluar 200 millones de movimientos por segundo, logra vencer a Gary Kasparov en un match a 6 partidas. [1][2]

3.1.2 Nacimiento de Arimaa

Luego de que el computador de Ajedrez Deep Blue derrotara a Garry Kasparov, el ingeniero en computación Omar Syed empezó a diseñar un nuevo juego, el cual fuera difícil que un computador lo jugara bien, pero con reglas lo suficientemente básicas para que un niño pueda entenderlas. A este juego, lo llamó “Arimaa” debido a que su hijo tenía como nombre “Aamir” (lo que al revés es “Arimaa” sumándole una ‘a’ al principio).

En el año 2002 Omar Syed publica las reglas de Arimaa, ofreciendo un premio de al menos 10.000 US$ hasta el año 2020 para la primera persona, compañía u organización que pueda vencer a tres jugadores humanos seleccionados en un match oficial de Arimaa. La idea es que el programa pueda ser corrido en un computador común, sin que necesite ocupar hardware especializado adicional.

Desde el año 2004 hasta el año 2008, el programa “Bomb” de David Fotland gana el campeonato mundial de Arimaa, con lo que se gana el derecho de pelear por el premio de los 10.000 US$. Sin embargo, no tuvo éxito en ninguna oportunidad. La misma suerte la corrieron los demás participantes de los otros años. Sin embargo, a pesar de esto último, hubo programas que sí le ganaron al menos a un jugador humano seleccionado; el bot Marwin el año 2010 y el bot Briareus el año 2012. [3]

3.1.3 Dificultad de implementar un buen programa de Arimaa

La implementación de los juegos con combinatoria en los computadores ha sido un tema a tratar durante muchos años. Y es que, como se mencionó anteriormente con el Ajedrez y el Arimaa, el nivel de complejidad que conllevan es muy grande. Pero, ¿cómo se puede medir la dificultad en la implementación de un cierto juego?

Existen varias maneras para medir la complejidad en los juegos, entre las cuales se encuentran:

- Complejidad Estado-Espacio: Número de posiciones legales dentro del juego accesibles desde la posición inicial del juego.
- Tamaño del Árbol del juego: Número total de posibles movimientos del juego. Esto quiere decir que cuando se forma el árbol de juego, el primer nodo sería la posición inicial.
- Árbol de Decisión: Es un subárbol del árbol de juego, en donde cada nodo adopta un valor como "jugador A gana", "jugador B gana" o "empate", según el estado de los nodos sucesores.
- Complejidad de Decisión: Número de nodos terminales en el árbol de decisión más pequeño que establece el valor de la posición inicial.
- Complejidad del juego-árbol: Número total de nodos terminales que hay en el árbol. Minimax es un ejemplo de algoritmo en donde se evalúan todos los nodos terminales.
- Coste Computacional: Consiste en el cálculo de la complejidad temporal a priori, en función del tamaño del problema. Esto permite establecer una cota superior para el tiempo de ejecución del algoritmo.

En la ***tabla 3.1***[1] se puede apreciar una comparación entre la dificultad que existe entre el ajedrez y el Arimaa. Como se mencionó anteriormente, el Ajedrez y el Arimaa son juegos bastante parecido, sin embargo, y como se puede ver en la tabla, la complejidad del árbol de juego en el Arimaa es mucho mayor que en el Ajedrez. Esto es debido básicamente a 2 razones: la gran cantidad de combinaciones de posiciones iniciales que pueden ocurrir y la gran cantidad de movimientos que pueden ocurrir en un turno. [4][5]

[1] Esta tabla es un resumen de [5]

Tabla 3.1: Complejidad de Ajedrez y Arimaa

Juego	Tamaño del tablero	Complejidad estado-espacio	Complejidad del árbol de juego	Longitud media del juego
Ajedrez	64	50	123	80
Arimaa	64	43	296	70

3.1.4 Reglas de Arimaa

Arimaa es un juego para dos personas, el cual utiliza un tablero similar al del ajedrez. Este tablero contiene 64 casillas (8x8), en donde habrán 32 piezas (16 para cada jugador) y 4 casillas de trampa. Dentro de las piezas con las que un jugador puede jugar (valga la redundancia), se pueden encontrar: un elefante, un camello, dos caballos, dos perros, dos gatos y ocho conejos; los cuales representarán el bando dorado o plateado.

Al inicio del juego, no habrá piezas en el tablero. El primero en jugar será el jugador que representa el bando dorado, el cual tendrá que ordenar sus piezas a su gusto, dentro de las primeras dos filas. Luego, será el turno del jugador que representa el bando plateado, el cual ordenará sus piezas a su gusto en las últimas dos filas. A continuación se presenta una imagen del tablero inicial vacío de Arimaa.

Figura 3.1: Tablero vacío Arimaa

Luego de que cada jugador haya posicionado sus piezas (por ejemplo, en la ***figura 3.2***), se pasará al movimiento de las piezas para poder llegar al objetivo principal del juego, el cual es llevar un conejo hasta el otro lado del tablero. En cada turno, un jugador podrá realizar entre 1 y 4 movimientos, el cual debe ser un cambio neto a la posición de las piezas del tablero; esto quiere decir, por ejemplo, que en un turno no se puede mover una pieza hacia delante y luego hacia atrás, pues quedaría en el mismo espacio, lo que sería lo mismo que realizar 0 movimientos en el turno. Cada pieza puede moverse un 1 celda hacia delante, hacia la derecha o hacia la izquierda, a excepción del conejo, el cual solo se puede mover hacia delante, derecha o izquierda.

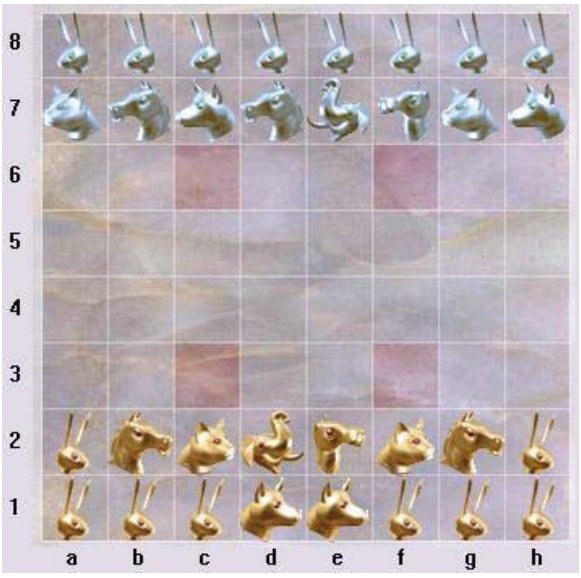


Figura 3.2: Posible posición de piezas iniciales en Arimaa

Además de los movimientos particulares de cada pieza, también existen movimientos especiales. Uno de estos es el movimiento empujar, el cual trata de que una pieza puede mover a una pieza enemiga adyacente más débil a una celda contigua vacía, y la pieza más fuerte se posiciona en el lugar donde estaba la pieza más débil. El otro movimiento especial, es el de tirar, el cual consiste en posicionar a la pieza más fuerte en cualquier cuadro adyacente vacío, y tirar la pieza más débil a la posición donde estaba la pieza más fuerte. Cada uno de estas jugadas, implica gastar 2 de los 4 movimientos disponibles por turno.

En la ***figura 3.3*** se puede apreciar que el caballo dorado en d4 puede mover al perro plateado en c4 a la trampa que está en c3. Este mismo caballo puede realizar el movimiento de tirar posicionándose en la celda d5 y moviendo el perro plateado de c4 a d4.

Otra de los beneficios que tienen las piezas más fuertes sobre débiles, es que tienen la capacidad de congelarlas. Esto quiere decir, que si hay una pieza de mayor valor que otra

enemiga, y esta última está adyacente a la primera y además no hay ninguna pieza del mismo color que esté contigua, entonces, la pieza se congela, no pudiendo moverse (solo puede moverse cuando el oponente decida realizar algunas de los movimientos especiales).

En la ***figura 3.3*** se puede ver que el camello que se encuentra en la celda e6 está congelado por el elefante que está en d6. En cambio, el perro que se encuentra en c4 no está congelado, pues hay una pieza de su mismo adyacente a esta.

En el tablero existen 4 casillas que en el juego son consideradas como trampa. Cada vez que una pieza entra en una casilla de trampa, esta es removida del juego, a menos que, adyacente a ella, haya otra pieza del mismo bando. Para que una pieza entre a una casilla de trampa, esta puede ser movida voluntariamente por el jugador de la pieza o por el enemigo con algún movimiento de empujar o tirar. Las casillas de trampa corresponden a las casillas C3, C6, F3 y F6 (en la imagen se representan de color rojo).

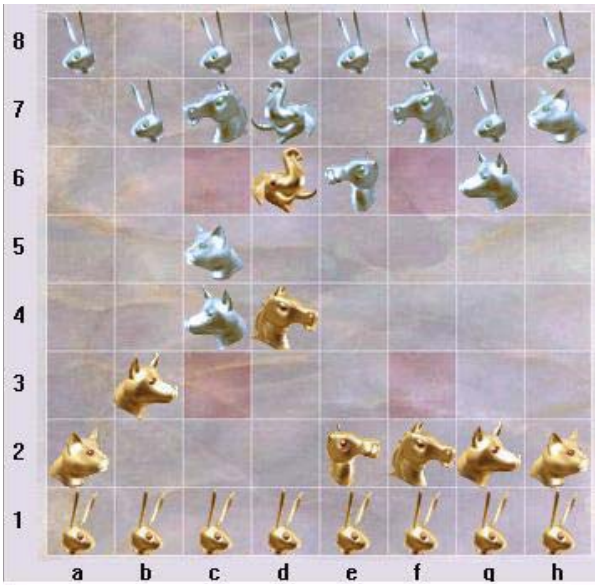


Figura 3.3: Posición de medio juego

Como se mencionó anteriormente, para que un jugador pueda obtener la victoria, debe llevar una pieza de un conejo al otro lado del tablero. Sin embargo, esta no es la única manera de poder salir victorioso o de terminar la partida. A continuación se presentan algunos escenarios posibles que pueden suceder para que se acabe el juego.

- Un conejo algún bando llega al otro lado del tablero, lo que origina la victoria del jugador de aquel conejo.
- Al principio de un turno, un jugador no se queda con ningún movimiento posible por realizar, provocándole la derrota inmediata.

- Si una posición de piezas se repite 3 veces, el jugador quien causó esto, perderá.
- Si un jugador se queda sin conejos, el jugador pierde el juego.

Cabe destacar que cada una de las condiciones anteriores se debe considerar luego de finalizar cada turno y no en medio de un turno. Esto quiere decir, por ejemplo, que si el conejo de un jugador es movido a la línea objetivo, y es reposicionado nuevamente en el mismo turno fuera de esta, el juego continúa. [3]

3.2 Algoritmos de Búsqueda

3.2.1 Árbol de Búsqueda

Antes de poder ejecutar algún algoritmo de búsqueda, es necesario crear un árbol con los posibles movimientos que se pueden ir dando dentro del juego. La idea es que en cada nivel del árbol estén los nodos que representan los movimientos que puede dar un jugador dentro del siguiente turno. Los hijos de estos nodos serán las jugadas del otro jugador.

En juegos como el Tic-Tac-Toe o Sudoku, es posible crear un árbol con todos los posibles movimientos que se pueden dar dentro del juego. Sin embargo, en juegos como el Ajedrez o el Arimaa, esto se hace imposible debido al gran número de combinaciones que se pueden dar, además del escaso tiempo que dura un turno.

En la ***figura 3.4*** se presenta un ejemplo de un Árbol de búsqueda con los nodos y sus respectivos valores.

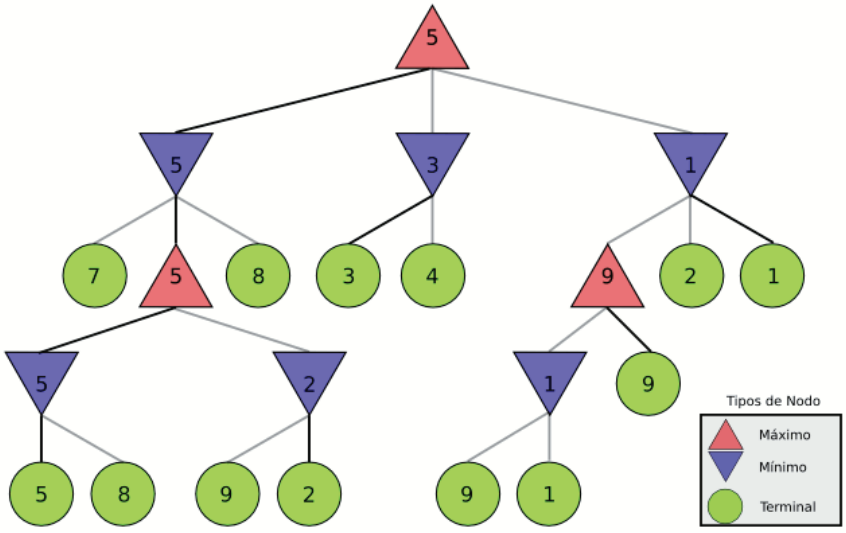


Figura 3.4: Ejemplo de Árbol de Búsqueda

3.2.2 Minimax

Minimax es un método de decisión para minimizar la pérdida máxima esperada en juegos con adversario y con información perfecta. Básicamente se trata de elegir el mejor movimiento sabiendo que el contrincante elegirá el menos conveniente para uno.

En este algoritmo, cada vez que se esté en un nivel en el cual se representan los movimientos del jugador que le toca mover, se elegirán los movimientos con mayor valor. En cambio, cuando se esté en un nivel que representa los movimientos del jugador contrario, se elegirán los movimientos con menor valor. Teniendo en consideración estos puntos, se podrá obtener la mejor jugada.

En resumen, para la implementación del algoritmo Minimax, se deben seguir los siguientes pasos:

- Generación del árbol de juego. Se generarán todos los nodos hasta llegar a un estado terminal.
- Cálculo de los valores de la función de utilidad para cada nodo terminal.
- Calcular el valor de los nodos superiores a partir del valor de los inferiores. Alternativamente se elegirán los valores mínimos y máximos representando los movimientos del jugador y del contrincante (de ahí el nombre Minimax).
- Elegir la jugada dependiendo de los valores que han llegado al nivel superior [6]

En la ***figura 3.5*** se puede observar el algoritmo Minimax en pseudocódigo.

Algorithm 1 $MM(n, d)$

```
1: S ← Successors(n)
2: if d ≤ 0 ∨ S ≡ ∅ then
3:    return f(n)
4: best ← −∞
5: for all n_i ∈ S do
6:    v ← −MM(n_i, d − 1)
7:    if v > best then
8:       best ← v
9: return best
```

Figura 3.5: Algoritmo Minimax

3.2.3 Poda Alfa-Beta

Debido a que el algoritmo Minimax requiere la evaluación de todos los nodos del árbol, en juegos como el ajedrez, en donde hay una gran cantidad de posibles movimientos, esto se hace poco práctico, pues se tardaría demasiado en encontrar el mejor movimiento. Es por esto, que se ha ideado una manera, la cual, si bien utiliza el mismo árbol que Minimax, solo va a analizar los mejores movimientos, ignorando los peores. Este algoritmo es conocido como "Poda Alfa-Beta".

La idea del algoritmo es utilizar dos variables auxiliares, que serán alfa y beta (de ahí el nombre), las cuales irán almacenando el valor máximo y mínimo respectivamente de los nodos que vaya analizando. A continuación se explica cómo el algoritmo va buscando la mejor opción realizando una poda a ciertos nodos del árbol. Para esto se utilizará el ejemplo de la ***figura 3.4***.

- Se evalúa el primer nodo del nivel terminal, y se anota de forma provisoria en el nodo padre. En el ejemplo el valor sería 5.
- Se vuelve al nivel terminal para evaluar al siguiente nodo de la misma rama. Este valor es 8.
- Se compara el valor provisorio del nodo padre con el último nodo evaluado. Como el nodo padre del nodo que se está evaluando es Minimizador, se conserva el menor de los valores, por lo que se mantiene el número 5 en el nodo padre (5<8).
- Este valor pasa como valor provisorio del nodo padre, y se compara con el valor provisorio del nodo hijo de la rama vecina. Teniendo en consideración que el nivel del nodo padre es un Nivel Maximizador y el nivel del nodo hijo es un nivel Minimizador, pueden ocurrir dos cosas:
    - Que el valor provisorio del nodo padre sea mayor que el valor provisorio del nodo hijo. Como el nivel del nodo hijo es un nivel minimizador, no es posible cambiar este valor por uno más alto. Por consiguiente, no es necesario seguir evaluando los otros nodos y se conserva el valor del nodo padre del nivel maximizador.
    - Que el valor provisorio del nodo padre sea menor que el valor provisorio del nodo hijo. Esto hace que se compare el valor provisorio del nodo hijo con los valores de sus propios nodos hijos que no han sido evaluados.
- En el ejemplo, el valor provisorio del nivel minimizador era 9, ya debido a que 9>5, se tuvo que seguir analizando los otros nodos hijos. Como se puede observar en la figura, el valor que quedó en el nodo del nivel minimizador es el valor, 2, pero como 2 es menor que 5, este último se mantuvo como valor del nodo padre. [6]

En la ***figura 3.6*** se puede observar el algoritmo Poda Alfa-Beta en pseudocódigo.

Algorithm 2 $\alpha\beta(n, d, \alpha, \beta)$

```
1: S ← Successors(n)
2: if d ≤ 0 ∨ S ≡ ∅ then
3:    return f(n)
4: best ← −∞
5: for all n_i ∈ S do
6:    v ← −αβ(n_i, d − 1, −β, −max(α, best))
7:    if v > best then
8:       best ← v
9:       if best ≥ β then
10:         return best
11: return best
```

Figura 3.6: Algoritmo Poda Alfa-Beta

3.2.4 Optimización por Colonia de Hormigas

Si bien este algoritmo no ha sido utilizado en juegos como el Ajedrez o Arimaa, ha podido solucionar una gran variedad de problemas de combinatoria. A continuación se explicará en qué consiste esta técnica, con el fin de que a futuro se pueda implementar en el juego de Arimaa.

3.2.4.1 Colonia de Hormigas Naturales

Uno de los comportamientos más llamativos de las hormigas es su habilidad para encontrar los caminos más cortos entre su hormiguero y la fuente de alimentos. Y es que muchas de las especies de hormigas son casi ciegas, lo que evita el uso de pistas visuales.

Las hormigas mientras cruzan el camino para llegar a la fuente de alimentos, van dejando un rastro de feromona. Esta feromona a medida que pasa el tiempo se va evaporando, por lo que si una hormiga no pasa por cierto camino después de cierto tiempo, la feromona puede desaparecer. Al principio, lógicamente no va haber ningún rastro de feromona, por lo cual las hormigas eligen un camino de forma aleatoria. Cuando ya existe feromona, las hormigas van a elegir un camino con una decisión probabilística sesgada por la cantidad de feromona; cuanto más fuerte es el rastro de feromona, mayor es la probabilidad de elegirlo.

En la ***figura 3.7*** se puede apreciar los pasos que ocurren desde que las hormigas están en el hormiguero hasta que llegan a la fuente de alimentos. En la primera imagen se ve a las hormigas en el hormiguero. En la segunda imagen las hormigas toman un camino al azar para llegar a la fuente de alimentos. En la tercera imagen se puede ver que empieza a haber una preferencia por el primer camino, debido a que en este hay una mayor concentración de feromona. En la última imagen se ve que cada vez son más las hormigas que van tomando el primer camino y menos el segundo.

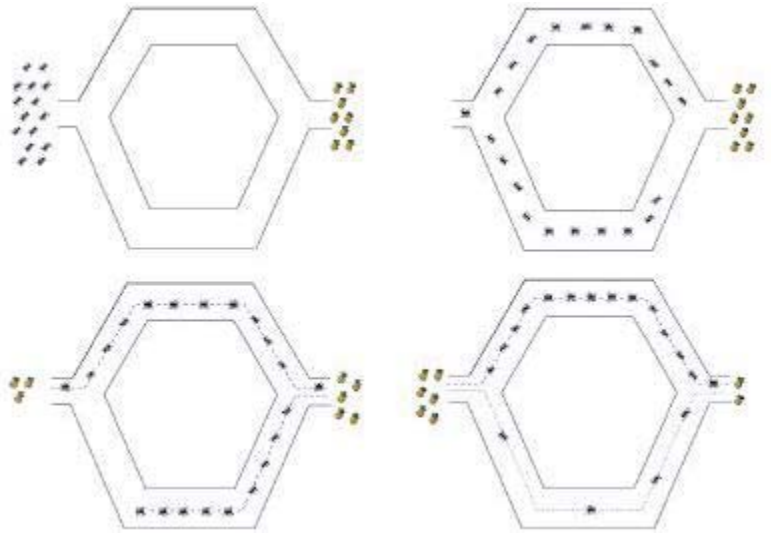


Figura 3.7: Comportamiento de la colonia de hormigas

3.2.4.2 Metaheurística de Optimización por Colonia de Hormigas

El algoritmo de Optimización por Colonia de Hormigas es una técnica meta heurística que busca los mejores caminos dentro de un grafo. Toma el nombre debido a que se basa en el comportamiento de las colonias de hormigas.

Los algoritmos de OCH se caracterizan por ser algoritmos constructivos; es decir, que en cada iteración del algoritmo, cada hormiga construye una solución al problema recorriendo un grafo de construcción.

Cada arista a_{ij} del grafo, la cual representa los posibles pasos que la hormiga puede dar, contiene dos tipos de información:

- Información Heurística: Mide la preferencia heurística de la arista. Las hormigas no modifican esta información durante la ejecución del programa.
- Información de los rastros de feromona artificiales: Mide la deseabilidad de la arista, representada por la cantidad de feromona depositada en él. Esta información es modificada durante la ejecución del algoritmo dependiendo de las soluciones encontradas por las hormigas.

El modo de operación básico de un algoritmo de OCH consiste en que existen m hormigas (artificiales) que se van moviendo de manera concurrente a través del grafo. Este movimiento se realiza siguiendo una regla de transición que está basada en la información local disponible en los nodos. Esta información local incluye la información heurística y memorística para guiar la búsqueda.

3.2.4.3 Modelos de Optimización Basada en Colonias de Hormigas

En la actualidad existen algunas variantes que siguen la Metaheurística OCH. A continuación se explicará en qué consiste cada una de estas variantes.

3.2.4.3.1 Sistema de Hormigas

En el SH la actualización de feromona se realiza cuando todas las hormigas han terminado su proceso. Esto se realiza siguiendo los siguientes pasos:

1. Todos los rastros de feromona se reducen en un factor constante, implementándose de esta manera la evaporación de feromona.
2. Cada hormiga de la colonia deposita una cantidad de feromona que es función de la calidad de su solución.

Para encontrar las soluciones en el SH, las hormigas en cada paso de construcción escogen ir al siguiente nodo con una probabilidad que se calcula como:

$$p_{rs}^{k} = \begin{cases} \frac{(\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}{\sum_{\mu \in N_i^n} (\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}, & si\ s \in N_k(r) \\ 0, & en\ otro\ caso \end{cases} \quad \textbf{(3.2.4.3.1.1)}$$

Donde $N_k(r)$ es el vecindario alcanzable por la hormiga k cuando se encuentra en el nodo r. α, β ∈ ℜ son dos parámetros que ponderan la importancia relativa de los rastros de feromona y la información heurística.

En el Sistema de Hormigas la deposición de feromona se realiza una vez que todas las hormigas han terminado sus soluciones. En primer lugar, los rastros de feromona asociados a cada arco se evaporan reduciendo todos los rastros de feromona en un factor constante:

$$\tau_{rs} \leftarrow (1 - \rho) \cdot \tau_{rs} \quad \textbf{(3.2.4.3.1.2)}$$

Donde ρϵ (0,1] es la tasa de evaporación. El siguiente paso de cada hormiga es recorrer de nuevo el camino que ha seguido (el camino está almacenado en su memoria local) y deposita una cantidad de feromona $\Delta\tau_{rs}^{k}$ en cada conexión por la que ha viajado:

$$\tau_{rs} \leftarrow \tau_{rs} + \Delta\tau_{rs}^{k}, \forall a_{rs} \in S_k \quad \textbf{(3.2.4.3.1.3)}$$

Donde $\Delta\tau_{rs}^{k} = f\big(C(S_k)\big)$, es decir la cantidad de feromona que se deposita depende de la calidad $C(S_k)$ de la solución S_k construida por la hormiga k.

3.2.4.3.2 Sistema de Colonias de Hormigas

En el SCH se realizan algunas modificaciones con respecto al Sistema de Hormigas. Entre los cambios que se realizan, se pueden considerar:

- Se utiliza una regla de transición distinta, denominada regla proporcional pseudo-aleatoria. Sea k una hormiga situada en el nodo r, q_0 ∈ [0,1] un parámetro y q un valor aleatorio en [0,1], el siguiente nodo s se elige de forma aleatoria mediante la siguiente distribución de probabilidad:

  Si $q \leq q_0$:

$$p_{rs}^{k} \begin{cases} 1, & si\ s = arg\ \max_{u \in N_k(r)} \left\{ \tau_{ru} \cdot \eta_{ru}^{\beta} \right\} \\ 0, & en\ otro\ caso \end{cases} \quad \textbf{(3.2.4.3.2.1)}$$

Si no:

$$p_{rs}^{k} = \begin{cases} \frac{(\tau_{rs})^{\alpha} \cdot (\eta_{rs})^{\beta}}{\sum_{u \in N_r^k} (\tau_{rs})^{\alpha} \cdot (\eta_{rs})^{\beta}}, & si\ s \in N_k(r) \\ 0, & en\ otro\ caso \end{cases} \quad \textbf{(3.2.4.3.2.2)}$$

En este caso se puede observar que cuando $q \leq q_0$se elige la mejor opción tomando en consideración la información heurística y los rastros de feromona. En cambio, cuando $q > q_0$ se aplica una exploración controlada, tal como se hacía en el SH.

- Para la actualización de feromona el SCH sólo considera a la hormiga que generó la mejor solución global, $S_{mejor-global}$. Esta actualización la realiza el demonio y no las hormigas individualmente.
  La actualización de feromona se realiza evaporando en primer lugar los rastros de feromona en todas las conexiones utilizadas por la mejor hormiga global:

$$\tau_{rs} \leftarrow (1-\rho) \cdot \tau_{rs}, \forall a_{rs} \epsilon\, S_{mejor-global} \quad \textbf{(3.2.4.3.2.3)}$$

A continuación, el demonio deposita feromona usando la regla:

$$\tau_{rs} \leftarrow \tau_{rs} + \rho \cdot f\left(C\left(S_{mejor-global}\right)\right), \forall a_{rs} \epsilon\, S_{mejor-global} \quad \textbf{(3.2.4.3.2.4)}$$

Además, el demonio puede aplicar un algoritmo de búsqueda local para mejorar las soluciones de las hormigas antes de actualizar los rastros de feromona.

- Se realiza una actualización local de feromona basada en que cada hormiga va registrando los rastros de feromona por cada arco que pasa. Cada vez que una hormiga pasa por la arista a_{rs}, aplica la regla:

$$\tau_{rs} \leftarrow (1-\varphi) \cdot \tau_{rs} + \varphi \cdot \tau_0 \quad \textbf{(3.2.4.3.2.5)}$$

Donde $\varphi \epsilon$ (0,1] es un segundo parámetro de decremento de feromona. En este caso, la actualización de manera local incluye tanto la evaporación de la feromona, como la deposición de la misma.

3.2.4.3.3 Sistema de Hormigas Max-Min

En el Sistema de Hormigas Max-Min la regla transición se mantiene al ocupado en el SCH. Luego de que las hormigas hayan construido su solución cada rastro de feromona sufre una evaporación:

$$\tau_{rs} = (1-\rho) \cdot \tau_{rs} \quad \textbf{(3.2.4.3.3.1)}$$

A continuación la feromona se deposita siguiendo la siguiente fórmula:

$$\tau_{rs} = \tau_{rs} + f\left(C\left(S_{mejor}\right)\right), \forall a_{rs} \in S_{mejor} \quad \textbf{(3.2.4.3.3.2)}$$

En este caso, la mejor hormiga que actualiza la feromona, puede ser la que tiene una solución mejor de la iteración o de la solución mejor global. Para el cálculo de los límites mínimo y máximo es necesario el uso de heurísticas.

En el SHMM en vez de inicializar los rastros de feromona en un valor pequeño τ_0, estos se inicializan en el valor máximo τ_{max}. Esto hace que al aplicar la regla de actualización, los arcos de las buenas soluciones mantengan valores altos, mientras que los de las malas soluciones reduzcan el valor de sus rastros.

3.2.4.3.4 Sistema de la Mejor-Peor Hormiga

En el Sistema de la Mejor-Peor Hormiga se mantienen ciertas reglas de otras variantes, dentro de las cuales se encuentran:

- Se utiliza la misma regla de transición de estados que en el Sistema de Hormigas.
- Se utiliza la misma regla de evaporación de feromona que en el Sistema de Hormigas Max-Min, en donde se aplica a todas las transiciones.
- Al igual que en el Sistema de Hormigas Max-Min, en el SMPH siempre se considera la explotación sistemática de optimizadores locales para mejorar las soluciones de las hormigas.

Entre las acciones del demonio en el Sistema de la Mejor-Peor Hormiga, se encuentran:

- La regla mejor-peor de actualización de rastros de feromona, en la cual se refuerza las aristas que se encuentran en la mejor solución global. Además, a la peor solución generada hasta el momento $S_{peor-actual}$, que no se encuentre en la mejor global, se realiza una evaporación adicional. La fórmula de actualización de feromona quedaría como:

$$\tau_{rs} \leftarrow \tau_{rs} + \rho \cdot f\left(C\left(S_{mejor-global}\right)\right), \forall a_{rs} \in S_{mejor-global} \quad \textbf{(3.2.4.3.4.1)}$$

$$\tau_{rs} \leftarrow (1-\rho) \cdot \tau_{rs}, \forall a_{rs} \in S_{peor-actual}\ y\ a_{rs} \notin S_{mejor-global} \quad \textbf{(3.2.4.3.4.2)}$$

- Se lleva a cabo una mutación de los rastros de feromona para introducir diversidad en el proceso de búsqueda. Esto se lleva a cabo mutando el rastro de feromona

asociado a cada una de las transiciones desde cada nodo con una probabilidad P_m utilizando cualquier operador de mutación con codificación real.

$$\tau'_{rs} \leftarrow \begin{cases} \tau_{rs} + mut(it, \tau_{umbral}), & si\ a = 0 \\ \tau_{rs} - mut(it, \tau_{umbral}), & si\ a = 1 \end{cases} \quad \textbf{(3.2.4.3.4.3)}$$

En este caso $mut(it, \tau_{umbral})$ corresponde al rango de mutación, el cual depende de la media de los rastros de feromona en las transiciones de la mejor solución global. La variable *a* corresponde a un valor aleatorio en {0,1}, *it* es la iteración actual y τ_{umbral} es la mejor solución global.

- Cuando la búsqueda se estanca se aplica una re inicialización de los rastros de feromona, poniendo a todos los rastros de feromona en τ_0. [7]

4 Análisis y Diseño de la Solución

4.1 Especificación de Requerimientos

4.1.1 Requerimientos Funcionales

- Permitir mover las piezas de un cuadro a otro adyacente.
- Restringir el movimiento hacia atrás para la pieza conejo.
- Informar al usuario qué movimientos puede hacer con cierta pieza seleccionada por él.
- Informar al usuario cuando una pieza ha caído en una celda de trampa.
- Informar al usuario cuando una pieza ha sufrido uno de los movimientos especiales de tirar o empujar.
- Informar al usuario cuando una pieza esté en estado “congelado”.
- Informar al usuario cuando se acabe el juego.

4.1.2 Requerimientos No Funcionales

- Proporcionar una interfaz amigable para el usuario.
- Que el pc pueda ejecutar archivos java.
- Tiempo de respuesta computacionalmente factible.

4.2 Diseño de la Solución

4.2.1 Casos de Uso

El modelado de Casos de Uso es la técnica más efectiva y a la vez la más simple para modelar los requisitos del sistema desde la perspectiva del usuario. Los Casos de Uso se utilizan para modelar cómo un sistema o negocio funciona actualmente, o cómo los usuarios desean que funcione. No es realmente una aproximación a la orientación a objetos; es realmente una forma de modelar procesos. Es, sin embargo, una manera muy buena de dirigirse hacia el análisis de sistemas orientado a objetos. Los casos de uso son generalmente el punto de partida del análisis orientado a objetos con UML. [9]

4.2.2 Caso de Uso General

En este diagrama se muestra de manera general cómo interactúan los actores con el sistema. En este caso hay un solo tipo de actor que es el Usuario, el cual se encarga de mover las piezas por el tablero. Este usuario puede jugar tanto con otro usuario o con la computadora.

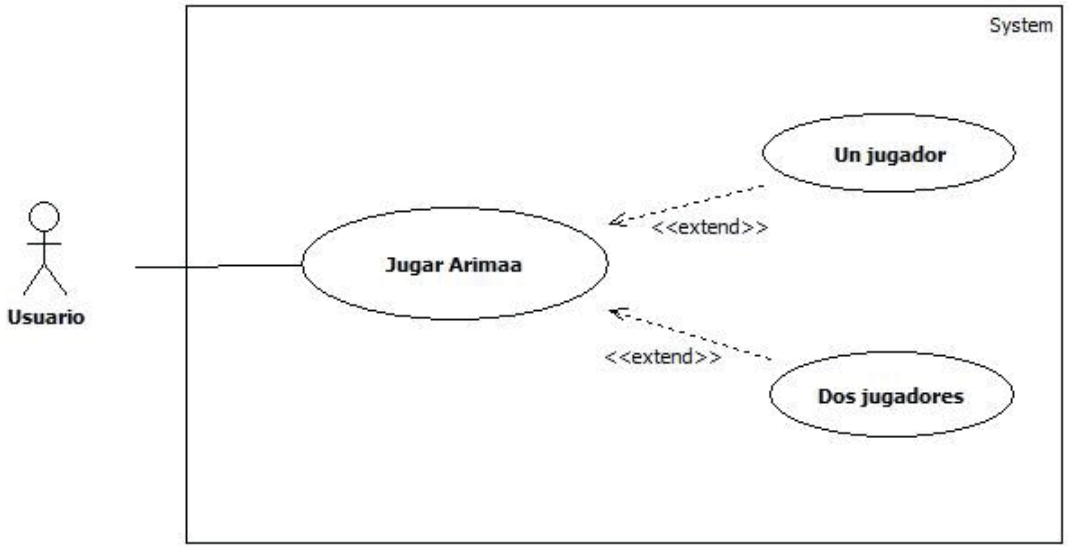


Figura 4.1: Caso de Uso General

4.2.3 Especificaciones de Casos de Uso

4.2.3.1 Especificaciones de Casos de Uso

En este diagrama se muestra de manera detallada cómo el usuario interactúa con el Sistema. Se utilizan 3 casos de uso, los cuales son “Posicionar piezas iniciales”, en el cual el usuario puede elegir los lugares donde quiere posicionar sus piezas; “Mover pieza”, en el que el usuario puede arrastrar una pieza desde una celda a otra; y “Terminar turno”, en el cual el usuario puede dar por término a su turno, no pudiendo realizar ningún tipo de movimiento hasta que vuelva a ser su turno. Las especificaciones formales de este diagrama se encuentran en Anexo A, sección Especificación Formal.

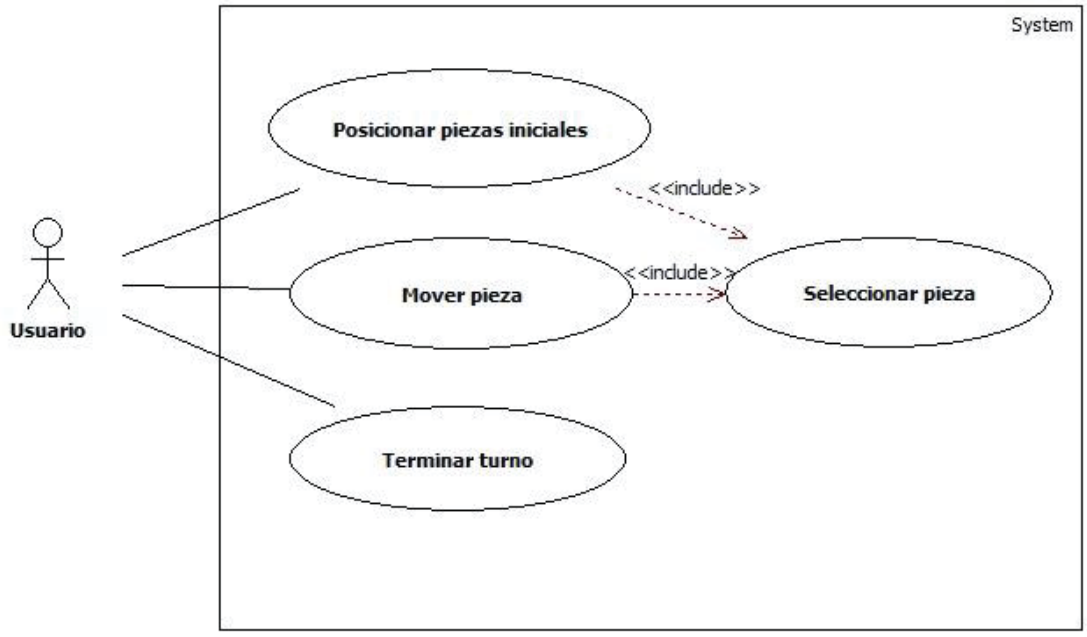


Figura 4.2: Caso de Uso “Jugar Arimaa”

4.2.4 Diagrama de Clases

El programa se divide en distintas clases que contienen la información necesaria para el buen funcionamiento del juego. En la ***figura 4.3*** se puede apreciar el Diagrama de Clases del juego Arimaa.

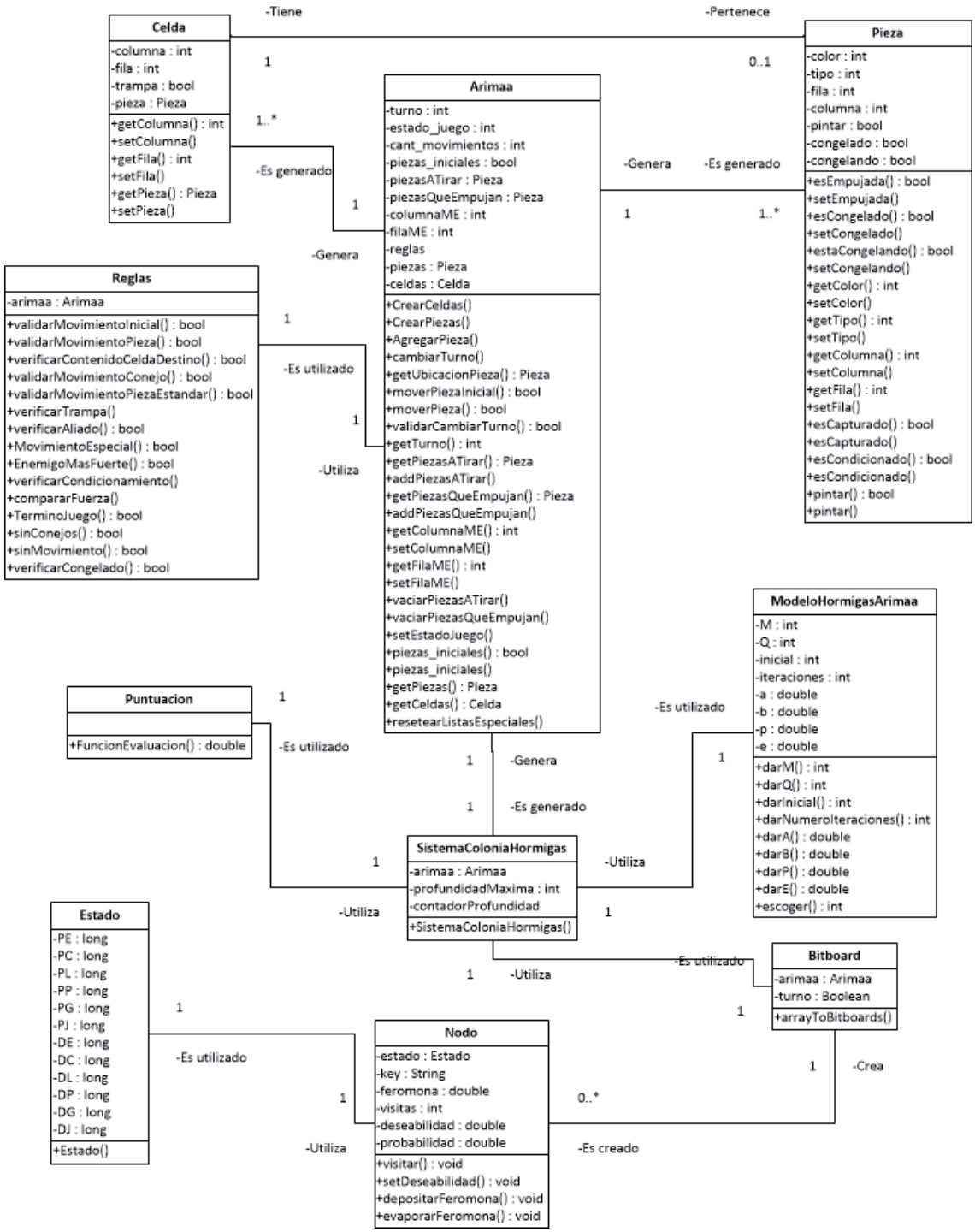


Figura 4.3: Diagrama de Clases

5 Representación de la Solución

Para poder representar la información se utilizan distintas clases. La clase más general es la clase Arimaa, la cual va a contener objetos de las otras clases más específicas del juego. A continuación se explica cómo se representan los distintos factores que hay dentro del juego.

5.1 Tablero

Para poder representar el tablero de Arimaa se ha utilizado un atributo en la clase Arimaa, la cual corresponde a una matriz de objetos de la clase Celda. En la ***tabla 5.1*** que se encuentra en el Anexo B, Sección Tablero, se puede ver la representación que se hace del tablero.

5.2 Piezas

Las piezas son representadas a través de una clase llamada Pieza, la cual va a contener diferentes atributos y métodos que permitirán interactuar de buena manera con los métodos de otras clases.

Como es sabido, en Arimaa existen dos tipos de jugadores (Dorado y Plateado), y 6 tipos de piezas, por lo cual se hace necesario poder diferenciar cada una de estas, ya sea por el tipo como por el color. Para esto se han usado dos variables. En la ***tabla 5.2*** y en la ***tabla 5.3*** que se encuentran en el Anexo B, Sección Piezas, se puede apreciar los valores que pueden tomar estas variables.

5.3 Turno

Para poder representar un turno de juego, se utiliza a atributo en la clase Arimaa llamado ‘cant_movimientos’ de tipo entero, la cual indica la cantidad de movimientos que se han realizado en un turno. Un turno puede terminar si, se ha llegado a cuatro movimientos, o si un jugador decide terminarlo después de haber realizado al menos un movimiento. En la ***figura 5.4*** se puede apreciar un diagrama de flujo que representa los distintos movimientos que se pueden ir dando durante un turno.

5.4 Trampas

Para poder comprobar si una pieza está en una celda de trampa se utiliza la matriz de celdas de la clase Arimaa, en la cual se comprueba si existen piezas en las casilla de trampas y si estas están o no adyacentes a una pieza aliada. En la ***tabla 5.1*** que se encuentra en el Anexo B, Sección Tablero, las celdas que representan las celdas de trampas corresponden a la Celda [5][5], Celda [7][5], Celda [5][7] y Celda [7][7].

5.5 Estado Congelado

Para poder representar si una pieza está en estado congelado se utiliza un atributo de tipo boolean en la clase Pieza llamada ‘congelado’. En la ***tabla 5.4*** que se encuentra en el

Anexo B, sección Estado Congelado, se puede apreciar los valores que puede tomar este atributo y los significados de cada uno de estos.

Además existe otro atributo de tipo boolean en la misma clase Pieza llamada congelando, la cual representa si la pieza está congelando a una pieza rival. Los valores que puede tomar este atributo se encuentran en la ***tabla 5.5***, que se encuentra en el Anexo B, sección Estado Congelado.

5.6 Movimientos Especiales

Para poder representar los movimientos especiales, ya sea el movimiento 'empujar' como el movimiento 'tirar' se crearon una serie de atributos en la clase Pieza. Estos atributos son:

- **piezasATirar**: Es una lista de objetos de la clase Pieza, la cual contiene las piezas que enemigas que están condicionadas a ser tiradas por una pieza aliada.
- **piezasQueEmpujan**: Es una lista de objetos de la clase Pieza, la cual contiene las piezas aliadas que pueden empujar a una pieza enemiga que ha sido movida.
- **columnaME y filaME**: son dos atributos de tipo entero, los cuales almacenan la columna y la fila de la pieza que ha sido movida.

5.7 Generación de posibles movimientos

Antes de poder implementar el algoritmo de búsqueda, es necesario conocer cuáles son los posibles movimientos que se pueden realizar. Para poder generar todos estos movimientos se ha utilizado Bitboard.

Un bitboard corresponde a un conjunto de bits, los cuales representan a un cierto estado del juego. La idea es utilizar operaciones Bitwise que permitirán mover los bits hacia la derecha o izquierda. Gracias a esto, será posible representar los distintos movimientos desde un cierto estado.

Esta técnica es muy utilizada en juegos como el Ajedrez, Damas y Othello, debido a que permite optimizar tanto velocidad como memoria.

En la ***figura 5.1*** se puede apreciar una posible posición de los conejos plateados dentro del juego. Para poder representarlo como un bitboard, se utilizará una variable long (64 bits), en donde, por cada bit que sea igual a 1, habrá un conejo plateado. [5]

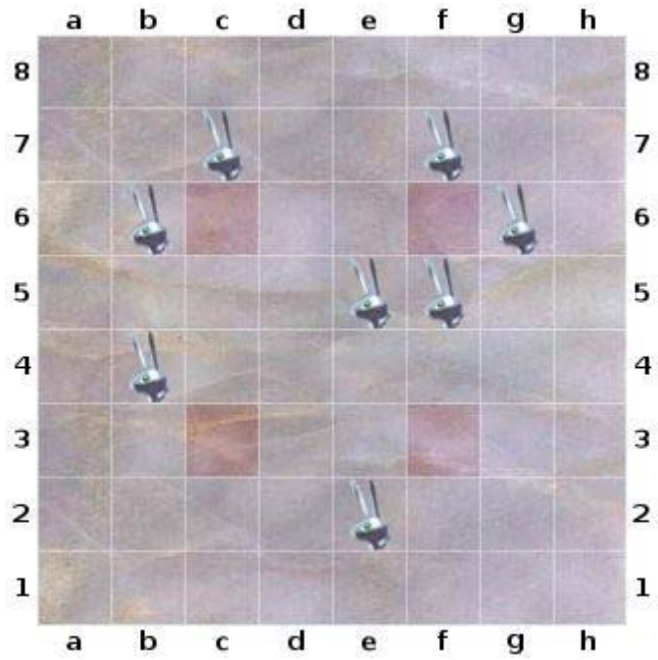


Figura 5.1: Bitboard Conejos Plateados

En este caso, la representación de la posición de los conejos plateados sería:

0000000000100100010000100000110001000000000000000000100000000000

Para poder apreciarlo de mejor manera, se puede ver este número como una matriz de 8x8:

0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0
0 1 0 0 0 0 1 0
0 0 0 0 1 1 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0

Además, se utiliza una técnica conocida como Zobrist Hashing [13], la cual consiste en darle una clave hash a cada posición del tablero. Teniendo estas claves, es posible realizar combinaciones de estas mismas, para así poder representar a los distintos estados que se van generando.

Esta técnica permite optimizar el tiempo de generación del árbol, ignorando movimientos que tengan un estado del tablero igual a uno generado anteriormente.

5.8 Algoritmo de Búsqueda

Tal como se explicó anteriormente, el algoritmo de Optimización por Colonia de Hormigas, al igual que en los algoritmos Minimax y Poda Alfa-Beta, trata de encontrar una solución óptima en un cierto árbol. Esto hace pensar que sería bastante factible poder implementar la metaheurística de Optimización por Colonia de Hormigas en el juego Arimaa.

Para poder realizar esto último es necesario conocer la utilización de esta técnica en otros juegos. Varios son los experimentos que se han llevado a cabo. Es el caso de, por ejemplo, el sistema que resuelve problemas de Sudoku basándose principalmente en cómo el algoritmo actúa en el Problema del Viajante (TSP en inglés) [10]; también está el caso del programa MARCH [11]; y finalmente una implementación para General Game Playing en donde realizan pruebas con juegos como Tic-Tac-Toe, Connect-4, Breakthrough y Checkers [12]. Es precisamente esta última técnica la que se usará como base para la implementación del juego Arimaa.

5.8.1 SCH para Juegos

Cada hormiga en el Sistema de Colonia de Hormigas es un jugador que tiene asignado un papel único de acuerdo a las reglas del juego. La colonia de hormigas es controlada por un agente central que es responsable de la creación de las hormigas, asignarles funciones, pasarles las reglas del juego y, almacenar y actualizar el nivel de feromona y la conveniencia de los caminos alimentados por las hormigas. La ruta, en este caso, es la secuencia de estados y los distintos movimientos que se han realizado a partir de estos estados. Cada combinación movimiento-estado tiene un depósito de feromona asociado con esto. Los movimientos son seleccionados desde un estado basándose de manera probabilística en estos valores. En la ***figura 5.2*** se puede apreciar la estructura básica de la colonia para Sistemas GGP.

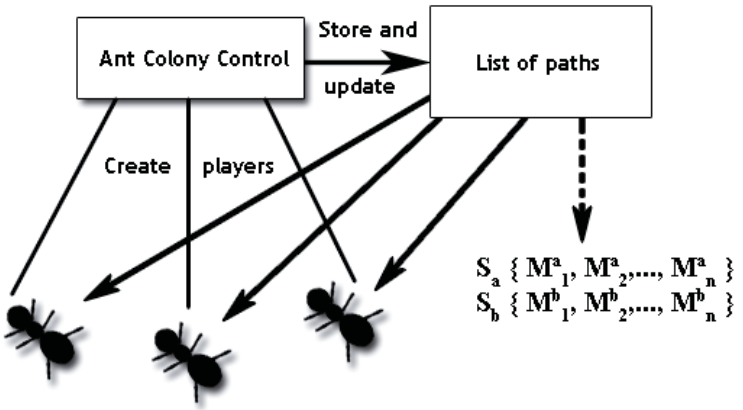


Figura 5.2: Estructura Básica de OCH para GGP

A diferencia del Sistema de Colonia de Hormigas convencional, la deseabilidad de la ruta es también modificada. Esto se hace debido a que no hay conocimiento predefinido del juego. Por lo tanto, la deseabilidad evoluciona con el tiempo a medida que más y más hormigas se alimentan a través de estas rutas. La hormiga ha tenido una búsqueda exitosa si es capaz de reproducir los movimientos que en última instancia lo condujo a un estado terminal ganador.

A continuación se explica cómo se realizará la búsqueda por parte de las hormigas para encontrar el mejor movimiento.

5.8.1 Implementación del SCH

En la ***figura 5.3*** se puede apreciar el algoritmo utilizado para los GGP. Como se denota en la línea 1, todas las hormigas son creadas con un rol único. Consecutivamente, el Control de la Colonia de Hormiga (ACC) mantiene una lista de caminos para cada rol.

```
Inicializar cada Hormiga ϵ Hormigas con un Rol único
SecuenciasDeJuego = lista vacía
Mientras numeroDeBúsquedas ≤ totalBusquedas hacer
        Para todas las hormigas hacer
                estadoActual = estado actual del juego
                secuenciaJuego = lista vacía
                Mientras estado terminal no es alcanzado hacer
                        Seleccionar movimiento m ϵ ListaMovimientosLegales
                        secuenciaJuego.add(estadoActual,m)
                        estadoActual = actualizarEstado(estadoActual,m)
                Fin Mientras
                SecuenciasDeJuego.add(secuenciaJuego)
        Fin Para
        Para todas las rutas t_sm(estado s y movimiento m) ϵ SecuenciasDeJuego hacer
                actualizarFeromona(resultado)
                actualizarDeseabilidad(resultado, distanciaDesdeTerminal)
        Fin Para
        SecuenciasDeJuego = lista vacía
Fin Mientras
```

Figura 5.3: Algoritmo SCH para GGP

Las actualizaciones de los rastros de feromona y conveniencia de cada ruta, se llevan a cabo como sigue. Se utiliza el promedio de los resultados finales de las jugadas como un valor para el depósito de feromonas para el estado y movimiento correspondiente. La actualización de feromona tiene lugar como:

$$\tau_{sm}(t) = \rho\tau_{sm}(t-1) + \sum_{\gamma\epsilon\Gamma} \frac{\Lambda_\gamma}{|\Gamma|} \qquad \textbf{(5.8.1.1)}$$

Λ_γ es el resultado final de la secuencia de juego γ. Γ es el conjunto de todas las secuencias de juego que incluye el movimiento m realizado desde el estado s.

La deseabilidad de una ruta se establece reflejando la puntuación media acumulada por medio del camino y la distancia acumulada, L_{avg}, de ese camino a un estado(s) terminal(es). Una unidad de distancia es un turno en el juego. Por lo tanto, la deseabilidad es actualizada como:

$$\eta_{sm}(t) = \eta_{sm}(t-1) + \frac{wins}{visits} + \frac{\vartheta}{L_{avg}} \qquad \textbf{(5.8.1.2)}$$

ϑ es una constante definida por el usuario que afecta a cómo el parámetro de distancia influye en la deseabilidad. Tener en cuenta que mientras que la feromona es el resultado promedio de un conjunto de búsquedas, la deseabilidad incluye el promedio de todos los resultados observados durante la búsqueda.

En la ***figura 5.5*** ubicada en el Anexo B, sección Algoritmo de Búsqueda, se muestra un diagrama de flujo que representa el funcionamiento del algoritmo de Optimización por Colonia de Hormigas en el juego Arimaa.

5.9 Evaluación de Posiciones

Para poder implementar cualquier tipo de búsqueda es necesario darle ciertos valores a los posibles movimientos que se pueden realizar. Para juegos como el Ajedrez o el Arimaa existen muchas variantes que influyen en la determinación de si una jugada es mejor que otra.

5.9.1 Evaluación de Material

Uno de estos aspectos que ha sido muy utilizado por programadores, es la comparación de piezas, en la cual se piensa que el jugador que tiene una mayor cantidad de piezas es quien posee una cierta ventaja con respecto al rival. Esto, en cierta medida, podría ser cierto, sin embargo, hay que tener en consideración la importancia de las piezas que están en juego. Y es que, no es lo mismo perder un Elefante que un Gato. Otro aspecto importante a tener en cuenta, es el relacionado a la piezas de Conejo, pues, si bien son las con menos fuerza en el juego, son el único método para ganar una partida.

A partir de esto último, en la ***tabla 5.6*** se puede observar el valor material de cada una de las piezas. En la ***tabla 5.7*** se observa que los Conejos son tratados de manera diferente, ya que no sólo se toma en cuenta el valor de la pieza individual, sino que también considera la cantidad de Conejos restantes en el tablero.

Las tablas se pueden encontrar en el Anexo B, sección Evaluación de Material.

5.9.2 Evaluación de la posición individual de las piezas

Otro aspecto importante a la hora de realizar una evaluación de los movimientos, es lo que tiene que ver con la posición individual de las piezas. Para poder implementar esto, es necesario entregarle al programa una serie de parámetros relacionados con "conocimiento ajedrecístico". En un comienzo, en el cual los computadores no tenían un gran nivel de procesamiento, era prácticamente imposible implementar este tipo de evaluación. Sin embargo, a medida que la tecnología fue avanzando, el incluir factores posicionales ya no iría en contra de la capacidad de cálculo.

En Arimaa, al igual que en el Ajedrez las piezas toman valores diferentes dependiendo del lugar en el que se encuentren. Además, otro aspecto a considerar será que tipo de pieza es la que se encuentra en tal casilla, pues, las piezas, al tener distintos valores materiales, cumplen distintas funciones dentro del juego. A continuación se explicará cuál es la función de cada pieza y qué valor toma en cada una de las casillas del tablero.

Conejos: Es una pieza débil, que es la clave para lograr ganar una partida. Como es natural, su mejor posición es la meta y sus cercanías. Al ser una pieza débil, debe tener cuidado con los cuadros de trampa, y por su cantidad en el tablero también puede ser utilizado en la defensa cubriendo espacios. En la ***tabla 5.8*** figuran los valores que se le asignaron a los conejos dependiendo de las distintas posiciones que ocupan en el tablero.

Se debe destacar que los Conejos son una pieza bastante expuesta a movimientos especiales o congelación por parte de las piezas rivales, por lo cual, se hace muy importante que estas estén adyacentes a alguna otra pieza aliada. En la ***tabla 5.9*** se puede observar la bonificación que existe para la pieza Conejo cuando está adyacente a una pieza aliada, en las distintas filas del tablero.

Gatos: Los gatos son piezas débiles que sólo pueden congelar a los Conejos enemigos, por lo que su misión será netamente defensiva, y sus mejores posiciones girarán en torno a los cuadros de trampa del lado propio. En la ***tabla 5.10*** se puede apreciar los valores que toma el Gato en los distintos lugares del tablero.

Perros: Los perros son piezas relativamente débiles que pueden congelar a los Gatos y Conejos rivales, por lo que su misión, al igual que los Gatos, será netamente defensiva, dándoles, de cierto modo, un apoyo a estos. Sus mejores posiciones girarán en torno al control de los cuadros de trampa del lado propio. En la ***tabla 5.11*** se aprecia los valores que toma el Perro en los distintos lugares del tablero.

Caballos: Los Caballos son piezas relativamente fuertes que pueden congelar a los Conejos, Gatos y Perros enemigos, por lo que su mayor utilidad en el tablero será en torno a los cuadros de trampa y su misión es de ataque y defensa; solo deben tener cuidado del

Camello y Elefante enemigo. En la ***tabla 5.12*** se puede apreciar los valores que toma el Caballo en los distintos lugares del tablero.

Camello: El Camello es una de las piezas más fuertes del tablero, pudiendo ser congelada solamente por el Elefante enemigo. Su mayor utilidad en el tablero será en torno a los cuadros de trampa, por lo que su mejor ubicación se encuentra en el centro del tablero que es donde le permite tener mayor movilidad para llegar a cualquier trampa. Es una pieza ideal para atacar y defender. En la ***tabla 5.13*** se aprecia los valores que toma el Camello en los distintos lugares del tablero.

Elefante: El Elefante es la pieza más fuerte del tablero, no pudiendo ser congelado por ninguna otra pieza. Al igual que el Camello, su mayor potencial será en las casillas centrales, lo cual le permite llegar rápidamente a los cuadros de trampa, ya sea para capturar o salvar una pieza aliada. En la ***tabla 5.14*** se puede apreciar los valores que toma el Elefante en los distintos lugares del tablero.

Las tablas se pueden encontrar en el Anexo B, sección Evaluación Individual de las piezas

5.9.3 Control de Trampas

Como se explicó en las Reglas de Arimaa, las trampas son el único método para poder eliminar a una pieza del tablero. Es por esto, que el control de estas casillas especiales es muy importante para mantener una ventaja con respecto al jugador enemigo.

Al comenzar el juego, cada jugador controla las dos trampas más cercanas. Existen diferentes estrategias que pueden ser utilizadas para poder hacer que una pieza enemiga caiga en un casilla de trampa. Por lo general, para poder controlar las distintas casillas, es preferible utilizar piezas de mayor jerarquía como el Elefante o Caballo, de tal manera que sea más accesible poder empujar o tirar piezas enemigas a la casilla de trampa, y así mismo, proteger a las piezas aliadas más débiles.

En la ***tabla 5.15*** se puede apreciar de mejor manera esto último, en donde las piezas con mayor jerarquía tienen un mayor valor en lo que respecta a la dominación de una trampa. Otro aspecto importante que hay que tener en cuenta, son las celdas cercanas a una celda de trampa. En la ***tabla 5.16*** se puede ver un ejemplo de los valores que toman las casillas más cercanas a una casilla de trampa.

Las tablas se pueden encontrar en el Anexo B, sección Control de Trampas.

5.9.4 Movilidad

En muchas ocasiones, un movimiento en juegos como el Ajedrez o el Arimaa, en primera instancia puede parecer bueno, sin embargo, hay veces que resulta que en un par de jugadas más, ese movimiento ocasionó una derrota o una considerable pérdida de material.

En esta función se realiza una evaluación de las posibles consecuencias que puede generar un movimiento dado. Además mide la capacidad de libertad de movimiento para cada pieza, y la capacidad de moverse sin ningún ataque de por medio.

La libertad de movimiento, en este caso, corresponde a la cantidad de casillas disponibles para cada pieza. Este factor es muy importante debido a que mientras más movimientos posibles se puedan realizar, va haber más probabilidad de encontrar una buena jugada.

Para poder llevar esto a cabo, es necesario realizar una penalización a una pieza cada vez que esta tenga enemigos más fuertes en su cercanía, y una bonificación a una pieza cuando esta tenga aliados más fuertes cerca. En la ***tabla 5.17*** y ***tabla 5.18*** se puede apreciar de mejor manera estos hechos, indicando el valor que tomarán las piezas cada vez que están cerca de una cierta pieza aliada o enemiga.

Las tablas se pueden encontrar en el Anexo B, sección Movilidad.

5.10 Función de Evaluación

Para poder implementar todos estos factores de evaluación, se debe realizar lo que se denomina Función de Evaluación. Esta es una función lineal de la forma $w_1f_1 + w_2f_2 + ... + w_nf_n$, donde w_i son los pesos asociados a las características f_i. A continuación se presentan las funciones para cada uno de los factores estudiados.

- Evaluación de Material

$$f_1 = \sum_{i=1}^{8}\sum_{j=1}^{8}\left(\frac{VC_{(i,j)}}{|VC_{(i,j)}|}\cdot VTM_{vc(i,j)}\right) \forall\, VC_{(a,b)} \neq 0 \qquad \textbf{(5.10.1}$$

- Evaluación de la Posición

$$f_2 = \sum_{i=1}^{8}\sum_{j=1}^{8}\left(\frac{VC_{(i,j)}}{|VC_{(i,j)}|}\cdot VTP_{vc(i,j)}\right) \forall\, VC_{(a,b)} \neq 0 \qquad \textbf{(5.10.2)}$$

- Control de Trampas

$$f_3 = \sum_{i=1}^{2}\sum_{j=1}^{2}\left(2VC_{(3i\pm1,3j)} + 2VC_{(3i,3j\pm1)} + VC_{(3i\pm1,3j-1)} + VC_{(3i\pm1,3j+1)} + VC_{(3i\pm2,3j)} + VC_{(3i,3j\pm2)}\right) \qquad \textbf{(5.10.3)}$$

- Evaluación de Movilidad

$$f_4 = \sum_{i=1}^{8}\sum_{j=1}^{8}\left(\frac{VC_{(i\pm1,j)}}{|VC_{(i\pm1,j)}|}\cdot VTB_{vc(i,j)\rightarrow vc(i\pm1,j)} + \frac{VC_{(i,j\pm1)}}{|VC_{(i,j\pm1)}|}\cdot VTB_{vc(i,j)\rightarrow vc(i,j\pm1)}\right) \forall\, VC_{(a,b)} \neq 0 \qquad \textbf{(5.10.4)}$$

Dónde:

VC(a,b): Valor de la Casilla en la fila **a** con la columna **b** (se obtiene de la matriz de la posición evaluada)

VTMvc(a,b): Valor de la tabla de material para la pieza correspondiente a VC(a,b) (tablas de la sección Evaluación de material)

VTPvc(a,b): Valor de la tabla posición para la pieza correspondiente a VC(a,b) (tablas de la sección Evaluación de la posición individual de las piezas)

VTBvc(a,b)→vc(c,d): Valor de la tabla de movilidad, de la pieza correspondiente a VC(a,b) en relación a la pieza correspondiente a VC(c,d) (tablas de la sección Movilidad)

Inicialmente se considerará $w1=w2=w3=w4=1$ por lo que la primera función de evaluación que se implementará es:

f.e. = f1 + f2 + f3 +f4

6 Implementación y Resultados Computacionales

6.1 Implementación

6.1.1 Interfaz de Usuario

Para la implementación de la interfaz del juego se han creado 3 ventanas. En primer lugar está la Ventana Principal, la cual contiene 2 botones para acceder a las demás ventanas, además de contener un radio button con las opciones "1 Jugador" y "2 Jugadores". La segunda ventana corresponde a la Ventana de Juego, en la cual se aprecia el tablero de juego con las piezas de juego, además de un botón que sirve para terminar un turno, y un texto que indica el turno. Finalmente la tercera ventana corresponde a la Ventana de Reglas la cual contiene una especie de pestañas para mostrar las distintas reglas del juego Arimaa.

Las Interfaces de Usuario se pueden ver en el Anexo C, sección Interfaz de Usuario.

6.1.2 Pruebas

Una vez implementado el juego Arimaa, este se sometió a pruebas con el fin de determinar si resuelve o no el problema de manera satisfactoria. Para ello se le suministraron distintos tipos de datos.

6.1.2.1 Pruebas de Caja Blanca

Las pruebas de Caja Blanca corresponden a aquellas en las que se asegura que la operación interna se ajusta a las especificaciones, y que todos los componentes internos se han probado de forma adecuada.

En este caso, para el probar los métodos internos del programa Arimaa se utilizaron Diagramas de Flujos, los cuales ya fueron presentados anteriormente.

6.1.2.2 Pruebas de Caja Negra

Las pruebas de Caja Negra son aquellas que comprueban que cada función del programa opere de buena manera, entregando los resultados esperados.

Para poder probar el funcionamiento del programa, se debe tomar en consideración tanto el turno del jugador como el turno del programa. En el caso del turno del jugador se tomaron en cuenta los siguientes casos de prueba:

- Turnos de un paso
    - PS
- Turnos de dos pasos
    - PS+PS
    - T
    - E
- Turnos de tres pasos
    - PS+PS+PS
    - E+PS
    - PS+E
    - T+PS
    - PS+T
- Turnos de cuatro pasos
    - PS+PS+PS+PS
    - E+PS+PS
    - T+PS+PS
    - PS+E+PS
    - PS+T+PS
    - PS+PS+E
    - PS+PS+T
    - E+E
    - T+T
    - E+T
    - T+E

Donde:

PS: Paso Simple

E: Empujar

T: Tirar

Para el testeo de posibles movimientos ilegales, se probaron los siguientes escenarios:

- Se intentó terminar el turno con cero movimientos.
- Se intentó terminar el turno con un movimiento de empujar inconcluso.
- Se intentó dejar inconcluso un empujar y mover otra pieza.
- Se intentó empujar piezas de igual o mayor fuerza.

- Se intentó tirar piezas de igual o mayor fuerza.

Para el turno del programa, se consideraron pruebas tanto para el generador de movimientos como para la función de búsqueda. Como se sabe, en el juego Arimaa se pueden realizar de 1 a 4 movimientos en cada turno, por lo que el árbol de juego que se va a generar es mucho mayor que en otros como el ajedrez, en el cual solo se puede realizar un movimiento por turno.

Para el caso del generador de movimientos, el programa funciona de manera correcta generando todos los movimientos posibles (1 a 4 movimientos por turno). Esto se pudo lograr gracias al uso de las técnicas 'Bitboard' y 'Zobrist Hashing' vistas anteriormente.

Otro punto que se debe tomar en cuenta es que en el algoritmo de Optimización por Colonia de Hormigas, el resultado puede ir variando dependiendo de la cantidad de hormigas que se ejecuten. En la ***tabla 6.1***, la ***tabla 6.2*** y la ***tabla 6.3*** se puede observar el tiempo que demoran los distintos puzles en ejecutarse. Además de la cantidad de hormigas, se tomaron en cuenta factores como el número de iteraciones y la profundidad de la búsqueda.

Tabla 6.1: Pruebas de tiempos (ms) para el SCH con 30 hormigas

	200 iteraciones		350 iteraciones	
	Profundidad = 1	Profundidad = 2	Profundidad = 1	Profundidad = 2
Puzle 1	PT	PT	PT	PT
Puzle 2	652	859	1075	1338
Puzle 3	PT	PT	PT	PT
Puzle 4	97	555	117	898
Puzle 5	294	706	428	1071
Puzle 6	20417	30424	36674	48188
Puzle 7	18079	70610	28327	449747
Puzle 8	324406	OVERFLOW	531300	OVERFLOW

Tabla 6.2: Pruebas de tiempos (ms) para el SCH con 50 hormigas

	50 Hormigas			
	200 iteraciones		350 iteraciones	
	Profundidad = 1	Profundidad = 2	Profundidad = 1	Profundidad = 2
Puzle 1	PT	PT	PT	PT
Puzle 2	1080	1364	1762	2256
Puzle 3	PT	PT	PT	PT
Puzle 4	158	887	185	1531
Puzle 5	457	1136	733	1964
Puzle 6	40120	49066	57906	81762
Puzle 7	30065	439967	48222	586448
Puzle 8	582327	OVERFLOW	909784	OVERFLOW

Tabla 6.3: Pruebas de tiempos (ms) para el SCH con 70 hormigas

	70 Hormigas			
	200 iteraciones		350 iteraciones	
	Profundidad = 1	Profundidad = 2	Profundidad = 1	Profundidad = 2
Puzle 1	PT	PT	PT	PT
Puzle 2	1438	1673	2531	2947
Puzle 3	PT	PT	PT	PT
Puzle 4	231	1158	250	2086
Puzle 5	593	1477	1017	2687
Puzle 6	44097	57015	82478	103511
Puzle 7	37514	387244	61650	877636
Puzle 8	727078	OVERFLOW	1293857	OVERFLOW

En estas tablas, se puede visualizar que existen celdas con un color “rojo”, y otras con un color “verde”. En el primer caso significa que el Puzle pudo ser resuelto con el mejor movimiento posible; en el segundo caso, significa que el Puzle no pudo encontrar la solución óptima. Se debe destacar además que tanto para el Puzle 1 como para el Puzle 3, ya existe una condición de término (CT), por lo que no es necesario que el algoritmo se ejecute. Otro punto importante, es que el Puzle 8 no pudo encontrar ninguna solución con más de 1 profundidad, debido que entraba a un estado de overflow.

En el Anexo C, sección Puzles, se encuentran las imágenes correspondientes a los resultados que el programa entregó para cada uno de los puzles. Estos puzles utilizaron la configuración de 50 hormigas, 350 iteraciones y 1 nivel de profundidad. Esto debido a que entre todas las combinaciones que se hicieron, esta otorga una mayor cantidad de puzles resueltos de manera óptima con el menor tiempo.

6.2 Análisis de Resultados

Teniendo los resultados de la implementación de la metaheurística de Optimización por Colonia de Hormigas, es posible realizar un análisis de estos mismos. Para esto es necesario compararlo con el algoritmo Poda Alfa-Beta. A continuación se presentan las tablas de resultados para el algoritmo Poda Alfa-Beta[2].

Tabla 6.4: Evaluación Poda Alfa-Beta (parte 1)

Puzle	Límite Jugadas	Profundidad	Tiempo Promedio (milisegundos)	Resuelve Puzle
Puzle 1	5000	1	480,6	si
Puzle 1	4000	1	474,8	si
Puzle 1	3000	1	479,4	si
Puzle 1	2000	1	475,4	si
Puzle 1	1000	1	486,2	si
Puzle 1	75	2	2181,8	si
Puzle 1	60	2	1127,2	si
Puzle 1	55	2	1062,8	si
Puzle 1	45	2	988,8	si
Puzle 1	30	2	909,2	si
Puzle 1	30	3	7930,4	no
Puzle 2	5000	1	228	si
Puzle 2	4000	1	228,6	si
Puzle 2	3000	1	227,4	si
Puzle 2	2000	1	226,8	si
Puzle 2	1000	1	225,8	si
Puzle 2	75	2	3283,4	si
Puzle 2	60	2	1932,8	si
Puzle 2	55	2	1501,8	si
Puzle 2	45	2	975	si
Puzle 2	30	2	469,4	si
Puzle 2	30	3	3323,4	si
Puzle 2	10	3	1131	si
Puzle 3	5000	1	139,6	si
Puzle 3	4000	1	136,6	si
Puzle 3	3000	1	138,2	si
Puzle 3	2000	1	132,2	si
Puzle 3	1000	1	138,2	si
Puzle 3	75	2	9479,2	no

[2] Estas tablas fueron extraídas de [1].

Tabla 6.5: Evaluación Poda Alfa-Beta (parte 2)

Puzle	Límite Jugadas	Profundidad	Tiempo Promedio (milisegundos)	Resuelve Puzle
Puzle 4	5000	1	57,8	si
Puzle 4	4000	1	57,6	si
Puzle 4	3000	1	59,6	si
Puzle 4	2000	1	58,4	si
Puzle 4	1000	1	58,4	si
Puzle 4	75	2	1132,2	si
Puzle 4	60	2	1085	si
Puzle 4	55	2	1097,2	si
Puzle 4	45	2	1081,4	si
Puzle 4	30	2	1076,8	si
Puzle 4	30	3	11650	si
Puzle 4	10	3	8839	no
Puzle 5	5000	1	139	si
Puzle 5	4000	1	131,8	si
Puzle 5	3000	1	130,4	si
Puzle 5	2000	1	129	si
Puzle 5	1000	1	129	si
Puzle 5	75	2	4760,6	no
Puzle 6	5000	1	179197,6	si
Puzle 6	4000	1	176704	si
Puzle 6	3000	1	66202	no
Puzle 7	5000	1	67530,8	si
Puzle 7	4000	1	67264,2	si
Puzle 7	3000	1	67869,6	si
Puzle 7	2000	1	26341,2	si
Puzle 7	1000	1	3472,6	si
Puzle 7	75	2	OM	no
Puzle 8	5000	1	66185,5	no

En primer lugar se debe considerar el número de puzles resueltos. Claramente, el algoritmo Poda Alfa-Beta presenta una ventaja pudiendo resolver 7 de los 8 puzles, a diferencia del algoritmo de Colonia de Hormigas que solo resuelve los primeros 5 puzles.

Otro factor a tener en cuenta es el tiempo que se demora el algoritmo en ejecutarse. En este caso, los tiempos de los primeros 5 puzles son relativamente similares. En los puzles 6, 7 y 8, el tiempo de ejecución del algoritmo Poda Alfa-Beta es menor que el de SCH, y esto debido principalmente al límite de jugadas de búsqueda. Aún con esta limitación, el algoritmo Alfa-Beta presenta buenas soluciones para el puzle 6 y 7.

Basándose en lo anterior, se puede concluir que para este tipo de problemas; en donde el tamaño del árbol de juego es muy grande y no existe un conocimiento previo de las distancias entre los nodos; el algoritmo Poda Alfa-Beta funciona de mejor manera.

Es por esto, y tomando en cuenta los resultados de los puzles de Arimaa y de otros juegos[3], se puede concluir que la metaheurística de Optimización por Colonia de Hormigas funciona de buena manera para juegos con un árbol de juego relativamente pequeño, no así para juegos con juegos con un árbol de tamaño grande.

[3] Los experimentos de estos juegos se encuentran en el Anexo D: Experimentos en otros juegos.

7 Conclusiones

Como se mencionó en el informe, Arimaa es un juego de estrategia relativamente nuevo, y que por lo tanto no tiene tantas referencias bibliográficas. Esto hace que este proyecto haya sido, de cierta forma, bastante interesante trabajarlo. Y es que si bien, ya las reglas que presenta este juego son fáciles de comprender para un humano, presenta una gran dificultad al tener que implementarlo en un computador. Esta es una de las razones por la cual aún no se ha creado un programa que sea capaz de ganar el desafío que impuso el creador de este juego.

Los algoritmos de inteligencia artificial que han sido utilizados tanto en el ajedrez como en Arimaa destacan por el uso de la fuerza bruta. En el informe se analizaron el algoritmo Minimax y Poda Alfa-Beta, en el que el segundo presenta una mejora con respecto al primero, pues permite ignorar los peores movimientos. Sin embargo, para poder realizar esto, es necesario realizar un orden en la lista de posibles jugadas para poder eliminar las peores. En el caso del Arimaa, en donde la probabilidad de captura es bastante baja, la dificultad de ordenar la lista es mucho mayor que en el ajedrez.

Otro de los algoritmos que se vieron en el informe es la Metaheurística de Optimización por Colonia de Hormigas, el cual si bien, no se ha implementado en el juego Arimaa, puede parecer una alternativa factible, pues al igual que en los algoritmos de Minimax y Poda Alfa-Beta, va analizando las soluciones mediante grafos. En OCH se pueden observar que existen diferentes variantes que permiten solucionar distintos tipos de problemas.

Antes de implementar el algoritmo en el juego Arimaa, fue necesario conocer en qué otros juegos se ha utilizado esta metaheurística. Dentro de estos juegos están: el Ajedrez, Sudoku, Tic-Tac-Toe, Connect-4, Breakthrough y Checkers. Esto fue muy importante, debido principalmente a que en los juegos, a diferencia del Problema del Viajante (TSP), no se tiene conocimiento previo de la distancia entre los nodos.

Para poder realizar el juego, se analizaron los distintos métodos de planificación, en el cual se decidió por el "Método en Cascada" debido principalmente a que durante el desarrollo del sistema, los requerimientos no van a cambiar. Siguiendo con este método, se analizaron los requerimientos del juego para luego diseñar los distintos casos de uso y un diagrama de clases en una versión alfa.

Finalmente se realizaron algunas pruebas, con distinta cantidad de hormigas, cantidad de iteraciones y nivel de profundidad. En este caso, la mejor combinación posible fue con 50 hormigas, 350 iteraciones y 1 nivel de profundidad.

Teniendo los resultados de la implementación con la metaheurística Colonia de Hormigas, fue posible realizar una comparación con el algoritmo Poda Alfa-Beta. En este

caso se pudo observar que el algoritmo Poda Alfa-Beta logra solucionar una mayor cantidad de puzles, por lo cual es una mejor alternativa para este tipo de juegos.

8 Referencias

[1] Andrés Cerón Lillo: "El Desafío de Arimaa", Julio 2010.

[2] Levy, David & Newborn, Monty (1991), Cómo Juegan las Computadoras al Ajedrez, Computer Science Press, ISBN 0-7167-8121-2

[3] Omar Syed y Aamir Syed, Arimaa: A New Game Designed to be Difficult for Computers, Junio 2003.

[4] Victor Allis (1994). Searching for Solutions in Games and Artificial Intelligence. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands. ISBN 90-900748-8-0.

[5] Carlini, S., Arimaa: From Rules to Bitboard Analysis, Knowledge Representation Thesis, University of Modena and Reggio Emilia, 2008.

[6] Yngvi Björnsson y Tony Marsland: Multi-Cut αβ-Pruning in Game-Tree Search, Theoretical Computer Science 252 (1-2), 2001..

[7] Sergio Alonso, Oscar Cordón, Iñaki Fernandez de Viana, Francisco Herrera. La Metaheurística de Optimización Basada en Colonias de Hormigas: Modelos y Nuevos Enfoques.

[8] S. Pressman, Roger. Ingeniería del Software: Un enfoque práctico, 3.ª Edición, Pag. 26-30.

[9] Modelado de Sistemas con UML. Disponible vía web en http://es.tldp.org/Tutoriales/doc-modelado-sistemas-UML/doc-modelado-sistemas-uml.pdf

[10] David Mullaney: Using Ant Systems to Solve Sudoku Problems.

[11] Alexis Drogoul: When Ants Play Chess (Or Can Strategies Emerge From Tactical Behaviors?). In: 5th European Workshop on Modelling Autonomous Agents in a Multi-Agent World. (1995).

[12] Shiven Sharma, Ziad Kobti, and Scott Goodwin: General Game Playing with Ants.

[13] Zobrist, A.: A new hashing method with application for game playing. Technical report 99, University of Wisconsin (1970).

[14] Bjornsoon, Y., H., F.: Simulation-based approach to general game playing. In: Proceedings of the Twenty-Third AAAI Conference on Arti_cial Intelligence, AAAI Press (2008).

[15] Christ – Jan Cox: Analysis and Implementation of the Game Arimaa, Marzo 2006.

[16] Zhong, H., Building a Strong Arimaa-playing Program, University of Alberta, 2005.

[17] Sitio Web de Arimaa. Disponible vía web en http://arimaa.com/arimaa/.

Anexos
A: Diseño de la Solución
Especificación Formal

Tabla 4.1: Especificación Formal Caso de Uso "Mover Pieza"

Caso de Uso	Mover Pieza
Actor Principal	Usuario
Participantes e Intereses	Usuario: Desea mover una pieza en el tablero
Precondiciones	1. El usuario debe ingresar a la aplicación 2. El usuario debe escoger el tipo de oponente 3. El usuario debe ingresar a la opción "Iniciar Juego"
Post condiciones	El usuario debe poder mover una pieza en el tablero
Escenario Principal	1. El usuario selecciona una pieza 2. El sistema muestra las opciones de movimiento que esa pieza puede realizar 3. El usuario elige una opción de movimiento de la pieza
Extensiones	2.1 Dentro de las opciones están: 2.1.1 Movimiento normal 2.1.2 Empujar 2.1.3 Tirar 3.1 Si elige empujar: 3.1.1 Usuario mueve la pieza a una celda adyacente donde hay una pieza rival más débil. 3.1.2 Usuario mueve la pieza del rival a una celda adyacente vacía. 3.2 Si elige tirar 3.2.1 Usuario mueve pieza a una celda adyacente vacía. 3.2.2 Sistema reposiciona la pieza rival más débil a donde estaba la otra pieza.
Requisitos Especiales	No hay requisitos especiales.
Frecuencia de Ocurrencia	Alta

Tabla 4.2: Especificación Formal Caso de Uso "Terminar turno"

Caso de Uso	Terminar turno
Actor Principal	Usuario
Participantes e Intereses	Usuario: Desea terminar su turno de juego
Precondiciones	1. El usuario debe ingresar a la aplicación. 2. El usuario debe escoger el tipo de oponente. 3. El usuario debe ingresar a la opción "Iniciar Juego". 4. El usuario debe haber posicionado las piezas iniciales en el tablero. 5. El usuario debe haber realizado al menos 1 movimiento de pieza durante su turno.
Post condiciones	El usuario debe poder terminar su turno.
Escenario Principal	1. El usuario presiona el botón "Terminar turno" 2. El sistema comprueba que se ha realizado al menos un movimiento
Extensiones	2.1 Si no se ha realizado ningún movimiento 2.1.1 Sistema indica que aún no se puede terminar el turno.
Requisitos Especiales	No hay requisitos especiales.
Frecuencia de Ocurrencia	Alta

Tabla 4.3: Especificación Formal Caso de Uso "Posicionar piezas iniciales"

Caso de Uso	Posicionar piezas iniciales
Actor Principal	Usuario
Participantes e Intereses	Usuario: Desea jugar una partida de arimaa
Precondiciones	4. El usuario debe ingresar a la aplicación 5. El usuario debe escoger el tipo de oponente 6. El usuario debe ingresar a la opción "Iniciar Juego"
Post condiciones	El usuario debe poder posicionar las piezas iniciales en el tablero.
Escenario Principal	1. El usuario selecciona una pieza. 2. El sistema muestra las celdas en las que la pieza se puede posicionar. 3. El usuario mueve la pieza a una de las celdas mostradas por el sistema.
Extensiones	
Requisitos Especiales	No hay requisitos especiales.
Frecuencia de Ocurrencia	Alta

B: Representación de los Datos

Tablero

Tabla 5.1: Representación del Tablero

Celda [0][11]	Celda [1][11]	…	Celda [10][11]	Celda [11][11]
Celda [0][10]	Celda [1][10]	…	Celda [10][10]	Celda [11][10]
…	…	…	…	…
Celda [0][1]	Celda [1][1]	…	Celda [10][1]	Celda [11][1]
Celda [0][0]	Celda [1][0]	…	Celda [10][0]	Celda [11][0]

Piezas

Tabla 5.2: Valores posibles del atributo 'color'

Valor	Significado
0	Jugador Dorado
1	Jugador Plateado

Tabla 5.3: Valores posibles del atributo ‘tipo’

Valor	Significado
1	Tipo Elefante
2	Tipo Camello
3	Tipo Caballo
4	Tipo Perro
5	Tipo Gato
6	Tipo Conejo

Turno

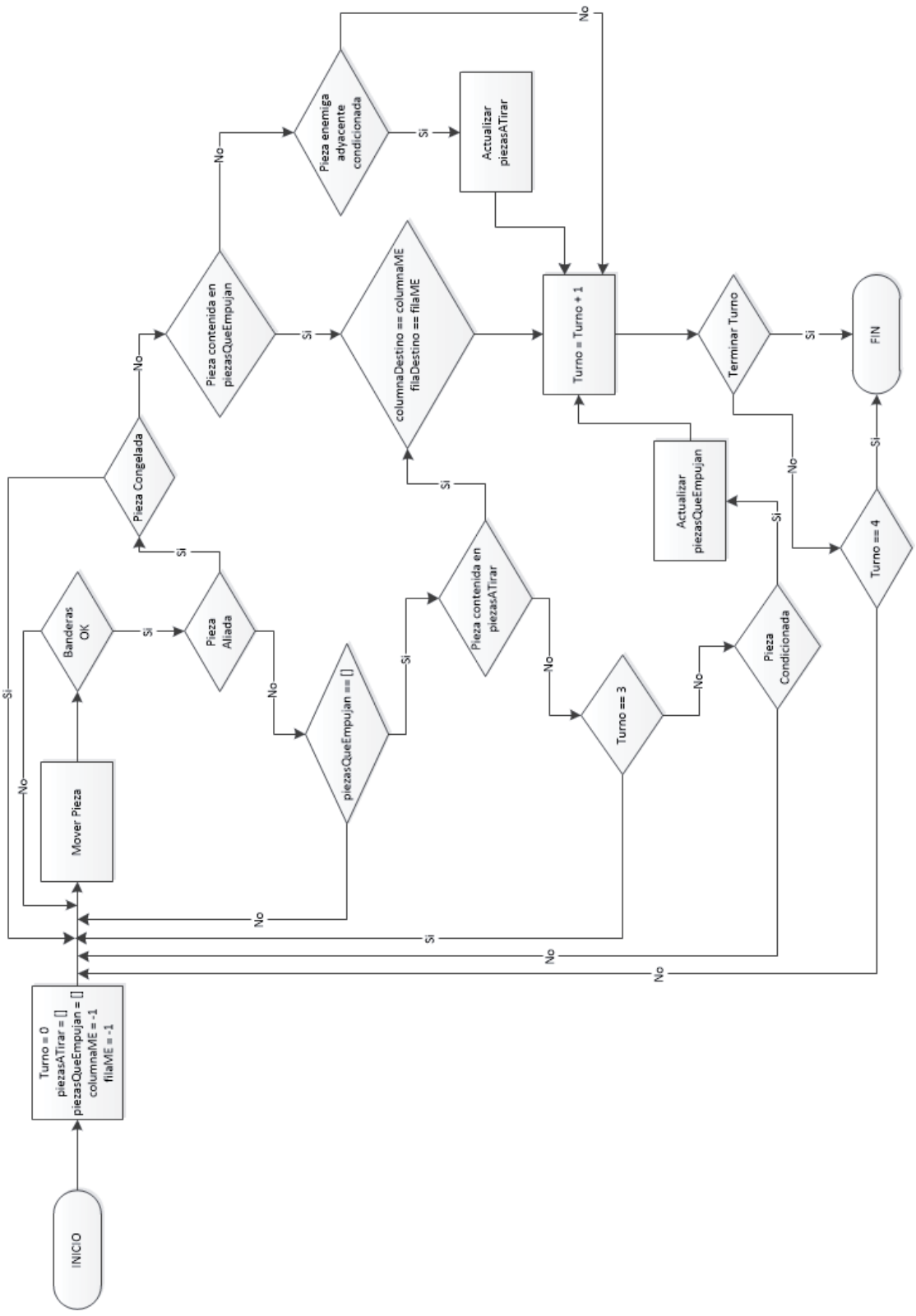


Figura 5.4: Diagrama de Flujo “Turno”

Estado Congelado

Tabla 5.4: Valores posibles del atributo ‘congelado’

Valor	Significado
True	La pieza está congelada
False	La pieza no está congelada

Tabla 5.5: Valores posibles del atributo ‘congelando’

Valor	Significado
True	La pieza está congelando
False	La pieza no está congelando

Algoritmo de Búsqueda

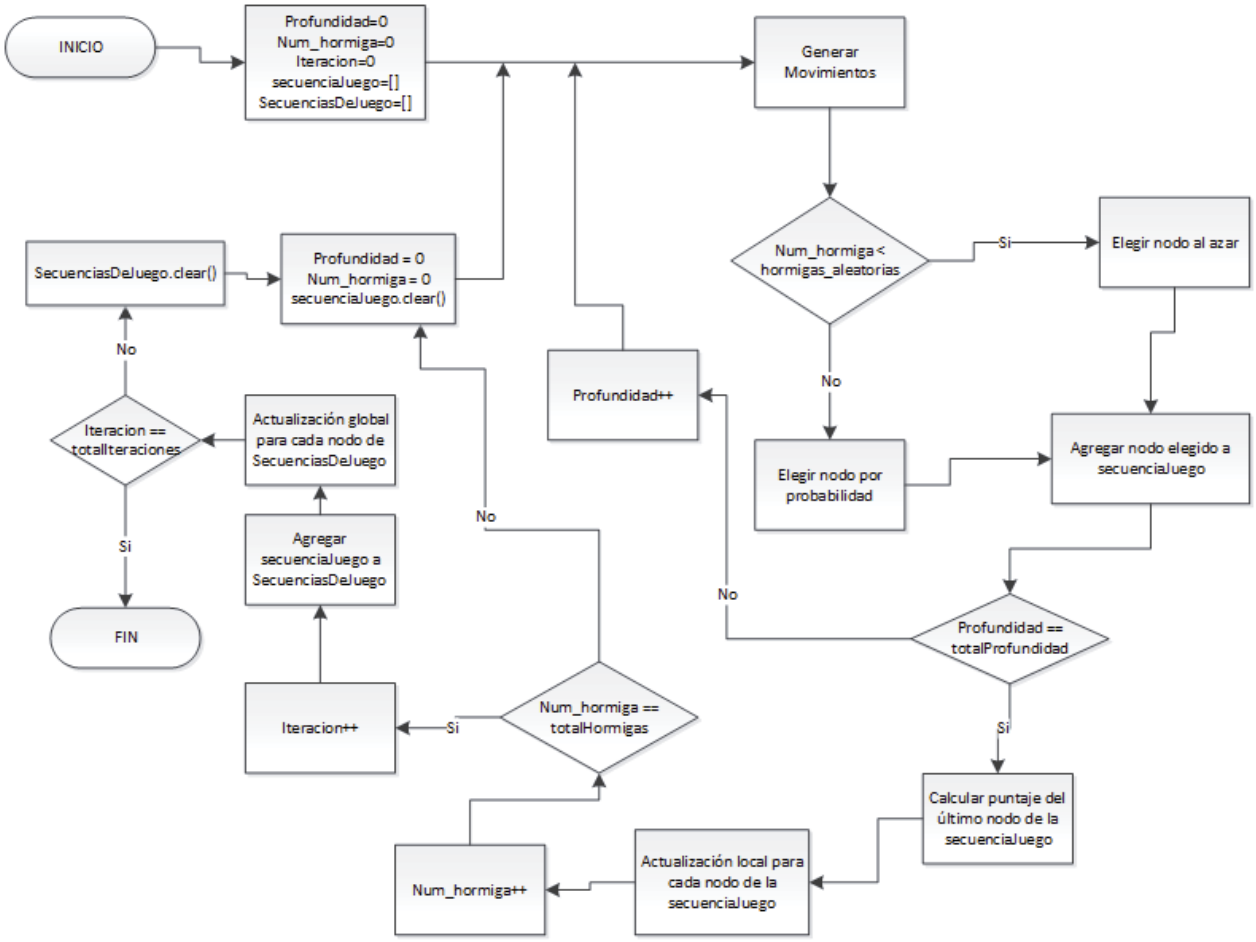


Figura 5.5: Diagrama de Flujo "SCH"

Evaluación de Material

Tabla 5.6: Valor Material de las piezas

Pieza	Valor
Elefante	17
Camello	13
Caballo	8
Perro	6
Gato	5

Tabla 5.7: Valor Material de la pieza Conejo

Cantidad Conejos	Valor Grupal
8	40
7	38
6	36
5	32
4	28
3	24
2	20
1	18

Evaluación de la posición individual de las piezas

Tabla 5.8: Valores de la posición de un Conejo

8	999	999	999	999	999	999	999	999
7	7	6	5	4	4	5	6	7
6	6	2	-1	0	0	-1	2	6
5	5	1	0	0	0	0	1	5
4	4	1	0	0	0	0	1	4
3	3	1	-1	0	0	-1	1	3
2	2	2	1	1	1	1	2	2
1	2	2	2	2	2	2	2	2
	A	B	C	D	E	F	G	H

Tabla 5.9: Bonificación de los Conejos por piezas aliadas

Posición	Bonificación
Fila 1	1
Fila 2	2
Fila 3	3
Fila 4	4
Fila 5	5
Fila 6	6
Fila 7	8
Fila 8	10

Tabla 5.10: Valores de la posición de un Gato

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	-1	0	0	-1	0	0
5	0	0	1	0	0	0	0	0
4	1	1	2	1	1	2	1	1
3	3	5	-1	5	5	-1	5	3
2	2	4	5	4	4	5	4	2
1	1	2	3	2	2	3	2	1
	A	B	C	D	E	F	G	H

Tabla 5.11: Valores de la posición de un Perro

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	-1	0	0	-1	0	0
5	0	0	1	0	0	0	0	0
4	1	1	2	1	1	2	1	1
3	3	5	-1	5	5	-1	5	3
2	2	4	5	4	4	5	4	2
1	1	2	3	2	2	3	2	1
	A	B	C	D	E	F	G	H

Tabla 5.12: Valores de la posición de un Caballo

8	0	0	1	2	2	1	0	0
7	0	0	2	3	3	2	0	0
6	1	2	-1	4	4	-1	2	1
5	2	3	4	5	5	4	3	2
4	2	3	4	5	5	4	3	2
3	1	2	-1	4	4	-1	2	1
2	0	0	2	3	3	2	0	0
1	0	0	1	2	2	1	0	0
	A	B	C	D	E	F	G	H

Tabla 5.13: Valores de la posición de un Camello

8	0	0	1	2	2	1	0	0
7	0	0	2	4	4	2	0	0
6	1	2	-1	4	4	-1	2	1
5	2	4	4	5	5	4	4	2
4	2	4	4	5	5	4	4	2
3	1	2	-1	4	4	-1	2	1
2	0	0	2	4	4	2	0	0
1	0	0	1	2	2	1	0	0
	A	B	C	D	E	F	G	H

Tabla 5.14: Valores de la posición de un Elefante

8	0	0	1	2	2	1	0	0
7	0	0	2	3	3	2	0	0
6	1	2	-1	5	5	-1	2	1
5	2	3	5	5	5	5	3	2
4	2	3	5	5	5	5	3	2
3	1	2	-1	5	5	-1	2	1
2	0	0	2	3	3	2	0	0
1	0	0	1	2	2	1	0	0
	A	B	C	D	E	F	G	H

Control de Trampas

Tabla 5.15: Valores por custodia de trampa

Pieza	Valor
Elefante	12
Camello	10
Caballo	8
Perro	6
Gato	4
Conejo	2

Tabla 5.16: Multiplicadores del control de la trampa

8								
7								
6			0			0		
5			0.5					
4		0.5	1	0.5				
3	0.5	1	0	1	0.5	0		
2		0.5	1	0.5				
1			0.5					
	A	B	C	D	E	F	G	H

Movilidad

Tabla 5.17: Penalización ante piezas enemigas

Enemigas	Elefante	Camello	Caballo	Perro	Gato	Conejo
Elefante	0	0	0	0	0	0
Camello	-5	0	0	0	0	0
Caballo	-4	-4	0	0	0	0
Perro	-3	-3	-3	0	0	0
Gato	-2	-2	-2	-2	0	0
Conejo	-1	-1	-1	-1	-1	0

Tabla 5.18: Conversión de la penalización de la bonificación ante piezas enemigas

Enemigas	Elefante	Camello	Caballo	Perro	Gato	Conejo
Elefante	0	5	4	3	2	1
Camello	0	0	4	3	2	1
Caballo	0	0	0	3	2	1
Perro	0	0	0	0	2	1
Gato	0	0	0	0	0	1
Conejo	0	0	0	0	0	0

Tabla 5.19: Bonificación ante piezas aliadas

Enemigas	Elefante	Camello	Caballo	Perro	Gato	Conejo
Elefante	0	0	0	0	0	0
Camello	5	0	0	0	0	0
Caballo	4	4	0	0	0	0
Perro	3	3	3	0	0	0
Gato	2	2	2	2	0	0
Conejo	1	1	1	1	1	0

C: Implementación

Interfaz de Usuario

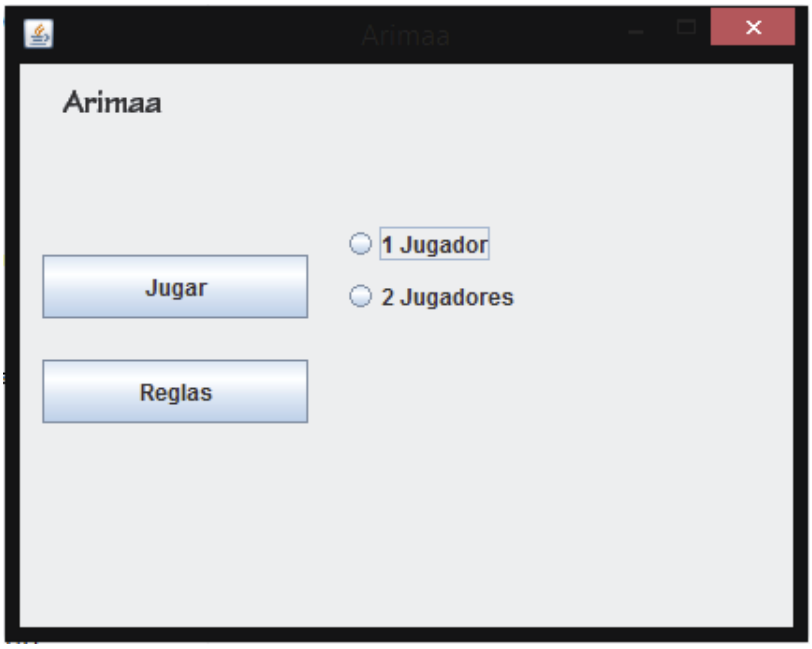


Figura 6.1: Ventana Principal

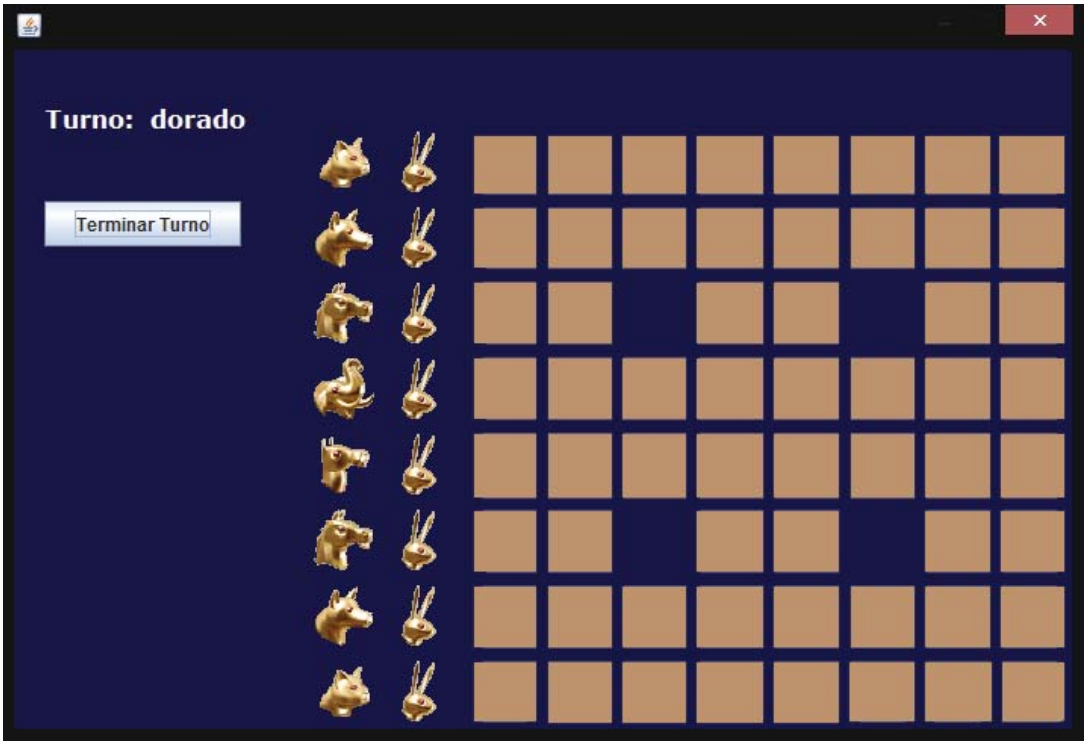


Figura 6.2: Ventana Juego

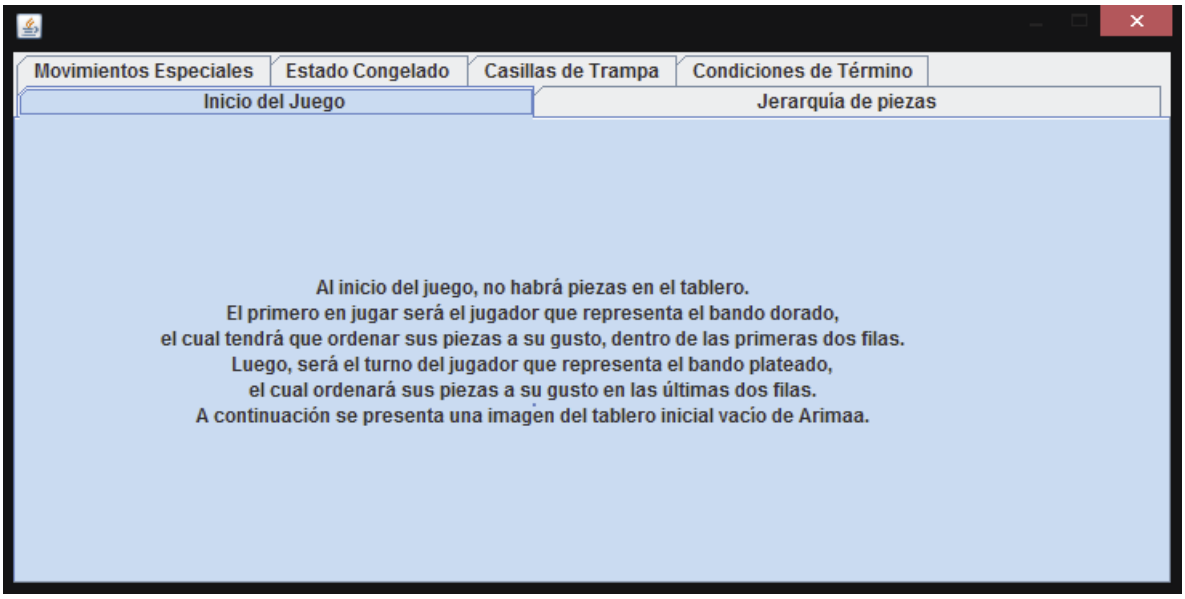


Figura 6.3: Ventana Reglas

Puzzles

Figura 6.4: Puzle 1

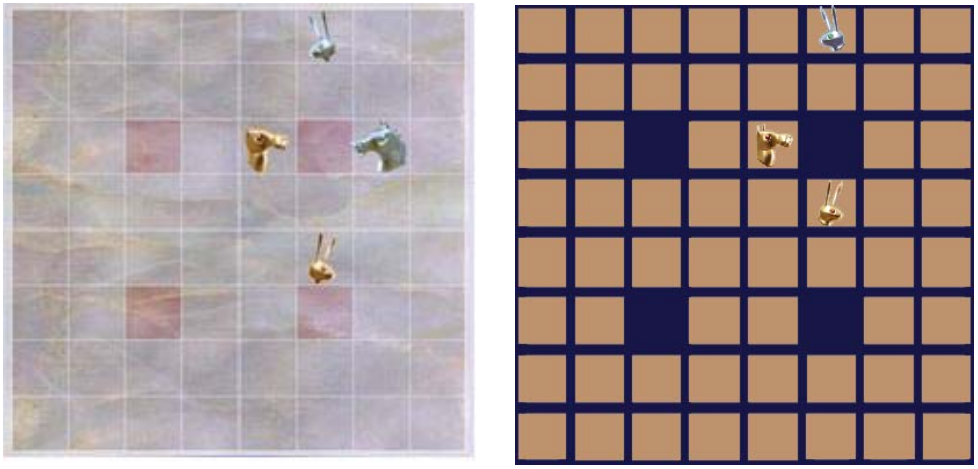


Figura 6.5: Puzle 2

Figura 6.6: Puzle 3

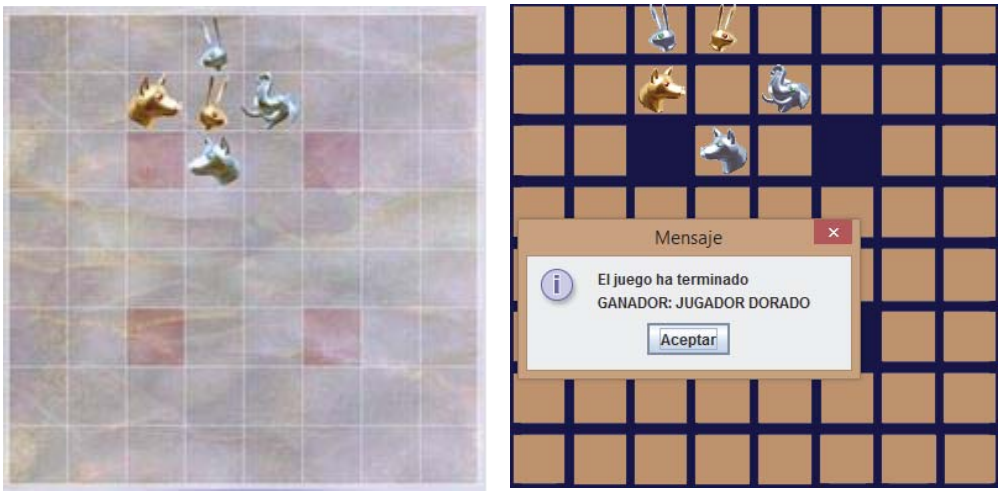


Figura 6.7: Puzle 4

Figura 6.8: Puzle 5

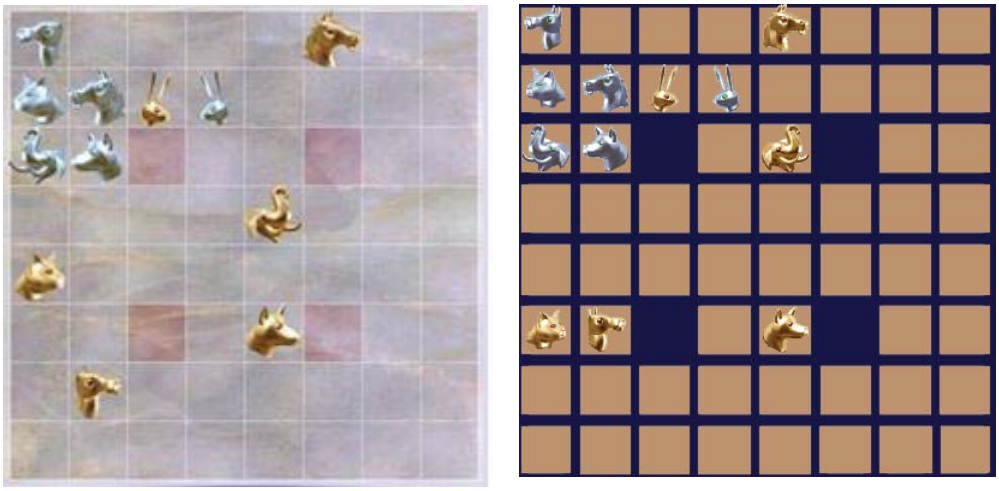


Figura 6.9: Puzle 6

Figura 6.10: Puzle 7

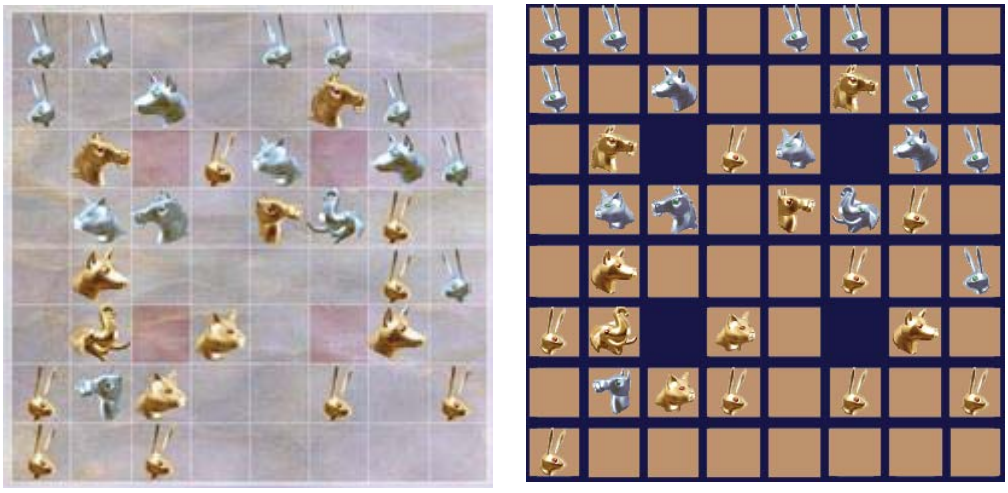


Figura 6.11: Puzle 8

D: Experimentos en otros juegos

El siguiente texto corresponde a una extracción de [12], correspondiente a las pruebas que se realizaron en distintos juegos.

Con el fin de probar la efectividad de la arquitectura de la Colonia de Hormigas, se han jugado partidas entre un jugador que usa el conocimiento desarrollado por las hormigas, ANT, y un jugador aleatorio, RAND D, el cual realiza movimientos al azar. Las selecciones se realizan con avidez con respecto al producto de la feromona y conveniencia.

Un total de 40 hormigas fueron creadas para cada juego, con los roles distribuidos igualmente entre las hormigas. El número total de búsquedas corriendo fue de 200. Los valores de los parámetros que dieron los mejores resultados, son los siguientes: el valor para α fue 0.6, el valor para β fue 0.8 y la probabilidad para consultar las rutas del oponente fue de 0.5. ϑ fue de 1.

Con el fin de tener un ambiente más controlado, se implementó un sistema de tipos GGP en nuestras máquinas locales. Este sistema es similar al implementado en Stanford, y utiliza el mismo protocolo de comunicación entre el Manager de juego y Jugadores. Las reglas fueron descargadas desde el sitio web de Stanford y fueron pasadas por el Game Manager a los jugadores. Los jugadores y manager fueron escritos en Java. Por inferencia, las reglas de juego se convirtieron al lenguaje Prolog. YProlog, un motor de inferencia estilo Prolog fue utilizado. Los juegos usados para las pruebas fueron: Tic-Tac-Toe, 5X5 Tic-Tac-Toe, Connect-4, Breakthrough y Checkers. Los resultados de las 100 partidas se resumen a continuación. El conocimiento recogido por las hormigas no se desarrolló durante el juego. Los roles para los juegos se distribuyeron uniformemente entre los jugadores.

Tabla 6.6: Resultados de Experimentos

Juego	Victorias ANT	Victorias RAND D	Total Empates
Tic-Tac-Toe (pequeño)	72	9	19
Tic-Tac-Toe (grande)	66	12	22
Connect-4	73	27	0
Breakthrough	74	26	0
Checkers	59	39	2

Los resultados muestran que las hormigas fueron capaces de generar con éxito las estrategias que permitieron a ANT ganar la mayoría de los juegos. Es importante señalar que para grandes juegos (tal como Checkers), no es práctico almacenar el conocimiento como una tabla de listas de estados de movimientos. Funciones de técnicas de aproximación para representar valores son más adecuadas, aunque a costa de precisión. Estos son discutidos brevemente en la siguiente sección.

Generación y formación del conocimiento para juegos que tienen largas secuencias es relativamente lento. En una competencia real, no podría ser posible generar una representación del conocimiento justo si no se le da una cantidad de tiempo suficiente. Sin embargo, el objetivo de estos experimentos era probar la eficacia del enfoque del Sistema de Colonia de Hormigas, sin tener en cuenta un escenario competitivo. El tiempo todavía se puede reducir mediante el uso eficiente de estructuras de datos y técnicas hashing, tales como Zobrist hashing [13], o la que se utiliza por [14] para la consulta de estado.