PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

EL DESAFÍO DE ARIMAA

Andrés Cerón Lillo

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

Julio 2010

Pontificia Universidad Católica de Valparaíso
Facultad de Ingeniería
Escuela de Ingeniería Informática

EL DESAFÍO DE ARIMAA

Andrés Cerón Lillo

Profesor Guía: **Silvana Roncagliolo de la Horra**

Profesor Co-referente: **José Miguel Rubio León**

Carrera: **Ingeniería Civil en Informática**

Julio 2010

Dedicatoria

Dedicado con mucho cariño a todas las personas que me han apoyado y creído en mí.

Como este trabajo trata de un juego, y el ajedrez además de servir como referencia es uno de mis juegos favoritos, voy a compartir una cita de un gran ajedrecista,

José Raúl Capablanca:

"El ajedrez es algo más que un juego; es una diversión intelectual que tiene algo de arte y mucho de ciencia. Es además, un medio de acercamiento social e intelectual..."

Andrés Cerón Lillo

Agradecimientos

Doy infinitas gracias:
A Dios, por el camino recorrido,
A mis padres, por su amor y apoyo,
Y a la vida por todo lo aprendido.

Resumen

En el presente trabajo de investigación se desarrolla un prototipo de Arimaa, que es un juego de tablero que comenzó a gestarse en 1997 cuando Deep Blue derrotó al campeón mundial de Ajedrez Gary Kasparov. De esta manera aparece Arimaa como un juego de reglas muy simples e intuitivas para los seres humanos, pero a la vez muy complejo para los computadores.

Para enfrentar esta problemática, en una primera instancia se analizó el juego, estudiando su historia, la manera de jugarlo, sus reglas oficiales, y las complicaciones que presenta el programarlo; en una segunda instancia se trazó el camino a seguir para conseguir este propósito, sin perder de vista las soluciones que se han planteado en el ajedrez, se definió como representar numéricamente los datos, se introdujo los conceptos del generador de movimiento, la función de búsqueda, Minimax con poda Alfa Beta, y la función de evaluación. Para finalmente proceder a la implementación del prototipo.

Abstract

The present research work aims to develop a prototype of Arimaa, a board game which began to take shape in 1997 when world chess champion Gary Kasparov was defeated by Deep Blue. Thus Arimaa appears as a game with simple rules and intuitive for humans, but simultaneously very complex for the computers.

To face this problem, at first instance the game was analyzed, studying its history, how to play it, its official rules, and the complications that presents to be programmed, in a second instance the way forward to achieve this purpose, without losing sight of the solutions that have been raised in chess, a numerical representation of the data was defined. Also, the concepts of the movement generator, search function, Minimax with pruned Alpha Beta, and the evaluation function were introduced to finally proceed with the implementation of the prototype.

Índice

Capítulo 1 1

1.1 Introducción 1

1.2 Definición de Objetivos 2

1.2.1 Objetivo General 2

1.2.2 Objetivos Específicos 2

1.3 Antecedentes del Proyecto 3

1.3.1 Los Computadores en el Ajedrez 3

1.3.2 El Desafío de Arimaa 4

1.3.3 Dificultad de Implementar un buen programa Arimaa 5

1.4 Estado del Arte 7

1.5 Metodología 8

Capítulo 2 9

2.1 Cómo jugar Arimaa 9

2.1.1 Reglas de encuentros oficiales 12

2.1.2 Notación para guardar partidas de Arimaa 13

2.1.3 Control de tiempo en partidas de Arimaa 16

Capítulo 3 18

3.1 Representación del Tablero 18

3.2 El generador de movimientos 19

3.3 Función de búsqueda: ¿Minimax? 20

3.3.1 Algoritmo Minimax 22

3.3.2 Mejorando Minimax: ¿Alfa Beta? 24

3.3.3 Alfa Beta 29

3.4 Estructura de Datos 30

3.5 Jugadas legales y pseudo legales 31

3.6 El efecto horizonte 32

3.7 Mejoras en la función de búsqueda 33

3.7.1 Profundización iterativa 33

3.7.2 Línea principal 33

3.7.3 Tablas de transposición 34

3.7.4 Jugada asesina 34

3.7.5 Ventana de aspiración 34

3.7.6 Factibilidad de la aplicación de las mejoras 34

3.8 Evaluación de posiciones *36*
3.8.1 Evaluación de material 38
3.8.2 Evaluación de la posición individual de las piezas 39
3.8.3 Control de Trampas 44
3.8.4 Movilidad 45
3.9 Función de Evaluación *47*
Capítulo 4 ..... **49**
4.1 Implementación *49*
4.1.1 Clases que representan la información 50
4.1.2 Representación de los Datos 51
4.1.3 Turno 52
4.1.4 Movimientos y Generador de movimientos 54
4.1.5 Trampas 56
4.1.6 Función de búsqueda 57
4.1.7 Diagrama de Clases 58
4.2 Testeo y evaluación del Prototipo *59*
4.2.1 Método de la Caja Blanca 59
4.2.2 Método de la Caja Negra 60
4.2.3 Pruebas 60
4.2.4 Configuración del prototipo y resultados 66
Capítulo 5 ..... **70**
5.1 Conclusiones *70*
Referencias **71**

Lista de Figuras

Figura 2.1: Tablero vacío de Arimaa 9
Figura 2.2: Posible configuración inicial de Arimaa 10
Figura 2.3: Posición típica de medio juego 11
Figura 2.4: Ejemplo de Notación de una posición 15
Figura 3.1: Matriz numérica de representación del tablero 18
Figura 3.2: Árbol generado por el proceso de búsqueda Minimax en el juego de tres en raya 22
Figura 3.3: Algoritmo Minimax 23
Figura 3.4: Ejemplo de un árbol de búsqueda completo usando Minimax 23
Figura 3.5: Primera parte 24
Figura 3.6: Segunda parte 25
Figura 3.7: Tercera parte 25
Figura 3.8: Cuarta parte 26
Figura 3.9: Quinta parte 27
Figura 3.10: Resultado de la poda 28
Figura 3.11: Algoritmo Alfa Beta 29
Figura 4.1: Pantalla del Programa Arimaa 49
Figura 4.2: Clase Tablero 50
Figura 4.3: Clase Casilla 51
Figura 4.4: Diagrama de Flujo de una jugada 53
Figura 4.5: Diagrama de flujo del generador de movimiento 55
Figura 4.6: Método aliado de la Clase Tablero 56
Figura 4.7: Diagrama de flujo de datos de Alfa Beta 57
Figura 4.8: Diagrama de clases del Prototipo Arimaa 58
Figura 4.9: Puzle 1 Gold “Trap the silver camel” 62
Figura 4.10: Puzle 2 Gold “Trap the silver horse” 63
Figura 4.11: Puzle 3 Gold “Trap any silver piece” 63
Figura 4.12: Puzle 4 Gold “Get the rabbit to goal” 63
Figura 4.13: Puzle 5 Gold “Get the rabbit to goal” 63
Figura 4.14: Puzle 6 Gold “Get the rabbit to goal” 64
Figura 4.15: Puzle 7 Gold “Gold to move and win in one” 64
Figura 4.16: Puzle 8 Gold “Gold to play and reach goal in one” 64

Lista de Tablas

Tabla 1.1: Complejidad de algunos juegos conocidos 5
Tabla 2.1: Las piezas de Arimaa [5].....13
Tabla 3.1: Valor material de las piezas.....38
Tabla 3.2: Valor de los Conejos 39
Tabla 3.3: Valores de la posición de un Conejo.....40
Tabla 3.4: Bonificación de los Conejos por piezas aliadas 41
Tabla 3.5: Cuadro de los valores de la posición de un Gato 41
Tabla 3.6: Cuadro de los valores de la posición de un Perro 42
Tabla 3.7: Cuadro de los valores de la posición de un Caballo.....42
Tabla 3.8: Cuadro de los valores de la posición de un Camello.....43
Tabla 3.9: Cuadro de los valores de la posición de un Elefante.....43
Tabla 3.10: Valores por custodia de trampas 44
Tabla 3.11: Multiplicadores del control de la trampa C3 45
Tabla 3.12: Penalización ante piezas enemigas.....46
Tabla 3.13: Conversión de la penalización de la bonificación ante piezas enemigas 46
Tabla 3.14: Bonificación ante piezas aliadas.....47
Tabla 4.1: Representación visual del objeto Tablero 50
Tabla 4.2: Valores que representan el estado de una casilla 51
Tabla 4.3: Posibles direcciones para pasos normales.....54
Tabla 4.4: Algunos datos de prueba del generador de movimiento 62
Tabla 4.5: Evaluación de los tiempos para los puzles (milisegundos) 65
Tabla 4.6: Cantidad de posibles jugadas 66
Tabla 4.7: Evaluación bajo distintos escenarios parte 1.....67
Tabla 4.8: Evaluación bajo distintos escenarios parte 2.....68

Glosario

Alfa Beta: Variación del algoritmo Minimax donde se desechan alternativas inferiores.

Arimaa: Juego de tablero, de reglas simples y fáciles de aprender.

Bitboard: Estructura de datos en bit muy utilizada en juegos de tablero.

Bot marwin: Programa de Arimaa que ganó, las preliminares del desafío 2010.

Botvinnik: Fue un ajedrecista soviético, campeón del mundo varias veces entre 1948 y 1963.

Caja Blanca: Tipo de pruebas que se realiza sobre las funciones internas de un módulo que están dirigidas a las funciones internas del mismo.

Caja Negra: es cuando un elemento es estudiado desde el punto de vista de las entradas que recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno.

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

ICGA: Asociación Internacional de Juegos de Computadores.

Información completa: se usa para describir un juego en el que todos los jugadores conocen el tipo del resto de jugadores, por ejemplo conocen las recompensas y espacios de estrategia de los demás jugadores.

Jugada Asesina: Es una técnica para mejorar la eficiencia del algoritmo Alfa Beta.

Minimax: Algoritmo para minimizar la pérdida máxima en juegos con adversario y con información perfecta.

Movimiento nulo: Es una técnica utilizada para aumentar la velocidad del algoritmo Alfa Beta.

Profundización iterativa: Es una estrategia de búsqueda por repetidas iteraciones en la que cada iteración alcanza un grado más de profundidad.

Capítulo 1

1.1 Introducción

Los juegos siempre han sido una de las maneras favoritas de los seres humanos para pasar el tiempo, y esto no es solamente porque sean entretenidos, sino también por el elemento competitivo. Este elemento competitivo es el que lleva a las personas a querer sobresalir en los juegos que practican, buscando llegar a la cima y ser los mejores.

En esta búsqueda por mejorar, los computadores aparecen como una herramienta muy útil, pero como es previsible, la consecuencia del uso de éstas al largo plazo, es que el computador será capaz de lograr un juego superior a cualquier ser humano, y el ejemplo más ilustrativo y prácticamente conocido por todos es el caso del ajedrez, donde la máquina Deep Blue derrotó a Gary Kasparov (1997).

Así como en el ajedrez, esta situación ha ocurrido en muchos juegos de estrategia, pero si se repara en un detalle, muchos de estos juegos han sido estudiados a fondo, ya que datan de mucho tiempo antes que los computadores, por lo que no se ha tenido ningún reparo en complicarle su manera de jugar. En este punto, ¿Qué sucede si se crea un juego especialmente pensado para que a un computador le sea difícil jugar bien, pero que a su vez tenga reglas tan simples que hasta un niño puede aprenderlo a jugar rápidamente?

Esta es la premisa de Arimaa, un juego que fue creado por Omar Syed un ingeniero en computación especializado en Inteligencia Artificial, quien se inspiró en la derrota de Gary Kasparov ante la Máquina Deep Blue para crear este juego, y quien además anunció un premio de al menos 10.000 US$, disponible anualmente hasta el año 2020, para el primer programa de computador que fuese capaz de vencer a un jugador humano de alto nivel en un match.

1.2 Definición de Objetivos

En este punto se debe definir el objetivo general y los objetivos específicos, para de esta manera definir las metas a alcanzar.

1.2.1 Objetivo General

"Generar un prototipo de programa Arimaa que sea capaz de jugar una partida de Arimaa".

1.2.2 Objetivos Específicos

- Estudiar, entender y analizar el juego de Arimaa
- Estudiar criterios utilizables en la función de evaluación e implementarlos
- Estudiar e implementar Minimax con Poda Alfa Beta
- Analizar alternativas para mejoras en la función de evaluación
- Evaluar el desempeño del programa con respecto al tiempo y al nivel de profundidad

1.3 Antecedentes del Proyecto

1.3.1 Los Computadores en el Ajedrez

La investigación sobre programas ajedrecistas se inició en 1950, cuando Claude E. Shannon publica el primer artículo en el cual hace notar la existencia de una solución perfecta en ajedrez y la dificultad práctica de encontrarla. Él describió dos estrategias basadas en la heurística de la función de evaluación, para analizar si la posición está a favor de un jugador o del otro. Los programas modernos de ajedrez siguen aún estos conceptos. En 1951 Alan Turing describe una simulación a mano de un programa de ajedrez, pero su juego es débil ya que pierde ante jugadores débiles.

En 1956 se documenta la primera ejecución de un programa ajedrecista cuyos experimentos corren en la máquina Univac Maniac I (11.000 operaciones por segundo). La máquina juega ajedrez en un tablero de 6x6 y con un set de piezas sin Alfiles demorándose 12 minutos en una búsqueda de 4 movidas de profundidad, al agregar los Alfiles le tomaba 3 horas la búsqueda de 4 movidas de profundidad. El equipo que programó Maniac fue liderado por Stan Ulam. Esta máquina sólo podía derrotar a jugadores débiles. Alex Bernstein escribió un programa de ajedrez para una máquina IBM 704 la cual podía realizar 42.000 instrucciones por segundo, esta máquina se demoraba 8 minutos en la búsqueda de 4 movidas de profundidad, y podría pasar como un jugador amateur.

En 1962 fue escrito el programa de ajedrez de Kotok-McCarthy, este fue el primer programa que podía jugar ajedrez regularmente de forma creíble. En 1967 el programa MacHack VI que corría en una máquina IBM 7090, era capaz de analizar 11.000 posiciones por segundo, se convirtió en el primer programa en derrotar a un buen jugador de ajedrez. Ese año MacHack VI participó en 4 torneos, logrando 3 victorias, 12 derrotas y 3 tablas.

En 1971 el programa de ajedrez Technology participa en seis torneos logrando 10 puntos de 26 posibles. Este programa fue el primero en ser escrito en un lenguaje de alto nivel. Corría en un PDP-10 (1 Mhz), y examinaba 120 posiciones por segundo.

En 1973 CHES 4.0 gana el torneo de programas de ajedrez, este analizaba entre 270 y 600 posiciones por segundo.

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, ya en 1983 éste buscaba entre 40.000 y 50.000 posiciones por segundo.

En 1977 Belle fue 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 Deep Thought, el antecesor de Deep Blue, fue creado por un grupo de graduados de la Universidad Carnegie-Mellon, esta máquina contenía 250 chips 2 procesadores, y era capaz de analizar 750.000 posiciones por segundos hasta con 10 movidas

de profundidad. En este año Deep Thought se convierte en la primera máquina en derrotar a un Gran Maestro en un Torneo.

En 1996 Deep Blue, la nueva máquina ajedrecista de IBM, que contaba con 32 procesadores en paralelo, 256 chips VLSI (Very Large Scale Integration), que buscaba entre 2 y 400 millones de movimientos por segundo, logra vencer al campeón del mundo Gary Kasparov en el primero de seis juegos, pero pierde el match.

Hasta que en 1997 Deep Blue logra derrotar a Gary Kasparov en un match a 6 partidas. Deep Blue tenía 30 procesadores IBM RS-600 SP acompañado de 480 chips, pudiendo evaluar 200 millones de movimientos por segundo. [1]

1.3.2 El Desafío de Arimaa

Actualmente el mejor jugador de Arimaa es un jugador humano, pero ¿cuánto tiempo seguirá siendo esto así? Cada año los avances de hardware aumentan la potencia de cálculo, lo que acorta la brecha entre el cerebro humano y los computadores. Sin embargo aún se piensa que en Arimaa seguirán siendo los humanos los que sigan dominando, y es por esto que se ofrece un premio de al menos 10.000 US$ hasta el año 2020 para la primera persona, compañía o organización que desarrolle un programa que pueda vencer a tres jugadores humanos seleccionados en un match oficial de Arimaa. El desafío consiste en jugar al menos tres partidas con cada uno de los jugadores seleccionados, este programa debe ser capaz de correr en un computar común, sin que se necesite ocupar hardware especializado adicional.

En el desafío 2010, el premio ascendió a 16.250 US$, y el match del desafío se programó entre el 11 y el 25 de Abril. Como resultado de esto el "bot_marwin" que ganó en los preliminares se enfrentó con los tres humanos designados: Patrick Dudek, Greg Magne y Daniel Scott, con Omar Syed como respaldo.

Las partidas se jugaron con un tiempo de control de 2/2/100/10/8[1], y el hardware permitido corresponde a un computador normal, que puede ser adquirido por 1000 US$, resultando un marcador de 6-3 a favor de los humanos designados.

Un aspecto importante a destacar es que este desafío de Arimaa es un incentivo para ayudar a promover la investigación en campos de la Inteligencia Artificial. Por lo que si un programa gana el desafío, este debe ser descrito en un paper y enviado a la revista técnica del International Computer Games Association (ICGA) describiendo la investigación y los resultados, todo esto previo a reclamar el premio del desafío.

[1] La interpretación de esta notación se explica en la sección "control de tiempo en partidas Arimaa"

1.3.3 Dificultad de Implementar un buen programa Arimaa

Hay varios aspectos de Arimaa que hacen que, para un programa de computador, sea difícil ganarle a un buen jugador humano. Los mejores programas de ajedrez se basan en una búsqueda por fuerza bruta, emparejada con la evaluación de posiciones estáticas, dominada por consideraciones de material. Los programas de ajedrez examinan muchas posibles jugadas, pero en comparación a los jugadores humanos son malos para distinguir quién va ganando, a menos que uno de los dos bandos tenga menos piezas.

Cuando se quiere aplicar la fuerza bruta en Arimaa, la profundidad de la búsqueda se ve limitada por la gran cantidad de diferentes jugadas que puede tener un jugador sólo en un turno, si en ajedrez un computador puede adelantarse 16 movimientos, puede adelantarse en 8 turnos de cada jugador, en Arimaa sólo se podría adelantar 2 turnos por cada jugador.

Como se puede apreciar en la ***tabla 1.1***, una de las dificultades para que un computador lo juegue bien radica en su gran factor de ramificación, lo que se puede ver con la complejidad del estado-espacio de un juego, que es el número de posiciones legales del juego, a las que se pueden acceder desde la posición inicial y con la complejidad del árbol del juego que es el número de nodos hoja del árbol de decisión completo en anchura que establece un valor en la posición inicial.

Tabla 1.1: Complejidad de algunos juegos conocidos[2]

Juego	Tamaño del tablero (casillas)	Complejidad estado-espacio (como log en base 10)	Complejidad del árbol del juego (como log en base 10)	Longitud media del juego (número de turnos)
Tres en raya	9	3	5	9
Ajedrez	64	50	123	80
Arimaa	64	43	296	70
Go	361	171	360	150

Para un programa de ajedrez, la profundidad de la búsqueda por fuerza bruta se ve casi doblada por la utilización de la técnica Alfa Beta, que permite que el programa concluya que movimiento es mejor que otro sin tener que analizar cada posible continuación de los movimientos más débiles. En esta técnica los movimientos de jaque y captura son cruciales para el proceso de eliminación, porque a menudo son mucho mejores que otros movimientos. En Arimaa las mejoras que provee Alfa Beta son mínimas, ya que la captura de piezas es menos frecuente, y en juegos nivelados que se han jugado sólo el 3% de los pasos resultan en captura, en cambio en el ajedrez son alrededor del 19%.

[2] **Esta tabla es un resumen de http://es.wikipedia.org/wiki/Complejidad_en_los_juegos**

En la mayoría de las posiciones de Arimaa, y principalmente en el principio, un jugador competente puede evitar perder piezas, a diferencia del ajedrez, en Arimaa es más fácil retrasar la captura. En Arimaa, la pelea inicialmente es más posicional y gira en torno a que las capturas sean inevitables en algún movimiento futuro. Por lo que es importante juzgar correctamente quien está ganando terreno en alguna forma no material. Por ende la fuerza de los programas de computador que analizan millones de posiciones no es tan significativa en comparación a su debilidad por juzgar la posición, independiente de quien tenga más piezas.

Otra debilidad de los programas de Arimaa es la configuración de la posición inicial. En el ajedrez por ejemplo la posición inicial es una sola, y por lo tanto se puede tener una lista compilada de las posibles respuestas a jugadas típicas, un programa de ajedrez puede hacer una docena o más de excelentes jugadas antes de comenzar a usar la fuerza bruta. En Arimaa en cambio hay muchas configuraciones, y muchas posibles primeras jugadas, lo que previene que un programa de Arimaa utilice algún tipo de librería de apertura.

1.4 Estado del Arte

Como se ha visto previamente, Arimaa es un juego relativamente nuevo que data del 2002, la fuente de información acerca del juego es principalmente su página oficial[3], en cuanto a la bibliografía solamente se sabe que Karl Juhnke quien fue Campeón Mundial de Arimaa en dos ocasiones, está escribiendo un libro titulado "*Beginning Arimaa: Chess Reborn beyon computer comprehension*". También hay artículos que hacen alusión al juego, en sus orígenes, en el desafío hacia los desarrolladores y además resaltan la premisa de Arimaa de ser un juego con reglas fáciles para los humanos, pero a la vez difícil para que lo jueguen bien los computadores.

En un aspecto más específico a lo que es el juego existen algunos trabajos de tesis, que tratan principalmente cómo analizar e implementar un programa de Arimaa pero no cuentan con un respaldo oficial serio. Respecto a estos trabajos se comenta la utilización de heurísticas y técnicas de la Inteligencia Artificial como búsqueda Alfa Beta, movimiento nulo, profundización iterativa, jugada asesina y Minimax para ayudarse en la creación del árbol de jugadas y la selección de las mismas, en estos trabajos también se establecen criterios de evaluación de posiciones, tomando en cuenta factores como la cantidad piezas, la ubicación de las mismas, su cercanía a los cuadros de trampas o la posibilidad de victoria.

Ahora si bien es cierto que se mencionan muchas técnicas y heurísticas, éstas no se estudian en profundidad, y sólo se hace una breve explicación de las mismas lo que no ayuda mucho al momento de querer entender y menos aún aplicar, en cuanto a los criterios de evaluación de las posiciones, hay distintas valoraciones dependiendo de los distintos trabajos. Como se puede apreciar no hay claridad en el tema, esto debido a la dificultad de poder medir efectivamente que posición es mejor que otra.

La otra gama de de trabajos se avoca más a la implementación, específicamente a la utilización de bitboard, pero sólo hace referencia a los elementos básicos, principalmente la representación del tablero, las piezas y sus movimientos, mostrando sólo fragmentos de código que por sí solos no son de mucha utilidad práctica.

Como era de esperarse, no hay mucha información importante respecto al tema, lo cual puede justificarse por un lado, por el desafío, ya que pese a haber muchos desarrolladores trabajando en sus proyectos de programa Arimaa, el premio de los 10.000 US$ (que actualmente va en casi 17.000 US$) aún está esperando un ganador, por lo cual pese a la licencia pública de Arimaa no comparten todos sus avances en la materia, por otro lado, esto también es debido a que Arimaa no es un juego popular, muy pocas personas tienen conocimiento de la existencia del mismo.

[3] Página Oficial de Arimaa http://arimaa.com

1.5 Metodología

En general este proyecto consiste en el estudio, análisis e implementación del juego de tablero Arimaa, como se ha visto, este es un juego que está pensado especialmente para que los programas de computadores no sean buenos jugándolo, la investigación se dividió en cuatro partes importantes, una primera parte donde se trataron los temas introductorios, y los aspectos básicos del juego. La segunda etapa consiste en un estudio relacionado a la función de evaluación, donde se buscan los criterios que se deben considerar, y se establece qué grado de importancia darles a cada uno de ellos. En una tercera etapa la investigación se centra en las técnicas y heurísticas, que permiten generar la gama de jugadas posibles en cada turno y la selección de los buenos movimientos, en este escenario se estudia el comportamiento del algoritmo Minimax, para poder así entender de mejor manera cómo funciona la búsqueda Alfa Beta. Para finalizar en una cuarta etapa donde se utilizan los conocimientos adquiridos anteriormente, para implementarlos en el prototipo de programa Arimaa.

Capítulo 2

2.1 Cómo jugar Arimaa

Arimaa es un juego de 2 jugadores, que puede ser jugado usando el mismo set de piezas que se utiliza en el ajedrez. En Arimaa existen dos bandos, el dorado (Gold) y el plateado (Silver). Para hacer más fácil su comprensión para las personas que no están familiarizadas con el ajedrez, las piezas son sustituidas por animales. Si se hace la conversión, el Rey es un Elefante, la Dama un Camello, las Torres son Caballos, los Alfiles Perros, los Caballos Gatos y los Peones Conejos.

La idea del juego es ser el primero en llevar un Conejo al otro lado del tablero. El juego comienza con un tablero vacío, primero el jugador dorado pone sus piezas a su gusto en las primeras dos filas, luego el jugador plateado hace lo mismo en las últimas dos filas, tal como se puede apreciar en la ***figura 2.1*** con el tablero vacio de Arimaa.

Figura 2.1: Tablero vacío de Arimaa

Una vez que se han situado las piezas de ambos jugadores como por ejemplo en la ***figura 2.2,*** el jugador dorado es quien juega primero, todas las piezas se mueven de la misma manera, un paso adelante, atrás, izquierda, o derecha. Con la salvedad que los Conejos no pueden retroceder por si solos, es decir sólo se mueven un paso hacia adelante, hacia la izquierda o hacia la derecha.

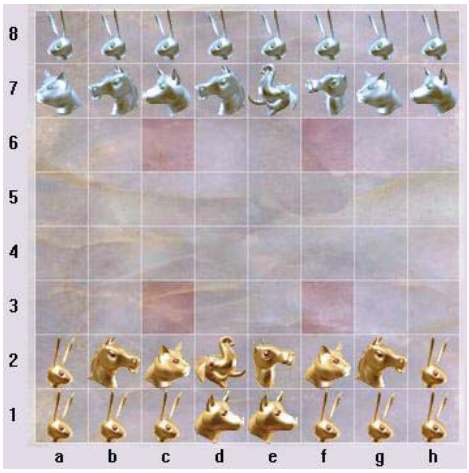

Figura 2.2: Posible configuración inicial de Arimaa

En un turno cada jugador puede dar hasta 4 pasos, con la obligación de dar al menos uno, un movimiento consiste en mover una pieza en un espacio en las direcciones anteriormente indicadas, para esto se necesita un paso. También hay dos movimientos especiales, en que se mueven piezas enemigas, como lo es el empujar, que consiste en mover una pieza enemiga y después ubicarse en su lugar, y tirar, que consiste en mover una pieza y luego arrastrar la pieza enemiga al lugar donde se encontraba la pieza, estos movimientos necesitan dos pasos.

Ahora las jugadas de empujar, y tirar, sólo se pueden aplicar a piezas enemigas más débiles, por lo que se hace necesario definir el orden de fuerza o jerarquía de las piezas. Ordenadas de mayor a menor el orden es: Elefante, Camello, Caballo, Perro, Gato y Conejo.

Las piezas más fuertes pueden congelar a las piezas enemigas más débiles, por ejemplo si un Caballo plateado está al lado del Camello dorado, el Caballo no se podrá mover, a menos que tenga una pieza aliada al lado. Al fijarse en la ***figura 2.3***, si fuera el turno del jugador plateado, éste tiene dos piezas que están al lado de una pieza dorada más fuerte, el Camello plateado en E6 está al lado del Elefante dorado ubicado en D6, y el Perro plateado en C4 está a un lado del Caballo dorado en D5. En el caso del Camello E6, éste no se puede mover, ya que no tiene ninguna pieza aliada en E5, E7 o F6, es decir, el Camello está congelado. En el caso del Perro la situación es diferente, ya que tiene una pieza aliada al lado, el Gato plateado en C5, está al lado del Perro plateado en C6 por lo que éste puede moverse independientemente a que tenga un Caballo dorado en D4.

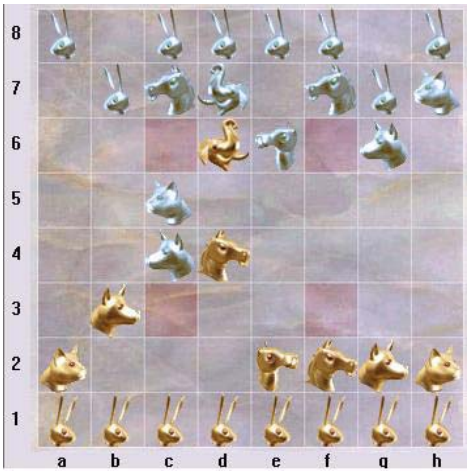

Figura 2.3: Posición típica de medio juego

En este juego hay cuatro casillas de trampas, si se considera las filas con números del 1 al 8 (desde abajo hacia arriba) y las columnas con letras de la A hasta la H (de izquierda a derecha), las casilla de trampa serian C3, C6, F3 y F6.

Cuando una pieza se mueve, o es movida a un cuadro de trampa, la pieza es removida del tablero, a menos que posea una pieza aliada al lado. Por ejemplo si en la ***figura 2.3*** le toca mover al jugador dorado, y éste en su turno empuja con su Caballo D4 el Perro plateado de C4 a C3, después obligatoriamente se mueve a C4 ocupando dos pasos, luego empuja el Gato plateado de C5 a C6, terminando su turno al mover obligatoriamente su Caballo a C5, se presenta el caso de que se tiene dos piezas que han sido empujadas a los cuadros de trampa. En el caso del Perro plateado que se encuentra en la casilla C3, éste se remueve del juego ya que no posee ninguna pieza aliada en B3, C2, C4 o D3, en el caso del Gato plateado que se empujó a C6, éste posee una pieza aliada en la casilla C7, por lo que en este caso la pieza permanece en el tablero.

Hay algunas situaciones especiales que se dan en el juego:

- Si un jugador pierde todos sus Conejos, ese jugador pierde el juego. Si ambos jugadores pierden su último Conejo en el mismo turno entonces se declara ganador al jugador que realiza la jugada
- Si un jugador no puede realizar ningún movimiento éste pierde el juego
- Si una posición se repite tres veces, se declarará perdedor al jugador que causa que se provoque la tercera repetición
- Si un jugador empuja o tira un Conejo enemigo a la meta, si el Conejo permanece en la meta al final del turno, el jugador pierde el juego
- Si al final de un turno los Conejos de ambos jugadores alcanzan la meta, entonces se declara vencedor al jugador que realizó la jugada

2.1.1 Reglas de encuentros oficiales

Existen algunas reglas que se aplican para las partidas oficiales por ranking, en torneos, desafíos o instancias similares.

- Las partidas deben ser guardadas, utilizando la notación para guardar partidas de Arimaa
- Se deben utilizar controles de tiempo
- El control de tiempo debe ser escogido para que la partida al menos alcance 80 movimientos
- Si un juego se detiene con motivo de haber alcanzado el límite de tiempo o jugadas, entonces el jugador con más piezas, o el que recientemente haya tenido más piezas, gana la partida, en el caso de que no se haya efectuado ninguna captura de piezas, entonces el jugador que mueve segundo gana la partida
- Un jugador en cualquier momento puede rendirse y terminar el encuentro, pero lo deseado es que la partida termine con un final natural

2.1.2 Notación para guardar partidas de Arimaa

Como se sabe el primer jugador en realizar movimientos es el dorado, y el segundo es el plateado. Para representar las piezas se utilizan letras mayúsculas para el jugador dorado, y minúsculas para el jugador plateado como aparece en la ***tabla 2.1***, cada cuadro del tablero se representa con una letra y un número como se denota anteriormente en la ***figura 2.1***, los movimientos de cada jugador se anotan en líneas diferentes, cada línea comienza con el número de la jugada y el bando que la realiza, por ejemplo 3g, significa que es la jugada 3 del jugador dorado, y que la siguiente línea empezará con 3s, que representa la tercera jugada del jugador plateado. La posición inicial de las piezas se registra indicando la pieza y el cuadro que ocupa, por ejemplo Da2, significa que el Perro dorado está situado en el cuadro a2. El movimiento de las piezas se representa indicando la pieza, el cuadro que ocupa junto con la dirección en la que se mueve, las direcciones son n (norte), s (sur), e (este) y w (oeste). Por ejemplo Ea2n, quiere decir que el Elefante dorado se mueve de a2 hacia a3. En el caso de que una pieza está en una trampa y es removida entonces se registra con la letra x, cuando un jugador se rinde, se coloca "resigns" en reemplazo de su jugada, en caso de que pierda porque un conejo llegó a la meta se escribe "lost", los movimientos que son saltados se dejan en blanco. Cuando se vuelve atrás en una jugada, se escribe en la jugada "takeback" y la siguiente jugada se cuenta como si fuera la anterior. El encabezado de la partida se puede escribir como en el ejemplo, indicando información como: el evento, el lugar, la fecha, los colores, y el resultado.

Tabla 2.1: Las piezas de Arimaa [5]

Nombre	Figura	Notación Dorado (g)	Notación Plateado (s)	Número	Fuerza
Elefante		E	E	1	Más Fuerte
Camello		M	M	1	2nd
Caballo		H	H	2	3rd
Perro		D	D	2	4th
Gato		C	C	2	5th
Conejo		R	R	8	Más Débil

2.1.2.1 Ejemplo de Notación de una partida de Arimaa

Event: Casual Game

Site: Cleveland, OH USA

Date: 1999.01.15

Round:?

Gold: Aamir Syed

Silver: Omar Syed

Result: 1-0

1g Ra2 Rb2 Mc2 Dd2 ...

1s ra7 rb7 rc7 rd7 ...

2g Ra2n Ra3e Rb3n Rb4e

2s ra7s ra6s ra5e rb5e

3g Dd2n Dd3n Mc2e Rc4s Rc3x

3s rc7s rc5e rc6x rd5e re5s

4g takeback

3s takeback

3g Rb2n Rb3n Rb4n

3s...

...

16s resigns

2.1.2.2 Ejemplo de cómo se anota un comentario

También se pueden realizar comentarios, los que pueden ocupar varias líneas, para esto se deben anotar entre la cadena de caracteres -=+=-. En el comentario, se puede agregar la jugada en que ocurre y también se le puede poner un nombre.

Chat: -=+=-

2s Gold: hi, how are u

2s Silver: fine, thanks

-=+=-

2.1.2.3 Ejemplo de Notación de una posición

En Arimaa también se pueden registrar posiciones puntuales, para lo cual se siguen los mismos principios anteriores, las letras mayúsculas para las piezas doradas y las minúsculas para las plateadas, como se puede apreciar en la ***figura 2.4***, existe una primera línea, la cual indica el turno del jugador al que le corresponde mover, en el caso del ejemplo es la movida número 7 del jugador dorado, generalmente se asume que el jugador no ha realizado ningún movimiento, pero en caso de haber realizado se anotan en la primera línea. Como se puede apreciar en el ejemplo la posición se grafica el tablero, y se anotan las piezas en sus respectivas posiciones, para el caso de los cuadros de trampa, si estos no tienen ninguna pieza que los ocupe, opcionalmente se pueden referenciar con la letra x.

```
    7g Da1n Cc1n
  +-----------------+
8 |   r   r r   r   |
7 | m   h     e   c |
6 |   r x r r x r   |
5 | h   d     c   d |
4 | E   H         M |
3 |   R x R R H R   |
2 | D   C     C   D |
1 |   R   R R   R   |
  +-----------------+
    a b c d e f g h
```

Figura 2.4: Ejemplo de Notación de una posición

2.1.3 Control de tiempo en partidas de Arimaa

Los controles de tiempos utilizados en Arimaa se definen por:

M/R/P/L/G/T

Donde:

M es el número de minutos: segundos por movimiento.

R es el número de minutos: segundos en reserva.

P es el porcentaje de tiempo no utilizado que se agrega a la reserva.

L es el número de minutos: segundos para el límite de la reserva.

G es el número de horas: minutos después de las cuales el juego es detenido y se determina al ganador. G también puede ser determinado como el número máximo de jugadas.

T es el número de minutos: segundos en los que un jugador debe realizar una jugada.

Cada jugador tiene un tiempo por movida (M), si se termina este tiempo, entonces comienza a correr el tiempo de reserva (R), si se termina el tiempo de reserva y el jugador no ha realizado su jugada, entonces el jugador pierde automáticamente la partida. Si un jugador termina su jugada en menos tiempo del que tiene para un movimiento, entonces un porcentaje (P) de ese tiempo se agrega a su reserva. Existe un límite de tiempo (L) que limita el tiempo máximo que puede haber en la reserva, si el tiempo de reserva excede el límite fijado por (L), entonces no se seguirá agregando tiempo a la reserva. Por razones prácticas, se determina un tiempo total de juego (G), si la partida no termina durante este periodo de tiempo, entonces se detiene la partida y se designa quien gano, este parámetro es opcional y si no se configura entonces quiere decir que la partida no tiene límite de tiempo. En las partidas en que el tiempo por movimiento (M) es menor a un minuto, se agrega un minuto para poner las piezas en la configuración inicial.

2.1.3.1 Algunos ejemplos de configuración de tiempos

0/5: quiere decir que cada jugador tiene 0 minutos por jugada y dispone de una reserva de 5 minutos

0:12/5: quiere decir que cada jugador tiene 12 segundos por jugada y dispone de una reserva de 5 minutos

3/0: quiere decir que cada jugador tiene 3 minutos por jugada, y la totalidad del tiempo que no utiliza se agrega a la reserva.

0:30/5/100/3:quiere decir que cada jugador tiene 30 segundos por jugada, con una reserva de 5 minutos, y que el 100% del tiempo que le queda se agrega a la reserva cuando ésta no es mayor que 3 minutos.

4/4/100/4/6:quiere decir que cada jugador tiene 4 minutos por jugada, con una reserva de 4 minutos, y que el 100% del tiempo que le queda se agrega a la reserva cuando ésta no es mayor a 4 minutos, y además al cabo de 6 horas si aún no hay ganador se dará por terminada la partida y se determinara quien gano.[4]

[4] Reglas oficiales Arimaa, obtenidas desde su página web oficial http://arimaa.com

Capítulo 3

3.1 Representación del Tablero

En el proceso de construcción de un programa de ajedrez la primera dificultad con que se encuentra es la necesidad de hacer llegar al computador la información que se refiere al juego. Cuando alguien aprende a jugar al ajedrez lo primero que tiene que saber es la forma de las piezas y cómo es el campo de batalla (el tablero). El primer paso en el aprendizaje del juego es conocer las piezas, distinguirlas por su forma y tamaño. Además es necesario saber que hay dos bandos y que se reconocen las piezas de uno y otro jugador por su color. Para un ser humano esta tarea es tan fácil que no se repara en ello, se hace de forma automática, sólo hay que mirar el tablero y la disposición de las piezas. Pero el computador no tiene ojos, ni forma de percibir el mundo real. Es necesario traducir la realidad física de las piezas y el tablero a una forma que pueda entender el programa. Y si hay algo que los computadores hacen bien es tratar con números. En Arimaa al igual que en el ajedrez, se consigue representar toda la información de una partida usando sólo números, lo que es un formato adecuado para ser tratado por el computador.

Para esta representación se construye una matriz de 8x8 (64 casillas), cada una de ellas tendrá un valor numérico. Si la casilla está vacía se tendrá un 0. Se utilizará un 1 para el Conejo, 2 para el Gato, 3 Perro, 4 Caballo, 5 para el Camello y 6 para el Elefante. Esto para las piezas doradas, para las plateadas se utiliza el mismo número pero negativo, tal como se puede apreciar en la ***figura 3.1*** donde se representa una posición inicial.

-4	-2	-3	-5	-6	-3	-2	-4
-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
4	2	3	5	6	3	2	4

Figura 3.1: Matriz numérica de representación del tablero

En esta representación cada casilla toma el valor de la pieza que la ocupa, y para los movimientos de éstas se intercambia el valor de las casillas. Una mejora que se utiliza frecuentemente en el ajedrez es el uso de una matriz más grande como una de 12x12 por

ejemplo, lo que le permite a los programas ser más eficientes en la generación de movimientos, en el caso de Arimaa esta mejora se puede realizar con una matriz de 10x10.

3.2 El generador de movimientos

Si se quiere que el programa juegue, el programa debe ser capaz de generar una lista con todos los posibles movimientos candidatos a ser jugados, más adelante se verá como decidir cuál es el mejor movimiento que se tiene que jugar, pero de momento se tiene que tener claro que inicialmente el programa generará dicha lista.

El programa debe ser capaz de realizar movimientos de tirar y empujar, además de la captura de piezas. Y por supuesto no debe hacer movimientos ilegales, por ejemplo mover una pieza congelada, aunque por motivos de rapidez se puede incluir los movimientos ilegales y más tarde eliminarlos.

Para generar la lista se tiene que pensar en el tablero y sus 64 casillas, por ejemplo cada casilla es asignada a un espacio en la matriz de 10x10[5], empezando desde A1 M ([1],[1]) hasta H8 M ([8],[8]). Para mover una pieza se debe variar uno de los índices, una pieza situada en la casilla M([x],[y]) se puede mover a M([x-1],[y]), M([x+1],[y]), M([x],[y-1]) o ([x],[y+1]). Todas las piezas tienen el mismo patrón de movimientos por lo que sólo se debe tener cuidado con los Conejos que tienen la limitante de que no pueden retroceder por si solos.

Aplicando las validaciones de los movimientos permitidos, entonces ya se es capaz de ir generando la lista de movimientos posibles.

[5] Aunque el tablero sólo tiene 64 casillas se utiliza una matriz de 10x10 para hacer más fácil la generación de los movimientos en las orillas del tablero.

3.3 Función de búsqueda: ¿Minimax?

Ahora cuando el programa ya tiene su motor, un generador de movimientos capaz de calcular todos los movimientos posibles en una posición determinada. La siguiente tarea es conseguir que el programa utilice adecuadamente su generador de movimientos. A esta parte se le suele llamar función de búsqueda. En esencia se trata de un generador de posiciones: que es llamado repetidamente, donde se llega a unas posiciones finales a las que se les asigna un valor determinado dependiendo de lo buenas o malas que sean para el programa. Esta parte de los programas de ajedrez no suele ser muy original, casi todos usan un algoritmo clásico introducido por Von Newman que recibe el nombre de Minimax. La misión de este algoritmo es construir el árbol de variantes y elegir entre todas las ramas la más beneficiosa para el programa.

En el caso del ajedrez cuando es el turno de juego del programa, que lleva las blancas, lo primero que se hace es llamar al generador para obtener la lista de todos los movimientos posibles en la posición. Como resultado de esta primera llamada al generador se tendrá una lista que contendrá las jugadas a3, a4, b3, b4, Cf3, etc. hasta 20 que son las posibles jugadas que tienen a su disposición las blancas desde la posición de salida. A continuación se procesa esta lista. Se trata de realizar cada una de estas jugadas en el tablero. Cada vez que se reproduce una jugada de la lista en el tablero se obtiene una nueva posición. Estas posiciones resultantes se guardan en otra lista y se vuelve una jugada atrás en el tablero virtual antes de hacer la siguiente. Al acabar la lista de movimientos se tiene como resultado tantas posiciones como jugadas posibles se tenían para realizar, posiciones que se ha tenido la precaución de guardar en memoria para seguir desarrollando el árbol. Tras esta primera pasada se tiene como resultado 20 posiciones. ¿Qué se hace con ellas? Pues lo mismo que se realizó con la posición inicial que sirve de punto de partida en este ejemplo, cada una de ellas se le pasa al generador de movimientos para obtener la lista de movimientos posibles. Y con esta nueva lista de movimientos posibles se repite el proceso anterior: se realiza sobre el tablero y se llama al generador de movimientos.

Se debe recordar que cuando se habla de realizar y volver jugadas atrás lo que se hace realmente es cambiar el valor de las casillas de nuestra matriz numérica. Por ejemplo, donde había un 1 que significa un peón se pone un 0 para indicar que la casilla ha quedado vacía. Para volver la jugada atrás se vuelve a poner un 1 en la casilla original. Estas operaciones deben estar optimizadas para ser muy rápidas, porque se van a repetir millones de veces durante el proceso de búsqueda. ¿Y cuándo termina este proceso? Para desarrollar esta parte del programa se utiliza una técnica de programación llamada recursividad, que básicamente significa que una parte del código se llama a sí mismo en un proceso sin fin. Resulta necesario por tanto cortar la búsqueda en algún punto, normalmente cuando se agota el tiempo asignado para esa búsqueda, pero también es posible que el programa siga buscando hasta una profundidad determinada, cancelando el proceso cuando ésta se alcance. De esta forma se construye el árbol de variantes, pero aún no se tiene ni idea de cuál es la mejor jugada. Para esto se tiene que utilizar la función de evaluación. Aunque luego se hablará de esta parte, de momento bastará la idea de que es una parte del código que asigna un valor numérico a una posición. El caso más sencillo posible sería una función de evaluación que simplemente sumara el valor material de las piezas y devuelve un valor resultado del valor material de un

bando menos el valor material del bando contrario. A la función de evaluación se le pasan únicamente las posiciones terminales, y esto es muy importante: sólo las posiciones finales del árbol se evalúan. Los nodos anteriores adquieren valor según sus nodos hijos. El truco para que todo esto funcione es que cada nodo toma el valor del máximo de sus nodos hijos si es el turno de juego del programa, o del mínimo en caso de que sea el turno de juego del rival. De este hecho toma su nombre el algoritmo: maximizar y minimizar alternativamente en cada nivel, según cambia el turno de juego. Repitiendo el proceso del Minimax el número suficiente de veces, en teoría, es posible construir el árbol completo de variantes del juego del ajedrez y jugar la partida perfecta. Lamentablemente la vida no es tan bonita y en la práctica por supuesto esto no es posible debido al inimaginable número de posiciones que habría que calcular.

En los primeros niveles hay pocas ramas, pero aumentan rápidamente con cada nivel. En la posición inicial de una partida de ajedrez las blancas pueden elegir entre 20 posibles movimientos. Puesto que existen 20 movimientos legales, al realizarlos sobre el tablero se tendrá 20 posiciones en el segundo nivel. Si se supone que para cada una de estas posiciones resultantes del primer nivel también hay 20 movimientos posibles. Tras llamar al generador de movimientos y aplicar la lista de jugadas en el tablero se obtiene 20 x 20 = 400 posiciones en el tercer nivel. En el tercer nivel ya se parte con 400 posiciones, si se sigue suponiendo que en cada una de ellas hay 20 jugadas para escoger se tiene por tanto 400 x 20 = 8000 posiciones en el siguiente paso. Si se sigue realizando las operaciones suponiendo siempre 20 jugadas legales por cada posición se encuentra con un árbol de 1.280 millones de posiciones en el nivel 6 y 25.600 millones en el nivel 7. Es fácil comprobar que el crecimiento del árbol de variantes es explosivo. En este ejemplo se ha llegado hasta el séptimo nivel, o lo que es lo mismo, 7 movidas jugadas o 3 turnos completos y un movimiento. Para ver una línea de tres turnos ya se está obligado a calcular 25.600 millones de posiciones. En la práctica esto es peor todavía, se ha supuesto 20 jugadas posibles en cada posición, una cifra más realista sería entre 30 y 35 movimientos posibles para una posición de medio juego. El hecho es que el conjunto de posiciones distintas que pueden darse en una partida de ajedrez es tan enorme que continúa siendo inabarcable para los computadores actuales. Es por eso que el algoritmo Minimax, que en teoría es capaz de encontrar la mejor jugada posible en los juegos llamados de información completa y suma cero, no funciona en la práctica con el ajedrez. La clave es el factor de ramificación, la rapidez de crecimiento del árbol de jugadas en cualquier juego está relacionada con el "factor de ramificación" que se puede definir como el número medio de jugadas legales que tiene un jugador a su disposición. Según esta definición un juego como el tres en raya tiene un factor de ramificación de 3. El ajedrez tiene un factor mucho más alto, alrededor de 35, esto lo convierte en un hueso muy duro de roer para el Minimax. En el caso de Arimaa, al tener un factor de ramificación más alto que el ajedrez se puede llegar a la misma conclusión en relación a la utilización de Minimax.

3.3.1 Algoritmo Minimax

En teoría de juegos, Minimax es un método de decisión para minimizar la pérdida máxima esperada en juegos con adversario y con información completa, es un algoritmo recursivo, y su funcionamiento puede resumirse como elegir el mejor movimiento para uno mismo suponiendo que el contrincante escogerá el menos conveniente para uno, en la ***figura 3.2*** se aprecia la primera aproximación a cómo trabaja Minimax.

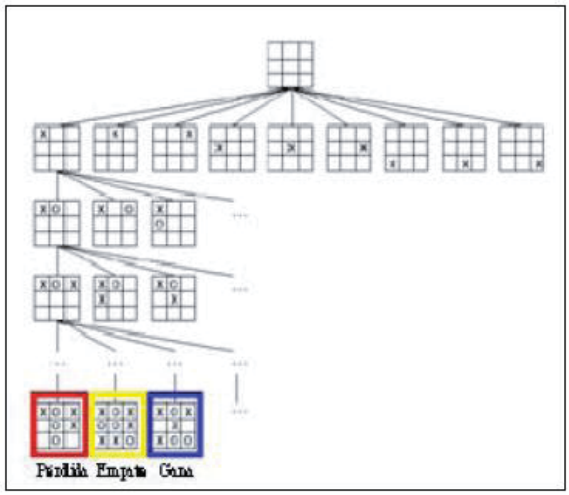

Figura 3.2: Árbol generado por el proceso de búsqueda Minimax en el juego de tres en raya

3.3.1.1 Pasos del Algoritmo:

- 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 oponente, de ahí el nombre de Minimax
- Elegir la jugada valorando los valores que han llegado al nivel superior

Al aplicar estos pasos o siguiendo el algoritmo de la ***figura 3.3***, se logrará representar de manera visual cómo se evalúa un árbol utilizando Minimax, como aparece en la ***figura 3.4***.

Entrada:
- Nodo N.

Salida
- Valor de utilidad de N

```
Si N es nodo Hoja
    ValorUtilidad← función de utilidad
De lo contrario
Para cada nodoHijo Hi de N hacer
    valorUtilidadi←MiniMaxR(Hi)
Fin para
Si N es max
    valorUtilidad ← max(H1,H2,…Hn)
De lo contrario
    valorUtilidad ← min(H1,H2,…Hn)

Devolver (valorUtilidad)
```

Figura 3.3: Algoritmo Minimax

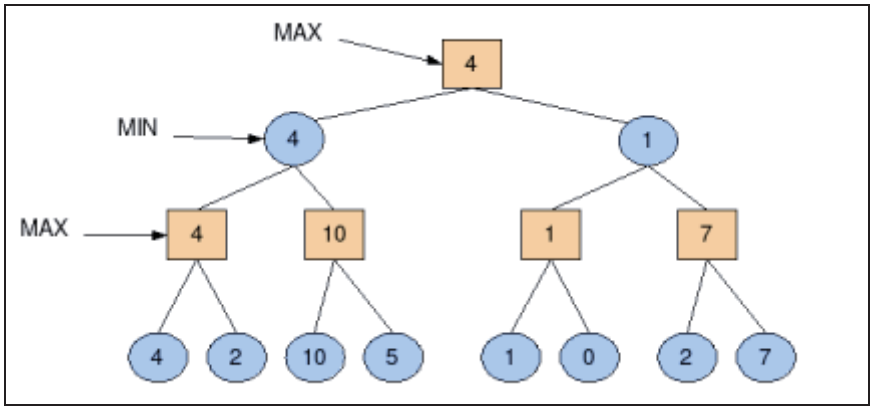

Figura 3.4: Ejemplo de un árbol de búsqueda completo usando Minimax

3.3.2 Mejorando Minimax: ¿Alfa Beta?

Si se considera la ***figura 3.3*** que es un ejemplo de la utilización del algoritmo Minimax en su totalidad, al calcular el árbol de variantes completo, sin aplicar ninguna técnica de poda. Este ejemplo de búsqueda completa con Minimax servirá de base para explicar cómo trabaja Alfa Beta.

A continuación se verá cómo aplicar una técnica que ahorra algo de trabajo. Para esto se seguirá con detalle el camino que sigue el algoritmo de búsqueda mientras se va construyendo el árbol de variantes.

3.3.2.1 Primera parte: Siguiendo de cerca la búsqueda

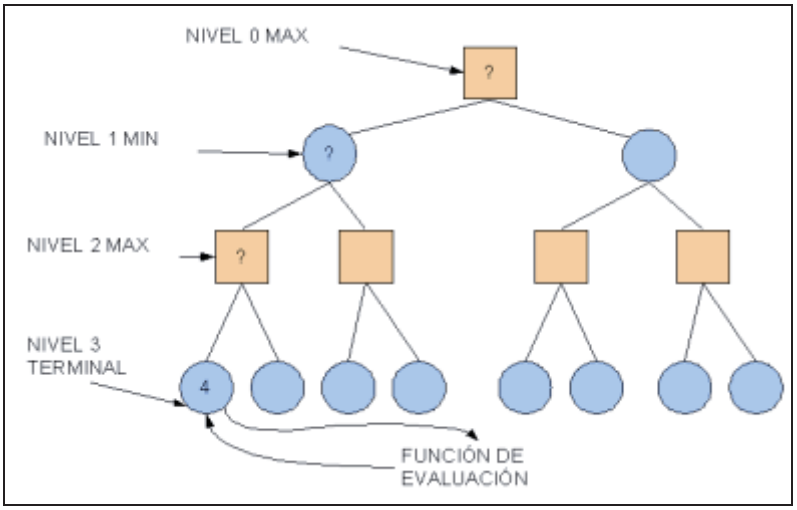

Figura 3.5: Primera parte

Nivel 0: Se parte de la posición de partida, el primer trabajo es generar su lista de jugadas. Una vez generada se considera la primera jugada de la lista y se aplica sobre el tablero. Como resultado se tiene una posición nueva en el nivel 1.

Nivel 1: Se genera la lista de jugadas para esta nueva posición. Otra vez se queda con la primera de la lista y se realiza en el tablero, así se llega a una nueva posición en el nivel 2.

Nivel 2: Se repite el proceso: se genera la lista de jugadas y se realiza la primera en el tablero. El resultado es una posición en el siguiente nivel, el 3.

Nivel 3: Se está en el final del árbol, los nodos de este nivel serán enviados a la función de evaluación y etiquetados con un valor numérico. El resultado de este primer nodo terminal evaluado es 4.

3.3.2.2 Segunda parte: Subiendo y bajando

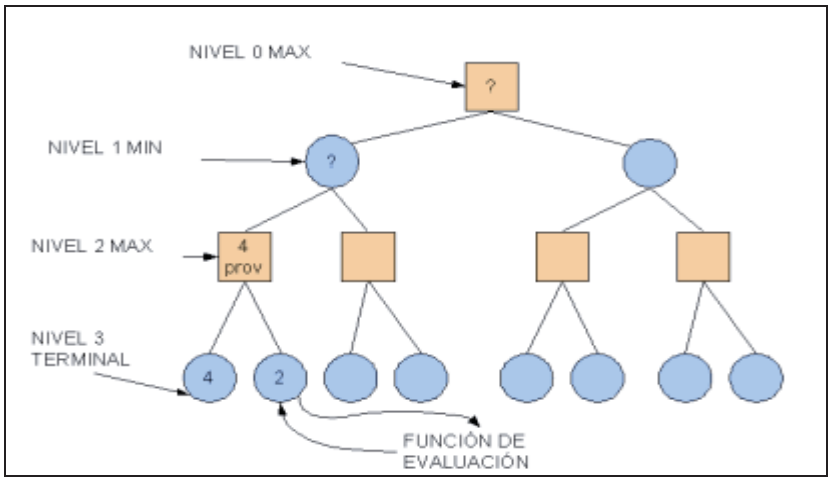

Figura 3.6: Segunda parte

Nivel 3: El algoritmo de búsqueda ha tocado fondo: es hora de iniciar su camino de regreso. El primer escalón de subida es ascender al nivel 2 con el resultado del primer nodo terminal evaluado. Este valor se anota como valor provisional del nodo padre.

Nivel 2: Este nodo toma como valor provisional 4. Ahora el algoritmo desciende de nuevo para examinar el siguiente nodo terminal en la lista de jugadas de esa rama.

Nivel 3: Se envía la posición a la función de evaluación y resulta que este otro nodo vale 2. El flujo del programa asciende de nuevo para informar al nodo padre del nuevo valor terminal.

Nivel 2: Se debe recordar que aquí se tiene un valor provisional de 4. El nuevo valor que se le propone es 2. Como se está en un nivel maximizador, el nodo conserva el valor anterior más alto, con lo que el valor definitivo de este nodo es 4.

3.3.2.3 Tercera parte: El flujo de trabajo del Minimax

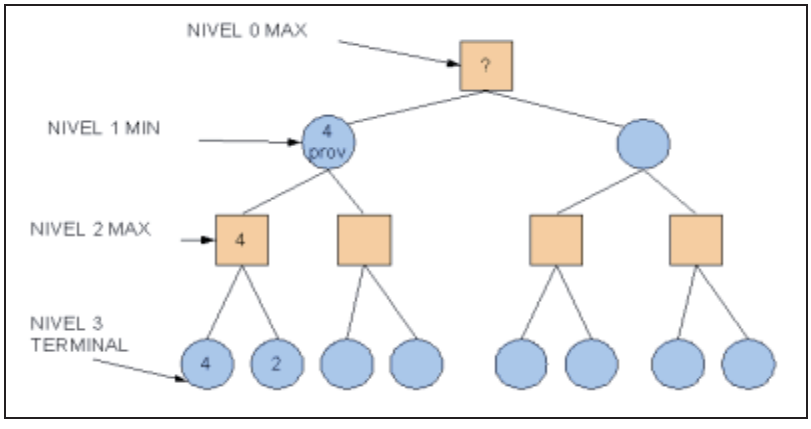

Figura 3.7: Tercera parte

Se hace una pequeña escala en el nivel 2. Si se observa el flujo de trabajo del Minimax: el árbol se va formando expandiéndose hacia abajo hasta encontrar un nodo con valor final. Cuando se asigna valor de forma definitiva a un nodo el algoritmo sube para informar al padre del valor recién obtenido. Estos valores definitivos se pueden obtener como resultado del juicio directo de la función de evaluación (nodos terminales) o porque todos los nodos hijos ya tienen valor definitivo. En este punto se está ahora: el nodo del nivel 2 tiene valor final porque ya se han valorado todos sus hijos. Por tanto la función de búsqueda sube de nivel para informar al nodo padre (minimizador) del valor de su primer hijo. Se anota por tanto un valor provisional 4 en este nodo del nivel 1.

3.3.2.4 Cuarta parte: Identificando límites

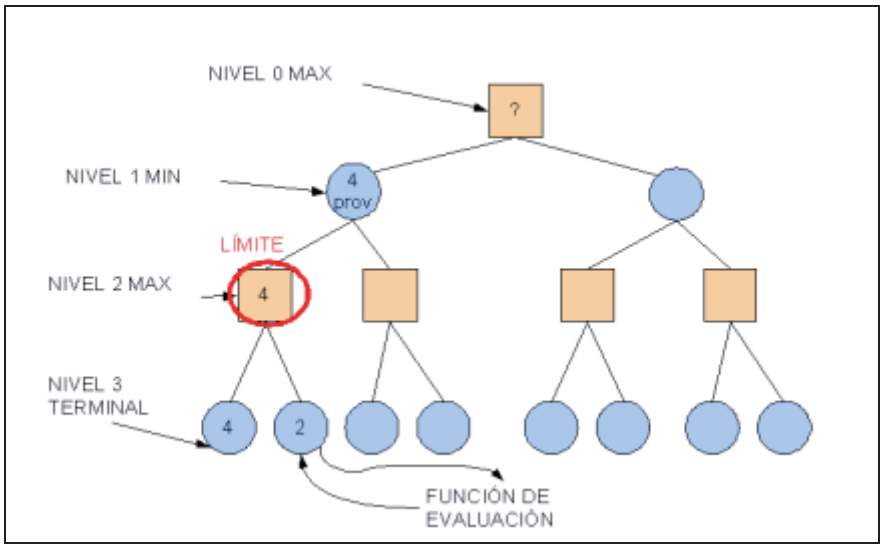

Figura 3.8: Cuarta parte

A continuación el flujo del programa se dirige otra vez hacia abajo para expandir el siguiente nodo de nivel 2. Pero esta vez hay una diferencia, se tiene un límite: el valor final del nodo hermano, es decir 4. ¿Qué significa que se tenga un límite = 4? El padre del nivel 2 es minimizador, elegirá el menor de sus nodos hijos en cada rama. O dicho de otra forma: entre todos los nodos hermanos del nivel 2 será escogido el menor. Luego como el primer nodo examinado tiene un valor 4, es seguro que ningún nodo con valor superior a 4 será elegido en esta rama. Por tanto en cuanto se detecte que un nodo va a tener un valor menor de esta cifra se puede ahorrar esa parte de la búsqueda sin miedo a perder nada: fuese cual fuese el resultado de esas posiciones que no se van a generar ni evaluar.

3.3.2.5 Quinta parte: Aplicando los límites

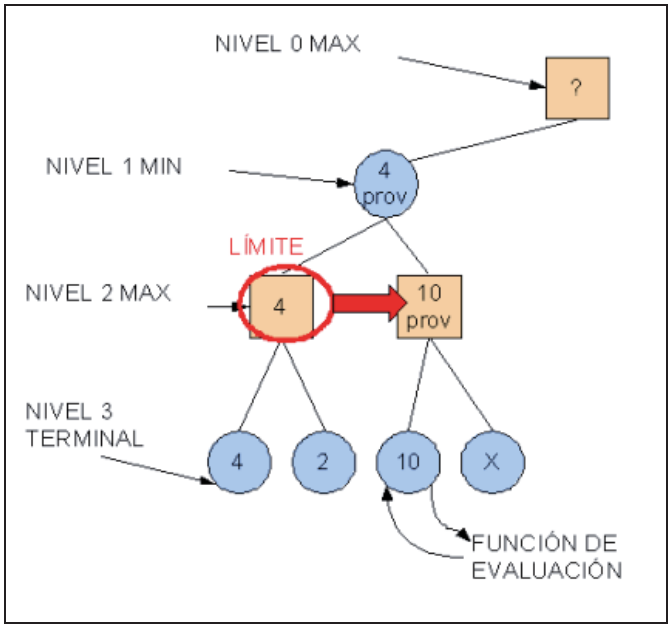

Figura 3.9: Quinta parte

Ahora se vuelve al ejemplo del diagrama. El algoritmo de búsqueda empieza a desarrollar el nodo del nivel 2 (el hermano del que vale 4). Desciende y valora el primer nodo terminal, que es puntuado con un valor de 10. Luego sube y se anota ese valor como provisional del nodo padre. ¿Se puede podar el árbol en esta rama? Se tiene un valor provisional de 10. Si se sigue con la búsqueda, se desciende y se le pide el siguiente valor a la función de evaluación, puede ser alguno de estos casos:

Caso 1: X < = 10

No es seleccionado por su padre del nivel 2 que preferirá el provisional más alto que tiene ahora, puesto que es maximizador.

Caso 2: X > 10

Es seleccionado por nivel 2. Pero un paso más arriba, el nivel 1 es minimizador, y lo descartará porque prefiere a su hermano el 4, que es más pequeño.

Por lo tanto se puede concluir que no es necesario calcular X porque no puede ser elegido por los niveles superiores en ningún caso. Por tanto cuando el nodo toma el valor provisional 10 ya se asegura de que no va a ser elegido, y se puede podar la búsqueda en esta rama.

3.3.2.6 Resultado de la poda

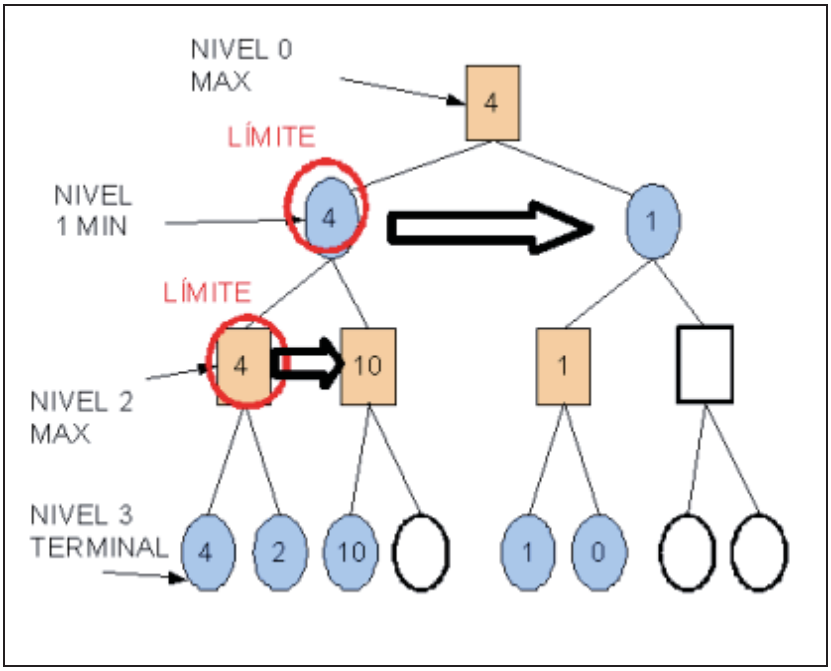

Figura 3.10: Resultado de la poda

Generalizando, los valores de los nodos ya evaluados son utilizados como límites para los nodos hermanos de ese nivel. Si el nivel es maximizador se podrá podar si se obtiene un valor provisional superior a los que ya se obtuvieron. Si se está en un nivel minimizador, se podará si se detecta un valor provisional inferior al límite de sus hermanos. En la ***figura 3.10*** se puede ver cómo quedó el árbol de variantes tras aplicar la poda Alfa Beta. Además de aplicar los límites que va encontrando, el algoritmo debe ir actualizándolos.

En el caso del siguiente ejemplo: se está en un nivel maximizador y se tiene un límite de 5. Si el siguiente nodo da un resultado de 3 no se puede podar, pero se actualiza el límite, que pasa a ser 3 para los siguientes hermanos. En el algoritmo de búsqueda clásico de los programas de ajedrez se utilizan normalmente dos límites de este tipo, uno representa el límite para los niveles maximizadores y otro para los minimizadores. A estos límites se les suele llamar alfa y beta, y es de ahí de donde recibe el nombre el algoritmo. Una buena aplicación de Alfa Beta acelera muchísimo el proceso de búsqueda.

Se debe tener presente que para un mejor funcionamiento del algoritmo Alfa Beta, es importante que la lista generada de los posibles movimientos esté ordenada de una manera conveniente, de modo que los mejores movimientos se encuentren en los primeros lugares, lo cual facilita la poda de los movimientos menos convenientes que siguen a continuación.[2]

3.3.3 Alfa Beta

Este algoritmo es el más utilizado en las aplicaciones referidas a juegos, dada su excepcional utilidad en el aumento de la velocidad de la búsqueda sin producir pérdida de la información. Es una extensión en particular del algoritmo de Búsqueda Minimax en juegos de dos contrincantes.

Cada vez que se evalúa un nodo no hoja, el algoritmo determina si los nuevos hijos generados pueden generar una mejor utilidad de la que ya posee el nodo estudiado y si afecta al nodo padre. De no ser así, eso significa que seguir analizando esa rama desperdiciará recursos como tiempo y espacio, por lo cual no se sigue generando y simplemente se le poda, de allí el nombre. El algoritmo detallado en la ***figura 3.11***, muestra como es el desarrollo para una búsqueda por medio de la poda Alfa Beta.

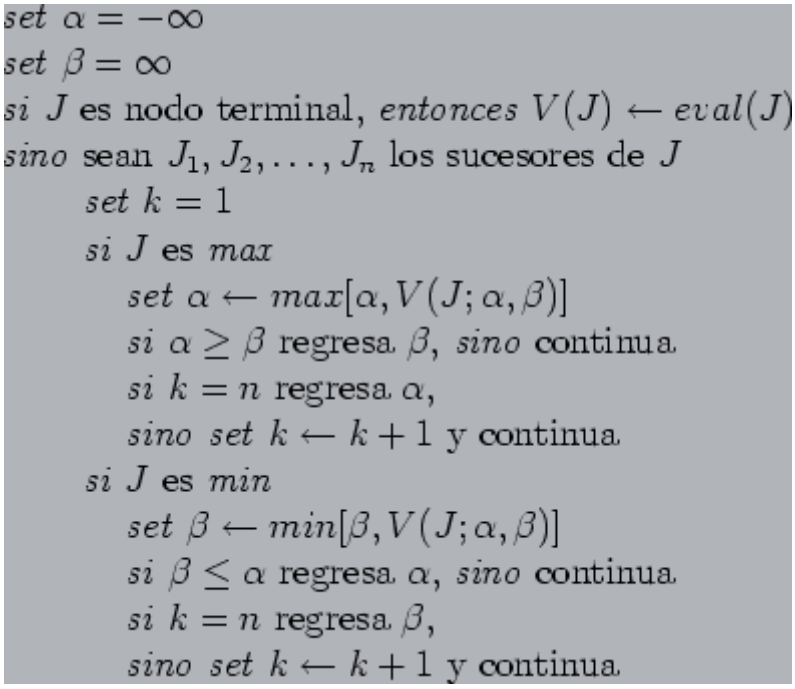

Figura 3.11: Algoritmo Alfa Beta

3.4 Estructura de Datos

Para poder manejar los datos importantes en el proceso de la búsqueda y selección del movimiento, se utilizará una estructura de datos que contiene los campos que se señalan a continuación:

- Identificador
- Identificador del padre
- Tipo de nodo
- Estado del tablero
- Función de utilidad
- Movimiento
- Profundidad

El primer campo es el identificador del nodo que se está generando. El siguiente dato, corresponde al identificador del padre que lo generó. El tipo de nodo hace referencia a la característica del nodo que se está analizando: si es Maximizador o si es Minimizador. En caso de ser Maximizador, el algoritmo tratará de obtener la jugada de mayor utilidad, pues genera beneficio para Max; sin embargo cuando es Minimizador, la máquina elegirá la menor utilidad posible, perjudicando al oponente. El estado, es la configuración del tablero, es decir, corresponde al ambiente que será analizado por el computador en este caso correspondería a una matriz de 8x8 o 10x10. La función de utilidad corresponde a la utilidad calculada por la función de evaluación. Movimiento es una variable que va recolectando la jugada que se genera con la mejor jugada posible, para esto se debe basar en la utilidad. Y finalmente la profundidad, indica, que tan profundo se va a analizar las jugadas, es decir, si se escoge una profundidad de 4, esto indicaría que se va a analizar 2 turnos para ambos jugadores y de allí se extraerá el mejor camino.

3.5 Jugadas legales y pseudo legales

Un aspecto decisivo es la velocidad de cálculo de jugadas y la manipulación de las posiciones en el tablero. Si el programa no es capaz de hacer esto ágilmente nunca se llegará muy lejos en la búsqueda. Muchos factores intervienen en la rapidez a la hora de construir el árbol de variantes, no sólo la rapidez del generador de movimientos. Sobre esto hay un punto muy interesante donde se puede ganar o perder mucho tiempo. Se trata de la detección de jugadas ilegales. Cuando se habla del problema de la legalidad de las jugadas en un motor de Arimaa se refiere principalmente sobre la detección de movimientos de piezas congeladas. Comprobar si en una posición determinada una pieza congelada realiza un paso es algo que lleva mucho tiempo de cálculo. Este proceso podría ser tan costoso que frecuentemente los generadores de movimientos calculan todos los movimientos posibles sin todas las validaciones correspondientes. Por lo que posteriormente hay que filtrar esta lista de movimientos pseudo legales de alguna manera.

Hay dos estrategias que se enfrentan de forma distinta al problema de la detección de jugadas no válidas:

- Tener un generador de movimientos que sólo incluya las jugadas completamente legales.
- Tener un generador de movimientos que incluya todas las jugadas posibles sin hacer ninguna comprobación. La lista se filtrará en la función de búsqueda, antes de realizar la jugada en el tablero se comprobará si el movimiento es completamente legal y se rechazarán las jugadas ilegales.

La ventaja aparente del segundo método es que no habría que hacer la costosa prueba de legalidad para toda la lista de jugadas, sino sólo para las que sobrevivan a la poda. [2]

3.6 El efecto horizonte

El efecto horizonte es un problema inherente al Minimax y que es imposible eliminar por completo. Se presenta cuando se envía a evaluar una posición intermedia en una secuencia táctica. Los siguientes ejemplos del ajedrez ilustran la situación:

En una secuencia de cambios tácticos el programa captura una torre. Se evalúa esta posición con un valor de +5 porque tiene una torre más que su adversario. A la siguiente jugada el rival corona un peón que estaba en la séptima fila. Se ha producido el temido efecto horizonte. El programa evaluó una posición intermedia en una secuencia táctica y obtuvo una valoración errónea.

Ahora, si se supone que es el turno de juego del programa con blancas. La función de búsqueda empieza a construir su árbol de variantes y en uno de los nodos terminales asigna un valor de +10 a la posición. La valoración tan alta se explica porque el programa acaba de comerse la dama de su rival. Si se cuenta el material que hay en el tablero se ve que las blancas tienen una dama de más y por tanto la función de evaluación valora esta posición con una ventaja de 10 puntos. Pero si se sigue mirando una jugada más allá se ve que las negras restablecen la igualdad material porque capturan la dama blanca en el siguiente movimiento. Es decir, lo que al programa le parece una captura de dama en realidad es un simple cambio. El desastre puede ser total si para llegar a ese cambio de damas el programa ha sacrificado material confiando en que al comerse la dama enemiga saldría ganando finalmente.

El efecto horizonte continúa apareciendo sea cual sea la profundidad de búsqueda. No importa si el programa puede ver hasta una profundidad de 4, 5 o 16 jugadas: si las posiciones de los nodos terminales son posiciones intermedias en una secuencia de cambios el efecto horizonte impedirá al programa jugar bien. Esta ceguera suicida es parte del algoritmo de búsqueda empleado por los programas de ajedrez y si bien no se puede eliminar completamente sí se puede mitigar en gran parte. La solución parcial que permite minimizar los efectos de este problema se basa en la búsqueda de posiciones estables. El algoritmo de búsqueda no se detiene hasta que encuentra una posición estable. A efectos prácticos se va a suponer que una posición estable en ajedrez es aquella en la que no existen capturas legales ni golpes tácticos como coronaciones de peón, etc. Si se trata de poner esto en práctica de forma directa extendiendo la búsqueda hasta llegar a posiciones estables pronto se descubre que no se va a llegar muy lejos: la búsqueda de las posiciones estables extiende demasiado el árbol, es peor el remedio que la enfermedad.

3.7 Mejoras en la función de búsqueda

3.7.1 Profundización iterativa

La primera de estas mejoras resulta sorprendente. Consiste en realizar varias búsquedas en cada turno de juego, en lugar de una sola. Primero se realiza una búsqueda hasta el nivel 1. Un vez terminada ésta se hace una segunda búsqueda que llega hasta el nivel 2, y así sucesivamente, profundizando un nivel más en cada ciclo hasta que se agota el tiempo disponible para esa jugada.

Resulta paradójico, aunque se hace todo lo posible por reducir la búsqueda, ahora se decide que es mejor hacer no una sino varias búsquedas por cada turno de juego. Hay varias razones que explican que hacer esto es una buena idea, por la propia naturaleza del Minimax sucede que cada nivel adicional en que se profundiza cuesta tanto o más tiempo de cálculo que la suma de todos los niveles anteriores. Con esta técnica se asegura de que siempre habrá una jugada disponible cualquiera que sea el tiempo que se tiene para realizar la jugada, ya que la búsqueda hasta los primeros niveles es casi instantánea. Lo más importante es que en cada búsqueda que se hace, se obtiene información muy valiosa que se utilizará en la siguiente búsqueda. Esta realimentación es la base de varias mejoras en el algoritmo de búsqueda. Es decir, para mejorar y hacer más rápida la función de búsqueda primero se tiene que introducir la profundización iterativa como paso previo. Es la base para todas las mejoras posteriores.

3.7.2 Línea principal

La principal mejora que aprovecha la técnica de la profundización iterativa es el uso de la línea principal. Para esto se debe tener la precaución de guardar y mantener la mejor variante cuando se está calculando dentro de la función de búsqueda. De esta forma cuando la búsqueda acaba se tiene no sólo la mejor jugada, sino también toda la línea prevista por el programa como posibles respuestas de uno y otro bando hasta la profundidad hasta donde haya llegado. Se llama a esta variante “línea principal”. La línea principal se utiliza para ordenar la lista de jugadas. Se tiene anotada la mejor jugada que el programa calculó para ese nivel, por tanto se aprovecha esta información para ponerla primera en la lista. Esto supone una mejora muy importante porque como ya se ha visto el algoritmo Alfa Beta es extraordinariamente sensible a la ordenación de la lista de jugadas.

3.7.3 Tablas de transposición

En esencia esto consiste en guardar las posiciones que se van generando durante el proceso de búsqueda junto con la valoración que les ha asignado la función de evaluación. Es decir, se guarda la posición de origen, la jugada para llegar a la posición final y la valoración asignada por la función de evaluación.

De esta forma si se vuelve a presentar esta posición, el programa no debe calcular de nuevo todas las variantes, utiliza la valoración obtenida anteriormente ahorrándose mucho tiempo de cálculo. El uso de estas tablas también es útil para la detección situaciones especiales por repetición.

3.7.4 Jugada asesina

Esta técnica consiste en mantener una lista de jugadas que demostraron ser buenas en posiciones anteriores. Estas jugadas se ponen cerca del principio de la lista para que sean procesadas primero, con la esperanza de que produzcan un corte en la poda Alfa Beta y ahorren tiempo de cálculo.

3.7.5 Ventana de aspiración

Normalmente el algoritmo Alfa Beta comienza su camino con unos límites extremos para alfa y beta, por ejemplo 9999 y -9999. La ventana de aspiración consiste en reutilizar los valores finales de alfa y beta para la búsqueda siguiente dentro de la profundización iterativa. Utilizar unos límites más pequeños hace al algoritmo ligeramente más rápido, pero si se sale de los límites habrá que hacer una búsqueda completa (con límites más amplios) para estar seguros de que la jugada escogida es correcta.

3.7.6 Factibilidad de la aplicación de las mejoras

- **Profundización Iterativa**: Esta es la mejora más tentadora, ya que una de las problemáticas de Alfa Beta es la manera de conseguir ordenar la lista de jugadas de una manera eficiente para aumentar el número de podas. Inicialmente se implementó esta mejora, pero dada la ramificación de Arimaa el árbol de jugadas crece demasiado rápido lo que no permite obtener mucha profundidad.

- **Línea Principal:** La línea principal se apoya en la profundización iterativa, dado que inicialmente se implementó la mejora anterior, también se decidió implementar esta mejora. El costo de implementación es bajo, aunque el requerimiento de implementar la profundización iterativa no lo es.

- **Tablas de transposición:** Esta mejora propone guardar posiciones con sus respectivas valoraciones, para cuando se vuelvan a presentar, reutilizar la información guardada para evitar volver a calcular nuevamente lo mismo, pero dada las características de Arimaa, el guardar una posición no garantiza que ésta se vuelva a presentar por lo que el costo en memoria y espacio no se justifica razón por la cual no se intentará implementar esta mejora.

- **Jugada asesina:** Al igual que en la mejora anterior, aunque una jugada demuestre ser buena, no hay garantías de que al guardarla, se vuelva a repetir una posición igual, para volver a utilizarla, por lo que tampoco se intentará implementar esta mejora.

- **Ventana de aspiración:** Esta es la más práctica de las mejoras, ya que no requiere mayor implementación, consiste simplemente en acotar los límites de alfa y beta para producir mayor cantidad de podas.

3.8 Evaluación de posiciones

Para guiar la selección de movimientos los primeros programas de fuerza bruta utilizaron una función de evaluación estática de la posición. Ésta se basaba en la premisa de que es posible examinar una posición y calcular un puntaje que representa su valor relativo a otras posiciones. Si esto es cierto, entonces la función de evaluación se reduciría a la simple tarea de buscar en el arreglo de posiciones posibles con tal de encontrar aquella con el mejor score.

Desde un punto de vista teórico esta premisa es razonablemente cierta. En el caso del ajedrez por ejemplo, sin límites de tiempo, recursos de hardware ilimitados y asistencia de grandes maestros de ajedrez, un equipo de buenos programadores podría presumiblemente escribir una muy detallada función de evaluación que consideraría factores que un jugador humano toma en cuenta al jugar al ajedrez. Con la ayuda de un reconocedor de patrones y una gran tabla de hash de posiciones un evaluador retornaría un valor de posición muy preciso para la mayoría de las posiciones. Desafortunadamente, esta función de evaluación sería tan grande y compleja que requeriría mucho tiempo y poder computacional para ejecutarse en un tiempo razonable.

Los programas que buscan a una profundidad de 8 movimientos como Belle o Crayblitz sólo podían disponer de 5 a 10 microsegundos para la evaluación de cada posición. Para búsquedas de 5 movimientos los programas deben evaluar posiciones en uno a 10 milisegundos. La función de evaluación debe ser muy simple y fácil de ejecutar, y debe funcionar en base a operaciones que un computador pueda manipular con facilidad en vez de conceptos que tendrían sentido bajo el punto de vista de un maestro de ajedrez.

La experiencia ha demostrado que el dar a un programa demasiado conocimiento es una tarea difícil y peligrosa. Si la información es incompleta se pueden tener efectos contrarios a lo deseado. El programa puede utilizar esta información en un momento y forma no adecuados. Si el código es exhaustivo en la manipulación de estas excepciones entonces puede influir en alterar la velocidad de funcionamiento del programa lo cual afectará su capacidad de búsqueda.

Las funciones de evaluación utilizadas en el mundo real no pretenden evaluar las posiciones con suficiente detalle. El objetivo es dar al programa suficiente información con tal que pueda usarla en su búsqueda dentro del árbol de variantes con tal de descubrir el efecto de factores dinámicos los cuales la función no puede evaluar, tales como el resultado de combinaciones. La función de evaluación sin embargo debe ser capaz de entregar al programa información acerca de aspectos estáticos de la posición tales como piezas congeladas, piezas sin defensa, movilidad, etc. Esta característica logra efectos a largo plazo en el juego, los cuales en la búsqueda muchas veces no los puede descubrir. Por todas estas razones la función de evaluación de un programa no puede ser considerada como una medida precisa del valor de la posición, aunque aún así es una herramienta indispensable.

Durante la búsqueda en el árbol de variantes el programa confirmará la evaluación de la posición con la búsqueda sobre el siguiente movimiento. La misión de la función de

evaluación es entonces bastante simplificada. No es su labor el evaluar en forma precisa factores dinámicos como el resultado de combinaciones, pues estos son dejados al motor de búsqueda. Tampoco debe hacer mediciones posicionales precisas puesto que los límites de tiempo que el programa dispone para realizar su movimiento le impiden perder tiempo en esta consideración.

Por lo tanto, la misión de la función de evaluación es el cooperar con el árbol de búsqueda en determinar en forma aproximada el valor de una posición observando qué puede ocurrir en el juego futuro. Para lograr esto en forma efectiva la función de evaluación debe ser capaz de medir el equilibrio material (cantidad de piezas para cada bando) y generar un razonable análisis de la situación estática de la posición en la cual, la búsqueda en profundidad es incapaz de determinar.

El mayor desafío o problemática en la construcción de funciones de evaluación es: ¿Cuánta información puede obtener el programa con el tiempo disponible? Como regla general la respuesta es "no lo suficiente". En este caso una buena función puede hacer una notable diferencia en la fuerza de juego del programa.

3.8.1 Evaluación de material

Uno de los aspectos más básicos e intuitivos de evaluar en una posición es la comparación de piezas, y como es natural, se piensa que el jugador que tiene un mayor número de piezas es quien posee una ventaja más evidente, si bien es cierto que esto es una manera acertada de ver, se deben tener en consideración la importancia de las piezas. En muchos juegos de ajedrez sencillos, ésta es la única función de evaluación usada, por su simplicidad y aparente efectividad para dar un valor a la jugada de turno. Esta función parte del concepto de que es muy importante y determinante en algunos casos, el poder tener una ventaja numérica respecto al adversario. Si no se logra obtener una ventaja numérica, en el peor caso sería aceptable estar en balance de fuerzas.

Además de ello, no todas las piezas son iguales, ya que no es lo mismo perder, un Elefante, que perder un Gato. Otro aspecto que se debe considerar y tratar de una manera diferente, es el relacionado con los Conejos, ya que estos en cuanto a fuerza son las piezas más débiles, pero son muy importantes ya que es la pieza con la que se gana el juego.

En base a la jerarquía de las piezas en la ***tabla 3.1***, se asignan los valores relativos a las piezas, como se puede ver en la ***tabla 3.2***, los Conejos son tratados de una manera distinta, ya que además de considerar la pieza, también se hace una diferencia en relación a la cantidad de Conejos restantes. Lo que en otras palabras se entiende, al considerar que el Conejo, es una pieza cuyo valor no es el mismo durante la partida, ya que es indispensable para ganar el juego y en la medida que quedan menos se hace más importante conservarlos para tener la opción de ganar la partida.

Tabla 3.1: Valor material de las piezas

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

Tabla 3.2: Valor de los Conejos

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

3.8.2 Evaluación de la posición individual de las piezas

La función de evaluación debe tomar en cuenta consideraciones de largo plazo tanto a favor como en contra dentro de la evaluación de una posición, es decir, efectos que puedan persistir sobre un número de movimientos mayor que los que el programa pueda calcular. Estos efectos se basan muchas veces en consideraciones estáticas sobre la posición, es decir, consideraciones basadas en la ubicación de las piezas, Conejos y sus posibilidades de evolucionar dentro de la partida. Estas consideraciones son las denominadas consideraciones posicionales.

Los primeros programas de ajedrez fallaban rotundamente en este tipo de consideraciones cometiendo errores similares a los novatos en el juego. El problema era que para dar a la máquina una fuerte evaluación posicional era necesario el entregarle una serie de parámetros relacionados con "conocimiento ajedrecístico", algo que los grandes maestros de ajedrez obtienen principalmente en base a su experiencia. La duda era entonces cuanto "peso" dar a los factores posicionales por sobre el conocimiento ajedrecístico a ocupar en la función de evaluación.

Los primeros programas debieron dar importancia a la capacidad de búsqueda y limitarse a una función de evaluación la cual considerara factores principalmente materiales. Con el paso de los años y la mayor capacidad de las máquinas, el incluir factores posicionales dentro de la función de evaluación ya no iría en contra de la capacidad de cálculo del programa, por lo cual empezó a ser una tarea para los programadores. Los primeros intentos no fueron fáciles, puesto que para esto los creadores de máquinas de ajedrez necesitaron del asesoramiento de expertos para incluir el conocimiento en sus programas. Aún así, las consideraciones posicionales dentro del ajedrez resultan ser tantas y con tantas excepciones lo cual hizo que la tarea de dar un conocimiento posicional a los programas fuese a ratos

extremadamente frustrante. De acuerdo a Botvinnik un factor posicional que logra un efecto positivo en una posición dada puede perfectamente ser negativo en otra. Por ejemplo, ¿los peones doblados son buenos o malos? La respuesta depende de la situación. En algunas ocasiones los peones doblados son un objetivo de ataque mientras que en otras pueden no ser una debilidad y defender casillas críticas. Lo mismo puede ser dicho para otros factores. La propuesta de análisis posicional se basaba más bien en un modelo matemático centrado en el control de casillas y consideraciones materiales.

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 3.3*** figuran los valores que se les asignaron a los Conejos dependiendo de las distintas posiciones que ocupen en el tablero.

Tabla 3.3: 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

En el caso particular de los Conejos además de definir sus mejores posiciones, también se tomará en cuenta su cercanía a piezas aliadas, esta bonificación aumentará dependiendo del avance del Conejo como se aprecia en la ***tabla 3.4***.

Tabla 3.4: 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

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 trampas del lado propio, los valores en relación a su posición se aprecian en la ***tabla 3.5***.

Tabla 3.5: Cuadro de los 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	1	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

Perros: Los Perros, son piezas relativamente débiles que sólo pueden congelar a los Conejos y Gatos enemigos, por lo que su misión también será netamente defensiva y en cierto modo de apoyo a los Gatos, y sus mejores posiciones girarán en torno al control de los cuadros de trampa del lado propio, sus valores de acuerdo a su posición se aprecian en la ***tabla 3.6***.

Tabla 3.6: Cuadro de los 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	1	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

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, sólo deben tener cuidado del Camello y del Elefante enemigo. Sus valores dependiendo de su posición se aprecian en la ***tabla 3.7***.

Tabla 3.7: Cuadro de los 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

Camello: El Camello, es una de las piezas más fuerte del tablero, sólo debe cuidarse del 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, como se aprecia en la ***tabla 3.8***.

Tabla 3.8: Cuadro de los 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

Elefante: El Elefante, es la pieza más fuerte del tablero, no puede ser congelado por ninguna otra pieza, al igual que el Camello el mayor potencial se obtiene en las casillas centrales que le permite llegar rápidamente a los cuadros de trampa, ya sea para capturar una pieza o salvar una pieza aliada. Sus valores se aprecian en la ***tabla 3.9***.

Tabla 3.9: Cuadro de los 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

3.8.3 Control de Trampas

El controlar un cuadro de trampa permite estar a salvo de la captura de piezas en esa trampa, y además permite ser capaces de capturar piezas enemigas en ella misma. Al comienzo del juego cada jugador es dueño de las dos trampas más cercanas. Una de las estrategias más simple, es la ofensiva que radica en la invulnerabilidad del Elefante que le permite ir al territorio enemigo y así poder empujar a las piezas enemigas en los cuadros de trampas que controlan, con la esperanza de lograr alguna captura de pieza.

Una estrategia ofensiva más ambiciosa es tratar de obtener el control de una de las trampas enemigas, la mayoría de las veces con un Elefante y Caballo. Que plantea el reto para tratar de invadir el territorio enemigo, de esta forma, si el Elefante enemigo no se opone a esta ofensiva y ayuda a defender, el resultado será una carrera de posición.

Un cuadro de trampa del territorio enemigo, por lo general tiene muchas piezas enemigas cerca, que pueden ser capturadas en una rápida sucesión, en contraposición con el laborioso proceso de mover varias veces una pieza enemiga, y así arrastrarla todo el camino a una trampa propia para poder capturarla. Igualmente un importante beneficio de ser dueño de una trampa del enemigo es que si se oponen, la dispersión de las piezas, puede dejar un agujero a través del cual un Conejo aliado puede marchar a la meta. Esta amenaza no sólo es una invasión del territorio enemigo, sino que además merma sus opciones de defensa y de un contra ataque.

Como se puede apreciar en la ***tabla 3.10***, las piezas tienen un valor asignado por jerarquía, donde la pieza más importante para dominar una trampa es el Elefante, el otro aspecto importante a definir, es dejar claro cuáles son los cuadros que se van a considerar, y como se muestra en la ***tabla 3.11*** con un ejemplo de la trampa en la casilla C3, se van a considerar las casillas que se encuentran a uno o dos pasos de distancia, haciendo una diferencia en las ponderaciones dependiendo de la distancia.

Tabla 3.10: Valores por custodia de trampas

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

Tabla 3.11: Multiplicadores del control de la trampa C3 [6]

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

3.8.4 Movilidad

En ajedrez es tal vez una de las funciones más importantes, permite definir las consecuencias de realizar un movimiento. Algunas veces realizar un movimiento x podría parecer buena opción, sin embargo muchas veces puede resultar que en dos movidas posteriores, haya sido una decisión catastrófica que pudo haber producido la derrota o una considerable pérdida de material.

Esta función a diferencia de las demás, es una función de carácter local con respecto a un nodo. No se evalúan superficialmente, sino que se hace una búsqueda de posibles consecuencias, para cada nodo, al realizar algún movimiento.

También mide la capacidad de libertad de movimiento para cada pieza, y la capacidad de moverse sin ningún ataque de por medio.

La cantidad de casillas disponibles para cada pieza entrega un factor de "libertad de movimiento" disponible para ellas. En ajedrez es muy importante el que las piezas tengan libre acción y a la vez entorpecer la acción de las piezas rivales. La evaluación de movilidad medirá simplemente la cantidad de movimientos legales disponibles para todas las piezas de cada bando considerando la diferencia para cada pieza en la cantidad de casillas atacadas por blancas y negras. Esta característica de movilidad hace que el programa priorice el desarrollo

[6] Esta vecindad se determina por ser la zona de riesgo, ya que en un turno una pieza enemiga sólo puede ser movida en dos espacios.

en la apertura de sus piezas menores ubicándolas en posiciones favorables y reservando el desarrollo de Torres y Dama para más tarde.

Para Arimaa se debe tener en cuenta las mejores posiciones para evitar tener piezas congeladas, y limitar la movilidad de las piezas enemigas. Con este propósito como se aprecia en la ***tabla 3.12*** se penalizará a las piezas que tengan enemigos más fuertes en su cercanía y como aparece en la ***tabla 3.14*** se bonificará a las piezas que tengan aliados más fuertes en su frontera. Por comodidad, la ***tabla 3.13*** contendrá valores equivalentes a los de la ***tabla 3.12***, pero en vez de penalizar por tener enemigos más fuertes, se bonificará por tener enemigos más débiles.

Tabla 3.12: 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 3.13: 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 3.14: Bonificación ante piezas aliadas

Aliadas	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

3.9 Función de Evaluación

La función de evaluación es la que se aplica a una posición estática con la misión de estimar la utilidad que se obtendrá y a favor de quien, es decir, buscará reflejar las posibilidades de ganar de un jugador en base a los criterios y pesos que considere más adecuados para dicho cálculo.

La función de evaluación, será una función lineal de la forma $w_1f_1 + w_2f_2 + \ldots + w_nf_n$, donde w_i son los pesos asociados a las características f_i. En base a los criterios estudiados, los criterios que se consideraron en la función son:

- Evaluación de material

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

- Evaluación de la posición

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

- Control de trampas

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

- Evaluación de movilidad

$$f_4 = \sum_{i=1}^{8}\sum_{j=1}^{8}\left(\frac{VC_{(i\pm1,j)}}{\left|VC_{(i\pm1,j)}\right|}\bullet VTB_{vc(i,j)-vc(i\pm1,j)} + \frac{VC_{(i,j\pm1)}}{\left|VC_{(i,j\pm1)}\right|}\bullet VTB_{vc(i,j)-vc(i,j\pm1)}\right) \quad \forall\, VC_{(a,b)} \neq 0$$

Donde:

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

$VTM_{vc(a,b)}$: Valor de la tabla de material para la pieza correspondiente a $VC_{(a,b)}$ (tablas de la sección Evaluación de material)

$VTP_{vc(a,b)}$: 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)

$VTB_{vc(a,b)\rightarrow vc(c,d)}$: 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 $w_1=w_2=w_3=w_4=1$ por lo que la primera función de evaluación que se implementará es:

$$f.e. = f_1 + f_2 + f_3 + f_4$$

Capítulo 4

4.1 Implementación

La implementación del programa Arimaa se realizó en el lenguaje de programación JAVA en complemento con el IDE Netbeans 6.8. Como se vio anteriormente, la primera dificultad radica en encontrar una manera adecuada de representar la información, para el caso de Arimaa una matriz 8x8 puede representar completamente la información de una posición, aunque para efectos prácticos se trabajará con matrices 10x10. Además de saber cómo representar la información es necesario encontrar como interactuar entre el usuario y el programa.

La interfaz gráfica, se diseñó manualmente con el IDE, creando un JFRAME que contiene JPANEL y JBUTTON. Inicialmente el primer panel representa el tablero de juego y contiene 64 botones que representan las 64 casillas de un tablero, el segundo panel contiene los botones necesarios para realizar el procedimiento de ordenar las piezas en el tablero y poder iniciar la partida como se muestra en la ***figura 4.1***.

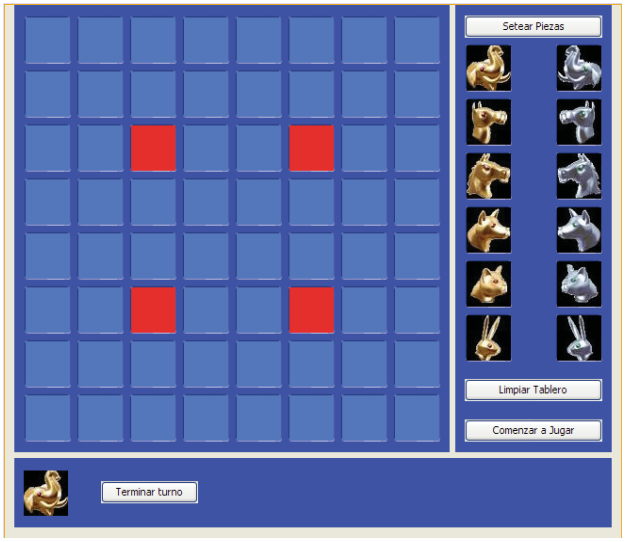

Figura 4.1: Pantalla del Programa Arimaa

Para representar la información se utilizan 2 clases, la clase Tablero y la clase Casilla. La clase Tablero será un arreglo bidimensional de objetos Casilla, y la clase Casilla será un objeto que va a contener la información de las coordenadas en las que se ubica la Casilla y de un valor que representara el valor de la pieza que la ocupa o en su defecto el valor 0 en el caso de ser una casilla desocupada.

4.1.1 Clases que representan la información

La clase Tablero: Es una matriz 10x10 de objetos Casilla, como se observa en la ***figura 4.2***, su único atributo es un arreglo bidimensional de objetos Casilla, y contiene varios métodos que se utilizan en otros procesos relacionados con la validación de los movimientos y la función de búsqueda. En la ***tabla 4.1*** se hace una representación más abstracta del objeto Tablero, anotando los índices de una manera más comprensible para la perspectiva de un jugador.

Figura 4.2: Clase Tablero

Tabla 4.1: Representación visual del objeto Tablero

Casilla [9][0]	Casilla [9][1]	Casilla [9][2]	Casilla [9][3]	Casilla [9][4]	Casilla [9][5]	Casilla [9][6]	Casilla [9][7]	Casilla [9][8]	Casilla [9][9]
Casilla [8][0]	**Casilla [8][1]**	**Casilla [8][2]**	**Casilla [8][3]**	**Casilla [8][4]**	**Casilla [8][5]**	**Casilla [8][6]**	**Casilla [8][7]**	**Casilla [8][8]**	Casilla [8][9]
Casilla [7][0]	**Casilla [7][1]**	**Casilla [7][2]**	**Casilla [7][3]**	**Casilla [7][4]**	**Casilla [7][5]**	**Casilla [7][6]**	**Casilla [7][7]**	**Casilla [7][8]**	Casilla [7][9]
Casilla [6][0]	**Casilla [6][1]**	**Casilla [6][2]**	**Casilla [6][3]**	**Casilla [6][4]**	**Casilla [6][5]**	**Casilla [6][6]**	**Casilla [6][7]**	**Casilla [6][8]**	Casilla [6][9]
Casilla [5][0]	**Casilla [5][1]**	**Casilla [5][2]**	**Casilla [5][3]**	**Casilla [5][4]**	**Casilla [5][5]**	**Casilla [5][6]**	**Casilla [5][7]**	**Casilla [5][8]**	Casilla [5][9]
Casilla [4][0]	**Casilla [4][1]**	**Casilla [4][2]**	**Casilla [4][3]**	**Casilla [4][4]**	**Casilla [4][5]**	**Casilla [4][6]**	**Casilla [4][7]**	**Casilla [4][8]**	Casilla [4][9]
Casilla [3][0]	**Casilla [3][1]**	**Casilla [3][2]**	**Casilla [3][3]**	**Casilla [3][4]**	**Casilla [3][5]**	**Casilla [3][6]**	**Casilla [3][7]**	**Casilla [3][8]**	Casilla [3][9]
Casilla [2][0]	**Casilla [2][1]**	**Casilla [2][2]**	**Casilla [2][3]**	**Casilla [2][4]**	**Casilla [2][5]**	**Casilla [2][6]**	**Casilla [2][7]**	**Casilla [2][8]**	Casilla [2][9]
Casilla [1][0]	**Casilla [1][1]**	**Casilla [1][2]**	**Casilla [1][3]**	**Casilla [1][4]**	**Casilla [1][5]**	**Casilla [1][6]**	**Casilla [1][7]**	**Casilla [1][8]**	Casilla [1][9]
Casilla [0][0]	Casilla [0][1]	Casilla [0][2]	Casilla [0][3]	Casilla [0][4]	Casilla [0][5]	Casilla [0][6]	Casilla [0][7]	Casilla [0][8]	Casilla [0][9]

La clase Casilla contiene la información sobre la ubicación de las piezas, asociándolas a una posición del tablero. Contiene tres atributos: una coordenada X, una coordenada Y, que representan la ubicación espacial de la casilla, y un valor, que representa mediante un número entero el estado ésta. Tal como se muestra en la ***figura 4.3***. La clase sólo contiene los métodos básicos para encapsular sus atributos.

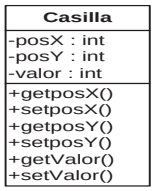

Figura 4.3: Clase Casilla

4.1.2 Representación de los Datos

Para definir en qué estado se encuentra la casilla, hay que asignarle un valor el cual representará, si tiene piezas, y en caso de tener, que pieza es la que tiene. En la ***tabla 4.2*** se muestran todos los posibles valores que puede tomar el atributo "valor" de la clase Casilla.

Tabla 4.2: Valores que representan el estado de una casilla[7]

Casilla ocupada por	Valor representativo
Elefante Dorado	6
Camello Dorado	5
Caballo Dorado	4
Perro Dorado	3
Gato Dorado	2
Conejo Dorado	1
Vacía	0
Conejo Plateado	-1
Gato Plateado	-2
Perro Plateado	-3
Caballo Plateado	-4
Camello Plateado	-5
Elefante Plateado	-6

[7] Los valores de esta tabla ya habían definido previamente en la sección "Representación del tablero"

4.1.3 Turno

Para llevar a cabo la implementación del turno, hay que tener presente algunas consideraciones en relación a lo que puede acontecer en cada paso del turno:

Paso 1:

- Si la pieza es aliada puede ser un paso normal o puede ser el inicio de un Tirar
- Si la pieza es enemiga sólo se puede tratar de un caso de Empujar

Paso 2:

- Si la pieza es aliada puede ser un paso normal, el inicio de un Tirar o el final de un Empujar
- Si la pieza es enemiga puede ser el final de un Tirar o el inicio de un Empujar

Paso 3:

- Si la pieza es aliada puede ser un paso normal, el inicio de un Tirar o el final de un Empujar
- Si la pieza es enemiga puede ser el final de un Tirar o el inicio de un Empujar

Paso 4:

- Si la pieza es aliada puede ser un paso normal, o el final de un Empujar
- Si la pieza es enemiga sólo puede ser el final de un Tirar

En la ***figura 4.4*** se muestra un diagrama de flujo de una jugada, en el cual se ven los posibles caminos que se pueden seguir para las distintas combinaciones de pasos que se pueden presentar en una jugada.

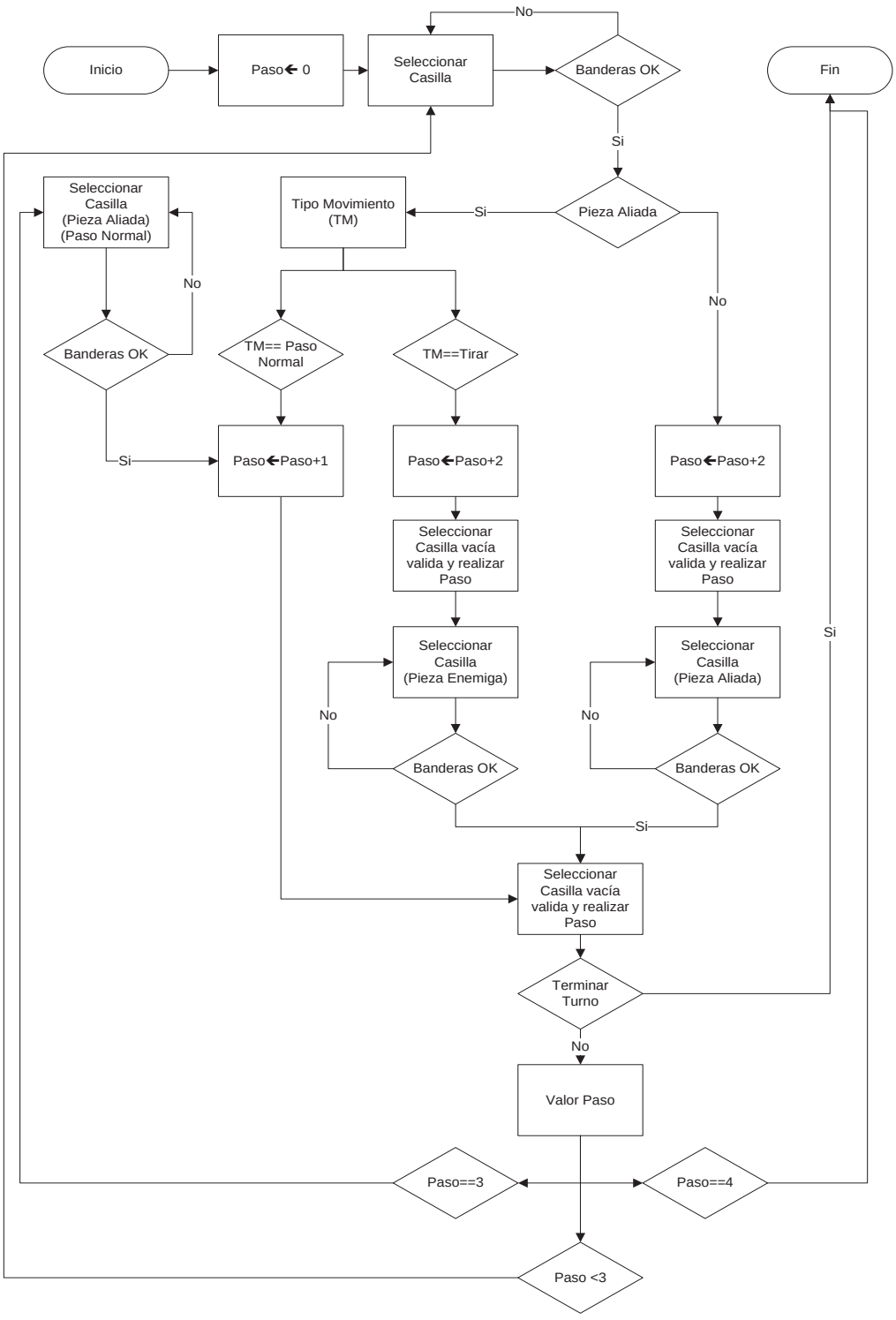

Figura 4.4: Diagrama de Flujo de una jugada

4.1.4 Movimientos y Generador de movimientos

Otro aspecto a considerar es el movimiento de las piezas, si bien es cierto que la mayoría de las piezas tienen el mismo patrón de movimientos hay que considerar que los Conejos no pueden retroceder en un movimiento normal, pero si lo pueden hacer en el caso de ser tirados o empujados por una pieza enemiga tal como lo muestra la ***tabla 4.3***, la única restricción que se debe considerar para el movimiento normal es la salvedad de los Conejos.

Tabla 4.3: Posibles direcciones para pasos normales

Pieza	Arriba	Abajo	Izquierda	Derecha
Elefante Dorado	SI	SI	SI	SI
Elefante Plateado	SI	SI	SI	SI
Camello Dorado	SI	SI	SI	SI
Camello Plateado	SI	SI	SI	SI
Caballo Dorado	SI	SI	SI	SI
Caballo Plateado	SI	SI	SI	SI
Perro Dorado	SI	SI	SI	SI
Perro Plateado	SI	SI	SI	SI
Gato Dorado	SI	SI	SI	SI
Gato Plateado	SI	SI	SI	SI
Conejo Dorado	SI	NO	SI	SI
Conejo Plateado	NO	SI	SI	SI

Por otro lado, además de la restricción de los Conejos para los pasos, hay que considerar otros aspectos del juego como las piezas congeladas y las piezas capturadas. En la **figura 4.5** se muestra un diagrama de flujo que señala como se generan las jugadas utilizando como principal apoyo listas.

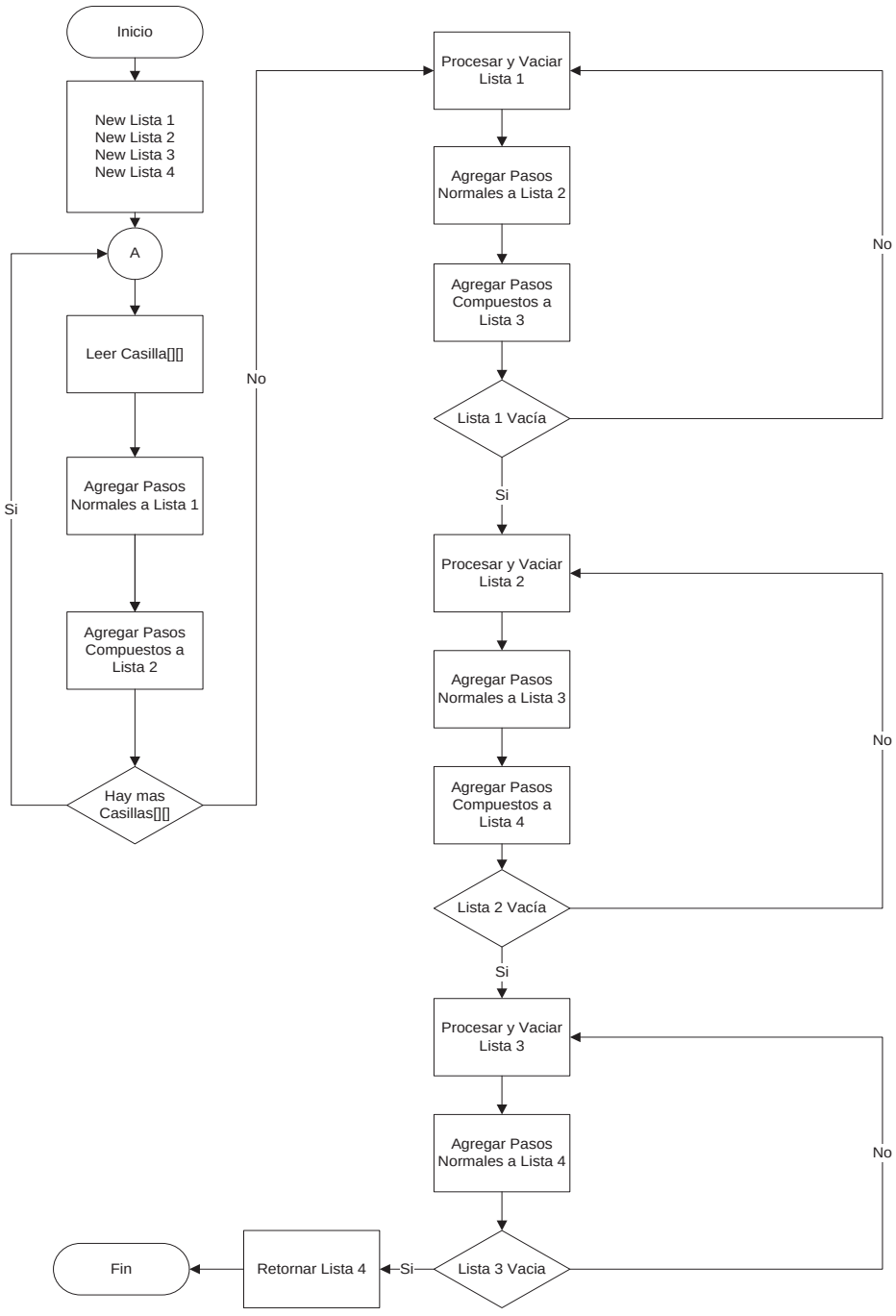

Figura 4.5: Diagrama de flujo del generador de movimiento

4.1.5 Trampas

En el caso de las trampas, estas casillas se deben verificar en todos los pasos para verificar si hay piezas que deben ser capturadas. Para esto, cuando se encuentre una pieza en alguna de estas casillas se utiliza el método aliado de la ***figura 4.6*** perteneciente a la Clase Tablero, que es un método bolean que determina si hay piezas aliadas a un paso de distancia.

```
boolean aliado(int x,int y){
  if (casillas[x][y].getValor()>0){
    if (casillas[x+1][y].getValor()>0) {return true;}
    if (casillas[x-1][y].getValor()>0) {return true;}
    if (casillas[x][y+1].getValor()>0) {return true;}
    if (casillas[x][y-1].getValor()>0) {return true;}
    }
  if (casillas[x][y].getValor()<0){
    if (casillas[x+1][y].getValor()<0) {return true;}
    if (casillas[x-1][y].getValor()<0) {return true;}
    if (casillas[x][y+1].getValor()<0) {return true;}
    if (casillas[x][y-1].getValor()<0) {return true;}
    }
return false;
  }
```

Figura 4.6: Método aliado de la Clase Tablero

4.1.6 Función de búsqueda

El diagrama de flujo de la ***figura 4.7***, explica a grandes rasgos, la implementación de la poda Alfa Beta, de una manera recursiva, como se aprecia en el diagrama, esta función necesita 4 parámetros, el primero es un nodo raíz, que corresponde a la estructura de datos que contiene toda la información relevante, el segundo parámetro es un límite que indica hasta que profundidad se quiere llegar, el tercer parámetro corresponde a ***alfa***, y el cuarto a ***beta.***

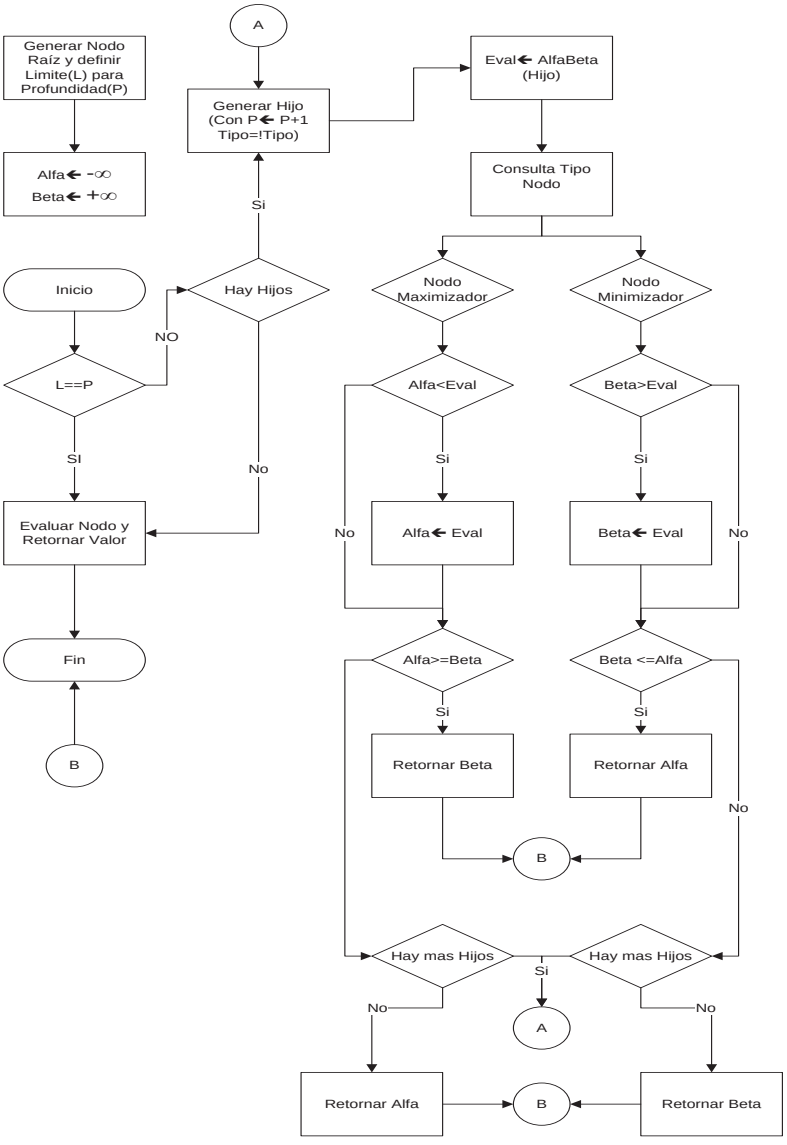

Figura 4.7: Diagrama de flujo de datos de Alfa Beta

4.1.7 Diagrama de Clases

En la ***figura 4.8***, se muestra un diagrama de clases del prototipo, donde se pueden apreciar los atributos y métodos que le permiten al prototipo manejar los procesos necesarios para que éste funcione. La clase NodoAB, es la que se encarga de realizar la poda AlfaBeta, y la clase Nodo la que guarda los pasos de la jugada.

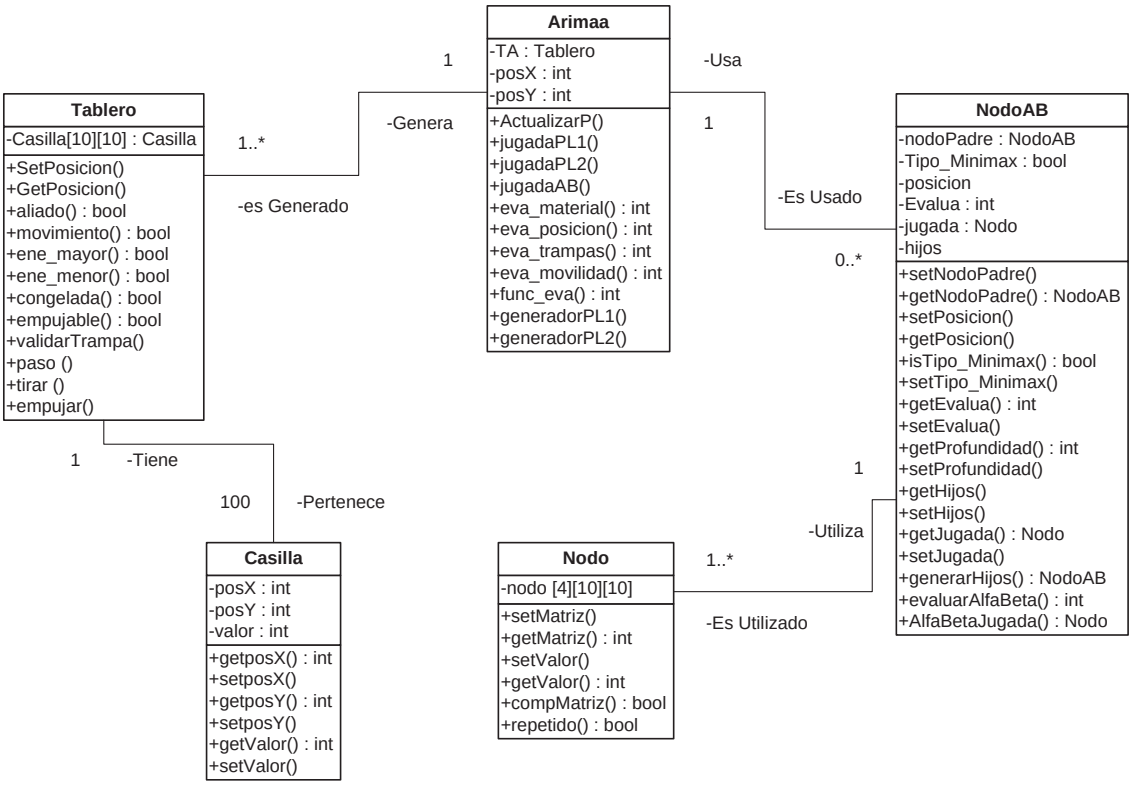

Figura 4.8: Diagrama de clases del Prototipo Arimaa

4.2 Testeo y evaluación del Prototipo

Una vez compilado el prototipo, este se sometió a pruebas a fin de determinar si resuelve o no el problema planteado en forma satisfactoria. Para ello se le suministró datos de prueba. Un programa codificado y compilado no garantiza que funcione correctamente. Por eso se debe depurar realizando pruebas continuas con datos y respuestas conocidas, verificando todas las posibles alternativas del programa y sus respuestas y haciendo el mayor número de variantes con sus combinaciones, a fin de determinar si resuelve o no el problema planteado en forma apropiada.

Las pruebas que se suelen aplicar a un programa son de diversa índole y generalmente dependen del tipo de problema que se está resolviendo. Comúnmente se inicia la prueba introduciendo datos válidos, inválidos e incongruentes y observando cómo reacciona en cada ocasión.

4.2.1 Método de la Caja Blanca

En programación, se denomina cajas blancas a un tipo de pruebas de software que se realiza sobre las funciones internas de un módulo. Así como las pruebas de caja negra ejercitan los requisitos funcionales desde el exterior del módulo, las de caja blanca están dirigidas a las funciones internas. Entre las técnicas usadas se encuentran; la cobertura de caminos, pruebas sobre las expresiones lógico aritméticas, pruebas de camino de datos y comprobación de bucles.

En los sistemas orientados a objetos, las pruebas de caja blanca pueden aplicarse a los métodos de la clase, pero según varias opiniones, ese esfuerzo debería dedicarse a otro tipo de pruebas más especializadas (un argumento podría ser que los métodos de una clase suelen ser menos complejos que los de una función de programación estructurada).

En el caso del prototipo Arimaa, los métodos de las clases son bastantes claros y muy específicos, por lo que se probaron a medida que se fueron implementando, la revisión de los bucles e iteraciones es hasta trivial. Para el caso de los métodos menos triviales como la jugada del jugador, generador de movimientos, y Alfa Beta, se utilizó la ayuda de los diagramas de flujos que aparecen en la sección “Implementación”. Por esta razón se dio más importancia a las pruebas de caja negra.

4.2.2 Método de la Caja Negra

A la mayor parte de los usuarios no les interesa los detalles del funcionamiento de un programa; lo único que desean es conseguir resultados. Es decir, desean tratar el programa como una caja negra a la cual se le introducen datos de entrada y se obtienen datos de salida esperados. De ahí el nombre de este método. De manera análoga, los datos de prueba se escogen atendiendo a las especificaciones del problema, sin importar los detalles internos del programa, a fin de verificar que éste corra bien. Una buena selección de casos de prueba deben incluir:

- Valores fáciles y realistas
- Valores extremos
- Valores ilegales

4.2.3 Pruebas

Para testear el prototipo de Arimaa en el turno del jugador se consideraron 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 4 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 o paso normal
E: Empujar
T: Tirar

Estas situaciones también se probaron en los límites del tablero. Que corresponde a la fila 1, la fila 8, la columna A y la columna H.

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

- 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 igual o más fuertes
- Se intentó tirar piezas igual o más fuertes

Para poder testear el generador de movimientos, se implementó un segmento de código, que además de contar los movimientos, muestra por pantalla las posibles jugadas. Considerando que el generador solamente considera turnos de cuatro pasos, sólo es fácil comprobar el valor esperado de piezas solitarias, como se aprecia en la ***tabla 4.4***. Aunque se realizaron muchas pruebas no se dejó constancia de todas.

Tabla 4.4: Algunos datos de prueba del generador de movimiento[8]

Casilla	Pieza	Movimientos esperados	Movimientos obtenidos
A1	Elefante	9	9
A2	Elefante	11	11
A3	Elefante	12	12
A4	Elefante	14	14
A1	Conejo Dorado	9	9
A2	Conejo Dorado	9	9
A3	Conejo Dorado	8	8
A4	Conejo Dorado	9	9
A5	Conejo Dorado	8	8
A6	Conejo Dorado	6	6
A7	Conejo Dorado	5	5
A8	Conejo Dorado	3	3

Para probar el funcionamiento de Alfa Beta, se utilizaron puzles[9] obtenidos desde la página oficial de Arimaa. A continuación desde la ***figura 4.9*** hasta la ***figura 4.16***, se mostrará, las soluciones que encontró el prototipo para los diferentes puzles, la imagen de la izquierda de cada figura es la posición inicial y la imagen de la derecha la solución que se encontró. En la descripción de la figura se indica el turno (el cual para todos los ejemplos corresponde al jugador dorado), y las instrucciones del puzle.

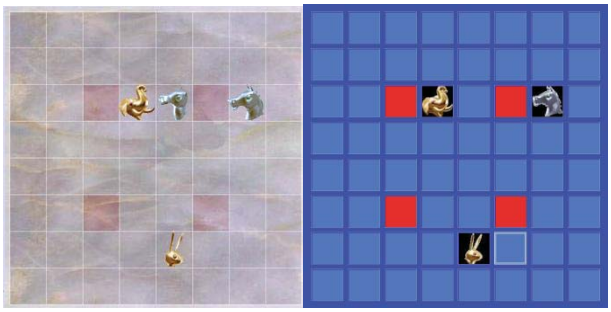

Figura 4.9: Puzle 1 Gold "Trap the silver camel"

[8] Hay que recordar que sólo los Conejos tienen una restricción de movimiento, y sólo en pasos normales

[9] http://arimaa.com/arimaa/puzzles/list.cgi

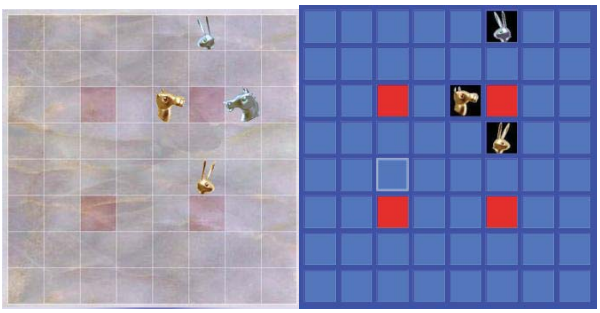

Figura 4.10: Puzle 2 Gold "Trap the silver horse"

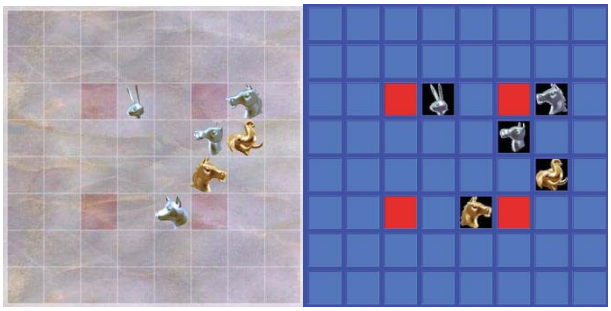

Figura 4.11: Puzle 3 Gold "Trap any silver piece"

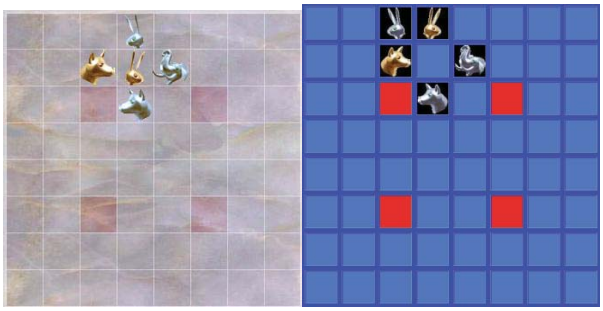

Figura 4.12: Puzle 4 Gold "Get the rabbit to goal"

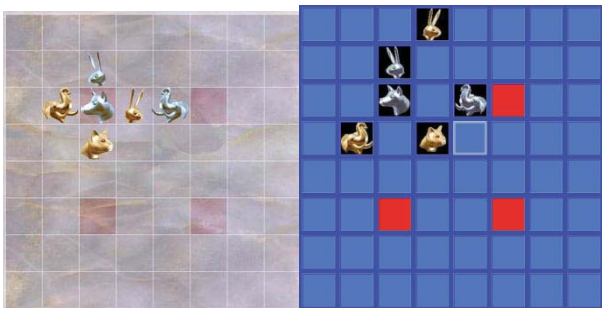

Figura 4.13: Puzle 5 Gold "Get the rabbit to goal"

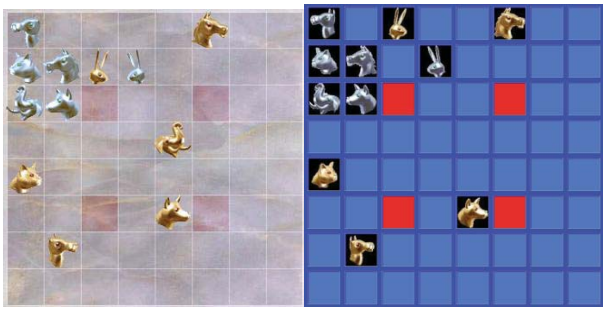

Figura 4.14: Puzle 6 Gold "Get the rabbit to goal"

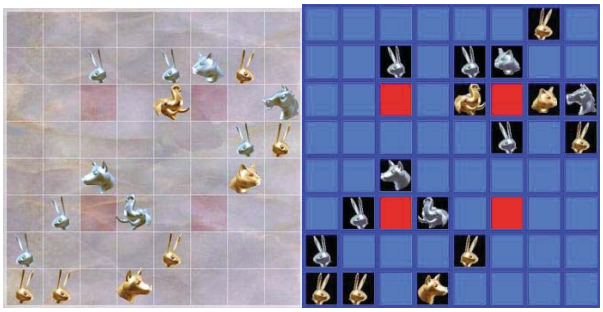

Figura 4.15: Puzle 7 Gold "Gold to move and win in one"

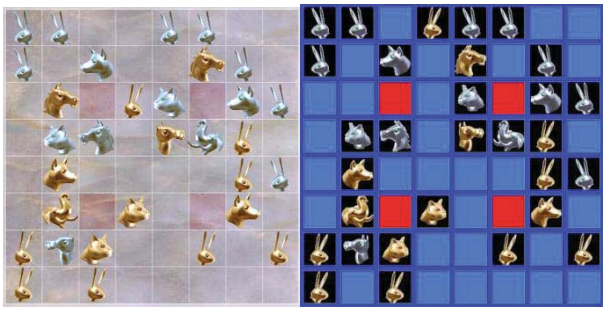

Figura 4.16: Puzle 8 Gold "Gold to play and reach goal in one"

En la ***tabla 4.5***, se midió el tiempo de respuesta que le tomó al prototipo aplicar Alfa Beta, para encontrar la solución a los puzles con diferentes profundidades, en la ***tabla 4.6*** se indica el número de jugadas posibles que encontró el generador de movimientos.

Tabla 4.5: Evaluación de los tiempos para los puzles (milisegundos)

Puzle	Alfa Beta 1	Alfa Beta 2	Alfa Beta 3
Puzle 1	484	43403	OM
	481	44974	OM
	482	42743	OM
	481	43830	OM
	497	43089	OM
Puzle 2	261	16244	53816
	240	16451	53137
	238	16568	53941
	229	16094	53596
	235	16172	53799
Puzle 3	164	OM	OM
	157	OM	OM
	161	OM	OM
	155	OM	OM
	160	OM	OM
Puzle 4	85	30662	OM
	75	30885	OM
	90	30183	OM
	80	30614	OM
	67	30672	OM
Puzle 5	141	164136	OM
	137	164622	OM
	142	164097	OM
	151	164034	OM
	139	163805	OM
Puzle 6	179227	OM	OM
	180801	OM	OM
	180921	OM	OM
	180379	OM	OM
	179888	OM	OM
Puzle 7	67828	TO	OM
	67848	TO	OM
	67623	TO	OM
	67757	TO	OM
	67541	TO	OM

Donde:

OM: Out of Memory, es un error por falta de memoria.

TO: Time out, en este caso, aunque la consola no lanzó el error por falta de memoria, se dejó correr el proceso por un tiempo prudente (más de 5 horas) sin que entregara una respuesta.

Tabla 4.6: Cantidad de posibles jugadas

Puzle	Jugadas Dorado	Jugadas Plateado
Puzle 1	245	28
Puzle 2	136	113
Puzle 3	159	131
Puzle 4	13	415
Puzle 5	145	359
Puzle 6	3980	768
Puzle 7	2530	7414
Puzle 8	17639	10181

4.2.4 Configuración del prototipo y resultados

Como etapa final de este proceso, hay que dejar expresado los resultados, si bien es cierto que el producto final es un programa, hay que tomar decisiones, para optimizar su rendimiento. En base a las pruebas y resultados, anteriores se configuró una serie de parámetros, como los pesos en la función de evaluación, y la vecindad de la ventana de aspiración.

Para terminar de configurar el programa, se buscó la forma de obtener una respuesta de una manera más rápida, dado que el tiempo que se demora el programa en realizar una jugada, cuando se encuentran todas las piezas en el tablero, es demasiado. Para la primera versión del programa prototipo, se decidió atacar este problema, restringiendo las jugadas que entrega el generador de movimiento, con la ayuda de la ***tabla 4.7***, y la ***tabla 4.8***, en las que se evaluaron distintas posibilidades, se decidió que se utilizará un límite variable que fluctuará en un inicio entre 1000 y 5000 jugadas, y que se regula en base al tiempo en que se demora en un turno.

Tabla 4.7: Evaluación bajo distintos escenarios 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

Tabla 4.8: Evaluación bajo distintos escenarios 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

4.2.4.1 La función de Evaluación

Para afinar esta función se ajustaron los parámetros, para poder resolver satisfactoriamente, los puzles que sirvieron de prueba. Las principales consideraciones fueron lograr la captura de piezas y la llegada de los Conejos a la meta. Para esto se establecieron las siguientes relaciones:

Captura de Pieza:

$W_1\Delta f_{1min} > W_2\Delta f_2 + W_3\Delta f_{3+} W_4\Delta f_4$

En esta relación matemática se ponen restricciones para determinar que una captura de pieza es más importante que las variaciones del resto de los criterios.

Condición de Victoria:

$W_1\Delta f_{1max} < W_2\Delta f_{2max}$

Al igual que en el caso anterior, se definió que la captura de pieza es más importante, hay un caso en particular en que esto no debe ser así, y es cuando un Conejo llega a la meta, que es una condición de victoria, por lo que se establece que la variación máxima de capturar piezas en un turno debe ser menor que la llegada de un Conejo a la meta.

Ahora para poder calibrar los pesos, hay que tomar valores para las variaciones, obteniendo los siguientes valores:

Δf_{1min}= 2 (captura de un Conejo)

Δf_{1max}=30 (captura de Elefante y Camello)

Δf_{2max} = 1000 (Conejo en la meta)

Δf_2 = 24 (sacar 4 piezas de cuadros de trampa)

Δf_3=22 (Mejorar ubicación de Elefante y Camello)

Δf_4=12 (Descongelar 4 piezas)

Si a lo anterior se le agrega la restricción $w_1>w_2>w_3>w_4>0$ como condición de importancia de los criterios, se puede calcular el valor mínimo de los pesos para que se cumplan las restricciones con lo que la función de evaluación con sus nuevos pesos queda:

$$\boldsymbol{f.e. = 64f_1 + 3f_2 + 2f_3 + f_4}$$

4.2.4.2 Ventana de Aspiración

Para poder definir los límites, se partió con una ventana, que consideró los límites alfa = ***f.e.***-0 y beta= ***f.e.***+0. Esta ventana era muy pequeña, por lo que se incrementó gradualmente los valores, hasta llegar finalmente alfa = ***f.e.***-12000 y beta= ***f.e.***+12000. Aunque se pudo utilizar una ventana más pequeña, se prefirió dar un margen para tener más seguridad.

Capítulo 5

5.1 Conclusiones

A modo de conclusión, puede decirse que no hay que dejarse llevar por las apariencias, Arimaa pese a ser un juego de tablero con reglas muy simples, presenta una gran dificultad a la hora de ser implementado. El análisis de Arimaa permite comprender la complejidad del juego y entender mejor los desafíos que plantea su autor.

Arimaa es un juego que resulta muy interesante, ya que es el primer juego pensado para ser difícil que un computador lo juegue bien, si a esto se le agrega que es un juego relativamente nuevo, y no muy conocido se suma la dificultad de que no existen estudios teóricos en profundidad. Lo cual se hace notar en el desafío de Arimaa, en el que aun no se logra crear un programa que derrote a los mejores exponentes humanos para ganar el desafío.

Otro aspecto importante a tomar en consideración, es que pese a la similitud y a las comparaciones que se puedan hacer entre el ajedrez y Arimaa, al aplicar las mismas soluciones no se obtiene el mismo nivel de efectividad en los resultados.

La teoría señala que el algoritmo Alfa Beta, mejora el rendimiento del algoritmo Minimax, pero esta mejora necesita un ordenamiento en la lista de jugadas que permita hacer la mayor cantidad de podas, a diferencia del ajedrez en Arimaa no hay jaques, y las capturas de piezas son menores, por lo que se hace más complejo buscar la manera de ordenar la lista. Por el lado positivo al no existir tanta captura de piezas el efecto horizonte no es tan relevante.

En la etapa de la implementación del prototipo, las mayores complicaciones fueron, la validación de los turnos y el generador de movimientos, que conceptualmente fueron temas simples, pero a la vez la parte fundamental de la implementación.

En base a las pruebas, se lograron realizar las estimaciones necesarias para afinar la función de evaluación, establecer los límites de la ventana de aspiración, terminar las validaciones, y ver las limitaciones del prototipo en relación al tiempo que demora en realizar las búsquedas debido a la gran cantidad de posibles movimientos.

Finalmente, se pueden considerar cumplidos los objetivos propuestos. Ya que en base a los estudios del juego de tablero Arimaa, se logró establecer y asignar valor a los criterios utilizados en la función de evaluación, lo que permitió la aplicación de Minimax con poda Alfa Beta. Obteniendo como resultado un prototipo de programa Arimaa capaz de jugar partidas enteras.

Referencias

[1] Arimaa the next Challenge: Arimaa Puzzles, Disponible vía web en http://arimaa.com/arimaa/puzzles/list.cgi

[2] Cheoss "chess engine": http://www1.freewebs.com/cheoss/index.html

[3] Christ - Jan Cox , Analysis and Implementation of the Game Arimaa, Marzo 2006

[4] David Fotland, Building a World Champion Arimaa Program, 2004

[5] Haizhi Zhong, Building a Strong Arimaa-playing Program, Septiembre 2005

[6] Jorge Edilberto Alvarado Valderrama, Lesly Rodríguez Pinillos, Aplicación del algoritmo poda alpha-beta para la implementación del juego "ajedrez"

[7] José A. Mañas, Prueba de programas, Marzo 1994, Disponible vía Web en http://www.lab.dit.upm.es/~lprg/material/apuntes/pruebas/testing.htm

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

[9] Russell-Norvig, Inteligencia Artificial, Un enfoque Moderno Capitulo 6 "Búsqueda entre adversarios", 2004

[10] Stefano Carlini, Arimaa: From Rules to Bitboard Analysys, Diciembre 2008

[11] Wikipedia La Enciclopedia Libre: "Arimaa", Disponible vía web en http://es.wikipedia.org/wiki/Arimaa

[12] Wikipedia La Enciclopedia Libre: "Complejidad en los juegos", Disponible vía web en http://es.wikipedia.org/wiki/Complejidad_en_los_juegos