Un Jugador de Go Basado en Técnicas de Aprendizaje Automático

Informe de Proyecto de Grado

Raúl Garreta Tompson

Tutores:
MSc Diego Garat
MSc Guillermo Moncecchi

Página Web:
`http://www.fing.edu.uy/~pgagrego/`

Facultad de Ingeniería,
Universidad de la República,
Montevideo,
Uruguay.

Mayo, 2006

Resumen

Desde los comienzos de la investigación en *Inteligencia Artificial (IA)*, los juegos han sido utilizados como campo de investigación para la prueba y desarrollo de nuevos algoritmos, técnicas y heurísticas para la resolución de problemas.

Esto se debe a que brindan reglas claras del problema, pero manteniendo una complejidad lo suficiente como para hacer que no sea trivialmente resuelto.

Luego del Ajedrez, el Go se ha llevado la mayoría de los esfuerzos para su resolución. Sin embargo, al día no existe ninguna máquina que pueda derrotar a un jugador humano profesional de Go. Por este motivo, la resolución del Go es considerada como uno de los retos en el campo de la IA.

Gran parte de la complejidad del Go se debe a que dada una posición, existen muchos movimientos posibles, por lo que analizarlos todos con gran profundidad y corto tiempo de respuesta es muy difícil. Otro problema reside en la dificultad de encontrar una función que permita evaluar una posición y decidir cuál movimiento es más favorable.

En este proyecto se propone investigar y diseñar un jugador de Go basado en técnicas de *Aprendizaje Automático*.

Por un lado, se desarrolla un sistema de aprendizaje de patrones basado en *Inducción de Programas Lógicos*. Los patrones inducidos se utilizan para generar un conjunto reducido de movimientos adecuados para analizar cada vez que se decide que movimiento realizar.

Por otro lado, se desarrolla una *Red Neuronal* cuyo objetivo es aprender una función que evalúe posiciones de Go. Para esto se utiliza *Diferencia Temporal* como algoritmo de aprendizaje y *Backpropagation* como algoritmo de ajuste de la red.

Ambos acercamientos son combinados en la implementación de un jugador artificial de Go. Los resultados son promisorios: se ha logrado una buena performance ante otros jugadores artificiales.

Palabras Clave: Inducción de Programas Lógicos, Redes Neuronales, Aprendizaje por Diferencia Temporal, Go.

Dedicatoria y Agradecimientos

Dedico este proyecto a mi familia, mi novia y mis amigos, que en todo momento han confiado en mí y me han dado un gran apoyo. Sin su ayuda y palabras de aliento, los problemas serían mucho más difíciles de resolver.

Agradezco en especial a mis tutores, Diego Garat y Guillermo Moncecchi por haberme apoyado y con excelente disposición ser mis tutores. Ambos aceptaron ser tutores de un proyecto propuesto por mí mismo, donde quedaban muchas interrogantes de si sería posible obtener algún resultado. Los dos compartieron su valioso tiempo y conocimiento conmigo para guiar este proyecto.

Es un orgullo haber podido llevar adelante un proyecto científico en el campo de mi interés: Inteligencia Artificial, Aprendizaje Automático y Robótica.

Tabla de Contenidos

Capítulo 1

Introducción

En este capítulo se plantea el problema y se motiva el trabajo, para luego establecer los objetivos a alcanzar. Por último se describe la organización general del documento.

1.1. Planteo del Problema y Motivación

¿Por qué el Go?

Los juegos son un excelente campo de pruebas de algoritmos, y el Go, en particular, reúne varias características interesantes como problema a resolver. Por un lado es un reto en el campo de la computación, ya que no se ha logrado desarrollar un jugador artificial que le gane a un jugador profesional. Por otro, por tratarse de un juego de tablero, sus reglas son claras, lo que ayuda a aislar los resultados del «ruido» y concentrarse casi completamente en la aplicación de las técnicas para su resolución.

El Ajedrez ha sido el juego en el cual se han dedicado la mayor cantidad de esfuerzos en crear una máquina que derrote al mejor jugador de Ajedrez humano del mundo. Este objetivo fue resuelto en el año 1997 cuando el programa Deep Blue de IBM derrotó al campeón mundial de Ajedrez Gary Kasparov. Un hecho muy interesante, dado que una máquina derrotó al campeón mundial en una disciplina a la que se asocia el uso intensivo de inteligencia para dominarla.

Luego del Ajedrez, el Go ha sido el juego que más se ha llevado los esfuerzos para su resolución [Müller, 2002]. La diferencia es que aún no se ha encontrado una solución aceptable; al día de hoy no existe ninguna máquina que pueda derrotar a un jugador humano profesional de Go. Por este motivo, la resolución del Go ha tomado el lugar del Ajedrez como reto en el campo de la Inteligencia Artificial, [Selman et˜al., 1996]. Como símbolo del ámbito competitivo, se encuentra el caso de organizaciones como la Fundación Ing [Bouzy and Cazenave, 2001], que ofreció un premio de aproximadamente 1 millón de dólares a quien creara un jugador artificial que ganara a un jugador humano profesional, premio que nunca tuvo dueño.

El acercamiento clásico en la resolución de juegos ha sido tradicionalmente la aplicación de técnicas de búsqueda. Básicamente, la estrategia consiste en observar varios movimientos hacia adelante. Esto se puede modelar como un árbol, llamado *árbol de juego*, cuyos nodos representan posiciones de tablero (estados), y las aristas

representan movimientos (acciones). A partir de este árbol, la estrategia de juego simplemente consiste en elegir la rama por la cual se llega a la posición final más conveniente.

En la gran mayoría de los casos se han obtenido buenos resultados con diseños simples y es por eso que las técnicas de búsqueda han tenido gran popularidad. Teóricamente en cualquier juego *determinístico*[1] como el Go, el árbol de juego puede ser expandido completamente, y de esta forma, elegir el mejor movimiento. Sin embargo, en el caso del Go tenemos un promedio de *factor de ramificación*[2] de 235 y un promedio de profundidad del árbol de 300, lo que significa un árbol de 235^{300} nodos [Burmeister and Wiles, 1994]. En la práctica, no es posible desarrollar completamente este árbol con la capacidad de una computadora actual, debido a los límites en capacidad de memoria y tiempo de procesamiento. Por lo tanto, no es suficiente aplicar una técnica exclusivamente de búsqueda en el Go.

Para intentar reducir el árbol de juego, en la mayoría de los casos se utiliza una *función de evaluación*. Una función de evaluación pretende asignar valores estáticos a situaciones dadas de un juego en base a ciertos atributos, el valor devuelto será mayor mientras más favorable sea la posición. La idea es utilizar funciones de evaluación para poder evitar recorrer todo el árbol de juego. Supongamos que encontramos una función de evaluación ideal, la cual asigna valores a posiciones de un tablero. Si tuviésemos esta función, una posible forma de resolver el problema sería: dado una posición inicial, generamos todas las posibles posiciones a las cuales es posible llegar mediante un movimiento (se generan todos los hijos del nodo inicial), evaluamos cada una de las posiciones con la función de evaluación y por último elegimos el movimiento que nos lleve a la posición con mayor valor. En muchos problemas simples este método puede ser utilizado con éxito. Sin embargo, en el caso del Go el diseño de una función de evaluación adecuada deja de ser una tarea trivial. Hasta ahora no se ha encontrado una función que evalúe correctamente tableros de Go y que al mismo tiempo sea eficientemente computable.

En resumen, gran parte de las dificultades del Go residen en el alto factor de ramificación de su árbol de juego y en la dificultad de encontrar una función de evaluación que permita decidir que movimiento es más favorable dada una posición. Es necesario diseñar un modelo que no solo resuelva el problema, sino que también sea posible llevarlo a la práctica respetando las restricciones de las computadoras actuales.

[1] Juegos en donde no interviene la aleatoriedad, el resultado depende únicamente de las habilidades de los jugadores

[2] Cantidad promedio de hijos por nodo.

¿Por qué utilizar Aprendizaje Automático?

En particular, en este proyecto interesa la aplicación de técnicas de *Aprendizaje Automático* (AA). AA es una subespecialidad de la *Inteligencia Artificial* que se ocupa del desarrollo de métodos que permitan a un programa aprender a través de la experiencia o extraer conocimiento de ejemplos. Se puede ver como un proceso por el cual un programa mediante el uso de algún algoritmo, logra mejorar su performance con el paso del tiempo [Mitchell, 1997]. En principio puede ser una gran herramienta para resolver problemas en donde no existe una solución analítica conocida, o el diseño e implementación de la misma están fuera de las capacidades.

El uso intensivo de conocimiento en la creación de un programa que juegue al Go es una técnica utilizada por la mayoría de programas más fuertes [Bouzy and Cazenave, 2001]. Este uso intensivo de conocimiento trae como consecuencia una gran dificultad asociada a la codificación, generación y mantenimiento manual. Esto significa un esfuerzo enorme y un alto riesgo de introducción de errores.

Por otro lado, cuando un programador intente mejorar su programa agregando más conocimiento, este nuevo conocimiento puede llegar a interactuar negativamente con conocimiento previo, produciendo resultados incorrectos. Más aún, agregar conocimiento analíticamente implica el requisito de que el programador debe ser un buen conocedor del Go. Además, debería lidiar con las dificultades de encontrar reglas sin pasar por alto casos excepcionales que podrían dar malos resultados.

En este proyecto se utilizaron técnicas de aprendizaje automático para la generación y aplicación automática de conocimiento en un jugador de Go. En particular, fueron utilizadas dos técnicas: aprendizaje de patrones de juego y aprendizaje de funciones de evaluación. A continuación describiremos las ideas de ambos conceptos.

Aprendizaje de Patrones de Juego

Los *patrones* son maneras de representar conocimiento y en el caso del Go están presentes en casi todos los prototipos. Un patrón puede verse como una regla compuesta por precondiciones y postcondiciones. Si la posición del tablero que estamos analizando cumple con las precondiciones, entonces el patrón puede ser usado. Las postcondiciones aconsejan que movimiento realizar en una situación donde se cumplen las precondiciones. En la figura 1.1 [Ramon et˜al., 2000] se muestra un ejemplo de un patrón, las precondiciones consisten en una conjunción de condiciones sobre ciertas intersecciones del tablero (relativas al movimiento aconsejado en la postcondición) del estilo «*posición = valor*».

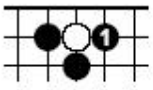


Figura 1.1: SI (-3,1)=vacío y (-2,0)=negro y (-2,1)=vacío y (-2,2)=borde y (-1,1)=negro y (-1,0)=blanco y (-1,1)=vacío y (0,1)=vacío y (1,0)=vacío ENTONCES negro juega en (0,0).

Tradicionalmente los patrones han sido generados analíticamente, tomados de expertos y codificados a mano. Recientemente se han propuesto acercamientos don-

de la generación de los patrones se realiza en forma automática, lo que reduciría el trabajo de codificación y el riesgo de introducción de errores. Un programa puede contener unos pocos patrones o llegar a manejar miles de ellos, esto depende obviamente en el diseño particular.

En conclusión, un patrón encapsula conocimiento que puede ser muy útil para un programa de Go. Si se cumplen las precondiciones de un patrón, entonces se podría evitar la búsqueda innecesaria en el árbol de juego. El patrón actúa como una especie de jugada precalculada, si el patrón puede ser usado y movemos en el lugar que éste indica, sabremos cual será el resultado final sin realizar una búsqueda en el árbol.

En este proyecto se pretende utilizar técnicas de aprendizaje automático para la generación automática de patrones de juego de Go. Los patrones obtenidos podrían luego ser usados para, dada una posición particular, obtener un conjunto reducido de buenos movimientos. De esta forma se pretende resolver uno de los problemas del Go: disminuir el gran factor de ramificación.

Hasta el momento no hay trabajos que hayan intentado la obtención de patrones de forma genérica, esto es, patrones acerca de todos los movimientos del Go. Casi todos los trabajos investigados en el estado de arte pretenden la obtención de reglas para casos particulares [Ramon et˜al., 2000, Kojima and Yoshikawa, 1999].

Aprendizaje de una Función de Evaluación

En muchos juegos se han encontrado funciones de evaluación de tableros adecuadas. Típicamente la situación se presenta de la siguiente forma: el programador está provisto de bibliotecas de rutinas que computan propiedades importantes de una posición (ejemplo: el número de piezas de cada tipo, el tamaño del territorio controlado por cada uno, etc). Luego, lo que queda por conocer es cómo combinar estas piezas de conocimiento y cuantificar su importancia relativa en la evaluación global. En el Go esta tarea es difícil de hacer analíticamente ya que requiere un amplio conocimiento del dominio para saber las importancias relativas de cada atributo dependiendo de la situación particular.

El ajuste automático de los pesos de la función de evaluación es uno de los problemas de aprendizaje más estudiados en los juegos [Fürnkranz, 2001]. En este proyecto se propone la utilización de algoritmos de aprendizaje automático para la resolución del segundo gran problema: el ajuste de una función de evaluación adecuada para el Go.

1.2. Objetivos

Los objetivos de este proyecto consisten en investigar y diseñar un jugador de Go basado en técnicas de Aprendizaje Automático. El trabajo se divide en tres etapas importantes.

En primer lugar, diseñar un sistema de aprendizaje de patrones de juego de Go. El objetivo de este sistema es su utilización como generador de un conjunto reducido de opciones a la hora de seleccionar un movimiento.

En segundo lugar, diseñar un sistema de aprendizaje de una función de evaluación para el Go, la función de evaluación es un punto muy importante necesario para la selección de movimientos.

Estos dos diseños serán utilizados finalmente en la implementación de un jugador de Go junto con otras técnicas tradicionales de resolución de juegos.

Se combinaran técnicas de aprendizaje automático «tradicionales» en la resolución de juegos, como ser el aprendizaje de funciones de evaluación basado en un acercamiento conexionista, junto con aprendizaje de patrones basado en un acercamiento simbólico. Ambas técnicas se basan en dos paradigmas de aprendizaje distintos.

1.3. Organización del Documento

El resto del documento se organiza de la siguiente forma:

En el capítulo 2 se da una breve introducción al Go, sus reglas, sus objetivos, conceptos importantes y un estudio de complejidad. Luego se introducen las técnicas tradicionales en la resolución de juegos para luego presentar el diseño de jugador.

En el capítulo 3 se presenta el diseño y evaluación de uno de los componentes principales: aprendizaje de patrones basado en *Inducción de Programas Lógicos*.

En el capítulo 4 se presenta el diseño y evaluación del segundo componente esencial: una función de evaluación basada en *Redes Neuronales* y algoritmos de *Backpropagation* y *Aprendizaje por Diferencia Temporal*.

En el capítulo 5 se describen detalles relacionadas a la implementación y evaluación del jugador de Go.

Por último, en el capítulo 6 se establecen las conclusiones, se evalúan los resultados, se plantean dificultades y se mencionan aportes del trabajo y sus posibles extensiones.

Capítulo 2

El Juego del Go

En la sección 2.1 se realiza una breve introducción al juego del Go, sus objetivos, reglas y conceptos importantes. En la sección 2.2 se busca identificar y entender en cuáles características reside su dificultad a la hora de resolver el problema. En la sección 2.3 se realiza una introducción básica de las técnicas clásicas en la resolución de juegos que sirven como punto de partida del diseño del jugador. Por último, en la sección 2.4 se presenta el diseño propuesto de un jugador de Go.

2.1. Introducción al Go

El Go[1] es un juego de dos jugadores. El juego depende únicamente de la habilidad de los jugadores ya que no existen elementos de aleatoriedad como los dados o cartas. Es uno de los juegos más antiguos del mundo. Tiene sus orígenes en China y de acuerdo a las leyendas fue inventado alrededor del 2300 A.C. El juego fue mencionado por primera vez en escrituras Chinas que datan del año 625 A.C. Alrededor del siglo séptimo el juego fue importado a Japón donde obtuvo el nombre de Go.

El juego tiene turnos donde cada jugador (blanco o negro) pone sus fichas (piedras) en el tablero. Las piedras son puestas en las intersecciones de las líneas verticales y horizontales del tablero, incluidos los bordes y las esquinas. El tablero de Go es usualmente de 19x19, pero también se utilizan tableros menores de tamaños 9x9, 11x11 y 13x13 (ver figura 2.1).

[1] http://www.usgo.org/ http://www.igochile.cl/ http://www.britgo.org/ http://www.go.org.ar/ http://gobase.org/

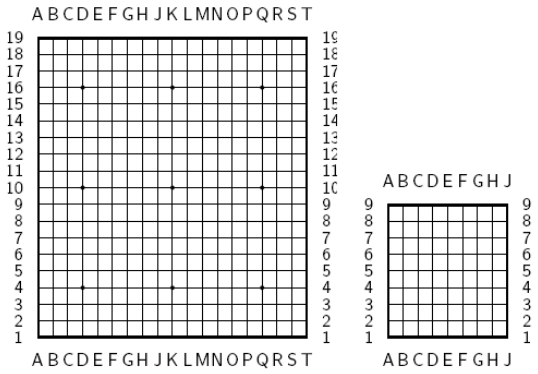


Figura 2.1: Tableros de Go de 19x19 y 9x9.

Objetivo del juego

El objetivo del juego es rodear espacios libres de ocupación para controlar un territorio o una suma de territorios mayor que los del oponente. Por lo tanto no se trata de meras ocupaciones físicas de piezas, sino de los espacios libres de ocupación rodeados por ellas. La ocupación física de las piezas no es más que un medio para alcanzar el fin de la conquista del territorio y de reducir el tamaño del territorio conquistado por el oponente.

Al inicio de la partida el tablero está completamente vacío, excepto en partidas con ventajas (*handicap*) concedidas por un jugador de categoría superior, aquí el jugador de menor nivel comienza la partida con piedras de su color puestas de ante mano en el tablero. Según la tradición el jugador que utiliza las piedras negras hace la primera jugada, el de las blancas la segunda y así sucesivamente, alternando las jugadas. Cada jugador, en cada jugada, sólo puede poner una piedra en una de las intersecciones vacías. Una vez en el tablero, la piedra no se moverá de allí, excepto cuando sea capturada y retirada por el contrario.

Fin de partida

Cuando ambos jugadores consideran que ya no existen territorios por disputar, la partida se da por terminada. Si uno de ellos no está de acuerdo y cree que todavía queda algún territorio por disputar, el juego puede seguir y el otro jugador puede pasar si lo desea o responder con un movimiento si lo cree oportuno. Luego de que se esté completamente de acuerdo en que la partida ha finalizado, se procede a contar los puntos. En una partida profesional, tal circunstancia la decidirá un juez. Una partida también puede terminar al abandonar el juego cualquiera de los dos jugadores reconociendo su derrota. Notese que la decisión de si una partida de Go ha finalizado no es tan obvia como por ejemplo en el Ajedrez donde tal situación se da cuando el rey está en jaque mate.

Conteo de Puntos

Luego de que la partida ha terminado es necesario contar los puntos para saber quién es el vencedor. Existen dos métodos de conteo, uno se basa en el territorio y el otro en área. Los dos métodos comienzan por quitar las piedras muertas y agregarlas a los prisioneros. En el conteo basado en territorio utilizado por las reglas Japonesas, luego se cuenta el número de intersecciones rodeadas (territorio) mas el número de piedras del oponente capturadas (prisioneros). En el conteo basado en área, utilizado por las reglas Chinas, se cuenta el número de intersecciones rodeadas mas el resto de las piedras del mismo color en el tablero. El resultado de los dos métodos es usualmente el mismo, salvo como máximo una diferencia de un punto.

Conceptos Importantes

Adyacencia Dos intersecciones son adyacentes si son contiguas. No se consideran piedras adyacentes a dos piedras en diagonal.

Bloque de piedras Un bloque de piedras es un conjunto de piedras del mismo color adyacentes entre si. Es un concepto fundamental, las piedras contenidas en un mismo bloque pasan a tener el mismo destino, es decir, si el bloque es capturado, todas las piedras del bloque serán capturadas. El bloque más pequeño y simple está compuesto por una sola piedra. La figura 2.2 muestra algunos ejemplos.

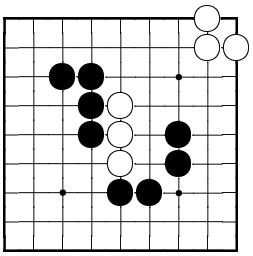


Figura 2.2: Ejemplo de bloques: 2 bloques blancos y 3 bloques negros.

Libertades Una libertad de una piedra es una intersección vacía adyacente a la piedra, es un concepto muy importante en el Go. El número de libertades de un bloque se obtiene como la suma de las libertades de cada piedra contenida en el bloque (sin repetir mismas libertades). El número de libertades de un bloque es el límite inferior en la cantidad de movimientos que necesita el adversario para poder capturar el bloque. Por lo tanto, un bloque con cero libertades está capturado, cuando un bloque es capturado, todas las piedras que lo componen son retiradas del tablero. En la figura 2.3 se muestran algunos ejemplos.

Atari Un bloque se encuentra en Atari si puede ser capturado en el siguiente movimiento del oponente, esto es, el bloque tiene una sola libertad. La figura 2.4 muestra algunos ejemplos.

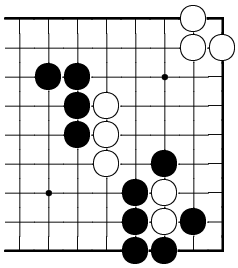


Figura 2.3: Ejemplo de libertades: el bloque negro de más a la izquierda tiene 7 libertades, el bloque blanco central tiene 6 libertades, el bloque blanco de la esquina superior tiene 5 libertades, el bloque blanco de la esquina inferior tiene 1 libertad (está en *atari*).

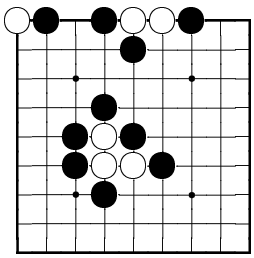


Figura 2.4: Ejemplo de bloques en atari, los 3 bloques blancos están en atari.

Conexión Si dos bloques son conectados pasan a formar un solo bloque, por lo que comparten las libertades y tienen mayor probabilidad de vivir. Buscar la conectividad entre los bloques es una estrategia muy importante, es más fácil de defender un bloque grande que muchos bloques pequeños no conectados (ver figura 2.5).

Ojo Una o más intersecciones rodeadas por piedras del mismo bloque. Ojos falsos son intersecciones rodeadas por piedras del mismo color, pero que pertenecen a diferentes bloques y no pueden ser conectados por un camino alternativo sin rellenar el ojo. Las intersecciones marcadas como e en la figura 2.6 son ojos. Los ojos tienen gran importancia a la hora de asegurar que los bloques no puedan ser capturados, ya que si un bloque posee dos ojos, este no puede ser capturado de ninguna forma. Si el oponente pusiera una ficha en alguno de los dos ojos, cometería un suicidio, ya que al bloque le quedaría otra libertad. Si el bloque en cambio tuviera un solo ojo, que represente su única libertad, entonces el oponente podría capturarlo ocupando su última libertad (salto con paracaídas).

Bloque vivo Un bloque está vivo si no puede ser capturado por el oponente. Esto es, no es necesario preocuparse por su defensa ya que su existencia es independiente de las acciones del oponente. Usualmente los bloques vivos tienen dos ojos. En la

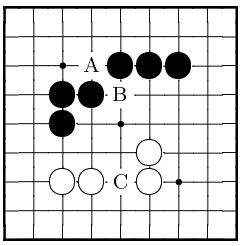


Figura 2.5: Ejemplo de conexiones. Si negro juega en A o B, conectará sus dos bloques. Si blanco juega en C, conectará sus dos bloques.

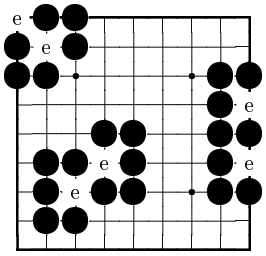


Figura 2.6: Ejemplos de ojos, marcados con *e*.

figura 2.6, los bloques están vivos.

Bloque muerto Un bloque está muerto si no puede escapar de ser capturado. Al final de una partida, los bloques muertos son quitados del tablero (ver figura 2.7).

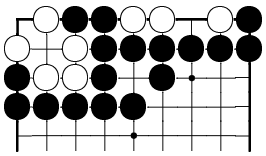


Figura 2.7: Los bloques blancos están muertos.

Cadena Una cadena es un conjunto de bloques que pueden ser conectados. Dos bloques pueden ser conectados si comparten más de una libertad. La figura 2.8 muestra un ejemplo.

Grupo Es un conjunto de bloques del mismo color «débilmente» conectados, que usualmente controlan un área del tablero. La figura 2.9 muestra un ejemplo.

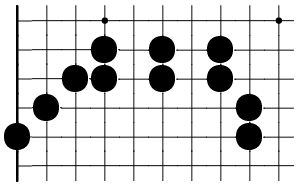


Figura 2.8: Ejemplo de una cadena compuesta por 6 bloques.

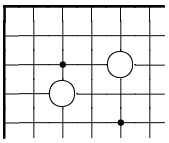


Figura 2.9: Ejemplo de un grupo.

Komi Un número predeterminado de puntos que serán agregados al puntaje final del jugador blanco. Es utilizado para compensar la ventaja que tiene el jugador negro al ser el que mueve primero. Un valor común de Komi utilizado entre jugadores de mismo nivel es 6.5 para Go 19x19 y 5.5 para Go 9x9.

Territorio Las intersecciones rodeadas y controladas por un jugador.

Piedras de Handicap Las piedras de handicap pueden ser puestas al inicio de la partida (en un tablero vacío), por el primer jugador de la partida para compensar la diferencia entre nivel de juego con el segundo jugador. La diferencia entre los grados de dos jugadores indica el número de piedras de handicap necesarias para que ambos jugadores tengan las mismas chances de ganar. Como máximo es posible colocar 9 piedras de handicap. Las piedras de handicap se colocan en las intersecciones marcadas con un punto en el tablero.

Prisioneros Piedras que son capturadas o muertas.

Jugadas prohibidas

En principio está permitido jugar en cualquier punto del tablero, pero existen dos jugadas prohibidas:

Suicidio Está prohibido cometer suicidio, esto es, ubicar una piedra en una posición que no captura ningún bloque del oponente y deja a su propio bloque sin libertades (ver figura 2.10).

Ko Dado que al capturar piedras las mismas son quitadas del tablero, es posible repetir posiciones previas del tablero. Por lo tanto, la repetición de posiciones debe ser evitada. El caso más común de repetición de una posición es el Ko de la figura

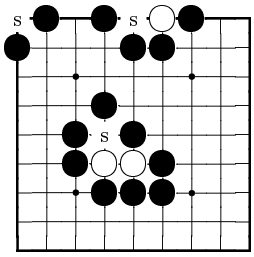


Figura 2.10: Ejemplos de suicidios, blanco cometería suicidio si jugara en cualquiera de las intersecciones marcadas con una *s*.

2.11. Negro captura la piedra blanca marcada al jugar en *a*. Luego blanco podría recapturar la piedra negra recién jugada al jugar en *b*. La misma secuencia se podría repetir indefinidamente. La regla de Ko prohíbe situaciones como esta, por lo que Blanco no puede jugar en *b* enseguida después de que Negro juega en *a*, la recaptura solamente la podría hacer luego de jugar un movimiento en otra posición del tablero (dejar pasar un turno).

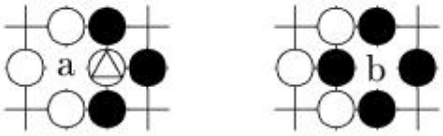


Figura 2.11: Ko básico.

2.2. Complejidad del Go

Categoría de juego

El Go entra en la categoría de juegos con las siguientes características generales:

- **Dos jugadores**.
- **Información perfecta**, los jugadores tienen acceso completo a la información del estado actual del juego en todo momento.
- **Determinístico**, el siguiente estado del juego está completamente determinado por el estado actual del juego y la acción del jugador, no está sujeto a eventos aleatorios tales como tirar dados.
- **Suma cero**, las ganancias o pérdidas de un jugador se equilibran con exactitud con las pérdidas o ganancias del otro jugador. Si se suma el total de las ganancias de los jugadores y se resta las pérdidas totales, el resultado es cero.

Aunque entra en la misma categoría que el Ajedrez, el Go posee varias características que hacen que sea difícil resolverlo exitosamente con las técnicas tradicionales de búsqueda en árboles de juego. Para entender esta dificultad, veremos en cuáles características radica.

Espacio de Búsqueda

Dadas sus propiedades, el Go puede ser representado mediante un grafo dirigido llamado *árbol de juego*. Recordemos que un árbol de juego, es un árbol cuyos nodos son estados del juego (posiciones de las fichas en un tablero) y sus nodos hijos son los estados a los que se puede llegar mediante una acción (movimiento). La raíz de este árbol, es el estado inicial (la posición inicial donde comienza el juego). Por lo tanto, las hojas del árbol serán los estados finales (posiciones finales, es decir, el juego terminó). Un *árbol de búsqueda* es una parte del árbol de juego que es analizada por un jugador (humano o máquina), este tiene su raíz en la posición que se está analizando. Cuando se dice que un nodo es expandido d veces, se refiere a que han sido analizadas hasta d posiciones adelante, es decir el árbol de búsqueda tiene como máximo una profundidad d. La figura 2.12 muestra un ejemplo (parcial) de un árbol de búsqueda para el juego *ta-te-ti*.

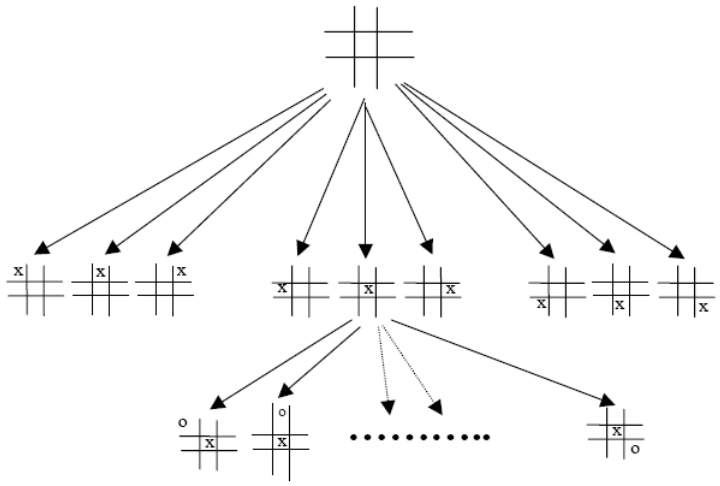


Figura 2.12: Ejemplo de un árbol de búsqueda para el ta-te-ti.

La complejidad del juego se podría traducir como el tamaño del árbol de juego. Las características más importantes que hacen al tamaño del árbol de juego son la *profundidad del árbol* y su *factor promedio de ramificación*[2].

Entonces, una posible forma de comparar la complejidad de dos juegos es comparar sus árboles de búsqueda, esto es, comparar los promedios de profundidades y factores de ramificación para cada juego. En la tabla 2.1 [Burmeister and Wiles, 1994] se muestran los valores correspondientes al Ajedrez y al Go.

Comparando las características anteriores, como primera conclusión es claro el

[2]Se entiende por factor de ramificación a la cantidad de hijos de un nodo, es decir la cantidad de estados a los que se puede llegar con un movimiento a partir de un estado particular.

	Ajedrez	Go
Promedio de profundidad del árbol de juego	80	300
Promedio factor de ramificación del árbol de juego	35	235

Cuadro 2.1: Promedios de profundidad y factor de ramificación.

hecho de que el Go posee un espacio de búsqueda mucho mayor que el Ajedrez.

Otra posible clasificación de la complejidad de los juegos de la categoría del Go es propuesta por L. V. Allis [Bouzy and Cazenave, 2001]. Allis define la *complejidad del espacio de estados* (E) como el número de posiciones que se pueden alcanzar de la posición inicial, y la *complejidad del árbol de juego* (A) como el número de nodos en el árbol más pequeño necesario para terminar el juego. El autor presenta una tabla con aproximaciones para diferentes juegos de la misma categoría que el Go junto con los resultados de partidas entre humanos y computadoras. En la tabla 2.2 [Bouzy and Cazenave, 2001] $>$, $>=$ y $\ll$ representan «es más fuerte», «más fuerte o igual» y «claramente más débil» respectivamente. H representa el mejor jugador humano.

Juego	$log_{10}(E)$	$log_{10}(A)$	Resultados Comp.-Humano
Damas	17	32	Chinook>H
Othello	30	58	Logistello>H
Ajedrez	50	123	Deep Blue >= H
Go	160	400	Handtalk $\ll$H

Cuadro 2.2: Complejidades por Allis

En una primera mirada de la tabla 2.2 se podría inducir una correlación entre la complejidad del árbol y los resultados obtenidos de las computadoras contra los humanos.

En la tabla 2.3 [Bouzy and Cazenave, 2001] se muestran los valores para un par de juegos adicionales, Go 9x9 y Go-moku 15x15 (5 en linea).

Go 9x9	40	85	Mejor programa $\ll$ H
Go-Moku 15x15	100	80	El juego está resuelto

Cuadro 2.3: Complejidades por Allis.

Con estos nuevos resultados vemos que la correlación inducida anteriormente se pierde. El Go-moku presenta una gran complejidad según A y E, pero el problema fue resuelto completamente. Por otro lado el Go 9x9 posee valores menores que el Ajedrez, pero los mejores programas creados aún son muy débiles comparados contra los jugadores humanos. Allis concluye que el resultado del Go 9x9 es un obstáculo en la credibilidad de la correlación anterior no porque se le haya dedicado poco esfuerzo en resolver el Go 9x9 comparado con los otros juegos, sino que el obstáculo viene por el hecho de que la complejidad no solo está dada por el tamaño del espacio de búsqueda, sino que también está dada por la incapacidad de poder obtener una *función de evaluación* eficiente para el Go. Es decir, la complejidad

del Go no solo viene dada por su gran espacio de búsqueda, sino también por otras dificultades inherentes al Go.

2.3. Resolución Clásica de Juegos

La *búsqueda por fuerza bruta*, es una técnica ampliamente utilizada con éxito en la resolución de muchos juegos de tablero, por lo que merece una descripción de los conceptos básicos involucrados. Además será utilizada en conjunto con técnicas de aprendizaje automático para el diseño de un jugador de Go.

Existen en la actualidad muchas técnicas basadas en búsquedas en el árbol de juego, entre las más conocidas se encuentran *minmax* y *alfa-beta*.

2.3.1. MinMax

En el algoritmo MinMax, los dos jugadores se representan con Max y Min, Max intentará maximizar el resultado del juego, mientras que Min intentará minimizarlo. El algoritmo primero construye el árbol de búsqueda a partir de la posición inicial hasta la profundidad máxima posible. Luego utiliza una función de evaluación estática para evaluar cada hoja del árbol construido. Por último se propagan los valores de los nodos desde abajo hacia arriba, en cada nodo si Max juega, el valor del nodo hijo con el máximo valor es tomado y propagado hacia el padre, si Min juega, el nodo hijo con el menor valor es tomado por el padre. Luego, el árbol construido es utilizado como estrategia de juego; si el jugador es Min, siempre tomará la rama que minimice el valor, si Max juega, este tomará la rama que maximice el valor. El cuadro 1 muestra el algoritmo minmax.

Cuadro 1 Algoritmo MinMax.

1. A partir del tablero b, construir un árbol de juego de profundidad d, esto es, un árbol con raíz b, y que consiste de:
   a) como nodos, tableros
   b) como aristas de un tablero $b1$ a otro tablero $b2$, un movimiento m que lleva del tablero $b1$ a un tablero $b2$.
2. Para cada hoja, calcular estáticamente su valor.
3. Dar a cada tablero del jugador Max, la mayor evaluación entre los nodos hijos.
4. Dar a cada tablero del jugador Min, la menor evaluación entre los nodos hijos.
5. Elegir el movimiento que lleva del tablero b al tablero con máxima evaluación.

Como ejemplo, en la figura 2.13 el jugador Max en la posición A tomará la rama que lleva al nodo D, ya que este busca maximizar el valor de la posición y los valores de las opciones son 3, 0 y 8 que corresponden a los nodos B, C y D respectivamente.

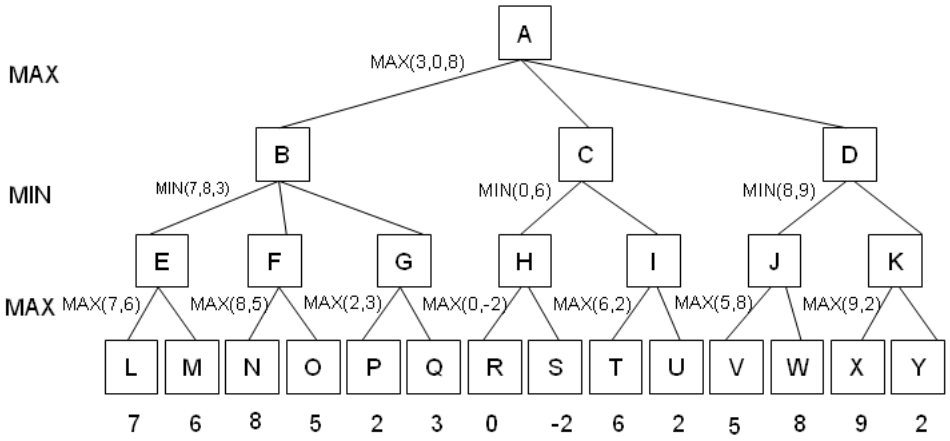


Figura 2.13: Ejemplo de un árbol MinMax.

Dado que no es posible expandir completamente el árbol de búsqueda en juegos con espacios de búsqueda muy grandes, las hojas del árbol de búsqueda, no corresponden a las hojas del árbol de juego, es decir, no corresponden con posiciones finales de juego. Para resolver esto, es necesaria la utilización de una función de evaluación. El propósito de esta función como se mencionó anteriormente consiste en evaluar una posición estáticamente a partir sus características, sin expandir más el árbol de búsqueda.

2.3.2. Poda Alfa-Beta

La poda alfa-beta es una optimización del algoritmo minmax. Simplemente se evita examinar (se podan) ramas (soluciones parciales) que se saben son peores que otra solución conocida o umbral. No aportan más información de cuál será la elección del movimiento. El cuadro 2 muestra el algoritmo alfa-beta.

La figura 2.14 muestra la aplicación de la poda alfa-beta al ejemplo de la figura 2.13. La evaluación del nodo E asegura al oponente (jugador minimizante) una cota superior con valor 7, es decir, el oponente obtendrá 7 o un valor menor en la evaluación de B (dado que los valores están en relación al jugador maximizante, los valores menores a 7 son mejores para el oponente). En este caso, 7 representa el valor *beta*. A continuación, cuando se examina el nodo N cuyo valor es 8, dado que es mayor que $beta = 7$, los nodos hermanos de N (en este caso el nodo O) pueden podarse (dado que nos hallamos en un nivel maximizante, el nodo F tendrá un valor igual o superior a 8, y por lo tanto no podrá competir con la cota asegurada *beta* en el nivel minimizante anterior).

Luego de evaluar los sucesores de B, se concluye que este nodo asegura al jugador maximizante una cota inferior con valor 3, es decir, obtendrá 3 o un valor mayor en la evaluación de A. En este caso, 3 representa el valor de $alfa$. A continuación, cuando se examina el nodo H cuyo valor es 0, dado que es menor que $alfa = 3$, los nodos hermanos de H (en este caso el nodo I) pueden podarse (dado que nos hallamos en un nivel minimizante, el nodo C tendrá un valor menor o igual a 0, y por lo tanto no podrá competir con la cota asegurada $alfa$ en el nivel maximizante

Cuadro 2 Algoritmo Alfa-Beta.

alfabeta(b**,**α**,**β**)**

1. Si el tablero b está al limite de profundidad, retornar la evaluación estática de b; sino continuar.
2. Sean $b_1, \ldots, b_n$ los sucesores de b, sea $k := 1$, y si b es un nodo MAX, ir al paso 3; sino ir al paso 6.
3. Poner $\alpha := max(\alpha, alfabeta(b_k, \alpha, \beta))$.
4. Si $\alpha \geq \beta$ retornar β ; sino continuar.
5. Si $k = n$ retornar α ; sino poner $k := k + 1$ e ir al paso 3.
6. Poner $\beta := min(\beta, alfabeta(b_k, \alpha, \beta))$.
7. Si $\alpha \geq \beta$ retornar α ; sino continuar.
8. Si $k = n$ retornar β ; sino poner $k := k + 1$ y volver al paso 6.

anterior).

En un nivel dado, el valor de umbral $alfa$ o *beta* según corresponda, debe actualizarse cuando se encuentra un umbral mejor. En el ejemplo anterior, al obtenerse el valor 8 del nodo D se puede actualizar el valor de $alfa = 3$ a $alfa = 8$. Esto tendría sentido en el caso de que existieran otros nodos hermanos de D que pudieran ser podados utilizando este valor de $alfa$. Análogamente, al obtenerse un valor de 3 en el nodo G se puede actualizar el valor de $beta = 7$ a $beta = 3$. Esto tendría sentido en el caso de que existieran otros nodos hermanos G que pudieran ser podados utilizando este valor de *beta*.

La efectividad del procedimiento alfa-beta depende en gran medida del orden en que se examinen los caminos. Un ordenamiento malo (considerar los peores movimientos primero) causaría que el algoritmo alfa-beta se comporte equivalente a un minmax. Un ordenamiento perfecto (considerar los mejores movimientos primero) causaría que la complejidad del alfa-beta sea del mismo orden que un min-max completo con un factor de ramificación de solo la raíz cuadrada del número de movimientos. Esto significa que se podría explorar el doble de profundidad que con un algoritmo minmax.

2.3.3. Funciones de Evaluación

La idea en la utilización de una *función de evaluación* es tener un método estático que permita evaluar una posición basado en características de la misma, sin la necesidad de realizar una búsqueda más profunda. De encontrarse una función de evaluación correcta y eficiente, se podrían resolver problemas con grandes espacios de búsqueda. En el Ajedrez existen varias funciones de evaluación que pueden ser utilizadas como una estimación del valor de una posición. Estas funciones permiten el uso de algoritmos de búsqueda para podar el árbol de búsqueda y de esta forma utilizar como estrategia de selección del movimiento óptimo. Un simple ejemplo de función de evaluación para el Ajedrez llamada «Balance de Material», que funciona

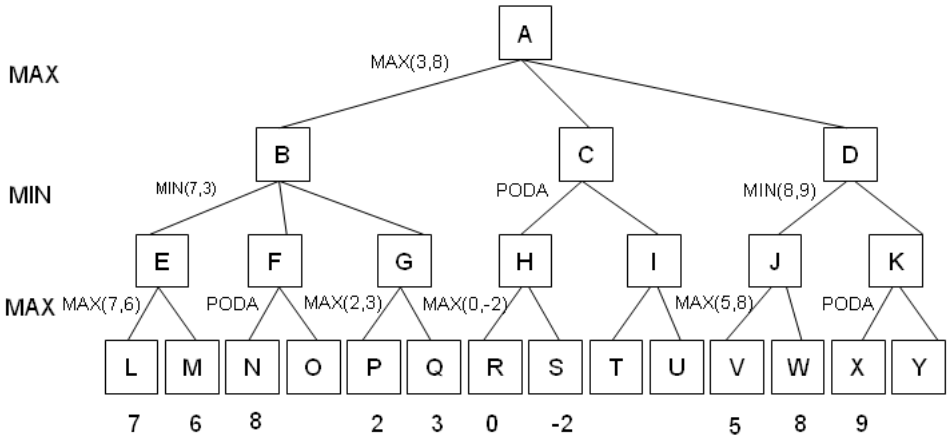


Figura 2.14: Ejemplo de un árbol Alfa-Beta.

bastante bien en relación a su simplicidad es:

$$MB = \sum_{p} N_p V_p$$

Donde N_p y V_p representan la cantidad y valor de piezas de tipo p. Por ejemplo el valor de la reina es de 9, una torre 5, un alfil 3, un caballo 3, un peón 1 y el rey 1000.

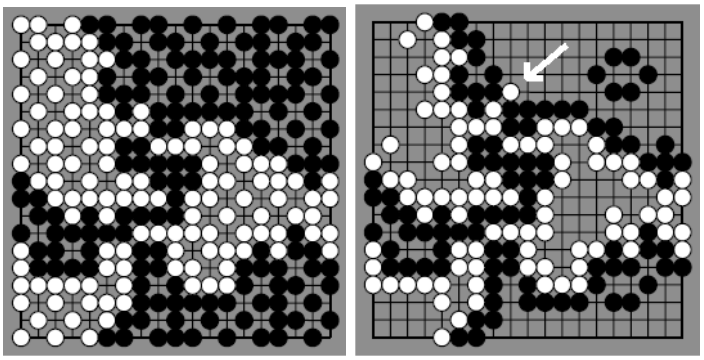
Sin embargo ninguna función de evaluación eficiente y rápida para el Go ha sido encontrada hasta el momento. A diferencia del Ajedrez, en el Go solo existe un tipo de pieza y solo el número de piezas que posee cada jugador en el tablero no es suficiente como para ser utilizado como material para la creación de una función de evaluación.

Para encarar el estudio de la creación de una posible función de evaluación, es muy clara la exposición de Bouzy y Cazenave [Bouzy and Cazenave, 2001]. La primera idea natural que surge es la definición de una función de evaluación concreta. Consiste en devolver +1 por cada intersección ocupada por negro e intersecciones vacías que son vecinas de intersecciones solo negras. Análogamente, asignar -1 por cada intersección ocupada por blanco e intersecciones vacías que son vecinas de intersecciones solo blancas. 0 para el resto de las intersecciones. Esta función de evaluación es completamente simple y fácil de calcular.

En la figura 2.15a las intersecciones son controladas explícitamente, esto es, una intersección controlada por un color tiene la propiedad de estar ocupada por una piedra de ese mismo color o la imposibilidad de poner una piedra del color opuesto.

A dicha posición se llega luego de una gran cantidad de movimientos y los jugadores podrían haber llegado a un acuerdo de quien controla cada territorio mucho antes, como muestra la figura 2.15b.

Una característica interesante del juego es que los jugadores humanos nunca hubieran llegado a la posición de la izquierda, sino que hubieran terminado en la posición de la derecha por mutuo acuerdo, ya que es imposible que el oponente pueda tomar el territorio del otro si el defensor jugara de forma óptima, que en esta situación es muy clara para cualquier jugador promedio. Por lo tanto en la



(a) Control Explícito. (b) Control Implícito.

Figura 2.15: Ejemplo de posiciones controladas explícitamente e implícitamente.

figura de la derecha las intersecciones son controladas implícitamente.

Desafortunadamente la función de evaluación mencionada anteriormente solamente da un resultado correcto en posiciones como la de la izquierda y comete grandes errores al evaluar posiciones como la de la derecha. Esto se debe a que para evaluar la posición de la derecha es necesaria una gran cantidad de conocimiento, y el conocimiento contenido en la función de evaluación concreta no es suficiente como para reconocer la figura de la derecha como una posición terminal. Por ejemplo en la figura de la derecha, la piedra blanca aislada en la parte central superior (apuntada con una flecha) está muerta, y la función de evaluación concreta tomaría como que suma un valor de +1. Por lo tanto la función de evaluación concreta puede ser utilizada solamente en posiciones de control explícito. El problema con esto es que a pesar de que la función de evaluación concreta puede ser computada rápidamente, las computadoras actuales no puede completar búsquedas hasta posiciones de control explícito dadas las grandes profundidades asociadas a estas posiciones y el gran factor de ramificación del Go.

El siguiente acercamiento entonces consiste en encontrar una *función de evaluación conceptual* que pueda evaluar correctamente posiciones de control implícito, y de esta manera acortar la profundidad del árbol de búsqueda drásticamente. La complejidad surge ahora en la creación de una función de evaluación conceptual basándose en las propiedades de las posiciones y que al mismo tiempo sea rápidamente computable. Es necesario no descuidar un equilibrio entre profundidad de búsqueda y complejidad de computación de la función de evaluación.

Una posible forma de acercamiento a la creación de una función de evaluación conceptual es la observación de los jugadores humanos. Intentar capturar los conceptos importantes de una posición y tratar de traducirlos a operaciones calculables. Algunos de esos conceptos pueden ser: bloques, cadenas, territorio, conexión, influencia, etc.

Hasta el momento no existe una función de evaluación de consenso general, mientras más exacto es el resultado, más conocimiento se requiere, resultando en un mayor costo de computación y por lo tanto una función poco eficiente. En este punto es importante tener en cuenta que en una partida de Go profesional se permite aproximadamente hasta un máximo de 24 segundos para realizar un movimiento,

mientras que en el Ajedrez se permiten hasta 180 segundos.

2.4. Diseño de un Jugador de Go

En esta sección se da una visión general del diseño del jugador de Go. En la figura 2.16 se muestra un bosquejo de los módulos más importantes y sus relaciones.

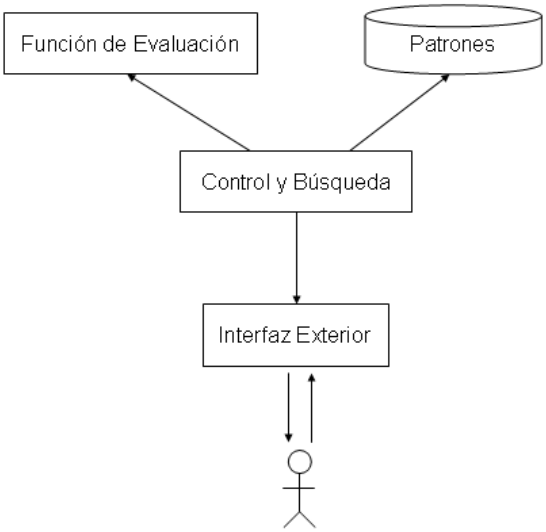


Figura 2.16: Componentes del diseño de un jugador de Go.

Control y Búsqueda

Este módulo centraliza el control del sistema. Para esto interactúa con el resto de los módulos. Implementa la búsqueda en el árbol de juego mediante el algoritmo alfa-beta. Tiene como entrada una posición de tablero y devuelve el movimiento a realizar, lo cual consiste en los pasos del cuadro 3.

Por otro lado, también mantiene una representación interna del tablero mediante estructuras de datos eficientes. En la figura 2.17 se muestra un bosquejo del funcionamiento de ese módulo. Dada una posición de Go, se desarrolla el árbol minmax de modo de «observar movimientos hacia delante» y así poder tener una mejor idea de cual es el mejor movimiento a realizar. Cada nodo representa una posición de tablero de Go, y sus nodos hijos, las posiciones que se pueden obtener al realizar un movimiento (las aristas representan movimientos).

Este módulo se sirve de un conjunto de patrones de Go que permiten, dada una posición, obtener un conjunto reducido de movimientos a considerar. De esta forma se evita analizar todos los movimientos válidos según las reglas de Go, y así, bajar el factor de ramificación del árbol de juego. En definitiva, los patrones son utilizados para podar el árbol de juego.

En el desarrollo del árbol minmax, se utiliza una estrategia alfa-beta, como se describió en secciones anteriores, esto permite podar ramas del árbol que se sabe que no aportarán una solución mejor a la encontrada hasta el momento. Otra propiedad importante de la aplicación de patrones relacionada con esto último, es

Cuadro 3 Algoritmo de selección de movimientos.

1. Repetir

   a) Obtener todos los movimientos válidos del nodo (posición) en consideración.

   b) Filtrar los mejores N movimientos que resultan de la aplicación de algún patrón de juego.

   c) Ordenar los movimientos según el orden de prioridad de los patrones aplicados.

   d) Expandir el árbol de búsqueda un nivel más mediante el algoritmo alfa-beta, recorriendo los nodos en el orden establecido en el paso anterior y podando ramas innecesarias.

2. Hasta la máxima profundidad posible permitida por los límites de tiempo y memoria.

3. Evaluar las hojas del árbol de búsqueda desarrollado en los pasos anteriores.

4. Realizar el movimiento que lleva a la hoja con mayor valor.

el hecho de que los patrones devuelven un conjunto reducido de movimientos en un orden de importancia. Como se describió en secciones anteriores, el algoritmo alfa-beta realiza mayores podas si analiza primero los movimientos de mayor valor para cada jugador, por lo que esta propiedad de ordenamiento aumenta la eficiencia del algoritmo.

En definitiva, al desarrollar el árbol de juego, se realizan dos tipos de podas:

- Podas por aplicación de patrones
- Podas por algoritmo alfa-beta

Gracias a estas dos formas de poda, el árbol de juego tiene un factor de ramificación más pequeño que si se expandiera completamente. Esto permite poder analizar el árbol a mayores profundidades, lo cual es algo deseable ya que de esta forma se puede ver más hacia delante en la partida y por lo tanto realizar una mejor elección de movimientos.

Como se establece en el paso 2 del algoritmo del cuadro 3, el desarrollo del árbol de juego se realiza hasta cierto límite, por lo que en la práctica no es posible desarrollar completamente el árbol de juego hasta llegar a nodos terminales (posiciones finales de Go). Como vimos antes, la solución a este problema consiste en utilizar una función que permita evaluar estáticamente posiciones de Go no terminales. De esta forma, se evita recorrer el árbol hasta llegar a posiciones terminales y la estrategia consiste en elegir la rama por la cual se llega a la hoja (no necesariamente posición terminal) con mayor valor.

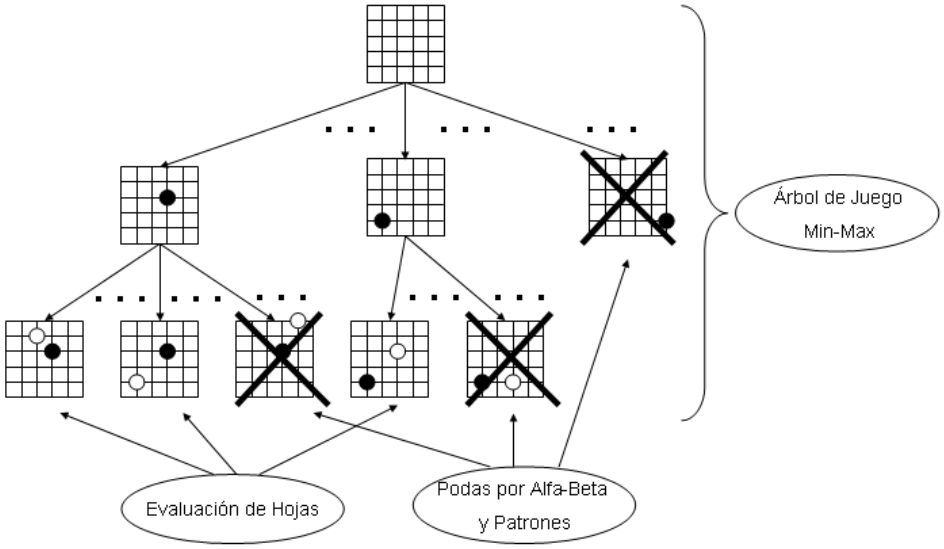


Figura 2.17: Bosquejo del funcionamiento del control y búsqueda.

Aprendizaje de Patrones

La mayoría de los jugadores de Go utilizan un conjunto de patrones programados manualmente. Esto requiere un conocimiento importante del Go y al mismo tiempo bastante esfuerzo de programación. La idea en este proyecto es utilizar aprendizaje automático para la generación de patrones. La figura 2.18 da un bosquejo de este proceso.

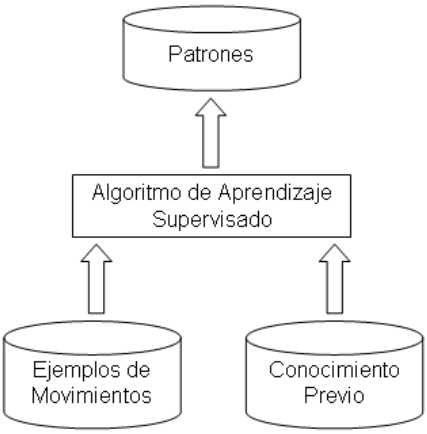


Figura 2.18: Proceso de aprendizaje de patrones.

El módulo utiliza como datos de entrada una base de ejemplos de entrenamiento, compuestos por movimientos realizados en partidas entre jugadores humanos profesionales en el formato estándar SGF [SGF, 2006]. Por otro lado, también recibe como entrada conocimiento previo, ya sea de las reglas del juego, como la

representación de conceptos importantes. Como salida, el módulo devuelve una base de patrones de Go. En el capítulo 3 se presentan todos los detalles involucrados en el diseño de este componente.

Función de Evaluación

Usualmente las funciones de evaluación en el diseño de jugadores de tablero se ajustan analíticamente. Al igual que en la creación de patrones, esto requiere conocimiento del dominio para saber la importancia relativa de cada atributo de una posición a la hora de asignar un valor global. Aquí también se intenta la utilización de técnicas de aprendizaje automático para el ajuste automático de los parámetros de una función de evaluación para el Go. El proceso de evaluación de una posición se muestra en la figura 2.19.

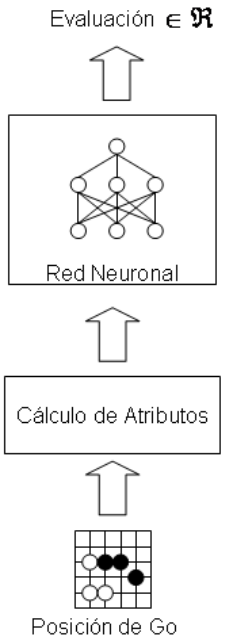


Figura 2.19: Proceso de evaluación de una posición.

El primer paso consiste en transformar la posición en un conjunto de atributos que describan y cuantifiquen las características más importantes. Luego, estos atributos son utilizados como entradas en una red neuronal, la cual devolverá un número real que indique que tan favorable es la posición.

La selección de buenos atributos tiene una gran importancia y puede verse como la incorporación de conocimiento previo. Luego, mediante técnicas de aprendizaje automático se ajustan los parámetros de la red de modo de obtener evaluaciones correctas. Este proceso se muestra en la figura 2.20.

Aquí se implementa un jugador cuya estrategia de selección de movimientos consiste en, dada una posición, obtener todas las posiciones sucesoras, evaluarlas con la red y por último elegir el movimiento que lleva a la posición con mayor valor. El proceso de entrenamiento inicia realizando una partida contra algún oponente. La secuencia de movimientos de ambos jugadores es guardada para luego

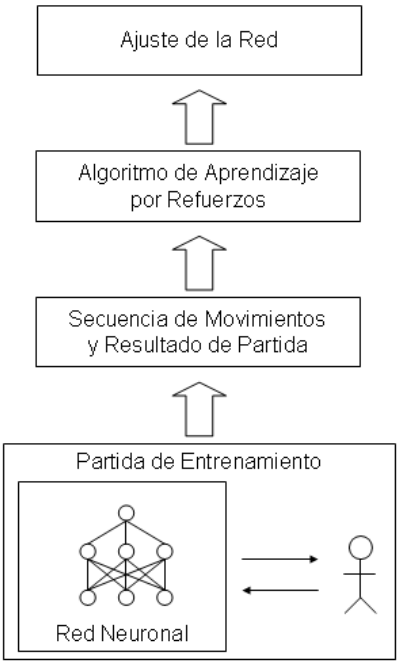


Figura 2.20: Proceso de aprendizaje de la función de evaluación.

ser analizada. A partir de la secuencia y el resultado final de la partida, se utiliza un algoritmo de aprendizaje por refuerzos para el ajuste de los parámetros de la red. Los detalles del aprendizaje de una función de evaluación del Go se presentan en el capítulo 4.

Interfaz Exterior

El último componente necesario en la construcción del jugador consiste en una interfaz que permita interactuar con oponentes humanos o artificiales. Básicamente implementa el protocolo GTP [Farnebäck, 2002]. Este protocolo es muy usado para la interacción de programas de Go. De este modo, con la implementación del protocolo GTP se tiene directamente la capacidad de interoperabilidad con servidores de Internet, interfaces gráficas y otros jugadores artificiales, que de otro modo requeriría la implementación de los componentes adecuados.

Capítulo 3

Aprendizaje de Patrones

En este capítulo se presenta el diseño de uno de los componentes principales del jugador de Go: la generación de patrones. Se profundiza en la utilización de un algoritmo basado en *Inducción de Programas Lógicos (ILP)*.

Para el diseño de este módulo existen tres decisiones importantes a tomar:

1. Lenguaje de representación.
2. Ejemplos de entrenamiento.
3. Algoritmo de aprendizaje.

En la sección 3.1 se discute cuál lenguaje de representación de patrones es más apropiado. En la sección 3.2 se buscan ejemplos de entrenamiento adecuados a las necesidades. En la sección 3.3 se resuelve que algoritmo de aprendizaje utilizar, en particular se introduce la técnica específica: *Inducción de Programas Lógicos*. Por último, en la sección 3.4 se muestran los resultados obtenidos.

3.1. Lenguaje de Representación

Un sistema puede aprender solamente lo que puede representar [Ramon and Blockeel, 2001]. De esto se desprende la importancia del lenguaje de representación. Clasificaremos los lenguajes de representación en dos clases, por un lado los *lenguajes proposicionales* y por otro lado los *lenguajes relacionales*. Veamos las características principales de cada uno y luego cual de los dos es más apropiado para las necesidades de este proyecto.

Lenguajes Proposicionales (Atributo-Valor)

La mayoría de los sistemas de aprendizaje automático utilizan lenguajes proposicionales, también llamados *atributo-valor*. Las reglas o patrones de una hipótesis son representadas como formulas proposicionales, con proposiciones de la forma «*atributo=valor*». Los atributos podrían ser por ejemplo casillas de un tablero y los valores: negro, blanco, vacío, borde, etc.

Gracias a la simplicidad de su representación, usualmente los lenguajes proposicionales son más eficientes que los relacionales. Sin embargo presenta una gran limitación en su capacidad de expresividad. Muchos de los conceptos manejados

en el Go son de naturaleza relacionales, y por lo tanto patrones proposicionales no son adecuados para describirlos. Es muy difícil poder representar en forma general conceptos de alto nivel o abstracción que comúnmente son manejados por jugadores humanos. Inclusive tareas comunes como contar las libertades son muy difíciles de implementar utilizando un lenguaje proposicional.

Lenguajes Relacionales (Lógica de Primer Orden)

En las cláusulas de primer orden, es posible referenciar variables en la precondición de la regla sin que tengan que ocurrir en las postcondiciones. La siguiente regla es un ejemplo de una cláusula de primer orden.

```
IF padre(Y,Z) and madre(Z,X) and mujer(Y) THEN nieta(X,Y)
```

La variable Z en esta regla, que refiere al padre de Y, no está presente en las postcondiciones de la regla. Cuando una variable ocurre solamente en las precondiciones, es asumido que está existencialmente cuantificada, esto es, las precondiciones de la regla son satisfechas si existe por lo menos una instanciación de la variable que satisface el correspondiente literal [Mitchell, 1997]. Una representación mediante lógica de primer orden presenta dos grandes ventajas sobre un lenguaje proposicional. Primero, una representación relacional provee de un lenguaje mucho más expresivo. Segundo, los resultados de un aprendizaje son más generales y entendibles. Más aún, una representación con lógica de primer orden podría ser independiente del tamaño de por ejemplo el tablero de juego y podría representar relaciones entre objetos del dominio del problema.

En este proyecto se decidió utilizar lógica de primer orden. La razón es fundamentalmente su capacidad de expresión. El Go es un juego en donde se manejan conceptos abstractos y la mayoría de las veces hacen referencia a relaciones entre diferentes objetos, por lo que sugiere un acercamiento relacional. Por otro lado, Blockeel y Ramon [Ramon et˜al., 2000] han propuesto un sistema relacional para la obtención de patrones de Go en situaciones tácticas como problemas de vida o muerte con resultados promisorios, lo que refuerza la elección.

3.2. Ejemplos de Entrenamiento

La siguiente decisión de diseño que se debió elegir es el tipo de ejemplos o datos de entrenamiento a utilizar. El tipo y la calidad de los ejemplos de entrenamiento pueden tener un impacto significativo en el éxito o falla del sistema de aprendizaje. Básicamente existen dos formas de proveer de ejemplos de entrenamiento, en forma *directa* o *indirecta* [Mitchell, 1997].

Supongamos que deseamos aprender patrones de Go: el sistema podría aprender de ejemplos de entrenamiento directos que consistan en posiciones de tablero y los movimientos correctos a realizar para cada uno.

Por otro lado, se podría proveer de ejemplos de entrenamiento indirectos, en este caso en el ejemplo consistiría de una secuencia de movimientos de una partida y el resultado final de la misma. En este último caso la información de la correctitud de los movimientos al inicio de la partida deben ser inferidos indirectamente del hecho de que la partida fue ganada o perdida. Aquí se presenta un problema propuesto

por Minsky conocido como *problema de asignación de crédito* (*credit assignment problem*), [Fürnkranz, 2001]. El problema consiste en decidir cómo y en qué grado cada movimiento influyó en el resultado final. Por lo tanto, un aprendizaje basado en un entrenamiento directo es típicamente más fácil que uno indirecto. A primera vista sería deseable realizar un entrenamiento directo.

Sin embargo, en la práctica surgió una limitación: no hay datos disponibles como para realizar un entrenamiento directo. Para poder realizar un entrenamiento directo en el caso del Go, serían necesarios datos sobre situaciones tácticas concretas y los correspondientes movimientos correctos. Luego de una búsqueda por diferentes sitios de Internet relacionados con recursos de información y datos de Go, los únicos datos disponibles se trataron de registros de juegos completos entre humanos. Estos registros se encuentran en el formato estándar SGF [SGF, 2006] y registran la secuencia de movimientos realizados por ambos participantes, junto con el resultado de una partida de Go. Por lo tanto, el sistema debería aprender con ejemplos de entrenamiento indirectos.

Por otro lado, en el caso de partidas de jugadores profesionales, existe la siguiente particularidad mencionada por conocedores del Go: la mayoría de los movimientos en una partida profesional de Go son muy buenos, casi todos los movimientos pueden ser considerados correctos independientemente del resultado de la partida. El resultado final de una partida profesional depende de quien realizó la mejor secuencia de movimientos. Este último hecho puede sugerir la utilización de cada movimiento individual de una partida profesional como un ejemplo de entrenamiento directo.

Otra consideración es el hecho de que al ser los ejemplos de entrenamiento muy generales, es decir, no estar previamente clasificados como movimientos de ataque, defensa, conexión, aumento de territorio, apertura, etc; el sistema de aprendizaje deberá intentar aprender un concepto muy general del estilo $mover : Tablero \rightarrow (X, Y)$. A primera vista este requerimiento puede ser un poco riesgoso, el sistema podría terminar aprendiendo movimientos demasiado generales. Más adelante veremos que es deseable tener también ejemplos negativos, esto es, ejemplos donde se enseñan movimientos incorrectos. En este punto se presenta una nueva limitación, los ejemplos de entrenamiento disponibles son solamente positivos.

En definitiva el sistema utiliza como ejemplos de entrenamiento registros de juegos completos entre profesionales. Las razones son básicamente la limitación para acceder a datos de entrenamiento más directos.

3.3. Algoritmo de Aprendizaje

La primera decisión hecha en el diseño del módulo de aprendizaje de patrones fue utilizar la lógica de primer orden como lenguaje de representación de los patrones. La inducción de teorías de cláusulas lógicas de primer orden es conocido como *Inducción de Programas Lógicos*, o ILP por sus siglas en inglés *(Inductive Logic Programming)*. El nombre viene por el hecho de que el proceso puede ser visto como la inducción de programas lógicos a partir de ejemplos.

3.3.1. Inducción Programas Lógicos

ILP [Lavrac and Dzeroski, 1993, Muggleton and Raedt, 1994, Mitchell, 1997, Peña, 2004], es una subespecialidad del campo de aprendizaje automático la cual utiliza conocimiento previo y ejemplos para inducir cláusulas Horn (cláusulas de primer orden con variables). Dado que conjuntos de cláusulas de primer orden tipo Horn pueden ser interpretadas como programas lógicos en el lenguaje PROLOG [Lloyd, 1987, Sterling and Shapiro, 1999], es por eso que este aprendizaje es usualmente llamado inducción de programas lógicos.

Los ejemplos, hipótesis y conocimiento previo se representan mediante cláusulas lógicas de primer orden. La figura 3.1 muestra un ejemplo de una cláusula lógica de primer orden para la selección de movimientos de Go.

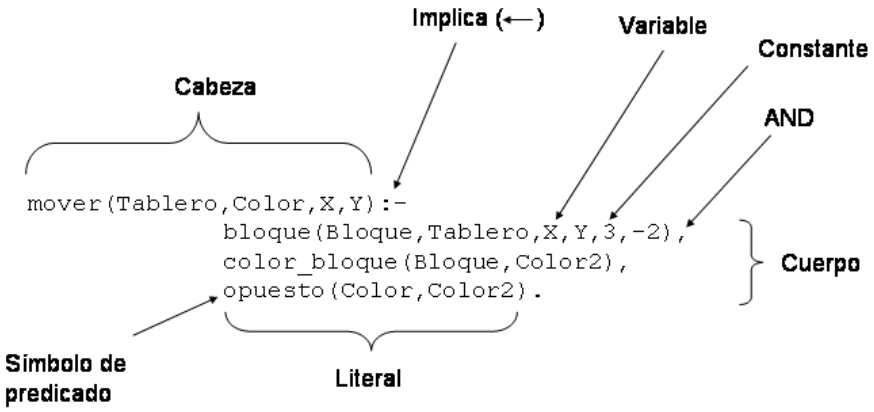


Figura 3.1: Ejemplo de una cláusula lógica en lenguaje Prolog. En lenguaje natural se puede interpretar como: «realizar el movimiento X, Y en el *Tablero* con *Color*, si existe un *Bloque* que se encuentre a distancia relativa 3,-2 de X, Y y tenga color opuesto a *Color*».

Típicamente un sistema de ILP utiliza como entrada el conocimiento previo B, que provee al sistema de información acerca del dominio. Por otro lado recibe también como entrada un conjunto de ejemplos E que por lo general consisten en ejemplos positivos $E+$ (el valor de la función debe ser verdadero en estos ejemplos) y ejemplos negativos E- (el valor de la función debe ser falso en estos ejemplos).

El objetivo es inducir las cláusulas de un predicado T que define una relación sobre el dominio. Este predicado T es utilizado para clasificar ejemplos nunca vistos $E?$, y puede ser visto como una definición de un concepto, es decir una aproximación de una función booleana cuyo valor es verdadero para todos los objetos pertenecientes a dicho concepto. En el cuadro 4 se muestra el algoritmo general de ILP.

Cuadro 4 Algoritmo de ILP

1. Inicializar $T = \emptyset$
2. Mientras $E^+ \neq \emptyset$ (La teoría no cubre todos los ejemplos positivos)
   a) $C = p \leftarrow$
   b) Mientras C cubra ejemplos negativos
      1) Seleccionar un literal l a agregar a $C = p \leftarrow body$.
      2) $C = p \leftarrow body, l$. (Especializo la cláusula)
      3) $T = T \cup C$ (Agrego la cláusula a la teoría)
      4) $E^+ = E^+ -$ ejemplos positivos cubiertos por T.

Algoritmo de ILP Progol

Los sistemas de ILP pueden variar en muchas de sus características. En este proyecto se decidió utilizar un acercamiento *top-down*[1] dado que es el acercamiento más utilizado en ILP. Más aún, se decidió utilizar la técnica de *aprendizaje guiado por ejemplos (example-driven learning)* dado que mejora significativamente le eficiencia del sistema comparado con un sistema puramente generador y testeador. Progol [Muggleton and Firth, 1999] es uno de los sistemas de ILP más utilizados, por lo que será utilizado como base de las experiencias en el aprendizaje de patrones. Progol toma como entrada:

- **Conocimiento de fondo** B
- **Ejemplos de Entrenamiento** E
- **Declaraciones de Modos**
- **Restricciones**

Las primeras dos entradas son clásicas en un algoritmo de ILP general como fueron descritas anteriormente. Sin embargo el tercer y cuarto punto son nuevos. Las *declaraciones de modos* definen qué predicados de B aparecerán en el cabezal de las cláusulas de la teoría y cuales aparecerán en el cuerpo de las cláusulas. También declara los tipos y modos (entrada, salida o constante) de cada argumento de cada predicado. La declaración de modos es un acercamiento muy útil para la reducción del espacio de hipótesis. Las *restricciones* son otra forma de acotar aún más el espacio de hipótesis.

Cada vez que se inicia una búsqueda de una cláusula para ser agregada a la teoría, Progol toma un ejemplo no cubierto y construye mediante *resolución inversa (inverse entailment)* la *cláusula más especifica* también llamada *cláusula base (bottom clause)*. De este modo Progol implementa un aprendizaje guiado por ejemplos.

En las siguientes secciones se describen las decisiones relacionadas a la utilización de ILP en la inducción de patrones de juego de Go.

[1] El acercamiento top-down consiste básicamente en comenzar con una teoría muy general para luego irla especificando incrementalmente.

3.3.2. Predicado Objetivo y Ejemplos de Entrenamiento

Predicado Objetivo

El primer paso consiste en definir el predicado que queremos aprender. En nuestro caso se trata de un único predicado «mover», que dados un tablero y un color indica el movimiento realizado. Para el aprendizaje del predicado todos los argumentos son de entrada (+). La declaración sería:

```
move(+board_type,+color,+coord_x,+coord_y).
```

Sin embargo para la aplicación en la práctica la situación consistirá en recibir un tablero y un color y devolver como salidas las coordenadas que indiquen el movimiento a realizar[2] . Por lo tanto, para la aplicación de los patrones inducidos, los predicados de conocimiento de fondo utilizados en la construcción de las cláusulas deberán ser generativos. De este modo lograremos poder utilizar el patrón como generador de movimientos, lo cual es el objetivo final.

Es claro que este predicado es muy general, se está intentando aprender un patrón que indique como jugar en cualquier situación. Como se discutió anteriormente, esta decisión se debe fundamentalmente por la limitación de no poder contar con ejemplos de entrenamiento más específicos. Sería deseable que los ejemplos de entrenamiento estuvieran previamente clasificados en movimientos de ataque, defensa, conexión, aumento de territorio, apretura, etc. Esto permitiría particionar el aprendizaje en espacios de búsqueda más reducidos.

El único acercamiento en el aprendizaje de patrones basado en ILP encontrado [Ramon et˜al., 2000] induce reglas tácticas para la resolución de problemas de vida o muerte. Como veremos al final de este capítulo, al intentar inducir patrones sobre ejemplos muy generales, se corre el riesgo de obtener reglas demasiado generales.

Ejemplos de Entrenamiento

Dada una relación R a aprender, y un símbolo de predicado p que denota la relación R. Un ejemplo de entrenamiento e de p es un *hecho* con el mismo símbolo de predicado p y aridad. Un ejemplo e es clasificado como positivo o negativo de p de acuerdo a si R es verdadera o falsa sobre los términos de e respectivamente.

En nuestro caso la relación R a aprender es: «seleccionar un movimiento de Go para un tablero y color dados».

En este caso un conjunto de ejemplos para tal relación podrían ser:

```
mover(tablero1,blanco,5,5).
mover(tablero2,negro,4,3).
mover(tablero3,blanco,6,1).
mover(tablero3,negro,2,7).
```

Donde *tablero1, tablero2, tablero3*, son instanciaciones de la variable *Tablero* (es decir, posiciones de tablero particulares), *blanco* y *negro* son las dos posibles instanciaciones de la variable *Color*. En este caso, los movimientos se representan

[2]En notación Prolog, el predicado a inducir en la práctica será instanciado: move(+Board_type,+Color,-Coord_x,-Coord_y).

por sus coordenadas cartesianas en el tablero, representadas con las variables X e Y.

A la hora de especificar los ejemplos también se debe decir cuáles de ellos son ejemplos positivos y cuáles negativos. En nuestro caso, en cuáles de los ejemplos anteriores el concepto *mover* es verdadero y en cuáles es falso.

Como ejemplos de entrenamiento se utilizaron 232 registros en formato SGF [SGF, 2006] de partidas de Go 9x9 entre jugadores humanos profesionales de nivel dan. Cada partida se compone en promedio de 47 movimientos, en total se generaron alrededor de 10800 ejemplos de movimientos. Se implementó un programa en Prolog el cual lee las secuencias de movimientos en cada partida en formato SGF y devuelve un archivo de texto con los hechos del predicado objetivo para luego ser utilizados como ejemplos en el algoritmo de aprendizaje basado en ILP.

3.3.3. Conocimiento Previo

En esta sección introduciremos los predicados que serán utilizados como conocimiento previo (B). Recordemos que B es un conjunto de cláusulas definidas (programa lógico), que define los predicados que serán utilizados para la construcción de la teoría a inducir. Por lo tanto, es una parte fundamental para el funcionamiento del algoritmo de aprendizaje. Dos puntos relevantes en el diseño deben tenerse en consideración.

En primer lugar, dado que el programa B define los predicados a utilizar en la construcción de las cláusulas de la teoría, la falta de relaciones suficientes y expresividad en las cláusulas de B limitan directamente la capacidad de representación y modelado de la teoría a inducir. En definitiva, si los predicados definidos en B son pocos o muy simples, probablemente será imposible poder inducir una teoría suficientemente compleja como para «resolver» el problema.

En segundo lugar, la eficiencia de B también afecta directamente la eficiencia del algoritmo de aprendizaje y la eficiencia de la teoría a inducir. Los predicados de B serán utilizados en las pruebas realizadas por el lenguaje Prolog, tanto en la construcción de las cláusulas base, como en las pruebas de cobertura de los ejemplos. Si los predicados son ineficientes, es probable que las restricciones en el tiempo de ejecución y capacidad de memoria típicos en la práctica limiten la capacidad de inducir una teoría compleja para un problema complejo.

Para el primer punto, no existe una receta para poder identificar cuáles conceptos deberán ser modelados en B, esto depende directamente del dominio del problema. Es necesario interiorizarse con el dominio en particular para hallar los conceptos más relevantes que puedan servir como material para la construcción de una teoría. En el caso particular del Go, si el diseñador es un experto en el juego (no es el caso), será más simple. De otra forma es necesario recurrir a información del dominio o trabajos relacionados para descubrir que conceptos deben ser incluidos. En el Go existen varios conceptos manejados por los jugadores humanos, así que el primer acercamiento será intentar modelarlos.

La idea es definir cuidadosamente un conjunto de predicados que además de ofrecer suficiente capacidad de expresión, sea un conjunto lo más reducido posible de modo de evitar redundancia y así reducir el espacio de búsqueda de hipótesis.

Como acercamiento inicial se decidió comenzar con el modelo propuesto por Blockeel y Ramon [Ramon et˜al., 2000]. Un conjunto de predicados básicos se

representan a partir del modelo relacional de la figura 3.2. Cada intersección en el tablero pertenece a un bloque, indicada por la relación *Board*. Un bloque puede ser una intersección vacía o un bloque de piedras. La relación *Block* especifica el identificador único del bloque y su color. Para cada bloque existe una lista de bloques adyacentes que son guardados en la relación *Link*.

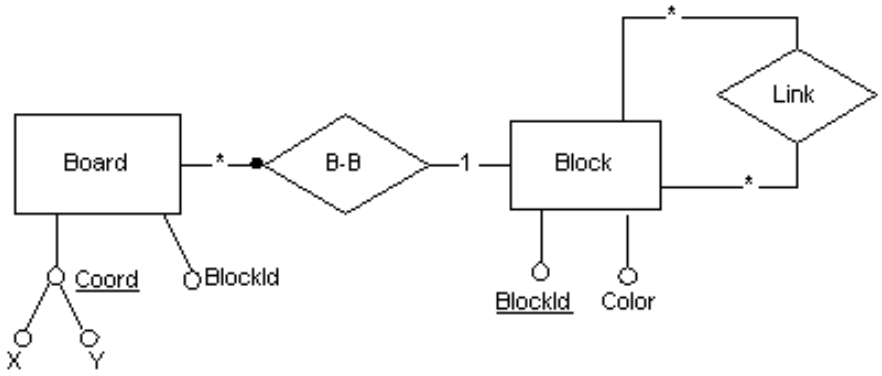


Figura 3.2: Representación relacional.

Este modelo puede ser representado con los siguientes predicados:

```
board(X,Y,BlockId)
block(BlockId,Color)
link(BlockId1,BlockId2)
```

Estos predicados dan una representación del tablero y de las relaciones entre los bloques del tablero. Sobre estos predicados básicos pueden ser representadas otras relaciones de *B*. Los siguientes ejemplos ilustran algunas posibles relaciones en lenguaje Prolog:

```
liberty(BlockId,Liberty):- link(BlockId,Liberty),
                           block(Liberty,vacío).
liberty_cnt(BlockId,Cant_libs):- findall(Lib,liberty(BlockId,Lib),L),
                                 length(L,Cant_libs).
atari(BlockId):- liberty_cnt(BlockId,1).
```

El predicado *liberty(BlockId, Liberty)* es verdadero si el bloque *Liberty* es una libertad del bloque *BlockId*. El predicado *liberty_cnt* devuelve la cantidad de libertades de un bloque. Utiliza el predicado *findall* que devuelve una lista de todas las soluciones del predicado *liberty(BlockId,Lib)*, de esta forma se obtiene una lista de todas las libertades de *BlockId*; el largo de la lista es la cantidad de libertades del bloque. El predicado *atari(BlockId)* es verdadero si el bloque se encuentra en atari.

A continuación presentaremos los predicados de *B* más importantes. En el apéndice A.1 se brinda un listado completo de los predicados de conocimiento previo diseñados para la inducción de patrones de Go.

En primer lugar fue necesario tener en cuenta una serie de simetrías en la aplicación de patrones. Las características del Go hacen que un patrón de movimiento

tenga 16 simetrías. Las 8 simetrías «topológicas» (rotaciones de 90 grados y la simetría axial en un eje), por 2 simetrías de «color» dan un total de 16 aplicaciones diferentes de un mismo patrón. La figura 3.3 ilustra esta situación.

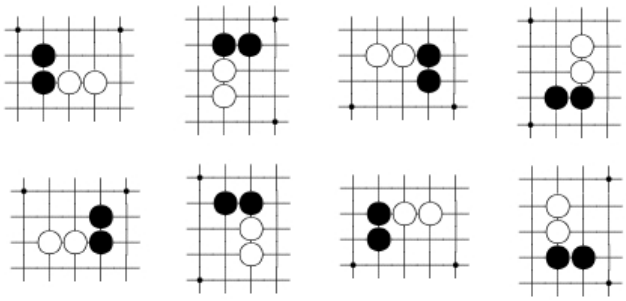


Figura 3.3: 8 simetrías topológicas. La primera fila se obtiene como rotaciones de 90 grados en sentido horario. La segunda fila se obtiene como la simetría según un eje vertical de la primer fila.

Para resolver esto, se utiliza un predicado de la siguiente forma[3]:

```
pos_gen(-pos)
```

Este predicado genera un término que indica la posición de simetría. En cada cláusula construida donde existan literales donde apliquen simetrías topológicas, existirá a lo sumo un literal del tipo *pos_gen* para generar el término que identifique la simetría topológica. El propósito es guardar una coherencia entre todos los predicados topológicos de modo que se apliquen en la misma «posición». Notar la importancia de representar tales simetrías, el espacio de búsqueda para las cláusulas con relaciones topológicas se reduce en un factor de 8, y las que implican relaciones de colores la reducción es de un factor de 2.

Para tener un predicado que represente la posición relativa a un bloque respecto a una intersección, se utiliza el siguiente predicado:

```
block_on_pos(+board_type,+coord_x,+coord_y,-block_id,#disp,#disp,+pos)
```

Dados un tablero, una posición de aplicación y las coordenadas de una intersección, genera los identificadores de bloques a distancias relativas.

Por otro lado, se utilizaron un par de predicados para indicar las distancias horizontal y vertical de una coordenada a su borde mas cercano:

```
edge_ver(+coord_x,+coord_y,#disp,+pos)
edge_hor(+coord_x,+coord_y,#disp,+pos)
```

Una posible alternativa a utilizar dos predicados podría haber sido utilizar un único predicado con un argumento que indique si la distancia es vertical o horizontal. Sin embargo, en la práctica se obtuvieron mejores resultados separando la vertical y horizontal en los predicados anteriores.

[3]Se utilizará la siguiente notación: + indica argumento de entrada, - indica argumento de salida, y # indica argumento constante.

Con respecto a los bloques, las propiedades de libertad y cantidad de piedras son conceptos relevantes en el Go. Las libertades tienen una relación directa con la seguridad del bloque, y la cantidad de piedras una relación directa con la importancia de un bloque. Por lo tanto, la idea es tener predicados que describan tales características de un bloque.

El predicado *liberty_cnt* devuelve la cantidad de libertades de un bloque. Se tienen dos tipos de instanciaciones, por un lado interesa devolver la cantidad como una constante y por otro lado interesa devolver el valor como una variable para luego, por ejemplo, ser relacionada en otro predicado:

```
liberty_cnt(+board_type,+block_id,#nat)
liberty_cnt(+board_type,+block_id,-nat)
```

El predicado *stone_cnt* devuelve la cantidad de piedras de un bloque. Al igual que el predicado anterior, se tienen dos posibles instanciaciones del predicado:

```
stone_cnt(+board_type,+block_id,#nat)
stone_cnt(+board_type,+block_id,-nat)
```

Por otro lado, interesa obtener el color de un bloque, esto se logra con un predicado de la forma:

```
block(+board_type,+block_id,-color)
```

Así, por ejemplo, si una teoría necesitara relacionar los colores de los bloques, utilizaría el predicado anterior. Para obtener el color opuesto a un color dado, se tiene un predicado de la forma:

```
opposite(+color1,-color2)
```

Dado un bloque, también es interesante obtener cada una de sus libertades, por ejemplo, para relacionarlas con las libertades de otros bloques. El siguiente predicado se encarga de esto:

```
liberty(+board_type,+block_id,-liberty_id)
```

Por otro lado, podemos definir predicados con un nivel más de abstracción. Por ejemplo los siguientes predicados definen los conceptos de atari, conexión directa, conexión indirecta y captura:

```
atari(Board,Block,X,Y,Color)
connects_direct(Board,Block,X,Y,Color)
connects_indirect(Board,Block,X,Y,Color)
captures(Board,Block,X,Y,Color)
```

El primer predicado se interpreta como: «Si el jugador *Color* juega en la posición X, Y; el bloque *Block* queda en atari». El segundo y tercer predicado se interpretan como: «Si el jugador *Color* juega en la posición X, Y; entonces conectará (directa/indirectamente) con el bloque *Block*». El último predicado se interpreta como: «Si el jugador *Color* juega en la posición X, Y; capturará al bloque *Block*». Estos predicados pueden ser representados en función de los predicados anteriores, por lo que a primera vista no serían necesarios ya que podrían ser inducidos como parte de la teoría a aprender. Sin embargo, como veremos más adelante, en la práctica la inducción de tales predicados no se produce o requeriría de mucho entrenamiento, por lo que es deseable introducirlos a priori como conocimiento previo.

3.3.4. Declaración de Modos

Las declaraciones de modos establecen los tipos y modos (entrada, salida o constante) de cada argumento de cada predicado. En la subsección anterior implícitamente definimos la instanciación de los argumentos con los signos de +, - y #. Por otro lado, es posible especificar un valor de *recall*, que es un entero mayor que 0 o * que establece el limite superior de la cantidad de soluciones alternativas de un predicado. El valor * indica todas las soluciones.

De los predicados mencionados en la subsección anterior:

```
block_on_pos(+board_type,+coord_x,+coord_y,-block_id,#disp,#disp,+pos)
liberty(+board_type,+block_id,-block_id)
```

Estos predicados deben tener un valor de *recall* = * ya que pueden devolver varias soluciones (no determinísticos). Esto se debe a que en el primer caso, se generan bloques a determinada posición relativa a las coordenadas de entrada (las constantes *disp* se generan y actúan como constantes de salida). Para el segundo caso, un bloque puede tener varias libertades, recordar que una intersección vacía adyacente a un bloque (libertad) también la identificamos como un bloque especial: un bloque que consta de una única intersección vacía.

El resto de los predicados tienen una única solución, por lo que sus valores de recall son 1.

3.3.5. Estrategia de Búsqueda

En la búsqueda de una cláusula para agregar a la teoría en construcción, implícitamente se realiza una búsqueda en el espacio de hipótesis. Las cláusulas pueden ser ordenadas parcialmente por una relación de especialización. En general, una cláusula c_1 es mas específica que una cláusula c_2 si $c_2 \models c_1$. Un caso especial utilizado usualmente, es que una cláusula c_1 es más específica que una cláusula c_2 si el conjunto de literales del cuerpo de la cláusula c_2 es un subconjunto de los literales del cuerpo de c_1 (c_2 *subsume* c_1). Esta relación de orden puede ser usada para estructurar un conjunto de cláusulas parcialmente ordenadas, llamado *grafo de refinamiento*. Una cláusula c_1 es un sucesor inmediato de una cláusula c_2, si c_1 puede ser obtenida agregando un literal al cuerpo de la cláusula c_2. Por lo tanto un grafo de refinamiento puede ser usado para saber como especializar una cláusula.

A la hora de construir una cláusula es posible utilizar diferentes estrategias de búsqueda, esto es, alterar la forma en que el grafo de refinamiento es recorrido. Usualmente los sistemas de ILP realizan una *búsqueda en amplitud (BFS)*, por lo tanto, las cláusulas más cortas serán recorridas primero. Otros acercamientos posibles podrían ser *búsqueda en profundidad (DFS)*, búsquedas estocásticas, etc. Algunas de las estrategias de búsqueda más utilizadas pueden verse [Peña, 2004], en nuestro caso se decidió adoptar la búsqueda BFS porque considera cláusulas más generales primero.

3.3.6. Restricciones

En un dominio de problema complejo, usualmente resulta en la inducción de teorías muy complejas. Esta complejidad se traduce en un gran tamaño del espacio de hipótesis y por lo tanto un gran árbol de búsqueda. En la práctica existen

restricciones de recursos y tiempo que deben ser respetados. Las declaraciones de modos y el aprendizaje guiado por ejemplos son buenas técnicas para reducir el espacio de búsqueda. Sin embargo es necesario reducir más aún el espacio. Las restricciones pueden ser clasificadas en:

- Restricciones de Lenguaje
- Restricciones de Búsqueda

Restricciones de Lenguaje

Estas restricciones buscan acotar el espacio de hipótesis a considerar. Estas pueden ser, por ejemplo, restringir el lenguaje a cláusulas que no contengan símbolos de funciones, cláusulas con a lo sumo n literales en el cuerpo, cláusulas que solo contienen literales con símbolos de predicados en el conocimiento de fondo B, etc. Mientras más restricciones agreguemos a las cláusulas, menor será el espacio de búsqueda, y por lo tanto el sistema terminará la búsqueda más rápido. Por otro lado las restricciones pueden evitar que buenas teorías sean descubiertas. Por ejemplo si las cláusulas están limitadas a 5 literales y la teoría correcta contiene cláusulas con 6 literales, esta nunca sería descubierta. Por lo tanto es necesario encontrar el equilibrio adecuado entre eficiencia del sistema y la calidad de la teoría encontrada. Las restricciones de lenguaje utilizadas fueron:

- **Longitud de cláusulas:** Limitar el tamaño de las cláusulas es una forma de reducir considerablemente el espacio de búsqueda. Concretamente la idea es limitar la cantidad de literales que pueden aparecer en el cuerpo de una cláusula. Al limitar a máximo 15 literales en el cuerpo de la cláusula se obtuvieron tiempos de entrenamiento manejables y al mismo tiempo suficiente capacidad de expresión.

  Por otro lado, también fue necesario establecer un límite inferior en el tamaño de las cláusulas. De otra forma, el sistema tendía a inducir reglas muy generales. Como cota inferior el mejor resultado se obtuvo estableciendo que las reglas debían tener más de 2 literales en su cuerpo.

Restricciones de Búsqueda

Estas restricciones buscan restringir el espacio de búsqueda. En un extremo el sistema podría realizar una búsqueda exhaustiva. Usualmente en la práctica este tipo de búsquedas lleva demasiado tiempo, por lo que es necesario utilizar alguna heurística para guiar cuales partes del espacio serán recorridas y cuales serán ignoradas. Estas restricciones pueden clasificadas un nivel más. A continuación se presentan las diferentes clasificaciones y el diseño ideado en este proyecto.

Restricciones de Integridad

Las restricciones de integridad usualmente son utilizadas para una definición explícita de restricciones tanto semánticas como sintácticas. A continuación se describen las restricciones de integridad utilizadas, en el apéndice A.3.1 se da un listado completo en la notación de Progol.

- **Restricción generativa:** Aquí se busca restringir que todas las variables de una cláusula aparezcan en al menos dos literales diferentes de la cláusula. Esto implícitamente impone una relación «proveedor-consumidor» entre los literales de la cláusula. Si un literal introduce una variable de salida, entonces debe haber un literal que utilice esa misma variable como argumento de entrada. Esta restricción produjo una reducción importante del espacio de búsqueda y tuvo un impacto muy importante en la práctica. Podría decirse que sin esta restricción sería prácticamente imposible obtener algún resultado útil, ya que de otra forma el sistema sistemáticamente introduciría nuevas variables sin utilidad alguna.

- **No permitir hechos:** Esta restricción es trivial pero no menos importante. En definitiva obligamos al sistema a devolver generalizaciones de los ejemplos de entrenamiento y no devolver los ejemplos mismos. Con esto se evita la teoría más trivial que se puede aprender: «memorizar» todos los ejemplos de entrenamiento.

- **Otras restricciones:** Otras restricciones de integridad que interesaron fue obligar cierta combinación de predicados. Por ejemplo, se utilizó una restricción que obliga a que si existe un literal *block_on_pos*, deberá haber un literal *block_color* que indique el color del bloque en cuestión. Otra restricción utilizada fue obligar a que luego de la etapa de apertura se obligue la aparición del predicado *block_on_pos*.

  Ambas restricciones fueron utilizadas con el objetivo de obtener patrones más específicos, de otra forma el sistema tiende a devolver resultados demasiado generales.

Restricciones de Poda

Estas restricciones permiten realizar grandes podas en el árbol de búsqueda. Si una hipótesis es descartada por no cumplir con una restricción de poda, entonces todas las hipótesis más específicas también serán descartadas. A continuación se describen las restricciones de poda utilizadas, en el apéndice A.3.2 se da un listado completo en la notación de Progol.

- **Evitar cláusulas recursivas:** Un ejemplo claro de una restricción de poda es no permitir que una cláusula sea recursiva, esto es, que ninguna cláusula tenga en su cuerpo un literal con el símbolo de predicado de su cabeza. Inducir cláusulas recursivas es un problema muy complejo, por lo que se decidió evitarlo.

- **Otras restricciones:** Evitar que dos literales diferentes en el cuerpo de una cláusula tengan el mismo símbolo de predicado *opposite*. Si hubiera más de un literal *opposite* sería redundante ya que existen solo dos colores, uno opuesto del otro. En la práctica, si no se introducía esta restricción, sistemáticamente se introducía este predicado sin utilidad alguna. También fue necesario evitar que existan dos literales con símbolo de predicado *liberty_cnt* o *stone_cnt* aplicados al mismo bloque.

Restricciones de Recursos

Por último, en la práctica se tiene una limitación en la capacidad de los recursos, por ejemplo, en la capacidad de memoria o de tiempo de ejecución. Por lo tanto muchas veces la cantidad de nodos que es posible recorrer en el espacio de búsqueda se ve limitado por restricciones en los recursos del sistema. Se manejaron los siguientes parámetros:

- Cantidad máxima posible de iteraciones en la construcción de la cláusula base. Cada iteración introduce nuevas variables. Este valor fue puesto en 10.
- Cantidad máxima de nodos que se pueden recorrer. Fue puesto en 5000.

Existen algunas guías de como configurar tales parámetros, pero la práctica demostró que es necesario ajustarlos por prueba y error.

3.3.7. Función Evaluación

Otra forma de introducir un sesgo en el aprendizaje mediante ILP es a través de su *función de evaluación*. En el momento de elegir qué cláusula agregar a la teoría entre las posibles cláusulas del espacio de hipótesis es necesario tener algún método automático de elección que mida la utilidad de una cláusula y elija la más deseable de todas. Esta selección actúa como un *sesgo preferencial*: se prefieren determinadas hipótesis sobre otras según algún criterio definido a priori.

Lo que se intenta a menudo es medir el poder de predicción de una cláusula. Usualmente esto se realiza utilizando los datos de entrenamiento o un conjunto de datos de testeo. Por otro lado, también se miden las complejidades de las hipótesis de modo de elegir la más simple de entre las posibles soluciones.

A continuación se describen algunas de las funciones de evaluación más populares utilizadas en los sistemas de ILP:

Entropía El valor de utilidad es $p.log_2(p)+(1-p)log_2(1-p)$ donde $p = P/(P+N)$ con P y N el número de ejemplos positivos y negativos cubiertos por la cláusula. Este tipo de formula proviene de Teoría de la Información y de alguna forma intentan medir la capacidad de «compresión» en bits de una hipótesis.

Cubrimiento El valor de utilidad es $P - N$ siendo P y N el número de ejemplos positivos y negativos cubiertos por la cláusula respectivamente.

Posonly La utilidad de la cláusula es calculada a partir de ejemplos solamente positivos. Esta función de evaluación puede ser utilizada en una modificación del algoritmo tradicional de ILP en la cual se busca un aprendizaje basado únicamente en ejemplos positivos [Muggleton, 1996]. La utilidad es calculada mediante un método Bayesiano. En muchos problemas los ejemplos de entrenamiento sólo son positivos, los ejemplos negativos son innaturales, por lo tanto este tipo de función de evaluación es muy práctica.

En nuestro caso se comenzó utilizando una función posonly. Tal elección se sustentó por el hecho de que se poseen ejemplos positivos pero no ejemplos negativos.

Como veremos más adelante, los patrones inducidos a partir de una función posonly fueron demasiados generales. Por lo tanto, luego se decidió utilizar la función de cubrimiento.

3.3.8. Sistema de ILP

Existen varios sistemas de ILP que implementan diferentes acercamientos. Esencialmente difieren en las características y decisiones de diseño e implementación introducidas anteriormente. Al inicio del proyecto se decidió utilizar el sistema CProgol de S. Muggleton [Muggleton and Firth, 1999]. Las razones fueron las siguientes:

- Es un sistema utilizado en varios dominios con éxito y se ha convertido en un estándar en ILP.
- Es de uso libre para investigación.
- Provee un fundamento teórico [Muggleton and Firth, 1999, Muggleton, 1995, 1996].
- Se basa en un aprendizaje dirigido por ejemplos lo cual reduce el espacio de búsqueda en el grafo de refinamiento de una cláusula.
- Permite la declaración de modos para restringir que predicados y funciones puede utilizar el sistema en la construcción de la teoría como así también los tipos y formatos de sus argumentos. Esto también reduce considerablemente el espacio de búsqueda.
- Permite la declaración de restricciones de integridad, poda y recursos. Esto también permite una considerable reducción del espacio de búsqueda y hace tratables en la práctica a problemas complejos.

Sin embargo, luego se encontró otro sistema de ILP llamado ALEPH (*A Learning Engine for Proposing Hypothesis*) de A. Srinivasan [Srinivasan, 2004], que posee varias mejoras con respecto a CProgol:

- Permite la configuración de muchos más parámetros relevantes en el algoritmo:
  1. Configuraciones relacionadas a la construcción de la cláusula base.
  2. El método de búsqueda puede ser alterado de modo de poder utilizar diferentes estrategias de búsqueda, funciones de evaluación y operadores de refinamiento.
- Se provee en código fuente Prolog, lo que permite explorar más a fondo el funcionamiento y realizar posibles modificaciones. Así fue posible acoplarlo con un sistema propio, CProgol está compilado en C y no provee una forma directa de interoperabilidad con otros programas.

- Puede ser utilizado tanto en SWI-Prolog [SWI, 2006] como en YAP [YAP, 2005], lo que permite utilizar cualquier predicado implementado en estos sistemas de Prolog como conocimiento de fondo. CProgol provee de un sistema de Prolog muy limitado.

- El manejo de tipos es más adecuado, en CProgol diferentes tipos de un mismo dominio pueden ser utilizados indistintamente lo que aumenta el espacio de búsqueda y produce resultados no deseados.

3.4. Resultados Obtenidos

En esta sección se muestran los resultados obtenidos en la generación de patrones según el diseño presentado en las secciones anteriores. Se finaliza con un análisis y evaluación de los patrones obtenidos.

Las sospechas de un riesgo en la obtención de patrones demasiado generales se confirmaron. Los patrones obtenidos se tratan efectivamente de patrones de Go, pero son demasiado primitivos como para tener una utilidad importante. Básicamente se obtuvieron dos clases de patrones. Por un lado se obtuvieron patrones del estilo de la regla 3.1, donde se indica donde poner una piedra en una intersección relativa a una piedra del oponente.

(3.1)
```
move(Board,Color,X,Y):-
    pos_gen(Pos),
    block_on_pos(Board,X,Y,Block_id,Disp_hor,Disp_ver,Pos),
    block_color(Board,Block_id,Color2),
    opposite(Color,Color2).
```

Como ejemplo particular, la regla 3.2 aconseja poner la piedra adyacente a una piedra del oponente. Esto es un movimiento fundamental para reducir las libertades del oponente.

(3.2)
```
move(Board, Color, X, Y) :-
    pos_gen(Pos),
    block_on_pos(Board, X, Y, Block_id, 0, -1, Pos),
    block_color(Board, Block_id, Color2),
    opposite(Color, Color2).
```

Por otro lado se encuentran reglas similares a la 3.3 que en su mayoría inducen a pensar que es más adecuado mover en la zona central del tablero. Este patrón es correcto, en tableros de 9x9 es muy importante mantener el control de la parte central del tablero.

(3.3)
```
move(Board, Color, X, Y) :-
    pos_gen(Pos),
    edge_ver(X, Y, -6, Pos),
    edge_hor(X, Y, -6, Pos).
```

Sin embargo, las reglas son demasiado generales. La razón más probable por la cual se aprendieron patrones tan generales se debe al hecho de no contar con

ejemplos negativos. Si se contara con ejemplos negativos, el sistema buscaría especializar la regla para no cubrirlo. Aparentemente el aprendizaje inductivo sobre ejemplos solamente positivos no da buenos resultados en nuestro problema.

El siguiente experimento consistió en descartar un aprendizaje solamente sobre ejemplos positivos. Se intentó ahora utilizar una función de evaluación de cobertura. Por lo tanto fue necesario contar con ejemplos negativos. Se crearon 27 ejemplos negativos que consistieron únicamente en ejemplos de movimientos donde se comete suicidio. Este intento padece el problema de que la cantidad de ejemplos positivos es muy superior a la cantidad de ejemplos negativos, por lo tanto en la función de evaluación cobertura[4] de $(P-N)$, P es mucho mayor que N, por lo que N no tiene un peso importante. Sin embargo, el sistema ALEPH mantiene un parámetro *noise* que indica la máxima cantidad de ejemplos negativos que una cláusula aceptable puede cubrir (incorrectamente). Estableciendo el parámetro noise a 0, es posible descartar patrones que cubran algunos de los ejemplos negativos creados, con lo que es de esperar obtener mejores resultados. En total se utilizó un conjunto de entrenamiento de 2257 ejemplos positivos y 27 ejemplos negativos.

Los resultados fueron más alentadores. A continuación se presentan algunos de los patrones obtenidos, muchos de ellos tienen una interpretación directa y conocida en el Go:

(3.4)
```
move(Board,Color,X,Y) :-
    pos_gen(Pos),
    block_on_pos(Board,X,Y,BlockId,2,1,Pos),
    block_color(Board,BlockId,Color).
```

(3.5)
```
move(Board,Color,X,Y) :-
    pos_gen(Pos),
    block_on_pos(Board,X,Y,BlockId,1,-1,Pos),
    block_color(Board,BlockId,Color).
```

(3.6)
```
move(Board,Color,X,Y) :-
    pos_gen(Pos),
    block_on_pos(Board,X,Y,BlockId,1,0,Pos),
    block_color(Board,BlockId,Color).
```

(3.7)
```
move(Board,Color,X,Y) :-
    pos_gen(Pos),
    block_on_pos(Board,X,Y,BlockId,-2,0,Pos),
    block_color(Board,BlockId,Color).
```

En el Go el concepto de conectividad entre bloques es fundamental. Es deseable tener grandes bloques a tener bloques pequeños separados. Por otro lado el objetivo del juego es controlar más territorio que el adversario. El hecho de buscar conectividad significa tender a ubicar las piedras más juntas entre sí, por otro lado, intentar maximizar el territorio implica dispersar más las piedras. Por lo tanto, el jugador debe de buscar un equilibrio adecuado entre buscar la conectividad entre sus bloques (tener bloques fuertes) y ocupar mayor territorio.

[4] P y N representan la cantidad de ejemplos positivos y negativos cubiertos por una cláusula respectivamente.

Dentro de los patrones de conexión se encuentra la regla 3.6 que se conoce como *conexión directa*: se ubica una piedra del mismo color adyacente a (en una libertad de) un bloque del mismo color para formar el mismo bloque.

La regla 3.5 se conoce como *conexión diagonal* (figura 3.4, movimiento 1) dado que dos piedras del mismo color se encuentran en diagonal con dos puntos posibles de conexión. En la figura, si blanco jugara en A, negro conectaría en B, y viceversa.

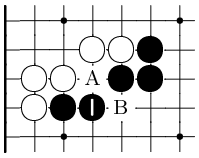


Figura 3.4: Conexión diagonal.

Las reglas 3.4 y 3.7 se conocen como *saltos*. En la figura 3.5, las piedras negras están separadas por saltos de un punto. En el Go se denomina *ikken-tobi* (3.7). Si Blanco pretende interferir en la conexión jugando en A, la respuesta negra en B ó C coloca a la piedra en *atari*. Es probable que del intento el que salga perjudicado sea Blanco. Por otra parte, si éste se «asoma» a la conexión con una jugada en C, entonces Negro debe conectar inmediatamente en A. En la misma figura, las dos piedras blancas están separadas según el movimiento del caballo en el ajedrez. Este salto se llama *keima* (3.4) en el Go. Si Negro trata de interferir en la conexión jugando en D, Blanco lo debe contener contestando en E. Si en cambio Negro ataca con una jugada en E, la respuesta blanca en D confina la piedra contra el borde.

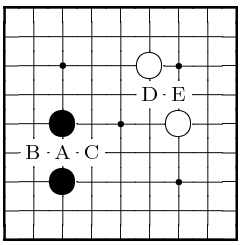


Figura 3.5: Ikken-tobi y Keima.

Las siguientes tres reglas inducidas:

(3.8)
```
move(Board,Color,X,Y) :-
    pos_gen(Pos),
    edge_ver(X,Y,-6,Pos),
    stage(Board,open).
```

(3.9)
```
move(Board,Color,X,Y) :-
    pos_gen(Pos),
    edge_ver(X,Y,-5,Pos),
    stage(Board,open).
```

(3.10)
```
move(Board,Color,X,Y) :-
        pos_gen(Pos),
        edge_hor(X,Y,-4,Pos),
        stage(Board,open).
```

Indican patrones de juego en la apertura de la partida. Según estos patrones en la apertura es preferible jugar en la zona central del tablero. Estos patrones son correctos, en una partida de Go 9x9 es importante realizar movimientos en el centro del tablero para tomar control inicial del mismo.

Las siguientes dos reglas:

(3.11)
```
move(Board,Color,X,Y) :-
        pos_gen(Pos),
        edge_ver(X,Y,-6,Pos),
        edge_hor(X,Y,-4,Pos).
```

(3.12)
```
move(Board,Color,X,Y) :-
        pos_gen(Pos),
        edge_ver(X,Y,-7,Pos),
        edge_hor(X,Y,-3,Pos).
```

Indican que es conveniente jugar en la zona central del tablero (evitar jugar en los bordes), pero intentar mantener el control de uno de los cuadrantes (figura 3.6).

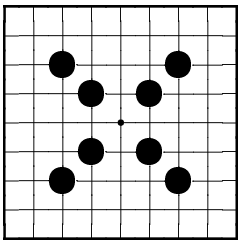


Figura 3.6: Intentar jugadas centrales que tengan influencia en cuadrantes.

El resto de las reglas no tienen una interpretación tan directa, pero es interesante ver que, como se previó al inicio del experimento, se obtuvieron reglas más específicas. A diferencia que en el caso anterior, estas reglas incorporan nuevos predicados como ser *liberty_cnt/3*, *range/3* y *stage/2*.

Por otro lado, es importante destacar que la expresividad del lenguaje permite interpretar fácilmente el significado de los patrones.

Luego, muchos experimentos fueron realizados variando los parámetros que involucran al algoritmo de aprendizaje. En nuestro caso se experimentó con diferentes valores en el parámetro de *noise*, la cantidad de ejemplos positivos, y la cantidad de ejemplos negativos. Por otro lado también es posible afectar de una forma importante la capacidad de aprendizaje del sistema al variar la cantidad y calidad del conocimiento previo.

El experimento (A) consistió en utilizar un conjunto de entrenamiento que consiste de:

1. Movimientos correspondientes a 3 partidas de jugadores profesionales de nivel dan como ejemplos positivos (cada partida consiste de alrededor de 50 movimientos por lo que serían 150 movimientos de ejemplo aproximadamente).

2. Movimientos realizados por un jugador randómico frente a GNU-GO como ejemplos negativos.

3. Un valor de $noise = 10$, es decir, una regla inducida tiene que cubrir menos de 10 ejemplos negativos.

4. Como predicados de conocimiento previo se utilizarán predicados que describan conceptos básicos: bloques, libertades, cantidad de piedras, distancia a los bordes, etapa de la partida.

5. Un conjunto de testeo, independiente del conjunto de entrenamiento, compuesto por 450 movimientos elegidos de partidas de profesionales de nivel dan.

El experimento (B) consistió en la misma configuración que A salvo el siguiente punto:

1. Se agregaron al conocimiento previo, predicados más avanzados, que representan conceptos importantes en el Go como ser: atari, captura, conexión.

La evaluación consistió en, dada una posición de testeo, seleccionar los mejores 20 movimientos según las reglas inducidas. Para cada regla inducida, el sistema devuelve dos valores: cantidad de ejemplos positivos cubiertos (P) y cantidad de ejemplos negativos (N) cubiertos por la regla (cobertura medida sobre los ejemplos de entrenamiento). El valor P - N (cobertura) será utilizado como medida de cuan buena es una regla, y de esta forma, generar un ordenamiento entre las reglas inducidas. De esta forma, para seleccionar los mejores N movimientos se procede de la siguiente forma:

1. Dada una posición, se obtienen todos los movimientos posibles (que sean válidos según las reglas del juego).

2. Para cada uno de estos movimientos, se obtiene un «puntaje» que se calcula como la suma de la cobertura de todas las reglas que aplican a dicho movimiento.

3. Se ordenan los movimientos según su puntaje y se devuelven los N mejores según dicha medida.

Los resultados en los experimentos A y B fueron los siguientes:

Experimento	A	B
Performance	57.9 %	71.3 %

Cuadro 3.1: Resultados experimento A y B.

Las experiencias muestran un resultado bastante claro: el solo hecho de agregar conocimiento previo más elaborado aumenta considerablemente la performance de las reglas inducidas. Ahora busquemos la razón de este resultado.

Por un lado es bastante directo suponer que este resultado tiene sentido, ya que al incorporar mayor conocimiento al sistema, es de esperar que este tienda a devolver mejores resultados. Por otro lado, los predicados avanzados que se agregaron al experimento B están compuestos pura y llanamente de predicados básicos utilizados en A. Entonces, ¿por qué en el experimento A no se obtuvieron los mismos resultados si es posible representar los mismos conceptos utilizando más predicados básicos? La respuesta viene por las limitaciones de la práctica. Para poder aprender conceptos como atari, o captura, el sistema debe realizar una búsqueda mucho más profunda en el árbol de refinamiento, esto lleva mucho tiempo y memoria, y en la práctica es necesario limitar la profundidad máxima a la que el sistema puede llegar.

Aquí es donde la práctica hace que muchas veces los resultados teóricos no sean alcanzables. Por lo tanto, en un sistema de aprendizaje de este tipo, es necesario buscar un equilibrio adecuado para poder lograr un aprendizaje útil. El hecho de agregar conocimiento previo más elaborado, hace que el sistema parta de un conocimiento que le permite representar conceptos más abstractos y complejos. De esta forma es posible combatir las limitaciones de la práctica que en nuestro caso se traducen en un límite en la profundidad de búsqueda.

Sin embargo, no todos los problemas son iguales. Si tratáramos con un problema completamente desconocido, en donde no conociéramos nada sobre como resolver el problema, no contaríamos con conocimiento previo elaborado, por lo que no habría más remedio que dejar al sistema, horas, días, meses, buscando en el árbol de refinamiento, y obtener el mismo resultado que si incorporáramos conocimiento previo (desconocido) que nos permita resolver el problema en minutos.

Los siguientes experimentos a probar consistieron en aumentar los ejemplos de entrenamiento. Las configuraciones y los resultados obtenidos se muestran en la tabla 3.2[5].

$\|E^+\|$	$\|E^-\|$	Noise	Perf.	Tiempo
750	70	10	68 %	10 min
1500	70	10	69.3 %	17 min
750	140	20	71.7 %	25 min
1500	140	20	74.3 %	40 min
1500	280	40	74 %	42 min
3000	280	40	72.7 %	61 min

Cuadro 3.2: Resultados experimento A y B.

[5]Se utilizó un PC Celeron 2.4 GHz 256 MB RAM en plataforma Linux

Capítulo 4

Aprendizaje de la Función de Evaluación

En este capítulo se presentan las decisiones de diseño del segundo componente principal en la implementación de un jugador de Go: el sistema de aprendizaje de una función de evaluación para tableros de Go.

En la sección 4.1 se discuten los temas relacionados a seleccionar un modelo de función adecuado. Para esto se utilizarán *Redes Neuronales*, por lo que se dará una breve introducción al concepto. En la sección 4.2 discutiremos los puntos importantes en la definición de la arquitectura de la red. En la sección 4.3 se introducen todos los conceptos necesarios para el ajuste de la red a través de aprendizaje automático, para esto se introducen las técnicas de *Backpropagation* y *Aprendizaje por Diferencia Temporal*. En la sección 4.4 se contemplan los puntos más importantes a definir con respecto a el método de entrenamiento a seguir. Por último, en la sección 4.5 se muestran los resultados obtenidos.

4.1. Representación de la Función: Redes Neuronales

El primer paso en la construcción de una función de evaluación consiste en elegir un modelo adecuado para representarla. Un acercamiento simple podría ser representar la función como una combinación lineal de atributos de una posición. Otras técnicas podrían utilizar funciones polinómicas o aún más complejas. La elección depende del dominio. Es necesario buscar el equilibrio adecuado entre poder de expresividad y tiempo de cálculo: una función podría tener la complejidad adecuada pero ser demasiado poco eficiente. Por otro lado, si la función es demasiado simple podría no poder evaluar correctamente la posición.

Las redes neuronales tienen una gran utilidad para el modelado y aprendizaje de funciones reales sobre entradas reales basado en un modelo conexionista [Mitchell, 1997]. Su gran capacidad de representación de funciones y su uso exitoso en otros problemas, sugirieron su utilización en este proyecto para el modelado de una función de evaluación para el Go. Básicamente consisten en redes de unidades de procesamiento elementales llamadas *neuronas* o simplemente *unidades*, que conectadas entre si, globalmente pueden exhibir un comportamiento complejo. Tal comportamiento depende de las conexiones entre las unidades de procesamiento, dichas conexiones se modelan como pesos. La inspiración original de esta técnica

surgió de la observación del comportamiento del sistema nervioso central y las neuronas biológicas. Su uso práctico surge del hecho de poder utilizarlas en conjunto con algoritmos diseñados con el objetivo de alterar los pesos de las conexiones entre las neuronas para producir la salida deseada.

A continuación se introducen los conceptos de *Perceptrón* y *Perceptrón Multicapa*, que serán utilizados para el modelado de una función de evaluación para tableros de Go.

4.1.1. Perceptrón

Una neurona se modela como una función compuesta por una combinación lineal de las entradas con los pesos y una función:

$$f(x) = K(\sum_i w_i x_i)$$

Donde w es el vector de pesos, x es el vector de entradas y K una función predefinida, conocida como *función de activación*.

Un caso muy conocido de unidad utilizado en redes neuronales es el *Perceptrón*, en este caso, la función de activación se trata de la función signo (ver figura 4.1 [Mitchell, 1997]).

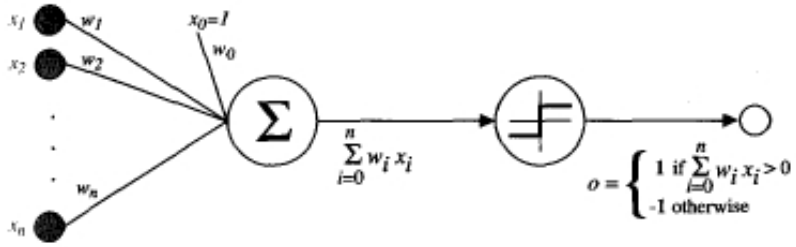


Figura 4.1: Diagrama de un Perceptrón

Como veremos más adelante, es deseable que la función de activación sea diferenciable. Por lo tanto es usual utilizar la función *sigmoidal* o la *tangente hiperbólica*. Estas funciones llevan todo el rango de reales a un rango fijo ([0,1] en el caso sigmoidal y [-1,1] en el caso tangente hiperbólica) y dan como resultado una función diferenciable y no lineal de sus entradas.

4.1.2. Perceptrón Multicapa

Un único perceptrón puede representar solo funciones lineales, en contraste un *Perceptrón Multicapa* (MLP por sus siglas en inglés: *Multi Layer Perceptron*), es capaz de representar una variedad muy rica de funciones no lineales [Mitchell, 1997]. Básicamente la idea es modelar una función a través de la composición de varias unidades. Esto se logra conectando las salidas de unidades con entradas de otras, formando así una red.

La arquitectura más popular en las redes neuronales es la de *redes de alimentación hacia delante (feedforward networks)*, en donde las señales de entrada son propagadas hacia la salida a través de cero o más capas ocultas. La figura 4.2

[Wikipedia, 2006] muestra un ejemplo de una red neuronal artificial con una capa oculta compuesta por m unidades ocultas.

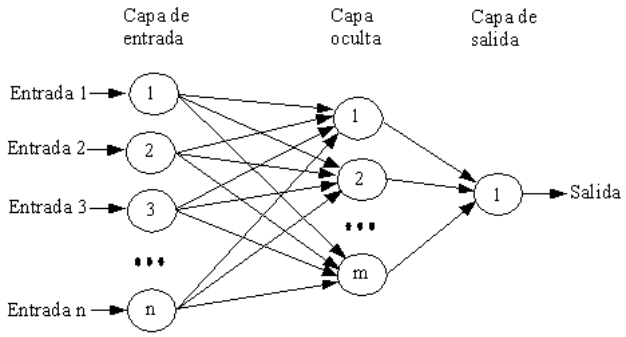


Figura 4.2: Ejemplo un MLP.

En este proyecto utilizaremos dos tipos de arquitecturas: redes sin capa oculta y redes con una capa oculta.

4.2. Arquitectura de la Red

Existen varias propiedades que ajustar a la hora de diseñar una red neuronal. En este proyecto se decidió no profundizar demasiado en estos aspectos y se decidió desarrollar arquitecturas no demasiado complejas. Entre los puntos que se consideraron a la hora de diseñar la red neuronal se incluyen:

- Definición de Entradas de la Red
- Cantidad de Niveles y Cantidad de Unidades Ocultas
- Función de Activación

4.2.1. Definición de Entradas de la Red

El siguiente paso consiste en definir cuáles son las entradas a utilizar, por lo que es necesario definir un conjunto de atributos que permitan caracterizar posiciones de tablero.

La definición de las entradas es una tarea sumamente importante y que afecta directamente al resultado. Si no definimos un conjunto de entradas que describan adecuadamente a un tablero, el sistema no tendrá el «material» suficiente sobre el cual modelar una función que describa correctamente el concepto objetivo. Por bueno que sea el algoritmo de aprendizaje, si las entradas no aportan los datos necesarios, el sistema no podrá aprender una función de evaluación útil.

En términos de aprendizaje, la definición de entradas podría verse como la definición de conocimiento previo; al seleccionar un conjunto dado de entradas, implícitamente estamos aportando conocimiento al sistema. Por lo tanto, según el grado de abstracción, calidad y cantidad de entradas, estaremos aportando más o menos conocimiento previo. Es de esperar que al proveer de una cantidad de entradas importantes y de gran calidad, el sistema tenga más oportunidad de poder

aprender el concepto deseado. Por otro lado, en la práctica puede ser deseable tener un número reducido de entradas, ya que esto implica un número de pesos menor que ajustar en la red, y por lo tanto, un tiempo de entrenamiento menor.

Las redes neuronales pueden ser vistas como un mapeo del espacio de entrada a un espacio de salida. El cubrir un gran espacio de entrada requiere recursos, y en la mayoría de los casos, la cantidad de recursos necesarios es proporcional al hipervolúmen del espacio de entrada. Esto puede causar que una red con una gran cantidad de entradas irrelevantes se comporte relativamente mal. Si la dimensión del espacio de entrada es alto, la red utilizará casi todos sus recursos en intentar representar partes irrelevantes del espacio. Inclusive, si las entradas son útiles, serán necesarios muchos datos de entrenamiento para poder ajustar la red. Este problema es conocido en el campo de las redes neuronales como la *maldición de la dimensionalidad (curse of dimensionality)* [AI-FAQs, 2006].

Por lo tanto, parte del objetivo de definir las entradas de la red consiste en lograr un conjunto que busque el equilibrio de ser suficientemente representativo, y al mismo tiempo, ser suficientemente reducido para evitar caer en los problemas prácticos descritos.

Es necesario entonces investigar en el dominio concreto cuáles entradas podrían ser útiles para aprender el concepto. En el caso del Go existen varios conceptos importantes a modelar. Algunos trabajos previos [van˜der Werf, 2004, Enzenberger, 1996, Zaman and Wunsch, 1999] proponen representar los siguientes conceptos:

- Cantidad de piedras de cada color capturadas
- Libertades de cada bloque de cada color
- Cantidad de piedras de cada bloque de cada color
- Cantidad de ojos de cada bloque de cada color
- Si hay bloques de cada color en atari
- Cantidad de territorio controlado por de cada color
- Conectividad de los bloques de cada color

Por otro lado, Zaman y Wunsch [Zaman and Wunsch, 1999] proponen la utilización de entradas más elementales, por ejemplo, en un tablero de 9x9 tener 81 entradas, una por cada intersección que devuelva 1,-1 o 0 si la intersección tiene una piedra negra, blanca o vacío respectivamente.

En este proyecto se decidió utilizar el primer tipo de atributos. De esta forma se posee un conjunto más reducido de entradas, que al mismo tiempo, representan conceptos importantes del Go.

No está de más recordar que si deseamos modelar la función de evaluación mediante una red neuronal, es necesario que las entradas puedan ser representadas como valores numéricos. Por lo tanto debemos investigar la forma de representar numéricamente conceptos abstractos como los de territorio y conectividad.

A continuación se describe cada atributo propuesto y su posible representación. Se adopta la siguiente convención: se utiliza el *signo positivo para magnitudes que favorecen al color blanco y el signo negativo para magnitudes que favorecen al color negro.*

Libertades Para representar el concepto de libertad de los bloques, en principio, surge la idea de tener una entrada por cada bloque de la posición que indique sus libertades, y agregándole un signo positivo o negativo dependiendo de si se trata de un bloque de piedras blancas o negras. El problema con esta idea es que la cantidad de bloques a medida que transcurre una partida es variable. Por lo tanto, esto obligaría a tener una cantidad variable de entradas, lo que requeriría una arquitectura de red neuronal dinámica. En nuestro caso se decidió por una arquitectura estática ya que simplifica el problema.

Para solucionar el problema de representar el concepto de libertad con una cantidad fija de entradas, básicamente tenemos dos posibles diseños. El primero consiste en representar el concepto de libertad como un valor global que se obtiene de la suma de todas las libertades de los bloques de un mismo color, de esta forma tendríamos una entrada por cada color. El segundo diseño es tener una sola entrada que sea la diferencia entre suma de libertades de bloques blancos menos suma de libertades de bloques negros. Es claro en el Go que el tener mayores libertades favorece a un jugador, por lo que un valor positivo de esta entrada favorecería al color blanco y un valor negativo al negro, tal como lo definimos en nuestra convención.

El concepto de libertad puede extenderse a varios órdenes. Las libertades usuales se pueden definir como libertades de grado 1. Se define libertades de grado 2 como libertades de libertades, es decir, cantidad de intersecciones adyacentes a las libertades de grado 1. Aplicando la misma idea podemos definir libertades de grado n.

Como ejemplo, Van der Werf [van˜der Werf, 2004] propone utilizar la diferencia de libertades de grado 1, 2 y 3 en la función de evaluación de un jugador artificial de Go 5x5, *PONNUKI*.

Otra posibilidad es tener una entrada por cada intersección, cada una devolverá *+/- libertades de la piedra en la intersección* (según la piedra sea blanca o negra) o 0 si la intersección es vacía. Sin embargo, en un tablero de 9x9 esto representaría 81 entradas diferentes.

En definitiva se decidió utilizar la cantidad de libertades de diferentes ordenes ya que es posible calcularlas eficientemente y ofrece una representación compacta.

Cantidad de Piedras Una opción similar al caso anterior es la de devolver la cantidad de piedras blancas menos la cantidad de piedras negras. Otra opción puede ser tener 81 entradas, una por cada intersección, que devuelvan +1 si hay una piedra blanca, -1 si hay una piedra negra o 0 si está vacía. Utilizaremos la primer opción ya que condensa mejor la información.

Conectividad y Ojos Las conectividad y los ojos son conceptos más difíciles de representar y calcular. Van der Werf describe una solución que afortunadamente es una manera eficiente de combinar las estimaciones de ambos conceptos en un solo número: el *número de Euler* [Gray, 1971].

El número de Euler en una imagen binaria es el número de objetos menos el número de agujeros en tales objetos. Por lo tanto al minimizar el número de Euler, se tiende a conectar las piedras y crear ojos. Una propiedad atractiva del número de Euler es que es localmente calculable: es posible calcularlo contando la cantidad de apariciones de los tres tipos de patrones mostrados en la figura 4.3 [van˜der Werf,

2004]. De la cantidad de patrones encontrados $n(Q1)$, $n(Q3)$ y $n(Qd)$, calculamos el número de Euler como:

$$E = \frac{n(Q_1) - n(Q_3) + 2n(Q_d)}{4}$$

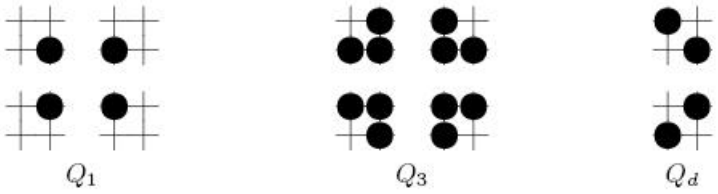


Figura 4.3: Patrones a contar para calcular el número de Euler.

Territorio El concepto de territorio usualmente es estimado en la literatura de Go mediante varios métodos [van˜der Werf, 2004]. Una opción conocida como *control por distancia* asigna una intersección al color que tenga una piedra más cercana. Sin embargo, esta forma de calcular territorio no tiene en cuenta el poder de cada piedra. Por ejemplo, una sola piedra sería tan importante como un bloque con varias piedras a la misma distancia.

Una forma de solucionar este problema es utilizar *funciones de influencia* las cuales fueron propuestas inicialmente por Zobrist. El algoritmo es simple: primero todas las intersecciones son inicializadas con los valores 50, -50 y 0 según haya una piedra blanca, negra o vacío. Luego el siguiente proceso es ejecutado 4 veces: para cada intersección se suman la cantidad de intersecciones adyacentes con valor positivo y se restan la cantidad de intersecciones adyacentes con valor negativo.

Una mejora de la influencia de Zobrist es el algoritmo conocido como *método de Bouzy*. Este algoritmo fue inspirado en un trabajo previo de Zobrist y algunas ideas de visión por computadora. Se basa en dos operaciones simples de morfología matemática: *dilatación* y *erosión*. La dilatación consiste en:

Primero se inicializan las intersecciones blancas con 128, las intersecciones negras con -128 y las intersecciones vacías con 0. El operador de dilatación consiste en:

- Para cada intersección diferente de 0 que no es adyacente a una intersección del signo opuesto, contar la cantidad de intersecciones del mismo signo y sumarlas a su valor absoluto.
- Para cada intersección 0 sin intersecciones negativas adyacentes, sumar la cantidad de intersecciones adyacentes con valor positivo.
- Para cada intersección 0 sin intersecciones positivas adyacentes, restar la cantidad de intersecciones adyacentes con valor negativo.

El operador de erosión consiste en:

- Para cada intersección diferente de 0, restar de su valor absoluto la cantidad de intersecciones adyacentes con valor 0 o de signo opuesto.

- Si esto causa que el valor de la intersección cambie de signo, el valor queda en 0.

Al aplicar 5 veces dilatación y 21 veces erosión se determina el territorio de una manera muy parecida a la forma en que los humanos lo hacen [GNU-GO, 2006].

Veamos dos ejemplos de la aplicación del método de Bouzy.

En este primer ejemplo, hay dos piedras, una negra (columna 2, fila 4) y una blanca (columna 6, fila 4) enfrentadas, cada color controla 2 intersecciones de territorio.

```
  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   0   0   0   0
  0  -2 -122   0   0   0 122   2   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   0   0   0   0   0
```

En el segundo ejemplo, hay dos piedras negras (columna 0, filas 4 y 5), dos piedras blancas (columna 6, filas 2 y 6). Blanco controla 20 intersecciones de territorio y Negro 7. Como vemos, blanco ha distribuido sus piedras de forma de tener influencia sobre mayor territorio que negro.

```
  0   0   0   0   0   0   0   5   4
  0   0   0   0   0   0   4  12   7
  0   0   0   0   0   0 127  14   8
  0   0   0   0   0   0   0   4   7
 -1  -9 -126  0   0   0   0   0   5
 -9 -14 -125  0   0   0   0   3   7
 -1   0   0   0   0   0 123  14   8
  0   0   0   0   0   0   0  12   7
  0   0   0   0   0   0   0   3   4
```

En definitiva, se utilizó el método de Bouzy para calcular la cantidad de territorio controlado por cada color y se representa una entrada que se calcula como la diferencia entre la cantidad de territorio blanco y la cantidad de territorio negro.

Cantidad de Bloques en Atari Recordemos que un bloque se encuentra en atari si posee una sola libertad. Este concepto claramente es importante, porque si un bloque se encuentra actualmente en atari, entonces en el siguiente movimiento podrá ser capturado (ver sección 2.1).

Una posible forma de representar esto es simplemente contar los bloques de cada color y devolver la diferencia. Otra forma podría ser devolver la diferencia de cantidad de piedras en atari. Este último método en principio es mejor ya que aporta más información: no es lo mismo tener 1 bloque con 1 piedra en atari, que tener 1 bloque con 5 piedras en atari; claramente es más importante concentrarse

en defender/capturar el bloque con 5 piedras. La segunda representación permitiría a la red distinguir estos dos casos, por lo que será utilizada.

En conclusión, las entradas a utilizar consistieron en las diferencias del valor blanco menos el negro en los siguientes atributos:

1. Cantidad de libertades de orden 1.
2. Cantidad de libertades de orden 2.
3. Cantidad de libertades de orden 3.
4. Número de Euler.
5. Cantidad de piedras capturadas.
6. Cantidad de territorio.
7. Cantidad de piedras en el tablero.
8. Cantidad de piedras atari.

En principio estas entradas representan varios conceptos importantes del Go que describen los atributos más importantes de una posición. Pueden ser calculados eficientemente lo cual influye mucho en la eficiencia de la función de evaluación.

4.2.2. Cantidad de Niveles y Cantidad de Unidades Ocultas

Con respecto a este punto, se utilizan dos diseños:

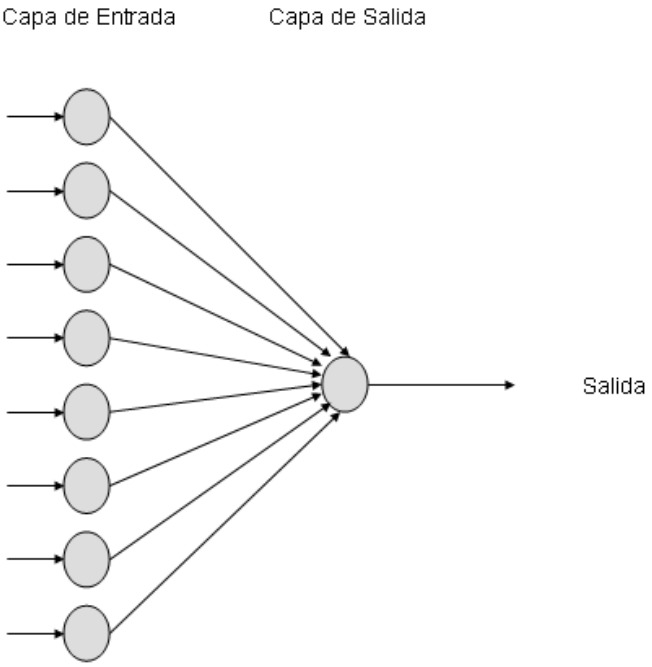
A Un Perceptrón

B Un Perceptrón Multicapa (MLP)

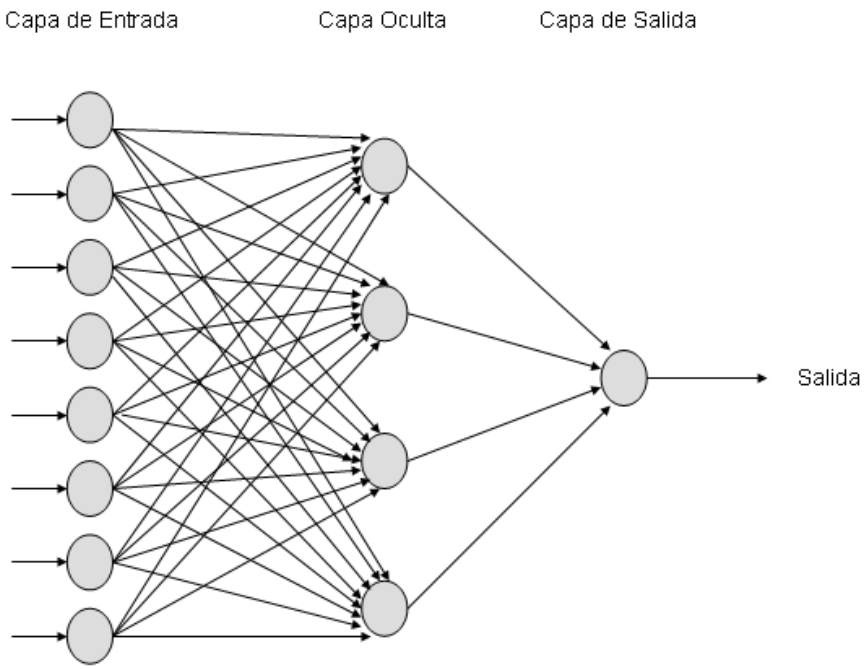
Para el caso A se decidió utilizar una red neuronal de un nivel, es decir, sin capa oculta. En el caso B se utilizó una red de dos niveles con una capa oculta compuesta por 4 unidades ocultas completamente conectadas. En la figura 4.4 se muestra un bosquejo de ambos diseños.

A priori sabemos que teóricamente la capacidad de expresión de la red B es mayor que la de A. Esto se debe a que la red A puede expresar solamente una relación lineal entre sus entradas. En el caso de la red B se ha demostrado que una red neuronal con una capa oculta puede aproximar prácticamente cualquier función con una cantidad suficiente de unidades ocultas [Mitchell, 1997]. Por lo tanto, es de esperar que la red B pueda representar un concepto más complejo (no lineal) mientras que la red A se limitará a dar la mejor representación lineal que ajuste al concepto buscado.

Por otro lado, si el concepto buscado es una relación lineal entre las entradas definidas en nuestra red, está demostrado teóricamente [Mitchell, 1997] que el algoritmo de aprendizaje por descendiente de gradiente (que más adelante se presentará) converge al concepto buscado. Sin embargo esta convergencia no está asegurada para redes no lineales, por lo que aquí el diseño A puede tener ventajas.



(a) Diseño A: Perceptrón de 8 entradas y 1 salida.



(b) Diseño B: Perceptrón Multicapa de 8 entradas, 4 unidades ocultas y 1 salida.

Figura 4.4: Diagramas de diseño de las redes A y B.

Por último, otra ventaja importante que posee el diseño A sobre el B es más que nada práctica: en la red A se deben ajustar mucho menos pesos que en la red B, por lo que, intuitivamente, es probable que la convergencia se dé mucho más rápido en A que en B. En el caso de A tendremos que ajustar *cant_entradas* pesos, mientras que en el caso B se ajustarán $(cant\_entradas \times cant\_unidades\_ocultas) + cant\_unidades\_ocultas$ pesos, lo cual es una gran diferencia.

La diferencia de velocidad no solo será diferente a la hora de entrenar las redes, sino también a la hora de evaluar la función. La función será utilizada en cada nodo a la profundidad máxima del árbol de búsqueda alfa-beta, por lo que el aumento de cómputos necesarios para calcular este valor tiene un efecto notorio en la cantidad de tiempo necesario para la selección de un movimiento.

Se resume en la tabla 4.1 la comparación a priori de ambos diseños.

	Perceptrón	**MLP**
Capacidad de expresión	funciones lineales	cualquier función
Convergencia	asegurada	no asegurada
Velocidad de convergencia	más rápida	más lenta
Velocidad de evaluación	más rápida	más lenta

Cuadro 4.1: Comparación de ventajas y desventajas de utilizar una red lineal o una red con una capa oculta.

4.2.3. Función de Activación

En ambas arquitecturas de red se decidió utilizar una función de activación tangente hiperbólica. Usualmente se utiliza una función sigmoidal que devuelve valores entre 0 y 1. A diferencia de ésta, la tangente hiperbólica devuelve valores entre -1 y 1. Se eligió esta ultima porque expresa mejor la simetría entre colores. De este modo representamos el valor de una partida con valor 1 si el jugador blanco gana, -1 si el jugador negro gana y 0 si empatan. El empate no es relevante ya que en el Go el valor de komi siempre tiene el valor de $n + 1/2$ (n natural). El resultado se calcula como: *territorio blanco - territorio negro + komi*, por lo que nunca se obtiene un valor entero y en definitiva nunca un valor de 0.

4.3. Algoritmo de Aprendizaje

Luego de haber seleccionado una arquitectura, es necesario utilizar algún algoritmo de aprendizaje para el ajuste de los pesos de la red. Los más exitosos y utilizados se basan en el *descendiente de gradiente*. Básicamente consiste en obtener las derivadas parciales de la red para cada peso respecto a una medida de error, que usualmente suele ser el cuadrado del error cometido por la función. El opuesto del gradiente indica con qué dirección y magnitud realizar la modificación en el peso para disminuir el error y por lo tanto hacer que la función se ajuste mejor al concepto buscado. En las redes sin unidades ocultas el cálculo es simple. Para el caso de contar con unidades ocultas es necesario utilizar repetidamente la regla de la cadena, lo que se conoce como *propagación reversa (backpropagation)*.

Usualmente, backpropagation es utilizado en un aprendizaje supervisado donde los datos de entrenamiento tienen la forma *(X,d)*, donde X representa el vector de entradas de la función y d el valor esperado. Sin embargo, en nuestro caso, tenemos el vector X de entradas (correspondiente a las entradas definidas en la sección 4.2.1), pero en solo un caso tenemos el valor d: en la última posición de la partida (posición final). El valor d de una posición final corresponde a 1 o -1 si gana blanco o negro respectivamente. Aquí surge entonces el problema de asignar valores de d a posiciones intermedias dentro de una partida. Para resolver esto, se utilizó *Aprendizaje por Diferencia de Temporal* (*Temporal Difference Learning*).

4.3.1. Backpropagation

Usualmente es utilizado el algoritmo de *Backpropagation* [Mitchell, 1997, Nilsson, 1996] para realizar los ajustes de los pesos. El objetivo es minimizar una medida de error. En nuestro caso la medida de error es:

$$\epsilon = \frac{1}{2}(d - f(X))^2$$

Donde f es la función a ajustar (la red neuronal) y d la salida deseada para el vector de entradas X. Deseamos minimizar este error, por lo que se suele utilizar el descendiente de gradiente. La idea es ajustar cada peso w de la red según la dirección opuesta a que crece la derivada parcial del error respecto a tal peso:

$$\triangle w = -\eta \frac{\partial \epsilon}{\partial w}$$

Donde η es conocida como la *constante de aprendizaje*. Por lo tanto queda por calcular la derivada parcial $\frac{\partial \epsilon}{\partial w}$. Recordemos que cada unidad elemental (neurona) de la red, devuelve el valor *tanh(net)*, siendo $net = W \cdot X$ ($net = \sum_i^n w_i x_i$), W el vector de pesos asociado a la unidad y X su vector de entradas.

En este punto debemos comenzar a diferenciar entre las dos posibles arquitecturas: una red lineal y una red con una capa oculta. A continuación se realizan los cálculos necesarios para una red lineal. Para el caso de una red con una capa oculta, el razonamiento es análogo.

Red Lineal En este caso se trata de una única unidad, por lo que se tienen n pesos, uno por cada una de las n entradas:

$$\triangle w_i = -\eta \frac{1}{2} \frac{\partial (d - f(X))^2}{\partial net} x_i$$

En en el caso de la red lineal *f(X) = tanh(net)*, por lo que:

$$\triangle w_i = -\eta \frac{1}{2} 2(d - f(X))(-\frac{\partial tanh(net)}{\partial net}) x_i$$

Una de las ventajas de la tangente hiperbólica es su fácil expresión de su derivada en función de su salida:

$$\frac{\partial tanh(net)}{\partial net} = 1 - tanh(net)^2$$

La expresión de actualización de los pesos de la red queda:

$$\triangle w_i = \eta(d - f(X))(1 - f(X)^2)x_i$$

Por lo tanto, ya contamos con la regla de actualización de los pesos, obtenida a partir de la regla de descendiente de gradiente.

4.3.2. Aprendizaje por Diferencia Temporal (TD)

Supongamos que queremos estimar el valor de una función f en el tiempo $m+1$, el valor exacto de f en el tiempo $m+1$ es d. Si tenemos una secuencia de ejemplos $X_1 \dots X_m$, en un acercamiento de aprendizaje supervisado buscaremos aprender una función f tal que $f(X_i)$ sea lo más parecido posible a d para cada i. Típicamente se necesita un conjunto de entrenamiento con varias secuencias, en cada paso i, se busca ajustar los pesos W de modo que la diferencia entre $f(X_i)$ y d sea mínima. Por lo tanto, para cada X_i, la predicción $f(X_i, W)$ se computa y se compara con d. Luego la regla de actualización de los pesos (cualquiera sea), computa el cambio $\triangle W_i$ que se realizara a los pesos W. Al final se obtiene un peso W_{m+1} final como:

$$W_{m+1} = W_0 + \sum_{i=1}^{m} \triangle W_i$$

Si se busca minimizar el cuadrado del error entre d y $f(X_i, W)$ mediante un descenso en el gradiente, la regla de ajuste de los pesos para cada ejemplo será:

$$\triangle W_i = c(d - f_i)\frac{\partial f_i}{\partial W}$$

siendo c un parámetro de razón de aprendizaje y $f_i = f(X_i, W)$ la predicción de d en el tiempo $t = i$.

A partir de la observación:

$$(d - f_i) = \sum_{k=i}^{m} f_{k+1} - f_k$$

definiendo $f_{m+1} = d$ y sustituyendo en la formula de $\triangle W_i$, se obtiene:

$$(\triangle W)_i = c(d - f_i)\frac{\partial f_i}{\partial W} = c\frac{\partial f_i}{\partial W}\sum_{k=i}^{m} f_{k+1} - f_k$$

En este caso, en vez de usar la diferencia entre la predicción $f(X_i)$ y el valor de d, se utilizan las diferencias entre las predicciones sucesivas $f(X_i)$ y $f(X_{i+1})$. Aquí es donde surge el algoritmo TD [Nilsson, 1996], por lo que se le conoce con el nombre de aprendizaje por diferencias temporales (*temporal difference learning*).

La ecuación anterior permite una generalización muy interesante:

$$(\triangle W)_i = c\frac{\partial f_i}{\partial W}\sum_{k=i}^{m} \lambda^{k-i}(f_{k+1} - f_k)$$

Con $0 < \lambda <= 1$. El parámetro λ le da pesos exponencialmente decrecientes a las diferencias más posteriores en el tiempo que $t = i$. Cuando $\lambda = 1$, tenemos un aprendizaje supervisado. Mientras que con $\lambda = 0$ solo pesa la diferencia $f_{i+1} - f_i$.

El método con el parámetro λ se lo conoce como $TD(\lambda)$. Para valores intermedios menores que 1, tenemos varios grados de aprendizaje no supervisado (*unsupervised learning*).

En definitiva utilizaremos diferencia temporal con $\lambda = 0$. Esto tiene una interpretación directa: para aproximar el valor de d para una posición intermedia, se utiliza el valor de la función de evaluación en la siguiente posición:

$$d_m = f(X_{m+1})$$

Esto es, al tener como entrada la m-ésima posición de la partida, el valor esperado d_m se aproxima como el valor de la función en la siguiente posición ($f(X_{m+1})$).

Esto desde el punto de vista de aprendizaje automático, se puede entender como que el sesgo inductivo del aprendizaje con diferencia temporal con $\lambda = 0$, es la suposición de que el concepto objetivo devuelve valores similares para posiciones contiguas, lo cual tiene una interpretación intuitiva bastante directa como una noción de continuidad o «suavidad» de la función.

La expresión de actualización de los pesos de la red lineal quedaría:

$$\triangle w_{m_i} = \eta(f(X_{m+1} - f(X_m))(1 - f(X_m)^2)x_{m_i}$$

En el caso de que m sea la última posición de la partida, el valor $f(X_{m+1})$ se define como 1 o -1 si gana blanco o negro respectivamente.

4.4. Método de Entrenamiento

Otro tema importante es como proveer los ejemplos de entrenamiento: se busca una convergencia rápida a una función de evaluación correcta y suficiente variación en los ejemplos como para que la función pueda ser utilizada en situaciones generales.

Para la red neuronal que se desea aprender, se utilizan dos algoritmos: backpropagation para el ajuste de pesos y diferencia temporal como algoritmo de aprendizaje. Este último, es un algoritmo de *aprendizaje por refuerzos*. En este tipo de algoritmos hay un punto muy importante a tener en cuenta: buscar el equilibrio entre la *exploración* y la *explotación*. En el aprendizaje por refuerzos, el agente influencia la distribución de ejemplos según la secuencia de acciones elegida. En nuestro caso, nuestra red influenciará los ejemplos que recibirá según los movimientos que decida hacer. Esto incorpora la siguiente pregunta: ¿cuál es la mejor estrategia de experimentación?

En un extremo el sistema de aprendizaje puede favorecer a explorar nuevos estados y acciones, buscar nueva información (exploración).

Por otro lado se puede favorecer a utilizar las acciones que ya se han aprendido, llevan a estados favorables (explotación). Esta última opción corre el riesgo de que el agente se «encierre» en un conjunto de acciones que aprendió que eran buenas al comienzo del entrenamiento y evite que explore otras acciones que le podrían dar mejor resultado. De hecho, la convergencia de algoritmos de refuerzo como el aprendizaje, requiere que cada transición estado-acción sea explorada infinitamente a menudo [Mitchell, 1997].

La técnica usual consiste en lograr un equilibrio, pero más precisamente se busca favorecer la exploración en las etapas tempranas del entrenamiento y luego gradualmente favorecer la explotación.

En el Backgammon la exploración de variedad de estados está asegurada ya que la aleatoriedad de los dados permite implícitamente explorar muchos estados diferentes. En el caso del Go, esta aleatoriedad no existe, por lo que seguir una estrategia de selección de estados determinística evita que se escojan opciones diferentes y por lo tanto el riesgo de caer un mínimo local, es decir, quedar estancado en una solución subóptima.

Para resolver este problema es necesario incluir la aleatoriedad explícitamente en la selección de movimientos durante el entrenamiento. La idea entonces es utilizar un *muestreo de Gibbs*. Dado un movimiento m de los n posibles, con un valor de v_m, se define la probabilidad de realizar el movimiento m según el siguiente muestreo de Gibbs:

$$P(m) = \frac{e^{\frac{v_m}{T}}}{\sum_i^n e^{\frac{v_i}{T}}}$$

Siendo $0 < T \leq 1$ un parámetro conocido como *temperatura*, el cual tiende a dar mayor probabilidad a movimientos con mayor valor mientras menor sea T.

Enzenberger [Enzenberger, 1996] utiliza una técnica similar, realiza un movimiento de exploración con probabilidad 0,15, de otra forma realiza el mejor movimiento según la función de evaluación. Para elegir el movimiento de exploración utiliza un muestreo de Gibbs con $T = 0{,}35$.

En nuestro caso se decidió utilizar valores intermedios. Con probabilidad PE (Probabilidad de Exploración) $= 0{,}5$ se realizarán movimientos de exploración y para la selección se utilizará un muestreo de Gibbs con $T = 0{,}5$. De esta forma se puede agregar la aleatoriedad suficiente y controlada de modo de asegurar la exploración y al mismo tiempo tender a elegir los movimientos que más valor tienen según la función de evaluación que se está aprendiendo.

4.4.1. Oponente de Entrenamiento

Estrategias de entrenamiento tales como jugar contra sí mismo, puede ser una opción adecuada para juegos no determinísticos tales como el Backgammon y por otro lado no ser adecuados para juegos determínisticos como el Go. Esto se debe a que gracias a la aleatoriedad (como tirar dados) se puede lograr una suficiente variedad de posiciones de entrenamiento, mientras que en caso determínistico esto no se da.

Con respecto al oponente durante el entrenamiento, podemos manejaremos básicamente con dos opciones:

- Entrenar contra una copia de sí mismo (autodidacta).
- Entrenar contra otro oponente artificial.

El entrenamiento contra un oponente humano se descarta completamente por razones prácticas. En la primera opción, la implementación es simple, se tiene un programa clon del que se entrena. Tesauro en el entrenamiento de su TD-Gammon

[Tesauro, 1995] obtuvo un gran éxito con esta forma entrenamiento. Enzenberger entrenó su NeuroGo [Enzenberger, 1996] con esta misma técnica.

Con respecto a la segunda opción (utilizar otro programa como oponente), podríamos utilizar al GNU-GO [GNU-GO, 2006] como programa oponente. Otros trabajos previos y la experiencia propia indican que se tiende a obtener un buen entrenamiento con oponentes con nivel similar al jugador a entrenar. Esto se debe a que el nivel de dificultad es el adecuado como para que el aprendiz pueda tener los dos refuerzos posibles: victoria y derrota. Si el oponente es demasiado superior, el jugador sólo tendrá estímulos de derrota, por lo que puede ser muy improbable que llegue a experimentar estados favorables y no obtenga estímulos de victoria tan necesarios como los de derrota para ajustar correctamente la función. El caso simétrico ocurre cuando el oponente es demasiado débil. Si se entrenara sólo contra un oponente débil, el jugador terminaría aprendiendo una estrategia suficiente para derrotarlo pero muy pobre como para derrotar a un jugador de mayor nivel.

Por lo tanto el oponente ideal sería un jugador con un nivel similar al aprendiz, a medida que el aprendiz adquiere una estrategia más fuerte, el oponente debe también subir su nivel de juego. GNU-GO es un programa muy fuerte, que participa en competiciones internacionales, por lo que para comenzar a entrenar contra el será necesario controlar su nivel de juego. La técnica que se podría utilizar es agregar al GNU-GO un parámetro PR, que indique la probabilidad con la cual realizará un movimiento aleatorio. De esta forma, ajustando tal parámetro, podemos lograr diferentes niveles de dificultad, desde un jugador randómico ($PR=1$) hasta GNU-GO con todo su potencial ($PR=0$).

Sería deseable intentar ambas opciones, sin embargo, aquí se decidió priorizar la utilización de entrenamiento autodidacta. Las razones son las siguientes:

- El entrenamiento con una copia automáticamente proporciona un oponente del mismo nivel del jugador a entrenar, propiedad como hemos mencionado, deseable. Por otro lado, para lograr esto con otro jugador artificial, deberíamos preocuparnos por ajustar el valor de PR analíticamente, lo que implica un seguimiento un tanto engorroso de decidir cuándo y cuánto modificar tal parámetro.

- Por otro lado, el entrenar contra un oponente específico tiene un efecto no siempre deseado: el agente aprende una estrategia que le sirve para derrotar a ese oponente específico y quizás tal estrategia no sea lo suficientemente rica para vencer a otros oponentes. Este punto se puede interpretar en el campo de aprendizaje supervisado como *sobreajuste* (*overfitting*). El algoritmo aprende una representación de un concepto que describe perfectamente un conjunto de ejemplos dado, pero no representa correctamente el concepto en todo el dominio.

- Otra razón práctica viene por el hecho de que nuestro jugador a entrenar utiliza un menor tiempo a la hora de seleccionar movimiento que el oponente artificial (GNU-GO), por lo que el tiempo de entrenamiento se reduciría considerablemente.

4.4.2. Profundidad de Búsqueda

Como describimos en la arquitectura general del sistema, el método de selección de movimientos básicamente consiste en realizar una búsqueda alfa-beta en el árbol de juego hasta una profundidad máxima y utilizar una función de evaluación (la que pretendemos aprender) en los nodos hojas.

Esto introduce una interrogante a la hora del entrenamiento: ¿Qué profundidad de búsqueda deberíamos utilizar durante el entrenamiento? Una primera idea podría ser utilizar simplemente una profundidad de búsqueda 1, es decir, solamente considerar los efectos de movimientos inmediatos.

Runarsson y Lucas [Runarsson and Lucas, 2005], por otro lado, sostienen que al expandir más el árbol de búsqueda, la función de evaluación es aplicada muchas más veces, por lo que se tendría una prueba más confiable de los aciertos y debilidades de la función de evaluación.

Lo ideal sería poder experimentar con varias profundidades. Sin embargo, al utilizar mayores profundidades la selección de movimientos es más lenta lo que produce tiempos de entrenamientos muy largos. Los limites de tiempo en este proyecto sugirieron utilizar la estrategia más simple y rápida: utilizar un nivel de búsqueda 1 durante el entrenamiento.

4.4.3. Optimizaciones

Para mejorar la velocidad de entrenamiento según la técnica propuesta en las secciones anteriores se realizaran un par de mejoras:

Por un lado, implícitamente al describir la técnica de ajustes mediante descendiente por gradiente, asumimos un *aprendizaje incremental* [Nilsson, 1996]; los ajustes de los pesos se realizan enseguida que se presenta un ejemplo de entrenamiento. En nuestro caso, se realiza el ajuste de pesos luego de realizar un movimiento. Por lo general (más común en aprendizaje supervisado), los pesos se ajustan luego de analizar unos cuantos ejemplos; esto se conoce como *aprendizaje por lotes* (*batch learning*). Se decidió utilizar aprendizaje incremental porque tiende a aumentar la velocidad de aprendizaje.

Por último otra forma de agilizar el entrenamiento puede ser utilizada. A medida que se juega una partida de entrenamiento, el sistema va guardando los estados (posiciones de tablero) que se fueron dando a través de la partida. Una vez finalizada la partida se cuenta con la secuencia completa de estados. En vez de realizar el aprendizaje descrito anteriormente comenzando desde la posición inicial, se realizará en sentido inverso. Comenzar desde atrás para adelante hace que los ajustes sean mucho más eficientes que de adelante hacia atrás. Intuitivamente esto se puede interpretar como hacer que el refuerzo directo del resultado de la partida se «propague» más rápido y produzcan mejores ajustes, resultando en un entrenamiento más rápido.

4.5. Resultados Obtenidos

En esta sección se muestran los resultados de la implementación y experimentación de los modelos presentados en secciones anteriores. En ambos casos se utilizan:

- Mismo conjunto de entradas.

- Función de activación $tanh(\frac{1}{10}x)$.
- Entrenamiento autodidacta.
- Partidas de testeo contra WallyPlus.

El único conocimiento previo que poseen los sistemas consiste en:

- Un conjunto de atributos (entradas) que en principio es adecuado (esto se describió en la sección 4.2.1).
- El conocimiento mínimo necesario para jugar el Go: las reglas del juego. Esto es necesario para que cada vez que el prototipo explore los movimientos a realizar, tenga un conjunto de movimientos permitidos. Este conocimiento previo es básico y muy fácil de implementar.
- Una función que calcule el resultado de una partida. Básicamente consiste en utilizar las reglas chinas de Go para calcular el resultado (conteo de territorio).
- Se considera una partida terminada cuando ya no existen movimientos permitidos a realizar. Usualmente una partida de Go termina mucho antes, cuando ambos jugadores coinciden en que no hay territorio por disputar. En principio la primera opción es más fácil de implementar y permite que el prototipo explore más movimientos por partida.

Durante el transcurso de este proyecto se realizaron varias experiencias relacionadas al aprendizaje de redes. Los siguientes experimentos y resultados, resumen y representan todas las experiencias realizadas.

Para simplificar las descripciones, llamaremos «agente» al programa que aprende la función de evaluación, es decir al programa que luego de una partida utiliza el algoritmo de aprendizaje para ajustar los pesos. Por otro lado, en el caso del aprendizaje autodidacta, llamaremos «clon» al programa que es lógicamente igual al agente, pero que tiene sus pesos fijos en determinados valores.

El entrenamiento autodidacta consiste en utilizar al clon como oponente del agente. Cada determinado determinado período se pasan los pesos aprendidos al clon. El procedimiento exacto que se siguió fue el siguiente:

1 Se inicializan los pesos del agente y el clon en 0.

2 Se realizan 200 partidas de entrenamiento entre el agente y el clon. 100 de esas partidas, clon es blanco y agente negro, en las segundas 100 se invierten los colores.

3 Luego de terminada cada una de las 200 partidas de entrenamiento, el agente ejecuta el algoritmo de aprendizaje según lo descrito en la sección 4.3.

4 Luego de terminadas las 200 partidas de entrenamiento, los nuevos pesos aprendidos son pasados al clon.

5 Se repite el proceso anterior a partir del punto 2 hasta que se decida terminar el entrenamiento.

Con respecto al ajuste de la constante de aprendizaje η, se suele aconsejar [AI-FAQs, 2006] comenzar probando con valores grandes, luego si se observa que los pesos fluctúan mucho y la performance del sistema es inestable y sin mejoras, entonces se reinicia con valores más pequeños. Si por otro lado, la performance mejora muy lentamente, es posible intentar valores mayores. Como ejemplo se propone empezar con una constante de 0,1 para aprendizaje por lotes y 0,01 para aprendizaje incremental. Como se describió anteriormente, utilizaremos aprendizaje incremental, por lo que comenzaremos con un valor de $\eta = 0{,}01$.

El primer experimento fue realizado con el Perceptrón. Al principio, luego de realizar varios intentos de «prueba y error» con cada valor de η, se observaron inestabilidades que no conducían a nada. Recién al utilizar el valor $\eta = 0{,}005$ se observaron resultados interesantes. En la figura 4.5 se muestra la performance del sistema frente a Wallyplus.

Si observamos la gráfica de porcentaje de partidas ganadas, en el rango de 0 a 1600 partidas de entrenamiento, se observa una curva de aprendizaje bien definida, luego de llegar a un pico de performance de 80 % al llegar a 1600 partidas de entrenamiento, esta se desploma para luego estancarse en alrededor del 30 %. El comienzo fue muy bueno, siempre mejorando, pero luego la inestabilidad en la performance es considerable.

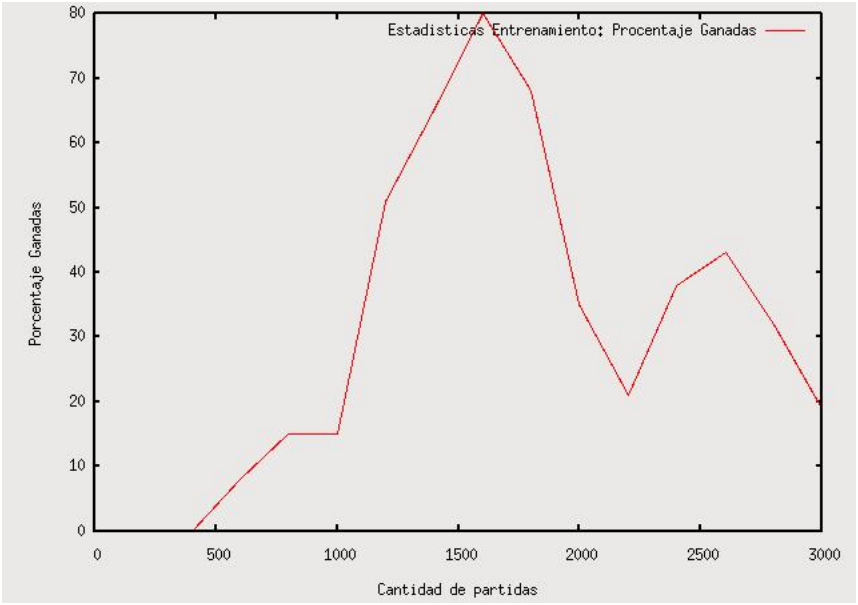
Por otro lado, si observamos la gráfica de promedio de puntaje, a grandes rasgos se observa una mejora a lo largo de todo el entrenamiento. En definitiva, aunque el porcentaje de partidas ganadas creció y luego disminuyó, el agente logró en general, mejorar su promedio de puntaje durante todo el entrenamiento.

Es posible que la causa de la inestabilidad en el porcentaje de partidas ganadas se deba a que el valor de la constante de aprendizaje η sea demasiado grande, por lo que provoque cambios bruscos en los pesos. Para remediar este problema, el segundo experimento consistió en comenzar con un valor menor de η; de esta forma se espera un aprendizaje más lento pero más seguro. En el momento de observar inestabilidades importantes, se disminuirá la constante de aprendizaje a un factor de aproximadamente 2/3, siguiendo los resultados de Runarsson y Lucas [Runarsson and Lucas, 2005], donde se propone utilizar tal factor cada cierto período.

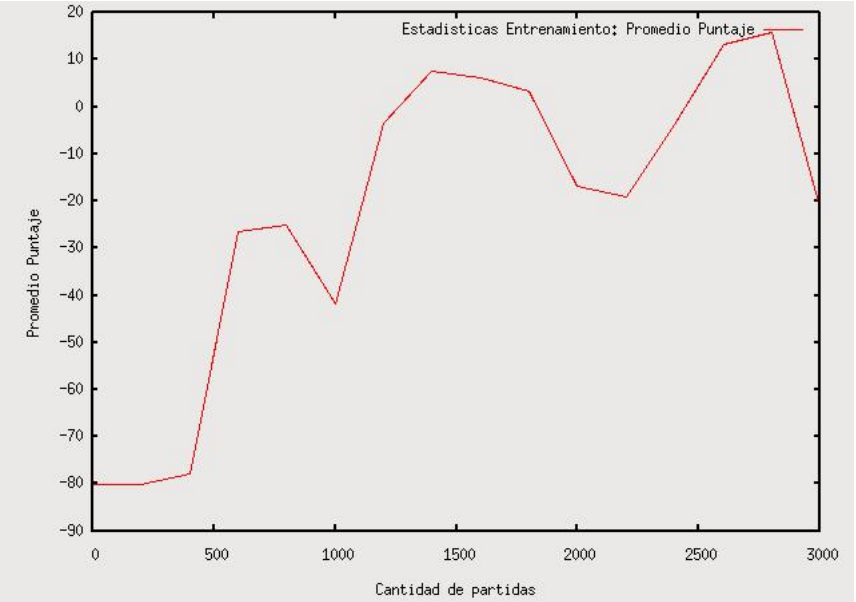
Como se puede observar en las curvas del Perceptrón de la figura 4.6, se logra una mayor estabilidad. El único punto donde se observa una inestabilidad importante es luego de los 1200 entrenamientos donde se pasa de un promedio de 5 % en 1200 a 75 % en 1400. Luego sigue un periodo de relativa estabilidad alrededor de 60 % con picos inferiores de 42 % y picos superiores de 82 %. Siendo el último valor, luego de las 8000 partidas de entrenamiento, el segundo mejor valor obtenido.

Los valores de η utilizados durante el entrenamiento del Perceptrón se muestran en la tabla 4.2.

Por último, se utilizó un MLP con una capa oculta compuesta por 4 unidades ocultas. Los resultados pueden verse en las curvas del MLP de la figura 4.6. Fue interesante ver como se obtuvo un aprendizaje importante comenzando con un valor de $\eta = 0{,}1$. Como se vio en el caso del Perceptrón, no se obtuvieron resultados positivos de aprendizaje sino hasta después de intentar con un valor mucho menor: $\eta = 0{,}005$. En principio no hay una interpretación directa, quizás se deba a que al tener que ajustarse más parámetros que en el caso lineal, sea necesario utilizar ajustes más «gruesos» al principio del entrenamiento, para así poder aumentar la



(a) Partidas Ganadas ante Wallyplus.



(b) Promedio de puntaje ante Wallyplus.

Figura 4.5: Gráficas de evolución de porcentaje de victorias y promedio de puntaje durante entrenamiento del Perceptrón con $\eta = 0{,}005$.

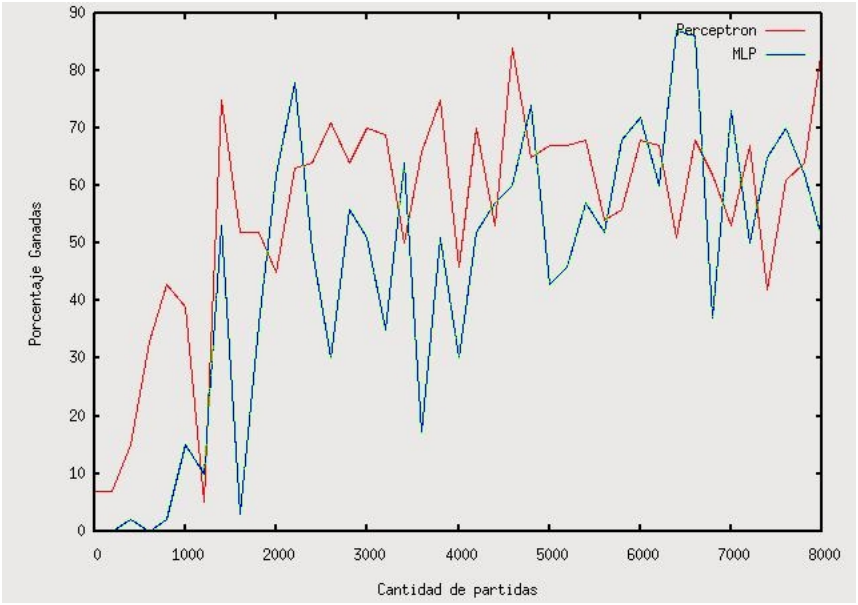
Rango	η
0-2000	0.001
2000-4000	0.0007
4000-5000	0.00047
5000-6000	0.0003
6000-8000	0.0001

Cuadro 4.2: Valores de η para el Perceptrón según rango de entrenamiento.

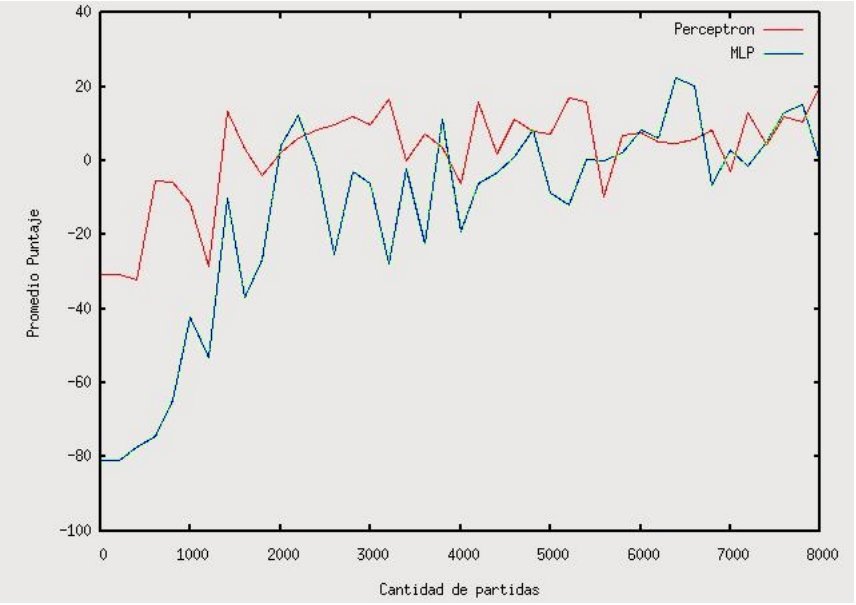
velocidad de convergencia.

En general, ambos modelos muestran una mejora de la performance del prototipo a medida que transcurre el entrenamiento. En este punto surge una pregunta razonable: ¿hasta cuándo seguir entrenando?, ¿cuándo parar? La respuesta usual a estas preguntas en aprendizaje supervisado es: terminar cuando el promedio de error en el conjunto de testeo sea menor a un valor establecido. Sin embargo, aquí no tratamos con aprendizaje supervisado, sino, que tratamos con aprendizaje por refuerzos. Nuestra medida de aprendizaje surge del testeo del agente ante un oponente con algunos conocimientos de Go (en nuestro caso Wallyplus). Por lo tanto, en principio deberíamos detener el aprendizaje cuando el agente logre niveles de performance adecuados ante el oponente.

En nuestro caso, según se observa en las gráficas con $\eta = 0{,}001$, ambos modelos se mantienen en un rango aproximadamente constante de performance, sin lograr mejoras sustanciales. Una posible hipótesis para explicar este «estancamiento» es que en ambos casos se ha llegado a la mejor aproximación al concepto objetivo que el modelo permite representar: la capacidad de expresión de los modelos planteados estarían limitadas a llegar a tales valores de performance. Por lo tanto, se decidió finalizar el entrenamiento en este punto al no esperar mejoras.



(a) Partidas Ganadas ante Wallyplus.



(b) Promedio de puntaje ante Wallyplus.

Figura 4.6: Gráficas de evolución de porcentaje de victorias y promedio de puntaje durante entrenamiento para el Perceptrón y el MLP.

Capítulo 5

Implementación del Jugador de Go

En este capítulo se describen algunos detalles respecto a la implementación del jugador de Go. En la sección 5.1 se describe la integración de los diferentes componentes. En la sección 5.2 se detalla la compilación de los patrones obtenidos en el capítulo 3. En la sección 5.3 se presenta la interfaz gráfica que permite la interacción con jugadores humanos. Por último, en la sección 5.4 se muestran los resultados de testeo del jugador ante otros jugadores artificiales.

5.1. Integración de Componentes

El objetivo es implementar un jugador de Go al que se llama *ANNILGOS* (*Artificial Neural Network and Inductive Logic Go System*) según el diseño propuesto. Este jugador utiliza los resultados obtenidos en el aprendizaje de patrones del capítulo 3 y el aprendizaje de una función de evaluación del capítulo 4.

Durante el aprendizaje de una función de evaluación para tableros de Go, se utilizó un prototipo cuya estrategia de selección de movimientos consistió en: dado un tablero, seleccionar todas las posiciones sucesoras, evaluarlas con la red neuronal y elegir el movimiento que lleva a la posición sucesora con mayor valor. Para este fin fueron necesarios implementar algunos componentes que fueron reutilizados en la implementación del jugador ANNILGOS.

Se construyó una interfaz que implementa el protocolo GTP [Farnebäck, 2002], que permite la comunicación a través de la entrada y salida estándar (protocolo basado en texto) con otros programas que implementen este mismo protocolo. Tanto en los entrenamientos como en los testeos se utilizó un script implementado en código Perl que actúa como intermediario entre los jugadores.

Por otro lado, fue necesario implementar un par de funciones. Una función que retorne todos los movimientos válidos dada una posición (encapsula las reglas de Go), y una función que devuelva el resultado de una partida de Go.

Para terminar de implementar el jugador ANNILGOS basado en el diseño propuesto en la sección 2.4, se utilizan los patrones generados en el capítulo 3. El procedimiento básicamente consiste en elegir todos los movimientos válidos según las reglas del Go y a partir de este conjunto, obtener el mejor subconjunto de movimientos según las reglas inducidas en el aprendizaje de patrones basado en ILP.

Como se evaluó en la sección 3.4, se obtuvo un promedio de 74 % de aciertos al elegir los mejores 20 movimientos.

Esto permite reducir el factor de ramificación en etapas tempranas de la partida y por lo tanto permitir realizar una búsqueda alfa-beta de profundidad 3 en un tiempo razonable. En definitiva, al momento de seleccionar un movimiento, se utiliza una búsqueda alfa-beta de profundidad variable: cada vez que se busca los movimientos posibles, se seleccionan los mejores 20 según los patrones; la profundidad se ajusta de modo de que si existen 20 movimientos sucesores, se utiliza una profundidad de 3, si existen menos movimientos sucesores (etapas finales de la partida), se incrementa la profundidad. Recordar que otra virtud de los patrones, además de devolver los mejores movimientos, es devolverlos en un orden de mayor a menor valor. Este orden hace que la búsqueda alfa-beta tienda a analizar los mejores movimientos primero, produciendo una optimización en el algoritmo.

Por último, como funciones de evaluación de posiciones se utilizan el Perceptrón y el MLP obtenidos en el capítulo 4.

5.2. Compilación de Patrones a Código C

En principio se simplificaría el trabajo si se utilizaran las reglas inducidas directamente: utilizar un sistema de Prolog o un intérprete de Prolog. De esta forma se podría utilizar alguna interfaz del sistema Prolog a C para poder utilizar los predicados dentro de un programa C.

Esta ventaja es menor a una desventaja práctica importante: eficiencia. En la práctica, utilizar directamente un interprete Prolog tiene una eficiencia muy baja con respecto a la codificación de las reglas en código C.

Para comenzar, es necesario contar con una infraestructura que permita mantener y actualizar el estado de un tablero. Esta estructura debe ser eficiente y contar con todas las primitivas necesarias: realizar movimientos, deshacer movimientos, obtener valores de intersecciones, propiedades de los bloques (libertades y cantidad de piedras), guardar la cantidad de piedras capturadas, el komi, etc. Dado que el objetivo del proyecto no pretende implementar dicha infraestructura desde cero, y existen a disposición pública muchas implementaciones eficientes, se decidió utilizar una de ellas.

GNU-GO [GNU-GO, 2006] es un programa que juega al Go, desarrollado dentro del proyecto GNU. Es un jugador no comercial, por lo que su código está disponible para uso gratuito. Este sistema posee una biblioteca en código C de rutinas muy eficientes para el manejo de tableros de Go, por lo tanto es la infraestructura eficiente que se necesita.

El siguiente paso es definir que funcionalidades son necesarias. El objetivo ahora es utilizar las reglas inducidas para que a partir de una posición dada, generar una lista de N movimientos candidatos ordenados de acuerdo a su valor. Por lo tanto se define la siguiente función:

```
mejores_movimientos(t:tablero, N:int, c:color, cant:int):conj_movs
```

Que devuelve los mejores *cant* movimientos ($cant <= N$) para el tablero actual y el color especificado. En *cant* devuelve la cantidad de movimientos devueltos. El siguiente peseudocódigo describe su funcionamiento:

```
movs:conj_movimientos;
Para cada movimiento m posible en el tablero t dado
   v = evaluar(m,c,t);
   agregar(movs,m,v);
ordenar(movs);
retornar movs;
```

Donde la función *evaluar*, retorna el valor de realizar el movimiento m, con el color c, en el tablero t, según las reglas inducidas. La función ordenar, ordena los movimientos de mayor a menor según su valor v.

Luego debemos definir una función *evaluar(m,c,t)*. Para esto se definió una función booleana

```
move(n:int,t:tablero,c:color,m:movimiento):boolean
```

Que devuelve true si la regla n aplica al realizar el movimiento m, con el color c, en el tablero t. A partir de esta función es fácil calcular el valor de un movimiento como se muestra en el siguiente pseudocódigo:

```
function evaluar(m:movimiento,c:color,t:tablero):int
   res=0;
   Para cada regla n
      si move(n,t,c,m)
         res = res + cobertura(n);
   retornar res;
```

Aquí debemos poder tener un estimativo del valor de cada regla. En este punto se decidió que una posible estimación de cuán buena es una regla podría ser la cobertura de la misma. Recordar que la cobertura de una regla inducida, se obtuvo como $P-N$, siendo P y N la cantidad de ejemplos positivos cubiertos y la cantidad de ejemplos negativos cubiertos durante el entrenamiento respectivamente.

Queda por definir como implementar la función *move*. Básicamente consiste en que según la regla n, devolver true o false según se cumplan las condiciones de tal regla. Recordemos que las reglas inducidas, son reglas lógicas Prolog, en nuestro caso tienen la forma:

```
move(n:int,t:tablero,c:color,m:movimiento) si condiciones.
```

Que indica que se trata de la regla n, y que «aconseja» realizar el movimiento m, con el color c, en el tablero t, si se cumple las condiciones. Donde tales condiciones no son otra cosa que una conjunción de predicados lógicos pertenecientes al conjunto de reglas de conocimiento previo. Es decir, el predicado *move* es el predicado objetivo que diseñamos en el capítulo 3. Un ejemplo de una tal regla en Prolog es:

```
move(20, Tablero, Color, X, Y) :-
   pos_gen(Pos),
   block_on_pos(Tablero, X, Y, Bloque, -2, 0, E),
   block_color(Tablero, Bloque, Color2),
   edge_hor(X, Y, -4, Pos),
   opposite(Color, Color2),
   stage(Tablero, start).
```

Que se interpreta como: «La regla 20 aconseja mover en la posición *X,Y* en el *Tablero* con el *Color* dado, si existe un bloque en la posición relativa -2,0 cuyo color es opuesto, la posición *X,Y* debe estar a una distancia horizontal de -4 y el *Tablero* debe ser de apertura». El predicado *pos_gen* es utilizado con el propósito de generar las 8 simetrías espaciales que posee el Go.

Por lo tanto cada predicado de conocimiento previo se debe implementar como una función booleana en código C. La función *move* revisará la regla especificada verificando si se cumple la conjunción de condiciones en alguna de las simetrías posibles.

Tal codificación presenta dificultad en el caso de los *predicados no determinísticos*, es decir, aquellos predicados que son positivos con más de una instanciación de sus variables. En nuestro caso debemos separar entre variables de entrada y salida. Otro punto importante en la implementación de reglas Prolog es implementar correctamente el *backtraking*. Por ejemplo la regla:

```
move(4,Tablero, Color, X, Y) :-
   connects_direct(Tablero, Bloque, X, Y, Color),
   stone_cnt(Tablero, Bloque, 1),
   block_color(Tablero, Bloque, Color).
```

se interpreta como: «La regla 4 aconseja mover en la posición *X,Y* en el *Tablero* dado, con el *Color* dado, si existe un *Bloque* en donde *X,Y* se conecte directamente, la cantidad de piedras del *Bloque* sea 1 y el color del *Bloque* sea el mismo del *Color* dado».

Notar que en este caso tenemos un cuantificador existencial en la regla, por lo que el predicado *connects_direct* debe devolver todos los posibles bloques que conecten directamente con *X,Y* y tengan el color dado. Para resolver esto, en el caso de los predicados no determinísticos se mantiene una estructura de datos que guarde el estado actual del predicado, esto es, las soluciones que ya fueron analizadas y cual es la siguiente solución a devolver. El predicado se modela como una máquina de estados: a medida que se llama la función correspondiente, se actualiza la estructura. Por otro lado, es necesario llamar una función de «reseteo» antes de utilizar la función por primera vez, de modo de llevar la máquina de estados a su estado inicial. Por lo tanto la implementación en código C del ejemplo exhibe el siguiente aspecto:

```
reset_connects_direct();
while(connects_direct(Tablero,&Bloque, X, Y, Color))
   if ( stone_cnt(Tablero, Bloque, 1) && block_color(Tablero, Bloque, Color))
      return 1;
```

5.3. Interfaz Gráfica

La interfaz gráfica que permite que un jugador humano interactúe en una partida contra ANNILGOS, se implementó según el esquema de la figura 5.1.

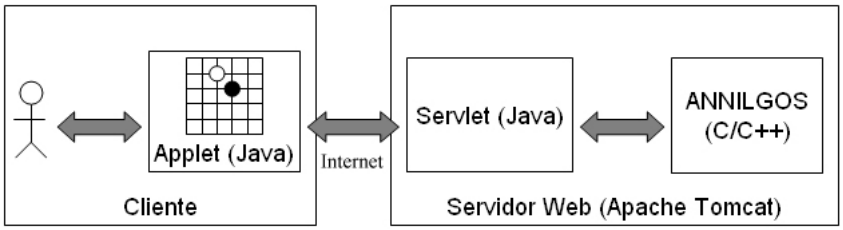


Figura 5.1: Esquema de interfaz gráfica.

El usuario interactúa con un tablero gráfico mediante un applet de Java, vía web. Este applet establece una conexión a través de internet (protocolo TCP/IP) con un servidor web mediante servlets de Java. Luego, el servlet interactúa con ANNILGOS (implementado en código C/C++), mediante el protocolo de texto GTP.

5.4. Resultados Obtenidos

En el capítulo anterior, los testeos de performance de la función de evaluación ante WallyPlus, obtuvieron una performance del 78 % de victorias, 20.0 de promedio de puntaje, utilizando un perceptrón y una performance de 80 % de victorias, 22.4 de promedio de puntaje, utilizando un MLP.

Se implementaron dos jugadores de Go que utilicen ambas funciones de evaluación (Perceptrón y MLP). En ambos casos, se utiliza como estrategia de selección de movimientos una búsqueda alfa-beta con profundidad variable. Los patrones aprendidos por ILP fueron utilizados para generar un conjunto reducido de 20 movimientos al analizar movimientos sucesores.

Los nuevos resultados de performance ante WallyPlus se muestran en la tabla 5.1. En ambos casos se realizaron 100 partidas, 50 de ellas ANNILGOS juega con color blanco y 50 con color negro. Se logró una mejora considerable al combinar ambos resultados de aprendizaje. Las mejoras son más importantes en el MLP que en el perceptrón. Esto puede deberse a que el MLP ha logrado ajustar una mejor función de evaluación para el Go que el perceptrón.

Perceptrón			**MLP**		
Color	**Victorias**	**Puntaje**	**Color**	**Victorias**	**Puntaje**
Negro	100 %	46.9	Negro	100 %	75.5
Blanco	64 %	13.2	Blanco	100 %	25.8
Total	82 %	30.0	Total	100 %	50.7

Cuadro 5.1: Estadísticas de resultados ANNILGOS (Perceptrón y MLP) vs Wallyplus.

Wallyplus es un jugador basado en heurísticas de un nivel de juego bajo. Por lo tanto, se decidió testear la performance de ANNILGOS ante un jugador de mejor nivel. GNU-GO [GNU-GO, 2006] ha sido desarrollado desde 1989 con varios autores dentro del proyecto GNU. Este jugador tiene un buen nivel de juego y su uso es público, por lo que se lo utilizó como nuevo oponente. Los resultados se muestran en la tabla 5.2. Igual que antes, en ambos casos se realizaron 100 partidas, 50 de ellas ANNILGOS juega con color blanco y 50 con color negro. Otra vez, el jugador que utiliza el MLP produce mejores resultados que un único perceptrón.

Perceptrón			**MLP**		
Color	**Victorias**	**Puntaje**	**Color**	**Victorias**	**Puntaje**
Negro	20 %	-40.0	Negro	50 %	-13.8
Blanco	0 %	-58.0	Blanco	6 %	-20.5
Total	10 %	-49.0	Total	28 %	-17.2

Cuadro 5.2: Estadísticas de resultados ANNILGOS (Perceptrón y MLP) vs GNU-GO.

El jugador logró vencer la gran mayoría de las veces y con muy buen promedio a un jugador de bajo nivel basado en heurísticas y codificación analítica de patrones de Go. Con respecto a las partidas ante GNU-GO, se observa un nivel inferior a este oponente. Sin embargo, los resultados son bastante prometedores teniendo en cuenta que este GNU-GO se encuentra entre las primeras posiciones dentro de las competencias de jugadores artificiales de Go. El aprendizaje de una función de evaluación mediante un MLP entrenado con aprendizaje por diferencia temporal, y el aprendizaje de patrones de Go mediante ILP han logrado una buena performance.

Capítulo 6

Conclusiones y Trabajo Futuro

En este proyecto se plantea como objetivo el diseño de un jugador de Go basado en técnicas de aprendizaje automático. Se logró el objetivo mediante la propuesta de una solución compuesta por las siguientes etapas:

- El diseño e implementación de un generador de patrones de juego de Go basado en aprendizaje por inducción de programas lógicos.
- El diseño e implementación de un sistema de aprendizaje de una función de evaluación de tableros de Go basado en redes neuronales y aprendizaje por refuerzos.

Los niveles de performance ante otros jugadores artificiales son buenos. Se destaca la combinación de dos paradigmas diferentes de aprendizaje automático: *aprendizaje simbólico* (inducción de programas lógicos) y *aprendizaje conexionista* (ajuste de una red neuronal mediante aprendizaje por refuerzos).

A continuación se analizan los dos componentes.

Aprendizaje de Patrones

Una medida del éxito de esta etapa es que se obtuvieron varias reglas que representan patrones de juego básicos de Go, muchos de ellos conocidos. Esto parece un logro interesante, dado que estos patrones fueron generados automáticamente a partir de ejemplos de entrenamiento de jugadores profesionales y un conjunto de reglas y conceptos muy básicos del Go.

Se ha logrado una buena performance en la predicción de movimientos de jugadores profesionales. Aunque los porcentajes de acierto son menores a los obtenidos en otros trabajos previos, aquí se propone la generación de patrones que aplican a una partida de Go en general. En los trabajos previos se realiza el aprendizaje de patrones de Go para situaciones específicas, donde los ejemplos de entrenamiento son más elaborados que los utilizados en este proyecto, por lo que los resultados son promisorios.

En el primer intento de aprendizaje de patrones se utilizó un acercamiento del tipo propuesto por Muggleton, donde se busca el aprendizaje a partir de ejemplos solamente positivos. Los patrones inducidos fueron demasiado generales, solamente conceptos muy básicos del Go fueron aprendidos.

Con respecto al conocimiento previo, como acercamiento inicial se utilizaron predicados que modelaban conceptos básicos del Go. La incorporación de nuevo conocimiento de fondo más elaborado significó una mejora sustancial en el aprendizaje. Sin embargo, tal incorporación debió de acompañarse con la adición de ejemplos negativos y restricciones de modo de contrarrestar el aumento en el espacio de búsqueda. Además, en la práctica se observó que recién luego de utilizar ejemplos negativos y restricciones, el algoritmo incorporó predicados más complejos debido a la necesidad de especificar más sus cláusulas, dando como resultado patrones más complejos.

Por lo tanto, la utilización de ejemplos negativos y restricciones pueden ser un punto vital sino necesario para este tipo de aprendizaje.

A la hora de utilizar ejemplos negativos, existió una dificultad importante: en la mayoría de los problemas no es usual poseer este tipo de ejemplos. Aquí fue necesario generar analíticamente un conjunto de ejemplos negativos en base a la violación de las reglas del Go y movimientos realizados por un jugador randómico. Esta solución tuvo el impacto suficiente como para forzar que algoritmo especifique más las reglas.

Aprendizaje de una Función de Evaluación

Como modelo de función de evaluación se utilizaron dos posibles representaciones: un perceptrón y un perceptrón multicapa (MLP). En ambos casos se utilizaron el mismo conjunto de atributos como entradas, misma función de activación y algoritmo de aprendizaje. Los resultados de testeo del prototipo muestran que se obtuvo una mejor performance utilizando el MLP.

Con respecto al entrenamiento, se decidió utilizar un entrenamiento autodidacta ya que de esta forma se simplifica el trabajo de poseer un oponente de entrenamiento de un nivel similar al agente a entrenar. Es interesante observar cómo fue posible el aprendizaje sin la necesidad de contar con un oponente con conocimientos de Go.

Se confirmó el hecho de que el ajuste adecuado de la constante de aprendizaje juega un papel muy importante y es un trabajo muy delicado, del cual depende directamente el éxito o fracaso del aprendizaje. Se siguieron las «recetas» usuales para esta tarea. Sin embargo, fue necesario realizar muchas experimentaciones de prueba y error antes de obtener buenos resultados.

6.1. Trabajo Futuro

Los distintos aspectos del trabajo admiten mejoras. A continuación se presentan algunos posibles trabajos a futuro de este proyecto.

Aprendizaje de Patrones

Intuitivamente el aumento de la cantidad de ejemplos de entrenamiento puede causar una mejora en la performance del sistema. Sin embargo, esto también produce un aumento importante en el espacio de búsqueda, y en la práctica, un aumento dramático en el tiempo de entrenamiento. Para resolver esto, sería interesante la investigación de un acercamiento híbrido entre ILP y algoritmos evolutivos [Reiser and Riddle, Reiser, 1999, Tamaddoni-Nezhad and Muggleton, 2000, Divina and

Marchiori]. Esta técnica permite combinar las ventajas de ILP con la capacidad de los algoritmos evolutivos de manejar espacios de búsqueda mayores.

Una posible dirección en la cual profundizar el aprendizaje de patrones de Go es el aprendizaje específico de patrones de apertura. El hecho de concentrar el aprendizaje en un caso particular podría aumentar la performance de predicción, pero la razón más importante es que en las etapas tempranas de la partida (apertura) es donde hay una mayor posibilidad de jugadas, por lo que es donde más interesa disminuir el factor de ramificación en el árbol de juego. A estas razones se le suma el hecho intuitivo de que la apertura es donde la mayoría de las funciones de evaluación tienen menos efectividad. Esto se debe a que la función cuenta con menos material de entrada y que se encuentra muy lejos de los valores de ajuste que usualmente se obtienen al final de la partida. Por lo tanto, concentrarse más en la apertura podría complementar mejor a una función de evaluación.

Por último, la incorporación de nuevos predicados de conocimiento previo podrían lograr mejoras en la performance y eficiencia tanto en el entrenamiento como en las reglas inducidas. Nuevamente, sería deseable encontrar nuevos ejemplos negativos y restricciones que permitan la mayor especialización de las reglas.

Aprendizaje de una Función de Evaluación

Con respecto a la las entradas de la red, podría intentarse buscar más y mejores atributos que colaboren en la obtención de una mejor función de evaluación. Para esto es necesario un análisis más profundo del dominio en particular. Por otro lado, existen algunas mejoras generales (no dependientes del dominio) que podrían intentarse mediante la *normalización*, también llamada, *estandarización* de las entradas. La idea aquí es reescalar los valores de entrada de modo que todas las entradas tengan sus valores en el mismo rango. De esta forma, todas las entradas contribuyen con el mismo orden de magnitud y se evita dar un peso implícito inicial mayor a entradas con mayor magnitud. Esto podría contribuir a aumentar la velocidad de convergencia al inicio del entrenamiento.

Con respecto a las salidas de la red, en este proyecto se propuso utilizar una sola salida que devolviera un valor positivo si la posición favorece al color blanco o negativo si favorece a negro. Sin embargo, podría utilizarse un refuerzo más fino. En nuestro caso, los valores +1 y -1 son dados independientemente de la magnitud de la victoria. Es decir, no es lo mismo que blanco gane con +86.5 (puntaje máximo con que puede ganar), que gane con +0.5 (puntaje mínimo con que puede ganar). Con esto se lograría que el agente obtenga refuerzos positivos menores cuando gana con puntajes bajos y refuerzos positivos mayores cuando gana con puntajes altos. Análogamente, el agente obtendrá refuerzos negativos en proporción a el puntaje de perdida.

Con respecto a la arquitectura de la red, se podrían buscar mejoras aumentando la cantidad de unidades ocultas o intentando con otro diseño de red. En este trabajo se optó por una arquitectura estándar: una red estática completamente conectada. Por otro lado, trabajos como NeuroGo [Enzenberger, 1996], proponen una arquitectura dinámica muy interesante de seguir explorando.

Durante este proyecto, el método de entrenamiento para las redes neuronales consistió en un entrenamiento autodidacta. Aunque se consideró el entrenamiento autodidacta como una mejor elección, sería interesante poder obtener resultados

utilizando otro jugador artificial como oponente durante el entrenamiento. Otra posible idea es intentar una estrategia mixta donde se combinen ambos métodos de entrenamiento.

Por otro lado, se decidió utilizar una profundidad de nivel 1 durante el entrenamiento. Una posible mejora se podría lograr utilizando profundidades mayores mediante algoritmos como *TD-directed*, *TD-leaf* y *TD(μ)* [Ekker et˜al., 2004]. La idea básica detrás de de los dos primeros algoritmos es permitir una mayor profundidad en la selección de movimientos durante el entrenamiento, y en el último, se busca lograr un mejor aprendizaje a partir de oponentes que cometen errores.

Apéndice A

Declaraciones de ILP

A.1. Conocimiento Previo

```
% El bloque Block queda en atari si Color juega en X,Y
atari(Board,Block,X,Y,Color):-
                        board(Board,Lib,X,Y),
                        block(Board,Block),
                        opposite(Color,Color2),
                        block_col(Board,Block,Color2),
                        liberty_cnt(Board,Block,2),
                        liberty(Board,Block,Lib).

% Se realiza una conexión directa con el bloque Block1 si se juega en X,Y
connects_direct(Board,Block1,X,Y,Color):-
                              board(Board,Lib,X,Y),
                              block(Board,Block1),
                              block_col(Board,Block1,Color),
                              liberty(Board,Block1,Lib).

% Se realiza una conexión indirecta con el bloque Block1 si se juega en X,Y
connects_indirect(Board,Block1,X,Y,Color):-
                                            board(Board,Pos,X,Y),
                                            block(Board,Block1),
                                            block_col(Board,Block1,Color),
                                            liberty(Board,Block1,Lib1),
                                            liberty(Board,Block1,Lib2),
                                            Lib1\=Lib2,
                                            link(Board,Pos,Lib1),
                                            link(Board,Pos,Lib2).

% Se captura el bloque Block si se juega en X,Y
captures(Board,Block,X,Y,Color):-
                            block(Board,Block),
                            block_col(Board,Block,Color2),
                            opposite(Color,Color2),
                            liberty_cnt(Board,Block,1),
```

```
                          liberty(Board,Block,Lib),
                          board(Board,Lib,X,Y).

% Devuelve todos los bloques del tablero
block(Board,Block):-
        setof(Bl,(between(1,9,X),between(1,9,Y),board(Board,Bl,X,Y),
        block_color(Board,Bl,_)),Ls),
        member(Block,Ls).

pos_gen(p1).
pos_gen(p2).
pos_gen(p3).
pos_gen(p4).
pos_gen(p5).
pos_gen(p6).
pos_gen(p7).
pos_gen(p8).

%------board
board([_,Is],G,X,Y):-
                      board2(Is,X,Y,G).

board2([_|Is],X,Y,G):-
                        Y>1,
                        Z is Y-1,
                        board2(Is,X,Z,G).

board2([I|_],X,1,G):-
                         board_x(I,X,G).

board_x([_|Bs],X,G):-
                         X>1,
                         Z is X-1,
                         board_x(Bs,Z,G).

board_x([G|_],1,G).

block_color(_,[_,b],b).
block_color(_,[_,w],w).

block_col(_,[_,C],C).

%----- link

link(B,G1,G2):-
               setof([X,Y],board_gen(B,G1,X,Y),L),
               block_border(L,L,[],L2),
               block_list(B,L2,[],Gs),
```

```
                member(G2,Gs).

%----block list  (de una lista de inters devuelve una lista de grupos, sin repetir)

block_list(B,[[X,Y]|Ls],Ac,Res):-
                                board(B,G,X,Y),
                                not(member(G,Ac)),!,
                                block_list(B,Ls,[G|Ac],Res).

block_list(B,[_|Ls],Ac,Res):-
                            block_list(B,Ls,Ac,Res).

block_list(_,[],Ac,Ac).

%-----block border
block_border([[X,Y]|Ls],L,Ac,Res):-
                            X2 is X+1,
                            Y2 is Y+1,
                            X3 is X-1,
                            Y3 is Y-1,
                            add_border(L,X2,Y,Ac,Ac2),!,
                            add_border(L,X,Y3,Ac2,Ac3),!,
                            add_border(L,X3,Y,Ac3,Ac4),!,
                            add_border(L,X,Y2,Ac4,Ac5),!,
                            block_border(Ls,L,Ac5,Res).

block_border([],_,Ac,Ac).

add_border(L,X,Y,Ac,[[X,Y]|Ac]):-
                        inside(X,Y),
                        not(member([X,Y],Ac)),
                        not(member([X,Y],L)).

add_border(_,_,_,Ac,Ac).


opposite(b,w).
opposite(w,b).

%----------liberty   liberty cnt   stone cnt ------------
liberty(B,G,Lib):-
                  block_col(B,G,C),
                  C\=e,
                  link(B,G,Lib),
                  block_col(B,Lib,e).

liberty_cnt(B,G,N):-
                    block_col(B,G,C),
```

```
                    C\=e,
                    setof(Lib,liberty(B,G,Lib),L),
                    length(L,N).

liberty_cnt(B,G,0):-
                    block_col(B,G,C),
                    C\=e.

stone_cnt(B,G,N):-
                  block_col(B,G,C),
                  C\=e,
                  setof([X,Y],board_gen(B,G,X,Y),L),
                  length(L,N).

%generador edge_ver
edge_ver(X,_,D,p1):-
                         var(D),
                         size(S),
                         S2 is (S+1)/2,
                         X>=S2,
                         D is S+1-X.

edge_ver(X,_,D,p1):-
                         var(D),
                         size(S),
                         S2 is (S+1)/2,
                         X<S2,
                         D is -X.

%chequeador edge_ver
edge_ver(X,Y,D,Pos):-
                         ground(D),
                         rotate(X,Y,D,0,Pos,X2,Y2,A2,B2),
                         U is X2+A2,
                         V is Y2+B2,
                         is_edge(U,V).


%generador edge_hor
edge_hor(_,Y,D,p1):-
                         var(D),
                         size(S),
                         S2 is (S+1)/2,
                         Y>=S2,
                         D is S+1-Y.

edge_hor(_,Y,D,p1):-
                         var(D),
```

```
                        size(S),
                        S2 is (S+1)/2,
                        Y<S2,
                        D is -Y.

%chequeador edge_hor
edge_hor(X,Y,D,Pos):-
                        ground(D),
                        rotate(X,Y,0,D,Pos,X2,Y2,A2,B2),
                        U is X2+A2,
                        V is Y2+B2,
                        is_edge(U,V).

is_edge(X,_):-
        size(S),
        edge(S,X).

is_edge(_,Y):-
        size(S),
        edge(S,Y).

board_gen([_,Is],G,X,Y):-
                        board3(1,1,Is,X,Y,G).


board3(U,V,[Row|_],X,Y,G):-
                             size(S),
                             S2 is S+1,
                             V<S2,
                             board3_row(U,V,Row,X,Y,G).

board3(U,V,[_|Rows],X,Y,G):-
                             size(S),
                             S2 is S+1,
                             V<S2,
                             V2 is V+1,
                             board3(U,V2,Rows,X,Y,G).

board3_row(U,V,[G|_],U,V,G).

board3_row(U,V,[_|Gs],X,Y,G):-
                                U2 is U+1,
                                board3_row(U2,V,Gs,X,Y,G).

%generador de block_on_pos
block_on_pos(Board,X,Y,G,A,B,p1):-
                               var(A),
                               var(B),
```

```
                                disp(A),
                                disp(B),
                                U is X+A,
                                V is Y+B,
                                inside(U,V),
                                board(Board,G,U,V),
                                block_col(Board,G,C),
                                C\=e.

%chequeador de block_on_pos
block_on_pos(Board,X,Y,G,A,B,Pos):-
                                ground(A),
                                ground(B),
                                rotate(X,Y,A,B,Pos,X2,Y2,A2,B2),
                                U is X2+A2,
                                V is Y2+B2,
                                inside(U,V),
                                board(Board,G,U,V),
                                block_col(Board,G,C),
                                C\=e.

%realiza  las rotaciones/simetrizaciones
rotate(X,Y,A,B,p1,X,Y,A,B).

rotate(X,Y,A,B,p2,X2,Y2,A2,B2):-
                                size(S),
                                X2 is S-Y+1,
                                Y2 is X,
                                A2 is -B,
                                B2 is A.


rotate(X,Y,A,B,p3,X2,Y2,A2,B2):-
                                size(S),
                                X2 is S-X+1,
                                Y2 is S-Y+1,
                                A2 is -A,
                                B2 is -B.

rotate(X,Y,A,B,p4,X2,Y2,A2,B2):-
                                size(S),
                                X2 is Y,
                                Y2 is S-X+1,
                                A2 is B,
                                B2 is -A.

rotate(X,Y,A,B,p5,X2,Y2,A2,B2):-
                                size(S),
```

```
                                X2 is S-X+1,
                                Y2 is Y,
                                A2 is -A,
                                B2 is B.

rotate(X,Y,A,B,p6,X2,Y2,A2,B2):-

                                X2 is Y,
                                Y2 is X,
                                A2 is B,
                                B2 is A.

rotate(X,Y,A,B,p7,X2,Y2,A2,B2):-
                                size(S),
                                X2 is X,
                                Y2 is S-Y+1,
                                A2 is A,
                                B2 is -B.

rotate(X,Y,A,B,p8,X2,Y2,A2,B2):-
                                size(S),
                                X2 is S-Y+1,
                                Y2 is S-X+1,
                                A2 is -B,
                                B2 is -A.

edge(_,0).
edge(S,X):-
          E is S+1,
          X==E.

inside(X,Y):-
           size(S),
           X>0,Y>0,X=<S,Y=<S.

stage([82,_],open).


stage([N,_],start):-
                     M is N-81,
                     between(2,20,M).

stage([N,_],mid):-
                 M is N-81,
                 between(21,40,M).

stage([N,_],end):-
                 M is N-81,
```

```
                M>40.

range(X,A,B):-
              between(1,20,A),
              between(A,20,B),
              between(A,B,X).

size(9).
```

A.2. Declaraciones de Modos

En Progol hay dos tipos de declaraciones de modos: aquellas que son utilizados para formar la cabeza de la cláusula, llamadas *modeh*, y las utilizadas para especificar los literales del cuerpo de la cláusula, llamadas *modeb*. Las declaraciones de modos tienen los siguientes formatos:

$$modeh(n, predicado\_objetivo(v_1 tipo_1, \ldots, v_n tipo_n)$$

$$modeb(n, predicado\_background(v_1 tipo_1, \ldots, v_n tipo_n)$$

Donde v_i indica el modo del argumento, los valores posibles son +, - o , # que indican que el argumento es un argumento de entrada, salida o constante respectivamente. $Tipo_i$ indica el tipo del argumento, los tipos son definidos como conocimiento de fondo en forma de cláusulas definidas. El parámetro n (*recall*) es un entero mayor que 0 o *, establece el limite superior de la cantidad de soluciones alternativas que pueden instanciar un literal. El valor * indica todas las soluciones.

Las declaraciones de modos utilizadas fueron:

```
:-modeh(1,move(+board_type,+color,+coord_x,+coord_y)).

:-modeb(1,pos_gen(-pos)).
:-modeb(*,block_on_pos(+board_type,+coord_x,+coord_y,-block_id,#disp,#disp,+pos)).
:-modeb(1,block_color(+board_type,+block_id,-color)).
:-modeb(1,opposite(+color,-color)).
:-modeb(1,edge_ver(+coord_x,+coord_y,#disp,+pos)).
:-modeb(1,edge_hor(+coord_x,+coord_y,#disp,+pos)).
:-modeb(*,liberty(+board_type,+block_id,-block_id)).
:-modeb(1,board(+board_type,+block_id,+coord_x,+coord_y)).
:-modeb(1,stage(+board_type,#stage_type)).
:-modeb(1,range(+nat,#nat,#nat)).
:-modeb(1,liberty_cnt(+board_type,+block_id,-nat)).
:-modeb(1,stone_cnt(+board_type,+block_id,-nat)).
:-modeb(1,stage(+board_type,#stage_type)).
:-modeb(1,range(+nat,#nat,#nat)).
```

A.3. Restricciones

A.3.1. Restricciones de Integridad

Las restricciones en notación Progol tienen la forma:

```
false:-
        Body.
```

Donde *Body* es un conjunto de literales que especifican las restricciones que no deberían ser violadas por una cláusula. Usualmente se utiliza el predicado *hypothesis/3*, el cual permite acceder al cabezal y cuerpo de la cláusula siendo analizada. Las restricciones de integridad fueron:

Evitar cláusulas no generativas:

```
false:-
       hypothesis(Head,Body,_),
       nongenerative(Head,Body).

nongenerative(Head,Body):-
                  expr_term(Body,L1),
                  expr_term((Head,Body),L2),
                  member(A1,L1),arg(_,A1,V1),var(V1),
                  not((member(A2,L2),A1\==A2,arg(_,A2,V2),V1==V2)).
```

Evitar devolver hechos:

```
false:-
       hypothesis(_,true,_).
```

Evitar cláusulas con menos de 3 literales en el cuerpo:

```
false:-
       hypothesis(_,Body,_),
       expr_term(Body,L1),
       length(L1,N),
       N<3.
```

Si se encuentra en una etapa posterior a la apertura de la partida (*stage(Board,open)*), entonces no considera cláusulas que no utilicen el predicado *block_on_pos*. Esto se utilizó para evitar la generación de cláusulas muy generales que utilizan solo los predicados de bordes:

```
false:-
               hypothesis(_,Body,_),
               expr_term(Body,Ls),
               member(A1,Ls),
               functor(A1,stage,2),
               arg(2,A1,V1),
               V1\==open,
               not((member(A2,Ls),functor(A2,block_on_pos,7))).
```

Si hay un átomo de *block_on_pos*, deberá haber un predicado *block_color* que indique el color del bloque considerado por *block_on_pos*:

```
false:-
                hypothesis(_,Body,_),
                expr_term(Body,Ls),
                member(A1,Ls),
                functor(A1,block_on_pos,7),
                arg(4,A1,V1),
                not((member(A2,Ls),functor(A2,block_color,3),
                arg(2,A2,V2),V1==V2)).
```

A.3.2. Restricciones de Poda

Progol define restricciones de poda de la forma:

```
prune((ClauseHead:-ClauseBody)):-
        Body.
```

Donde *ClauseHead* y *ClauseBody* son la cabeza y el cuerpo de la cláusula que se está analizando respectivamente. A continuación se introducen las restricciones de poda junto con una breve descripción.

Predicados auxiliares:

```
expr_term(+Body,-List)
```

Dado *Body* un conjunto de átomos separados por comas, devuelve en *List* la lista de los átomos.

```
member(?Elem,?List)
```

Elem es un elemento de la lista *List*.

```
functor(?Atom,?Functor,?Arity)
```

Atom es un átomo con functor *Functor* y aridad *Arity*.

```
arg(?N,?Atom,?Value)
```

El N-esimo argumento del átomo *Atom* tiene valor *Value*.

```
Term1==Term2
```

Term1 es equivalente a *Term2*.

Restricciones de poda:

No pueden haber dos átomos diferentes con mismo símbolo de predicado *opposite* en el cuerpo de la cláusula:

```
prune((_:-Body)):-
```

```
                expr_term(Body,Ls),
                member(A1,Ls),
                functor(A1,opposite,2),
                member(A2,Ls),
                A1\==A2,
                functor(A2,opposite,2).
```

No pueden haber dos átomos diferentes con símbolo de predicado *liberty_cnt* aplicados al mismo bloque:

```
prune((_:-Body)):-
                expr_term(Body,Ls),
                member(A1,Ls),
                functor(A1,liberty_cnt,3),
                member(A2,Ls),
                A1\==A2,
                functor(A2,liberty_cnt,3),
                arg(2,A1,V1),
                arg(2,A2,V2),
                V1==V2.
```

No pueden haber dos átomos diferentes con símbolo de predicado *stone_cnt* aplicados al mismo bloque:

```
prune((_:-Body)):-
                expr_term(Body,Ls),
                member(A1,Ls),
                functor(A1,stone_cnt,3),
                member(A2,Ls),
                A1\==A2,
                functor(A2,stone_cnt,3),
                arg(2,A1,V1),
                arg(2,A2,V2),
                V1==V2.
```

Apéndice B

Listado de Patrones Inducidos

Para cada regla se indica entre paréntesis rectos la cantidad de ejemplos positivos y negativos cubiertos.

```
[Rule 1] [Pos cover = 25 Neg cover = 0]
move(1,_A, _B, C, D) :-
    pos_gen(E), edge_ver(C, D, 4, E), edge_hor(C, D, 4, E).

[Rule 2] [Pos cover = 183 Neg cover = 17]
move(2,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 2, 2, E),
    liberty_cnt(A, F, 4), block_color(A, F, G),
    opposite(G, B).

[Rule 3] [Pos cover = 599 Neg cover = 17]
move(3,A, B, C, D) :-
    opposite(B, E), connects_direct(A, F, C, D, E),
    stone_cnt(A, F, 1).

[Rule 4] [Pos cover = 391 Neg cover = 20]
move(4,A, B, C, D) :-
    connects_direct(A, E, C, D, B), stone_cnt(A, E, 1),
    block_color(A, E, B).

[Rule 5] [Pos cover = 383 Neg cover = 20]
move(5,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, -4, 0, E),
    block_color(A, F, B).

[Rule 6] [Pos cover = 433 Neg cover = 20]
move(6,A, _B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 1, 2, E),
    block_color(A, F, G), block_on_pos(A, C, D, H, 2, 1, E),
    block_color(A, H, G).

[Rule 7] [Pos cover = 235 Neg cover = 15]
```

```
move(7,A, _B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, -1, -2, E),
    block_color(A, F, G), block_on_pos(A, C, D, H, 2, -2,E),
    block_color(A, H, G).

[Rule 8] [Pos cover = 11 Neg cover = 0]
move(8,A, _B, C, D) :-
    pos_gen(E), edge_hor(C, D, 5, E), stage(A, open).

[Rule 9] [Pos cover = 204 Neg cover = 2]
move(9,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 2, 0, E),
    block_color(A, F, G), opposite(G, B),
    stage(A, start).

[Rule 10] [Pos cover = 409 Neg cover = 13]
move(10,A, B, C, D) :-
    opposite(B, E), connects_direct(A, F, C, D, E),
    block_color(A, F, E), stage(A, mid).

[Rule 11] [Pos cover = 5 Neg cover = 0]
move(11,A, _B, C, D) :-
    pos_gen(E), edge_hor(C, D, 3, E), stage(A, open).

[Rule 12] [Pos cover = 89 Neg cover = 0]
move(12,A, _B, C, D) :-
    pos_gen(E), edge_ver(C, D, 4, E), stage(A, start).

[Rule 13] [Pos cover = 99 Neg cover = 0]
move(13,A, _B, C, D) :-
    pos_gen(E), edge_hor(C, D, 5, E), stage(A, start).

[Rule 14] [Pos cover = 103 Neg cover = 4]
move(14,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 3, -4, E),
    block_on_pos(A, C, D, F, 4, -4, E), block_color(A, F, B).

[Rule 15] [Pos cover = 79 Neg cover = 3]
move(15,A, _B, C, D) :-
    pos_gen(E), edge_hor(C, D, 3, E), stage(A, start).

[Rule 16] [Pos cover = 87 Neg cover = 0]
move(16,A, _B, C, D) :-
    pos_gen(E), edge_ver(C, D, 3, E), stage(A, start).

[Rule 17] [Pos cover = 70 Neg cover = 0]
move(17,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 2, -1, E),
```

```
    liberty_cnt(A, F, 6), block_color(A, F, B).

[Rule 18] [Pos cover = 8 Neg cover = 0]
move(18,A, _B, C, D) :-
    pos_gen(E), edge_ver(C, D, -4, E), stage(A, open).

[Rule 19] [Pos cover = 7 Neg cover = 0]
move(19,A, _B, C, D) :-
    pos_gen(E), edge_hor(C, D, -4, E), stage(A, open).

[Rule 20] [Pos cover = 42 Neg cover = 0]
move(20,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, -2, 0, E),
    block_color(A, F, G), edge_hor(C, D, -4, E),
    opposite(B, G), stage(A, start).

[Rule 21] [Pos cover = 109 Neg cover = 2]
move(21,A, _B, C, D) :-
    pos_gen(E), edge_ver(C, D, -4, E), stage(A, start).

[Rule 22] [Pos cover = 93 Neg cover = 1]
move(22,A, _B, C, D) :-
    pos_gen(E), edge_ver(C, D, 5, E), stage(A, start).

[Rule 23] [Pos cover = 83 Neg cover = 0]
move(23,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 2, -3, E),
    liberty_cnt(A, F, 5), block_color(A, F, B).

[Rule 24] [Pos cover = 115 Neg cover = 0]
move(24,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 3, 3, E),
    liberty_cnt(A, F, 4), block_color(A, F, B).

[Rule 25] [Pos cover = 61 Neg cover = 0]
move(25,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, -4, -3, E),
    liberty_cnt(A, F, 5), block_color(A, F, B).

[Rule 26] [Pos cover = 11 Neg cover = 0]
move(26,_A, _B, C, D) :-
    pos_gen(E), edge_ver(C, D, 2, E), edge_hor(C, D, 1, E).

[Rule 27] [Pos cover = 66 Neg cover = 0]
move(27,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 3, -4, E),
    liberty_cnt(A, F, 5), block_color(A, F, G),
    opposite(G, B).
```

```
[Rule 28] [Pos cover = 121 Neg cover = 0]
move(28,A, B, C, D) :-
    captures(A, E, C, D, B), block_color(A, E, F), opposite(F, B).

[Rule 30] [Pos cover = 70 Neg cover = 0]
move(30,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 1, 4, E),
    block_color(A, F, B), edge_hor(C, D, -1, E).

[Rule 31] [Pos cover = 53 Neg cover = 0]
move(31,A, B, C, D) :-
    pos_gen(E), block_on_pos(A, C, D, F, 3, -1, E),
    liberty_cnt(A, F, 4), block_color(A, F, B),
    edge_ver(C, D, -3, E).

[Training set performance]
            Actual
          +            -
      + 1469         65         1534
 Pred
      -   0           12          12

        1469         77         1546

 Accuracy = 0.957956
 [Training set summary] [[1469, 65, 0, 12]]
 [time taken] [2428.89]
 [total clauses constructed] [40782]
```

Glosario

Alfa-beta Mejora del algoritmo minmax que se basa en la poda del árbol de búsqueda.

Algoritmo de aprendizaje Algoritmo a través de cual, un programa puede mejorar alguna medida de performance a través de la experiencia o análisis de ejemplos.

Aprendizaje Automático (AA) Subespecialidad de la inteligencia artificial que estudia algoritmos de aprendizaje y generación automática de conocimiento.

Aprendizaje por Refuerzos Un tipo de técnica en aprendizaje automático que consiste en un agente que explora un ambiente en donde puede percibir su estado actual y toma acciones. El ambiente provee refuerzos según las acciones tomadas. El objetivo es obtener una política de acciones que busquen maximizar los refuerzos acumulados.

Aprendizaje Supervisado Es una técnica de aprendizaje automático usualmente utilizada para el ajuste de una función a partir de datos de entrenamiento. Los datos consisten en pares de entrada y su salida deseada. La salida de la función puede predecir valores continuos (regresión), o puede predecir una etiqueta de clase de un objeto de entrada (clasificación).

Arbol de juego Grafo dirigido, donde cada nodo representa un estado del juego (posición) y cada arista representa una acción (movimiento).

Atari Un bloque se encuentra en atari si tiene una única libertad.

Atributo Característica de un objeto. Por ejemplo, en una posición de Go, posibles atributos podrían ser: cantidad de piedras de cada color en el tablero, cantidad de piedras de cada color capturadas, si hay bloques en atari, etc.

Backpropagation Algoritmo de ajuste de pesos de de redes neuronales mediante la reducción de alguna medida de error. Se basa en el descendiente por gradiente y la regla de la cadena.

Bloque Conjunto de piedras del mismo color conectadas.

Cabezal de cláusula definida Postcondición de una cláusula definida.

Captura Un bloque es capturado cuando queda sin libertades.

Cláusula definida Regla lógica del estilo: «*si* condiciones *entonces* resultado», donde las condiciones son una conjunción de predicados lógicos y el resultado es un único predicado lógico.

Conexión Una piedra esta conectada a otra piedra si son adyacentes, es decir, se encuentran en intersecciones contiguas (no se consideran piedras contiguas en diagonal).

Conocimiento previo Conjunto de predicados lógicos que son utilizados como material (predicados a agregar en el cuerpo de la cláusula a aprender) para la inducción de reglas lógicas en un algoritmo de ILP.

Constante de aprendizaje Constante numérica que regula la magnitud de los cambios realizados en los parámetros de una función durante el aprendizaje.

Cuerpo de cláusula definida Conjunción de precondiciones en una cláusula definida.

Declaración de modos Conjunto de declaraciones en un algoritmo de ILP del tipo Progol en donde se definen los tipos de las variables, su forma de instanciación (entrada, salida o constante), y la cantidad de instanciaciones de los predicados de conocimiento previo.

Descendiente por gradiente Técnica utilizada para el ajuste de pesos en una función real. Se busca minimizar el error cometido por la función. Básicamente consiste en obtener las derivadas parciales de la función con respecto a cada peso y efectuar los cambios en los pesos según la magnitud y opuesto del signo de la derivada parcial.

Determinístico Situación donde no interviene la aleatoriedad.

Ejemplo de entrenamiento Usualmente consiste en un par $< entrada, salida >$ donde se indica la salida deseada para una entrada dada, para un concepto a aprender.

Ejemplo negativo Ejemplo donde la aplicación de un concepto es negativa.

Ejemplo positivo Ejemplo donde la aplicación de un concepto es positiva.

Entrenamiento autodidacta Forma de entrenamiento donde por ejemplo, un programa que desea aprender a jugar al Go, juega contra una copia de si mismo.

Factor de ramificación Cantidad de hijos de un nodo en un árbol de juego.

Función de activación Función que se aplica a la combinación lineal de las entradas de una neurona con sus pesos. Usualmente se utilizan la función sigmoidal, tangente hiperbólica o la función signo.

Función de evaluación En el ámbito de resolución de juegos, es una función que permite evaluar que tan favorable es una posición a partir de un conjunto de atributos de la misma. En el ámbito de ILP, es una función que permite elegir que cláusula es más favorable para agregar a una teoría.

Generador de patrones Sistema que utiliza algún algoritmo de aprendizaje para la generación de patrones.

GNU-GO Jugador artificial de Go de nivel alto, desarrollado por el proyecto GNU.

Go Juego de origen oriental donde dos jugadores alternan poner sus piezas en las intersecciones de una cuadrícula para gobernar mayor territorio que su oponente.

GTP Go Text Protocol, protocolo basado en texto para la intercomunicación entre jugadores artificiales de Go, interfaces gráficas de Go y servidores de Go.

ILP Inductive Logic Programming, inducción de programas lógicos, técnica de aprendizaje automático donde se inducen reglas lógicas a partir de un conjunto de ejemplos positivos y negativos, y un conjunto de predicados lógicos de conocimiento previo.

Komi Valor que se le suma al puntaje final del jugador blanco debido a la desventaja de ser el jugador negro el primero en comenzar una partida de Go

Libertad Intersección vacía adyacente a un bloque.

Literal Un predicado o su negado.

Minmax Estrategia de resolución de juegos basada en árboles de juego. Se basa en el concepto de que un jugador pretende maximizar el resultado de la partida y el otro jugador pretende minimizarlo.

MLP Multi Layer Perceptron, perceptrón multi capa, es una red neuronal compuesta por perceptrones. Se los suele organizar en capas de unidades.

Ojo Intersección vacía, rodeada por piedras de un mismo bloque.

Operador de refinamiento Operador que se le aplica a una cláusula lógica para obtener refinamientos de la misma. Por lo general consiste en la especialización o generalización de una cláusula lógica.

Patrón Un conjunto de condiciones que si se cumplen en un caso particular permiten predecir algún resultado.

Perceptrón Unidad elemental o neurona. Posee un conjunto de entradas numéricas que combina linealmente con un conjunto de pesos y luego compone con una función de activación para devolver otro valor numérico.

Piedra Fichas que se utilizan en el Go.

Posonly Función de evaluación utilizada en un sistema de ILP que busca el aprendizaje a partir de ejemplos solamente positivos.

Predicado objetivo Predicado que se pretende aprender en un algoritmo de ILP.

Problema de vida o muerte Simplificación del Go donde donde solo importa la captura y defensa de piedras.

Progol Implementación de un sistema de ILP.

Prolog Lenguaje de programación lógica.

Red Neuronal Artificial Conjunto de neuronas artificiales (unidades elementales) que conectadas entre si producen un comportamiento complejo, de acuerdo a los pesos de cada conexión.

Restricción Declaración que permite restringir el espacio de hipótesis en el aprendizaje basado en ILP.

SGF Simple Go Format, formato para registro de partidas de Go, donde es posible guardar la secuencia de movimientos realizada por cada jugador, el resultado, komi, y otros datos importantes relacionados a una partida de Go.

Suicidio Movimiento por lo general no válido, donde un jugador hace que uno de sus bloques quede sin libertades y por lo tanto sea capturado.

Territorio Cantidad de intersecciones bajo el control de un jugador.

Técnica de búsqueda Técnica tradicionalmente utilizada en la resolución de juegos. Básicamente consiste en desarrollar el árbol de juego a partir de una posición dada y elegir la rama (movimiento) que lleva a la mejor situación final.

WallyPlus Jugador artificial de Go de nivel bajo.

Referencias

AI-FAQs. comp.ai.neural-nets newsgroup faqs, 2006. URL http://www.faqs.org/faqs/by-newsgroup/comp/comp.ai.neural-nets.html. Último acceso: marzo de 2006.

B. Bouzy and T. Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

J. Burmeister and J. Wiles. The challenge of go as a domain for AI research: A comparison between go and chess. In *In Proceedings of the 3rd Australian and New Zealand Conference on Intelligent Information Systems*, 1994.

F. Divina and E. Marchiori. Learning in first order logic using greedy evolutionary algorithms.

R. Ekker, E. van der Werf, and L. Schomaker. Dedicated TD-Learning for stronger gameplay: applications to Go. In *Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning*, 2004.

M. Enzenberger. The integration of a priori knowledge into a Go playing neural network, September 1996.

G. Farnebäck. Specification of the go text protocol, version 2,draft 2, October 2002. URL http://www.lysator.liu.se/~gunnar/gtp. Último acceso: setiembre de 2005.

J. Fürnkranz. Machine learning in games: A survey. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games*, chapter 2, pages 11–59. Nova Science Publishers, Huntington, NY, 2001.

GNU-GO. GNU Project - Free Software Foundation, 2006. URL http://www.gnu.org/software/gnugo/gnugo.html. Último acceso: mayo de 2006.

S. Gray. Local properties of binary images in two dimensions. May 1971. IEEE Transactions on Computers, 20:551-561.

T. Kojima and A. Yoshikawa. Knowledge acquisition from game records. 1999.

N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Routledge, New York, NY, 10001, 1993. ISBN 0134578708.

J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

S. Muggleton. Learning from positive data. In *Inductive Logic Programming Workshop*, pages 358–376, 1996.

S. Muggleton and J. Firth. CProgol4.4: a tutorial introduction, 1999.

S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

M. Müller. Computer Go. *Artif. Intell.*, 134(1-2):145–179, 2002. ISSN 0004-3702.

N. J. Nilsson. Introduction to machine learning. An Early Draft of Proposed Textbook, 1996.

M. L. Peña. *Search Improvements in Multirelational Learning*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2004.

J. Ramon and H. Blockeel. A survey of the application of machine learning to the game of Go. 2001.

J. Ramon, T. Francis, and H. Blockeel. Learning a Go heuristic with TILDE. In A. van den Bosch and H. Weigand, editors, *Proceedings of the 12th Belgian-Dutch Artificial Intelligence Conference*, pages 149–156, 2000. URL http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=32247.

P. G. K. Reiser. *Evolutionary Algorithms for Learning Formulae in first-order Logic*. PhD thesis, Computer Science, Aberystwyth, University of Wales, 1999.

P. G. K. Reiser and P. J. Riddle. Scaling up inductive logic programming: An evolutionary wrapper approach.

T. P. Runarsson and S. Lucas. Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. 2005. IEEE Transactions on Evolutionary Computation – Special Issue on Evolutionary Computation and Games.

B. Selman, R. A. Brooks, T. Dean, E. Horvitz, T. M. Mitchell, and N. J. Nilsson. Challenge problems for artificial intelligence. pages 1340–1345, 1996.

SGF. User guide, 2006. URL http://www.red-bean.com/sgf/. Último acceso: setiembre de 2005.

A. Srinivasan. The aleph manual, June 2004. URL http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/. Último acceso: octubre de 2005.

L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1999.

SWI. SWI Prolog, 2006. URL http://www.swi-prolog.org/. Último acceso: noviembre de 2005.

A. Tamaddoni-Nezhad and S. Muggleton. Searching the subsumption lattice by a genetic algorithm. In *ILP '00: Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 243–252, London, UK, 2000. Springer-Verlag. ISBN 3-540-67795-X.

G. Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38 (3):58–68, 1995. ISSN 0001-0782.

E. van der Werf. *AI techniques for the game of Go*. PhD thesis, Universitaire Pers Maastricht, 2004.

Wikipedia. The free encyclopedia, 2006. URL http://en.wikipedia.org/. Último acceso: mayo de 2006.

YAP. The YAP Prolog System, 2005. URL http://www.ncc.up.pt/~vsc/Yap/. Último acceso: setiembre de 2005.

R. Zaman and D. Wunsch. TD methods applied to mixture of experts for learning 9x9 Go evaluation function. In *Neural Networks, 1999. IJCNN '99. International Joint Conference on , Volume: 5 , Page(s): 3734 -3739 vol.6*, 1999.