Proyecto de Grado
Un jugador de Go basado en técnicas de IA
Informe de investigación del estado de arte

Raúl Garreta Tompson

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

10 de septiembre de 2005

Índice general

1. Introducción **5**
1.1. ¿Por qué un jugador artificial de Go? 5
1.2. ¿Puede una maquina pensar? 6

2. Introducción al Go **7**
2.1. Objetivo del juego 7
2.2. Jugadas prohibidas 8
2.2.1. Suicidio 8
2.2.2. Ko 8
2.3. Conceptos Importantes 8
2.4. Fin de partida 11
2.4.1. Conteo de Puntos 11

3. Complejidad del Go **13**
3.1. Categoría de juego 13
3.2. Espacio de Búsqueda 13
3.3. Función de Evaluación 16

4. Resultados Obtenidos **19**
4.1. Breve Historia 19
4.1.1. Décadas 60-70 19
4.1.2. Décadas 80-90 20
4.2. Competiciones entre programas 21
4.3. Técnicas aplicadas 22
4.3.1. Generación de movimientos 22
4.3.2. Metas y submetas 22
4.3.3. Funciones de evaluación 23
4.3.4. Búsqueda táctica 24
4.3.5. Funciones de influencia 25
4.4. Utilidades y herramientas 25
4.4.1. Servidores de Internet 25
4.4.2. Protocolo GTP 26
4.4.3. Formato SGF 26

5. Técnicas de búsqueda 27
5.1. Búsqueda por fuerza bruta en la resolución de juegos 27
5.2. MinMax 27
5.3. Poda Alfa-Beta 29
5.4. Otras técnicas de búsqueda 31
5.5. Aplicaciones en el Go 31

6. Técnicas de aprendizaje automático 33
6.1. Mejora de la poda alfa-beta 33
6.2. Ajuste de la función de evaluación 34
6.2.1. Aprendizaje Supervisado (Supervised learning) 35
6.2.2. Aprendizaje por Refuerzos (Reinforcement learning) . . . 36
6.2.3. Aprendizaje por Diferencias de Tiempos (Temporal Difference (TD) learning) 36
6.3. Aprendizaje de aperturas de juego 38
6.4. Aprendizaje de patrones 38
6.4.1. Explanation Based Generalization (EBG) 40
6.4.2. Metaprogramación y EBG 43
6.4.3. Inductive Logic Programming (ILP) 44
6.4.4. Algoritmos ecológicos 47

Capítulo 1

Introducción

1.1. ¿Por qué un jugador artificial de Go?

Desde los comienzos de la investigación en inteligencia artificial (IA), los juegos (y en especial los juegos de tablero) 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 los juegos de tablero modelan una versión simplificada de la realidad, esto es, se brindan reglas claras de juego, pero manteniendo una complejidad lo suficiente como para hacer que el problema no sea trivialmente resuelto. Tal es el ejemplo del ajedrez en donde se han desarrollado algoritmos de búsqueda tales como alfa-beta, minimal window search, hash table, etc. En definitiva, el campo de jugadores artificiales es uno de los más amplios dentro de la IA y de donde se han obtenido grandes resultados aplicables a la resolución de otros problemas. Otra característica que hace a los juegos un medio muy interesante para probar algoritmos es el hecho de que existe un método simple y que da una forma de medida muy clara de la habilidad del prototipo para resolver un problema en particular: simplemente se lo hace jugar contra un oponente humano y se observan los resultados. Por lo tanto el modelado de la realidad mediante juegos, permite concentrarse en resolver problemas fundamentales, cumplir con un objetivo claro y con reglas claras, y evitar la confusión o "ruido"de los detalles de la realidad. La siguiente cita ilustra dicha afirmación [8]:

"Newton no hubiera podido descubrir las leyes del movimiento si se hubiera concentrado en intentar comprender las leyes que gobiernan las cataratas y los huracanes. En vez de eso, simplificó el problema de las leyes del movimiento en un problema mas claro y original: planetas moviéndose a través del vacío."

Douglas Hofstadter.

La siguiente pregunta es quizás un resumen de la dificultad que una maquina debe afrontar en la resolución de ciertos problemas:

1.2. ¿Puede una maquina pensar?

Esta pregunta reúne discusiones en el campo de la ciencia, filosofía y teología. Pero quizás una definición de maquina pensante aplicable en el campo de la ingeniería podría ser: "Una maquina pensante es aquella que resuelve problemas en los cuales solo un ente inteligente a través de su razonamiento y conocimiento podría resolver". Varias respuestas famosas a dicha pregunta fueron formuladas, entre ellas, Alan Turing propuso su famoso "test de Turing"[17] basado en un acercamiento conductivista como forma de comprobar si una maquina podría pensar. Por otro lado Claude Shannon propuso como test la habilidad de una maquina en jugar ajedrez como un medio por el cual decidir si una maquina es inteligente o no.

El ajedrez ha sido el juego en el cual se han dedicado la mayor cantidad de esfuerzos en su resolución, es decir, en crear una maquina que derrote al mejor jugador de ajedrez humano del mundo. Este objetivo fue resuelto en el año 1997 cuando el programa Deep Blue derrotó al campeón mundial de ajedrez Gary Kasparov. Hoy en día este hecho parece ser una situación normal para la mayoría, pero debería destacarse como un hecho muy importante en la historia. ¿Acaso no es de destacar que una maquina haya derrotado al campeón mundial de ajedrez, siendo esta una disciplina que se la asocia a el uso intensivo de inteligencia para dominarla?. Si seguimos la definición anterior, Deep Blue seria una maquina pensante. Deep Blue podría ser considerado inteligente pero en una realidad muy reducida, un subconjunto muy reducido de la realidad como la conocemos, simplemente es inteligente a la hora de jugar ajedrez y nada más. Inclusive esta ultima afirmación se puede poner en duda si uno investiga mas profundamente la forma en que fue diseñado, el cual básicamente consiste en una búsqueda por fuerza bruta.

Luego del ajedrez, el Go ha sido el juego que mas se ha llevado los esfuerzos para su resolución. La diferencia es que aun no se ha encontrado una solución aceptable, es decir, al día no existe ninguna maquina que pueda ganarle a un jugador humano profesional de Go. Por este motivo, la resolución del Go ha tomado el lugar del ajedrez y es considerado por muchos como uno de los retos en el campo de la Inteligencia Artificial. Como símbolo del ámbito competitivo, se encuentra el caso de organizaciones como la Ing, 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.

La creación de un jugador de Go es un problema interesante por muchos aspectos: Desde el punto de vista científico es un problema que presenta una complejidad importante a la hora de plantear modelos teóricos que resuelvan el problema de una manera eficaz. Desde el punto de vista ingenieril presenta varios retos el hecho de llevar dichos modelos a una instancia practica que no solo resuelva el problema sino que se logre sujeto a las restricciones de eficiencia inherentes a un sistema físico, esto es, optimizar el funcionamiento del programa para cumplir con restricciones en el tiempo de respuesta, restricciones de recursos computacionales ya sea en capacidad de almacenamiento en memoria como capacidad de computo.

Capítulo 2

Introducción al Go

El Go 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 sigo 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.

2.1. 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 enemigo.

Al inicio de la partida el tablero está completamente vacío, excepto en partidas con ventajas (*handicap*) concedidas por un jugador de categoría superior. 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.

Las piedras del mismo color ubicadas en intersecciones adyacentes (*conectadas*) forman bloques. Los bloques permanecen en el mismo lugar a menos, como

se describió anteriormente, sea capturado por el oponente. La adyacencia es solo vertical u horizontal, es decir, dos piedras en diagonal no son adyacentes. Las intersecciones directamente adyacentes a un bloque son llamadas *libertades*. Un bloque es capturado y quitado del tablero cuando el oponente pone una piedra en la última libertad del bloque.

2.2. Jugadas prohibidas

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

2.2.1. 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.

figura

2.2.2. Ko

Dado que al capturar piedras las mismas son quitadas del tablero, es posible repetir posiciones previas del tablero. Dado que un juego infinito no es práctico, la repetición de posiciones debe ser evitada. El caso más común de repetición de una posición es el Ko como se muestra en la figura 2.1. 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 blanco juega en *a*, la recaptura solamente la podría hacer luego de jugar un movimiento en otra posición del tablero (dejar pasar un turno).

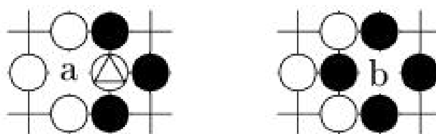


Figura 2.1: Ko básico.

2.3. Conceptos Importantes

Bloque de piedras Un bloque de piedras es un conjunto de piedras del mismo color conectadas entre si. Es un concepto fundamental, las piedras contenidas

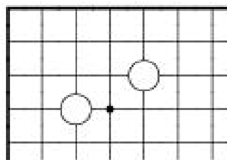


Figura 2.2: Ejemplo de grupo.

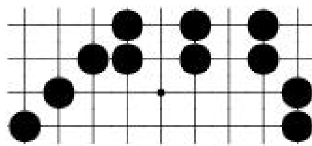


Figura 2.3: Ejemplo de una cadena de 6 bloques.

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.

Libertades Es un concepto fundamental en el Go, una libertad de una piedra es una intersección vacía adyacente a la piedra. El número de libertades de un bloque se obtiene como la suma de las libertades de cada piedra contenida en el bloque. 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.

Adyacencia Dos grupos de diferente color son adyacentes si comparten libertades. Dos intersecciones son adyacentes si son contiguas. No se consideran piedras adyacentes a dos piedras en diagonal.

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.

Grupo Es un conjunto de bloques del mismo color "débilmenteçonectados, que usualmente controlan un área del tablero.

Cadena Una cadena es un conjunto de bloques que pueden ser conectados. Dos bloques pueden ser conectados si comparten más de una libertad.

Conexión Es otro concepto fundamental, si dos bloques son conectados pasan a formar un solo bloque, por lo que comparten las libertades y tienen mayor

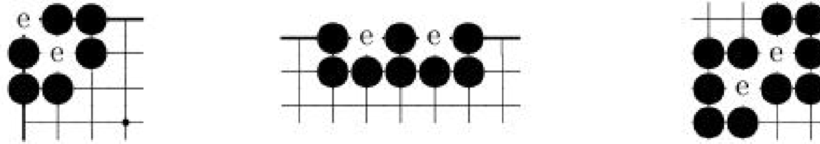


Figura 2.4: Bloques vivos, los ojos están marcados con una *e*.

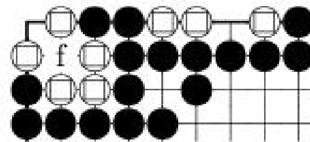


Figura 2.5: Los grupos marcados están muertos.

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.

Bloque vivo Un bloque esta 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 o están en Seki.

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.

Ojo Una o mas 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 2.4 son ojos.

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 numero de piedras de handicap necesarias para que ambos jugadores tengan las mismas chances de ganar.

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.

Suicidio Un movimiento que no captura ningún bloque del oponente y deja a su propio bloque sin libertades. Este movimiento es ilegal en la mayoría de los sistemas de reglas.

Territorio Las intersecciones rodeadas y controladas por un jugador.

Seki Dos o más bloques vivos que comparten una o más libertades y que no tienen dos ojos.

Ko Una situación de capturas repetitivas prohibida.

Dame Intersecciones vacías que no son controladas por ninguno de los dos jugadores. Usualmente son llenadas por los jugadores al final de la partida.

Prisioneros Piedras que son capturadas o muertas.

2.4. 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.

2.4.1. Conteo de Puntos

Luego de que la partida ha terminado es necesario contar los puntos para saber quien es el vencedor o si hay un empate. 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 numero de intersecciones rodeadas (territorio) mas el numero de piedras del oponente capturadas (prisioneros). En el conteo basado en área, utilizado por las reglas Chinas, se cuenta el numero 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.

Capítulo 3

Complejidad del Go

3.1. 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.
- Deterministico, 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.

Aunque entra en la misma categoría que el ajedrez, el Go es demasiado complejo como para ser resuelto exitosamente con las técnicas tradicionales de resolución de juegos, esto es: *minmax*, *poda alfa-beta*, etc. Para entender dicha complejidad, veremos en que características radica.

3.2. Espacio de Búsqueda

Dadas las propiedades de la categoría antes mencionada, un juego con dichas características puede ser representado mediante un grafo dirigido llamado *árbol de juego*. 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 dicho 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 maquina), este tiene su raíz en la posición que se esta 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 3.1 muestra un ejemplo (reducido) de un árbol de búsqueda para el juego *ta-te-ti*.

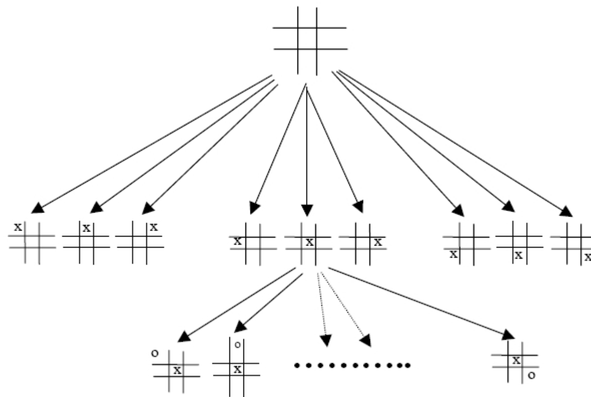


Figura 3.1: Ejemplo de un árbol de búsqueda para el ta-te-ti.

La complejidad del espacio de búsqueda, se podría traducir entonces como el tamaño del árbol de juego. Las características mas importantes que hacen al tamaño del árbol de juego son la *profundidad del árbol* y su *factor promedio de ramificación*[1].

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 3.1 se muestran los valores correspondientes al Ajedrez y al Go.

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

Cuadro 3.1: Promedios de profundidad y factor de ramificación.

Comparando las características anteriores, como primera conclusión es claro el 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 presentada en [1]. L. V. 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. Allis presenta una tabla con

[1]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.

aproximaciones para diferentes juegos de la misma categoría que el Go junto con los resultados de partidas entre humanos y computadoras. En la tabla 3.2 $>$, $>=$ y $\ll$ representan “es más fuerte”, “más fuerte o igualz “claramente más débilrespectivamente. 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 3.2: Complejidades [?]

En una primera mirada de la tabla 3.2 se podría inducir una correlación entre la complejidad del árbol y los resultados obtenidos de las computadoras contra los humanos.

Veamos ahora la tabla 3.3, es la misma que la anterior, a diferencia de que se han agregado los valores para un par de juegos adicionales (Go 9x9 y Go-moku 15x15).

Juego	$log_{10}(E)$	$log_{10}(A)$	Resultados Comp.-Humano
Damas	17	32	Chinook>H
Othello	30	58	Logistello>H
Go 9x9	40	85	Mejor programa $\ll$ H
Ajedrez	50	123	Deep Blue>=H
Go-Moku 15x15	100	80	El juego está resuelto
Go	160	400	Handtalk $\ll$ H

Cuadro 3.3: Complejidades [?]

Al agregar los otros resultados, en la Tabla 3, 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. En [1] es importante la conclusión de 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 esta dada por el tamaño del espacio de búsqueda, sino que también esta dada por la incapacidad de poder obtener una *función de evaluación* eficiente para el Go. Es decir, la complejidad del Go nos solo viene dada por su gran espacio de búsqueda, sino también por otras dificultades inherentes al Go. En la siguiente sección daremos una breve descripción de una función de evaluación y la aplicación en la resolución de juegos de tablero.

Figura 3.2: Ejemplo de posiciones controladas explícitamente e implícitamente

3.3. Función de Evaluación

La idea en la utilización de una *función de evaluación* es tener un método estático que permita evaluar el valor de una posición basado en características de la posición actual sin la necesidad de realizar una búsqueda más profunda, de encontrarse una función de evaluación correcta y rápida, 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.

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 numero 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 en [1]. 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 3.3 a la izquierda 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 de la derecha. 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 obvia para cualquier jugador promedio. Por lo tanto en la figura de la derecha las intersecciones son controladas implícitamente. Desafortunadamente la función de evaluación concreta 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 esta 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 dichas 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 el trade-off 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 computables. Algunos de esos conceptos pueden ser: grupos, cadenas, territorio, interacción, influencia, dentro, fuera, vivo, muerto, etc. Hasta el momento no existe una función de evaluación de consenso general, mientras mas exacto es el resultado de una función de evaluación, mas conocimiento se requiere, resultando en un mayor costo de computación y por lo tanto una función de evaluación lenta. En este punto es importante tener en cuenta que para un partido 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.

Capítulo 4

Resultados Obtenidos

En términos de esfuerzos en investigación y programación, el Go ocupa una segunda posición luego del ajedrez. Sin embargo, en nivel de juego, los programas de Go están muy detrás del nivel de juego de sus contrapartidas en otros juegos. Aunque los programas de Go han avanzado considerablemente en los últimos 15 años, aun pueden ser derrotados fácilmente por jugadores humanos de un nivel moderado. La fundación Ing hasta el año 2000 había ofrecido un premio de aproximadamente 1 millón de dólares a quien diseñe un programa de nivel profesional, el premio ni siquiera fue retado por ningún prototipo. Sin embargo, este ha servido el propósito de dar un gran impulso a programadores e investigadores en el área. Muchas competencias de menor escala son llevadas a cabo en Asia, Europa, Norte América y a través de Internet. Luego del éxito de Deep Blue en ajedrez ante el campeón mundial Kasparov, el Go ha tomado su lugar y es considerado la frontera final. Por otro lado en ámbitos académicos muchos científicos han dedicado tiempo considerable en la investigación de la aplicación de técnicas a la resolución de problemas y subproblemas en la creación de un jugador artificial de Go. A partir de dichos estudios se han publicado una considerable cantidad de artículos y se han realizado tesis a nivel master y doctorado. La creación de un programa que juegue al Go es un formidable reto conceptual y técnico. La mayoría de los programas competitivos han requerido entre 5 y 10 años/persona de esfuerzo y contienen entre 50 y 100 módulos que trabajan en los diferentes aspectos del juego. El nivel total del programa se limita al nivel del componente más débil, muchos programas juegan movimientos de nivel maestro, pero su nivel global en el juego es mucho menor. La figura 4.1 muestra la posición actual de los programas de Go en la escala de nivel humano.

4.1. Breve Historia

4.1.1. Décadas 60-70

Los inicios de la programación de jugadores de Go se remontan a la década de los sesenta. Aparentemente el primer programa de Go fue escrito por D.

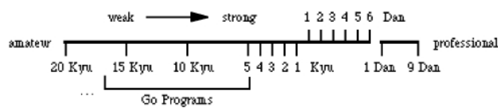


Figura 4.1: Escala de nivel de juego humano-computadora [13].

Lefkovitz. El primer artículo científico fue publicado en 1963 por Remus. El primer programa de Go en derrotar a un jugador humano (un principiante), fue creado por Zorbist. Los primeros programas se basaban exclusivamente en la computación de una función que aproximaba la influencia de las piedras. Esto es, las piedras radian influencia en las intersecciones cercanas (piedras de color opuesto radian valores opuestos) y la radiación decrece con la distancia. Desde los primeros estudios en el campo, también se ha trabajado en la resolución de sub problemas del Go, es decir, tableros mas pequeños o problemas localizados.

4.1.2. Décadas 80-90

La creación de programas de Go tuvo su mayor impulso en la década de los 80 con la aparición de los computadores personales de bajo costo, y el apoyo de sponsors en las competiciones internacionales. También se crearon las primeras publicaciones dedicadas a la computación para el Go, y también la liberación de las primeras versiones de programas comerciales. En la decada del 90 las competiciones entre programas tuvieron un gran impulso, regularmente asistieron hasta 40 participantes de todas las nacionalidades. En las primeras competiciones, los programas taiwaneses como Dragon fueron exitosos. Desde 1989-91 el programa Goliath de M. Boon dominó todas las competiciones, este acercamiento representó un gran avance gracias a la utilización intensiva de patrones para el reconocimiento de situaciones típicas para luego sugerir movimientos apropiados. Algunos de los programas más destacados que compiten actualmente en competencias internacionales y de los cuales se tiene mayor información son:

- Goliath, por M. Boon
- Go Intellect, por K. Chen
- Handtalk y Goemate, por C. Zhixing
- Many Faces of Go, por D. Fotland
- Go4++, por M. Reiss
- Explorer, por M. Müller

4.2. Competiciones entre programas

La competencia entre programas de go más antigua fue la copa Ing. Fue organizada desde el año 1987 hasta el año 2000. El ganador de la copa jugaba contra jugadores humanos para testear su capacidad. La fundación Ing ofreció durante esos años un premio aproximado de 1 millón de dólares al primer programa de Go que derrotara a un jugador humano profesional, el premio nunca fue entregado y ni siquiera tuvo un retador.

Los ganadores de las competiciones de la copa Ing fueron:

1987	Friday
1988	Codan
1989	Goliath
1990	Goliath
1991	Goliath
1992	Go Intellect
1993	Handtalk
1994	Go Intellect
1995	Handtalk
1996	Handtalk
1997	Handtalk
1998	Many Faces of Go
1999	Go4++
2000	Wulu

Cuadro 4.1: Ganadores de la copa Ing

Otras competiciones fueron desarrolladas, entre las más importantes, la copa FOST que tiene lugar todos los años en Tokio. También se llevan adelante la *"Mind Sport Olympiad"*, los campeonatos Europeos y Norte Americanos. Por otro lado existen competiciones a través de Internet que son permanentemente organizadas en el Computer Go Ladder. En esta competencia cada participante provee piedras de handicap (según las reglas) a un contrincante en un nivel inmediatamente inferior. Cada vez que el autor de un programa siente que su programa ha mejorado, puede realizar un reto, ya sea al programa de nivel inferior (incrementando las piedras de handicap) o al programa de nivel superior (decrementando las piedras de handicap). Cuando un programa entra a la competencia comienza por retar al programa en el nivel más inferior y a medida que va ganando a los programas que tiene en el nivel superior, va subiendo de nivel. Las partidas son jugadas usualmente en servidores de Internet específicamente para el Go.

4.3. Técnicas aplicadas

A continuación se presentan las técnicas aplicadas por los programas más competitivos, las secciones se estructuran según los sub problemas más comunes. La información sobre las técnicas en cada programa fue tomada de [2].

4.3.1. Generación de movimientos

Tradicionalmente las técnicas de búsqueda han sido utilizadas exitosamente en la resolución de juegos. Como se ha visto en secciones anteriores, una búsqueda global completa no es posible en el Go, y por otro lado la utilización de una función de evaluación se ve limitada por la complejidad de la misma. Sin embargo, el objetivo en la resolución de un juego no es buscar en los árboles de juego ni evaluar funciones, estas actividades son solo medios, el objetivo es generar movimientos y seleccionar el más adecuado. Típicamente los movimientos candidatos son generados a partir de reglas heurísticas basadas en patrones, a cada posible movimiento se le asigna un valor de prioridad o urgencia. Por último, el problema se reduce a elegir el movimiento con mayor prioridad. Los primeros programas de Go se asemejaban a sistemas expertos sin la utilización de búsquedas en árboles ni funciones de evaluación. Actualmente la generación de movimientos sigue siendo muy importante. El número de movimientos candidatos que son evaluados varia según el diseño del programa. En [2], como ejemplo se mencionan, Go Intellect que evalúa hasta 12 movimientos, Many Faces of Go hasta 10, y Go4++ un mínimo de 50. El número de reglas generadoras de movimientos contenidas en un programa varía, cerca de 100 para Explorer, 200 para Many Faces of Go. Go Intellect contiene cerca de 20 generadores de movimiento que se basan en bibliotecas de patrones, búsquedas orientadas por metas y reglas heurísticas. De todas formas en muchos casos es necesario asistir la generación de movimientos con funciones de evaluación y búsquedas. Las funciones de evaluación son necesarias a la hora de saber parar el juego correctamente cuando ya no hay más puntos por disputar. Por otro lado, técnicas de búsqueda en el árbol de juego se utilizan para chequear el estado táctico de ciertas posiciones locales o para elegir el mejor movimiento global entre los movimientos locales candidatos.

4.3.2. Metas y submetas

Cuando un programador intenta una utilización extensiva de conocimiento y se basa en la generación de movimientos, naturalmente entra en el dominio de la generación de metas. En vez de generar movimientos, el programa primero genera las metas que usualmente son útiles en la victoria de un juego. Una vez que la meta es seleccionada, se busca un movimiento específico que cumpla con esa meta. La ventaja de este método es que simplifica la complejidad del problema, ya que reduce el objetivo de buscar un movimiento a cumplir con una meta concreta. Por otro lado, presenta algunas desventajas, puede sufrirse de una pérdida de globalidad y por otro lado se presenta el problema de elegir

cual meta es más importante. En la mayoría de los casos, un gran territorio es más deseable que uno pequeño, por otro lado se busca mantener una gran influencia, lo que implica concentración de piedras. Esto genera un trade-off entre territorio e influencia, particularmente en el inicio y mitad del juego. Otras metas decisivas que deberían ser tenidas en cuenta son: atacar/defender grupos, expandir/reducir territorio, mientras que es posible catalogar como submetas a: crear/destruir ojos, conectar/desconectar grupos. Aparentemente no existe un consenso en el campo acerca de la relativa prioridad de las diferentes metas y submetas. Fotland propone una jerarquía de metas que podrían ser utilizadas para construir un programa de Go. Como ejemplos en [2], Handtalk pone énfasis en los combates tácticos. Many Faces of Go también lo hace y también se concentra en la conectividad, ojos y fuerza de los grupos. Go4++ se concentra casi exclusivamente en la conectividad, casi todo eventualmente se deriva (directa o indirectamente) de un mapa de probabilidad de conectividad. Por lo tanto una manera de generar un movimiento global podría ser generar todas las metas, obtener los mejores movimientos para cada meta, elegir dentro de cada meta el movimiento de mayor valor y por ultimo elegir el movimiento que sirve para la meta con mayor prioridad. Es deseable combinar alguna técnica de búsqueda para chequear que el movimiento elegido en cada caso (dentro de cada meta y global) cumple con la meta correspondiente.

4.3.3. Funciones de evaluación

Como se ha visto anteriormente una función de evaluación tiene una gran importancia, permite asociar un valor a una posición, puede ser usada tanto para seleccionar mejores movimientos como para decidir si el juego ha terminado. Encontrar una función de evaluación eficiente es uno de los retos mas importantes y difíciles, por lo general dada su complejidad, el proceso de evaluación es muy lento, sumado a los limites de tiempo, significa la posibilidad de realizar pocas evaluaciones de posiciones. Como ejemplo en [2] se mencionan, Many Faces of Go evalúa menos de 10000 posiciones completas en un juego entero a menos de 10 evaluaciones por segundo, comparado con entre 10000 y 100000 evaluaciones por segundo que los programas de ajedrez pueden realizar. En Go4++ se evalúan 50 movimientos candidatos, el proceso toma 6 pasos.

1. Se genera un mapa de probabilidad de conexión, esto es, para cada piedra blanca y negra en el tablero, se calcula la probabilidad de poder conectarla con otra piedra (real o hipotética), este paso involucra una búsqueda táctica.

2. A partir del mapa anterior, los grupos son determinados.

3. Se determinan los ojos utilizando patrones, el mapa y los grupos.

4. Se determina la seguridad de los grupos en base a la cantidad de ojos que tiene.

5. La seguridad de cada piedra es radiada en proporción a la probabilidad de conexión y es sumada a las piedras que la conectan.

6. El territorio de cada jugador es estimado en base a los valores de las radiaciones. La diferencia entre los valores de territorio es retornada como la evaluación.

En Many Faces of Go, la evaluación también consiste en un proceso de varios pasos que se basan fuertemente en la búsqueda táctica. Todas las cadenas con menos de 4 libertades y algunas con 4 son examinadas mediante una búsqueda táctica para así determinar cuales de ellas están muertas. Esta búsqueda táctica también se utiliza para determinar posibles conexiones y ojos. De este paso resulta la identificación de grupos formados por cadenas. La fuerza de cada grupo es determinada basándose en conectividad, ojos, etc. Esta información se utiliza para determinar la cantidad de control de cada color (entre -50 y 50). El territorio es determinado basándose en la suma de los valores de cada intersección, y esto da la evaluación final.

En Go Intellect la evaluación se utiliza durante una búsqueda global. Si el valor de uno de los movimientos candidatos es significativamente mayor que los valores de los otros movimientos candidatos, entonces se utiliza como el siguiente movimiento. Sin embargo, si varios de los movimientos candidatos tienen un valor aproximadamente equivalente, se utiliza una búsqueda global junto con una función de evaluación para determinar cual de los movimientos elegir. Para la evaluación, la seguridad de los grupos es utilizada como base para la asignación de valor a cada intersección del tablero (entre -64 y +64), lo cual indican el grado de control de cada color. El valor de cada intersección es sumado y se retorna el valor final. La búsqueda global examina no más de 6 a 7 movimientos y la búsqueda no es más profunda que 6 niveles.

4.3.4. Búsqueda táctica

La búsqueda táctica es una búsqueda local, selectiva orientada a una meta específica. Es utilizada para una variedad de propósitos, determinar si una cadena esta viva o muerta (Go4++, Many Faces of Go, Explorer, Go Intellect), determinar si una conexión es segura o se puede cortar (Go4++, Many Faces of Go), determinar si pueden formarse ojos (Many Faces of Go), generar movimientos candidatos (Go Intellect), determinar vida o muerte de grupos (Explorer). Al igual que una búsqueda global, una búsqueda táctica también requiere generación de movimientos y una función de evaluación. Debido a las restricciones de tiempo, la búsqueda táctica esta limitada por el número de nodos, el factor de ramificación, y la profundidad. Hay que tener en cuenta que aunque una cadena puede estar viva a nivel táctico puede estar muerta a nivel estratégico. En Many Faces of Go, un módulo táctico examina cada cadena con 4 o menos libertades mediante una búsqueda táctica. Cada cadena es examinada dos veces, una vez con el blanco moviendo primero y la otra con el negro moviendo primero. Se determina si una cadena es capturada (esto es, no puede vivir aunque se

mueva primero), amenazada (vive si se mueve primero y muere si el oponente mueve primero), o estable (vive cualquiera que juegue primero). Este módulo tiene dos generadores de movimiento, uno para generar movimientos de ataque y otro para generar movimientos de defensa. Los movimientos candidatos son ordenados según varias características, luego una búsqueda alfa-beta basada en DFS es utilizada, la eficiencia de dicha búsqueda depende de la calidad del ordenamiento anterior. La generación de movimientos y su ordenamiento ocupan la mayoría del tiempo en la búsqueda táctica. La búsqueda táctica esta limitada por la cantidad de nodos que pueden ser examinados. El número de nodos que se permiten visitar en una búsqueda son calculados en función del valor de cada movimiento asignado en el primer nivel por el generador de movimientos. Por lo tanto diferentes ramas tienen diferentes profundidades, dependiendo del valor precalculado de cada una.

4.3.5. Funciones de influencia

La influencia es un concepto que se asocia como un indicador del control potencial de territorio vacío que pueden tener un grupo de piedras. Asegurando una distribución de piedras evitando la sobre-concentración en el inicio de la partida, el jugador maximiza la chance de controlar mayor territorio al final del juego. Como se mencionó al inicio el concepto de influencia fue modelado computacionalmente por funciones de influencia (Zorbist, 1969; Ryder, 1971; Chen, 1989). Las piedras radian influencia a las intersecciones cercanas (colores opuestos radian valores opuestos). En cada punto se suman los valores de las influencias de todas las piedras. La influencia radiada por las piedras decae en función de la distancia. En Go Intellect decae como $(1/2)^d istancia$, en Many Faces of Go decae como $1/distancia$. Los programas que dependen fuertemente en la influencia no juegan muy bien (Explorer y Go4++ ya no utilizan influencias, aunque estos utilizan algunas técnicas similares). El uso heurístico de la influencia incluye la determinación de la conectividad entre grupos (Go Intellect) y la determinación de territorio (Many Faces of Go). En este último, el valor de radiación de un grupo también depende de la fuerza del mismo, es decir un grupo mas fuerte radia mayor influencia. Los grupos muertos pueden radiar influencia negativa, es decir, benefician al oponente.

4.4. Utilidades y herramientas

4.4.1. Servidores de Internet

Las competencias a través de Internet permiten que cualquier jugador del mundo pueda jugar contra cualquier oponente, también se tiene la oportunidad de observar juegos y comentarios en cualquier momento, lo que es una fuente muy útil de datos para entrenamiento de programas. Entre los servidores mas importantes están:

- IGS (Internet Go Server)

- NNGS (No Name Go Server)
- KGS (Kisedo Go Server)

4.4.2. Protocolo GTP

El Go Text Protocol (GTP), fue creado con la intención de proveer un protocolo de comunicación entre programas de Go que sea flexible y fácil de implementar. El objetivo es permitir que dos programas de Go puedan jugar entre ellos, permitir testeos de regresión, comunicación con GUIs y comunicación con servidores de Internet para competencias de Go. El protocolo fue desarrollado dentro del proyecto GNU Go con la intención de simplificar la automatización de testeos de regresión y la conexión de programas a los servidores. Existe un manual de especificación disponible en [6].

4.4.3. Formato SGF

El Smart Game Format (SGF), es un formato de archivo utilizado para guardar registros de juegos de dos jugadores. Es un formato basado en texto, y con una estructura de árbol. Provee propiedades como marcas, comentarios, información de juego, etc. Por lo tanto dado que es un estándar utilizado por muchos programas y servidores, es útil como forma de obtener datos de partidas jugadas, compartir información, etc.

Capítulo 5

Técnicas de búsqueda

5.1. Búsqueda por fuerza bruta en la resolución de juegos

La búsqueda por fuerza bruta o simplemente búsqueda, es una técnica ampliamente utilizada con éxito en la resolución de muchos juegos de tablero. Por esta razón merece una descripción de los conceptos básicos involucrados.

Básicamente la estrategia consiste en expandir el árbol de búsqueda a partir de la posición actual y evaluar las posiciones finales obtenidas, de esta forma la estrategia de juego simplemente consiste en elegir la rama por la cual se llega a la posición final más conveniente. Teóricamente en cualquier juego de la categoría del Go, el árbol de juego puede ser expandido completamente y de esta forma elegir el mejor movimiento. Pero en la práctica dicha técnica es imposible (por lo menos con los recursos actuales). Esto se debe a que expandir completamente el árbol no es posible debido a su enormidad, su crecimiento es exponencial, en el caso del Go tenemos un promedio de factor de ramificación de 250 y un promedio de profundidad del árbol de 150, lo que significa un árbol de 250^{50} nodos, lo cual es computacionalmente imposible de desarrollar con la capacidad de las computadoras actuales.

Existen en la actualidad muchas técnicas basadas en búsquedas en el árbol de juego, entre ellas *minmax*, *alfa-beta* y otros. Veamos una breve descripción de las técnicas más conocidas y que sirven como base de algunas más complejas.

5.2. MinMax

En el algoritmo MinMax, los dos jugadores se representan con Max y Min, Max intentara maximizar el resultado del juego, mientras que Min intenta 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 ultimo

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 algoritmo entonces consiste en los siguientes pasos:

1. Elegir el primer movimiento del tablero b realizado por primer jugador
2. 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 $b2$.
3. Para cada hoja, calcular estáticamente su valor.
4. Dar a cada tablero del jugador Max, la mayor evaluación entre los nodos hijos.
5. Dar a cada tablero del jugador Min, la menor evaluación entre los nodos hijos.
6. Elegir el movimiento que lleva del tablero b al tablero con máxima evaluación.

En la figura 5.1 Se muestra un ejemplo de un árbol minmax.

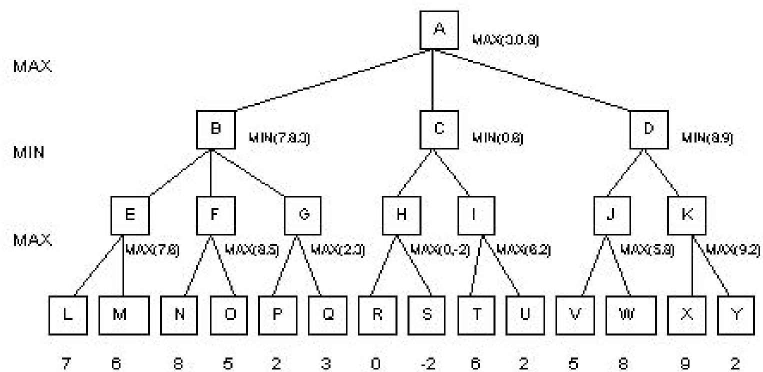


Figura 5.1: Ejemplo de un árbol MinMax.

En la figura, 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.

Dado que no es posible expandir completamente el árbol de búsqueda en juegos con espacios de búsqueda muy grandes, las hojas de dicho árbol, 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 dicha función como se mencionó anteriormente consiste en evaluar una posición estáticamente a partir de las características de la misma, sin expandir mas el árbol de búsqueda. En el caso del Go es muy difícil construir una función de evaluación adecuada, por lo que la aplicación de dicho método se complica.

5.3. Poda Alfa-Beta

La poda alfa-beta es una simple optimización del algoritmo minmax. Simplemente se evita examinar (se podan) ramas (soluciones parciales) que se puede saber con seguridad que son peores que otra solución conocida o umbral. No aportan más información de cual será la elección del movimiento. El algoritmo consiste en los siguientes pasos:

alfabeta(b**,**α**,**β)

1. Si b está al limite de profundidad, retornar la evaluación estática de b; sino continuar.
2. Sean $b1, \ldots, bn$ 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 $\alpha := max(\alpha, alfa - beta(b_k, \alpha, \beta))$.
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.

La figura 5.2 muestra la aplicación de la poda alfa-beta al ejemplo anterior. 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).

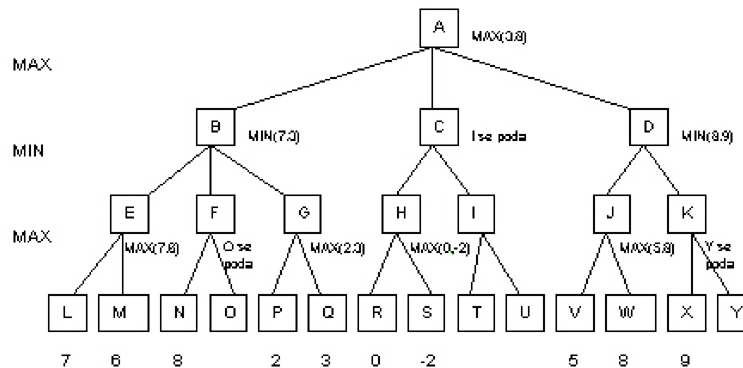


Figura 5.2: Ejemplo de un árbol Alfa-Beta.

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 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 *alfabeta* depende en gran medida del orden en que se examinen los caminos. Si se examinan primero los peores caminos no se realizará ningún corte. Pero, naturalmente, si de antemano se conociera el mejor camino no necesitaríamos buscarlo La efectividad de la técnica de poda en el caso perfecto ofrece una cota superior del rendimiento en otras situaciones. Según Knuth y Moore (1975), si los nodos están perfectamente ordenados, el número de nodos terminales considerados en una búsqueda de profundidad d con *alfabeta* es el doble de los nodos terminales considerados en una búsqueda de profundidad $d/2$ sin *alfabeta*. Esto significa que en el caso perfecto, una búsqueda *alfabeta* nos da una ganancia sustancial pudiendo explorar con igual costo la mitad de los nodos finales de un árbol de juego del doble de profundidad.

5.4. Otras técnicas de búsqueda

Existen otras posibles mejoras en los algoritmos de búsqueda basados en minmax, el desarrollo de dichos acercamientos están fuera del alcance de este documento. La presentación de los algoritmos minmax y alfa-beta son suficientemente representativos como para tener una introducción de las técnicas de búsqueda para luego relacionarlos con la dificultad de su aplicación en el Go.

5.5. Aplicaciones en el Go

Estas y otras técnicas de búsqueda en el árbol de juego han sido aplicadas con éxito en muchos juegos. Este no es el caso del Go que como se mencionó anteriormente tiene un enorme árbol de juego. Por lo tanto la aplicación de estas técnicas se ve limitada por el enorme árbol de búsqueda y por la carencia de una función de evaluación adecuada. Una posible forma de buscar reducir el árbol de búsqueda mediante podas podría consistir en la aplicación de conocimiento en forma de patrones. De esta forma el conjunto de movimientos a considerar en cada posición se podría reducir a aquellas que aplican a patrones de resolución de situaciones locales. Por otro lado, como se mencionó al tratar las funciones de evaluación, la evaluación de una posición requiere la utilización intensiva de conocimiento para poder evaluar las características de la misma y así utilizar estas propiedades como material para el ajuste de una función de evaluación. En las siguientes secciones se proponen posibles aplicaciones de técnicas de aprendizaje automático que en parte podrían ser aplicadas conjuntamente con técnicas de búsqueda.

Capítulo 6

Técnicas de aprendizaje automático

¿Por que utilizar aprendizaje automático? Hasta ahora hemos visto el acercamiento tradicional en la resolución de juegos: búsquedas en el árbol de juego. Como hemos visto una búsqueda completa en el árbol de búsqueda es imposible. 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. Este uso intensivo de conocimiento trae como consecuencia una gran dificultad asociada a la codificación, generación y mantenimiento manual. La dificultad de codificación viene dada por la gran cantidad de información que debe ser manejada manualmente, 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. Mas aún, agregar conocimiento manualmente implica el requisito de que el programador debe ser un muy buen jugador de Go. Por otro lado debería de lidiar con las dificultades de encontrar reglas sin pasar por alto casos excepcionales que podrían dar malos resultados. Por estas razones el uso de técnicas de aprendizaje automático y generación automático de conocimiento tiene un futuro prometedor. La generación y mantenimiento de grandes cantidades de conocimiento se vería simplificado.

¿Que puede ser aprendido? En [15] se mencionan algunas aplicaciones interesantes del aprendizaje a las técnicas de búsqueda las cuales serán desarrolladas a continuación.

6.1. Mejora de la poda alfa-beta

En una búsqueda, la poda alfa-beta puede ser optimizada considerablemente según el orden en que se investiguen las ramas. Un ordenamiento malo (conside-

rar los peores movimientos primero) causa que el algoritmo alfa-beta se comporte equivalentemente al minmax. Un ordenamiento perfecto (considerar los mejores movimientos primero) hace que el algoritmo alfa-beta tenga una complejidad del mismo orden que una búsqueda completa minmax pero con un factor de ramificación de solo la raíz cuadrada de la cantidad de movimientos. Esto significa que se podrían expandir las búsquedas en el doble de profundidad de una búsqueda minmax.

Por lo tanto una heurística que utilice conocimiento para el reconocimiento de movimientos tácticos vitales en una posición, ayudaría a determinar que movimientos tienen más importancia y cuales se considerarían malos movimientos. De esta forma se podría tener una ordenación en la búsqueda del árbol de juego más favorable y por lo tanto realizar más podas, traducido en la práctica como menor utilización de recursos de memoria y calculo o visto de otra forma, la capacidad de poder analizar más profundamente el árbol de búsqueda lo que significaría un mejor juego. Algunas propuestas de este tipo pueden verse en [9, 15].

6.2. Ajuste de la función de evaluación

A pesar de las optimizaciones en los algoritmos de búsqueda y el aumento en el poder de cálculo de los computadores, la profundidad de los árboles de búsqueda está limitada. Como hemos visto anteriormente, en árboles de juego muy grandes, las hojas del árbol de búsqueda no coinciden con las hojas en el árbol de juego, por lo tanto es necesaria la evaluación estática de la posición. En muchos juegos se han encontrado funciones de evaluación adecuadas. En el Go esta tarea es mucho más difícil de hacer manualmente. Un posible acercamiento a la solución sería la aplicación de técnicas de aprendizaje automático para el ajuste de los parámetros de la función de evaluación, es decir, aprender una heurística para el ajuste de los pesos de cada propiedad de una posición, y de este modo obtener una función de evaluación cada vez mas adecuada. El ajuste automático de los pesos de la función de evaluación es uno de los problemas de aprendizaje más estudiados en los juegos, ver [7] Típicamente la situación se presenta de la siguiente forma: El programador esta 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). Lo que queda por conocer, es como combinar estas piezas de conocimiento y cuantificar su importancia relativa. En el diseño del algoritmo de aprendizaje de una función de evaluación se deben de tener varias decisiones de partida, muchas veces dependientes del problema a resolver. Entre las decisiones más importantes se encuentran:

Función de evaluación lineal vs no lineal La función de evaluación debe de ser fácilmente computable. Una combinación lineal de algunas propiedades que caractericen a las posiciones del juego puede ser una primera aproximación. Por otro lado una función no lineal tiene la ventaja de que pueden aproximar

un conjunto mayor de funciones. La desventaja de una función de evaluación no lineal radica en un cómputo más lento y un entrenamiento más largo. Otro posible acercamiento a la solución podría ser la utilización de diferentes funciones de evaluación dependiendo de la fase del juego.

Estrategias de entrenamiento Otro tema importante es como proveer los ejemplos de entrenamiento, por un lado se busca una convergencia rápida a una función de evaluación correcta, y por otro lado se busca suficiente variación en los ejemplos como para que la función de evaluación pueda ser utilizada en situaciones generales. Estrategias de entrenamiento tales como jugar contra si mismo, puede ser una estrategia adecuada para juegos no deterministicos tales como el backgammon y por otro lado no ser adecuados para juegos deterministicos. Esto se debe a que gracias a la aleatoriedad se puede lograr una suficiente variedad de posiciones de entrenamiento, mientras que en caso deterministico esto no se da. Algunos estudios experimentales realizados por Epstein [7], presentan evidencia de que el entrenamiento autodidacta no obtiene buenos resultados en juegos deterministicos. Como resultado, propone el entrenamiento lección y práctica (lesson-and-practice) en donde se alterna fases de entrenamiento con un jugador experto y luego con entrenamientos autodidactas. Otra estrategia interesante propuesta por Samuel consiste en entrenar el programa con una copia de si mismo pero con los pesos de la función de evaluación fijos. Luego de una cierta cantidad de juegos y cuando se haya observado la superioridad del primer jugador, se transfieren los nuevos pesos de la función de evaluación al segundo jugador y se repite el proceso. Esta técnica evita el hecho de quedar trancado en un óptimo local. Otra técnica propuesta por Lee, Mahajan y Fogel consiste en asegurar una amplia exploración del espacio de posibilidades haciendo que los primeros movimientos sean realizados aleatoriamente. Por último otro posible método es el entrenamiento ante jugadores a través de Internet, esto posibilita una gran variedad de oponentes de diferentes niveles.

Existen varios acercamientos para resolver el problema del ajuste de una función de evaluación, en las siguientes secciones se categorizarán los posibles acercamientos según el tipo de entrenamiento.

6.2.1. Aprendizaje Supervisado (Supervised learning)

El aprendizaje supervisado, es un proceso por el cual se tienen como entrada ejemplos de entrenamiento en los cuales para cada entrada se conoce la salida correcta. En el ajuste de los pesos de una función de evaluación, los ejemplos consisten en posiciones del tablero y el correspondiente valor exacto de la función de evaluación. El programa luego intenta ajustar los pesos de forma de minimizar el error de la función de evaluación en dichos puntos. La función obtenida es luego utilizada para obtener los valores en nuevas posiciones desconocidas. Un ejemplo bastante descriptivo, aplicado por Mitchell en la creación de una función de evaluación para el juego de Othello, consistió en seleccionar 180 posiciones ocurridas en un campeonato luego de la jugada 44 de cada juego y evaluar el valor exacto de la posición utilizando un algoritmo minmax. Luego estos valores

fueron utilizados para computar los pesos de 28 parámetros en una función de evaluación lineal. Como otro ejemplo, Tesauro desarrolló TD-GAMMON, cuya función de evaluación fue ajustada mediante la utilización de miles de posiciones evaluadas por expertos. Las posiciones de entrenamiento fueron obtenidas de varias fuentes: libros, juegos de él mismo, y juegos de Internet. El problema más importante del aprendizaje supervisado es la obtención de ejemplos apropiados para el entrenamiento. La generación de los ejemplos no es tarea fácil, ya sea por medios automáticos o por medios manuales. Además se debe de tener en cuenta que los ejemplos tengan una distribución adecuada, que se asemeje a la realidad, pero que tampoco oculte casos excepcionales. Otra forma de encarar el problema de la distribución de ejemplos es elegir los mismos de forma de indicar la dirección en la cual los pesos deben ser ajustados.

6.2.2. Aprendizaje por Refuerzos (Reinforcement learning)

Cuando no existen ejemplos de entrenamiento como para utilizar una técnica de aprendizaje por supervisión, entonces una técnica de aprendizaje por refuerzo puede ser más adecuada. En esta técnica las diferentes acciones posibles son exploradas y luego el sistema recibe retroalimentación del ambiente (recompensa). Esta recompensa luego es utilizada para evaluar el éxito de cada acción. En los juegos, esta recompensa es obtenida típicamente al final del juego, con el resultado del mismo. Para utilizar dicha recompensa, el algoritmo de aprendizaje debe de distribuir dicha recompensa sobre las acciones que contribuyeron al final del juego. El problema más importante que se presenta es el llamado problema de *asignación de crédito* (*credit assignment problem*, Minsky), esto es, el problema de distribuir la recompensa entre las acciones responsables del resultado. Varias situaciones complicadas pueden darse, por ejemplo, en la derrota de un juego, puede ser responsable una sola acción mientras que las demás pueden haber sido todas buenas acciones. Algunas soluciones al problema han sido propuestas, una es dar a todas las acciones intermedias el mismo valor de acuerdo a la recompensa, aunque esta solución intuitivamente no seria coherente con el caso anterior. Otra posible solución es darle mayor impacto a las posiciones cercanas al final del juego. Aunque a primera vista puede haber casos que no se ajusten a esta solución, se han probado teoremas de convergencia, que confirman que esta solución es correcta a medida que se avanza en el entrenamiento.

6.2.3. Aprendizaje por Diferencias de Tiempos (Temporal Difference (TD) learning)

Para entender la idea de TD learning conviene tomar primero la idea del aprendizaje supervisado. En el aprendizaje supervisado, cada posición encontrada en etapas intermedias de una partida es evaluada según el resultado final del juego. Naturalmente esta técnica no es perfecta, por ejemplo, en el caso en que un jugador realice buenas jugadas la mayor parte de la partida, pero que termine perdiendo por un error al final del juego, las jugadas buenas que fueron

hechas serán catalogadas como malas. Si por otro lado, cada posición es evaluada según el valor de posiciones hasta cierto límite de profundidad del juego, los ajustes de los pesos serán hechos de una mejor manera. Esto se debe a que las posiciones cercanas al final van a ser evaluadas según el valor final del juego, mientras que el valor final del juego no tendrá tanto efecto en la evaluación de las primeras jugadas de la partida. 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 z. 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 z para cada i. Típicamente se necesitaría un conjunto de entrenamiento con varias de dichas secuencias, en cada paso i, se busca ajustar los pesos W de modo que la diferencia entre $f(X_i)$ y z sea mínima. En el caso del aprendizaje TD, el aprendizaje se basa en la diferencia entre las estimaciones sucesivas $f(X_{i+1})$ y $f(X_i)$ en vez de las diferencias entre z y $f(X_i)$. Se busca entonces, una función f que depende del valor X y los pesos W. Por lo tanto en el caso de aprendizaje supervisado, para cada X_i, la predicción $f(X_i, W)$ se computa y se compara con z. Luego la regla de aprendizaje (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:

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

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

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

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

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

definiendo $f_{m+1} = z$ y sustituyendo en la formula de $\triangle W_i$:

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

En este caso en vez de usar la diferencia entre la predicción y el valor de z, se utilizan las diferencias entre predicciones sucesivas, por lo que se conoce con el nombre de diferencias temporales (temporal difference learning). Esto 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 mas posteriores en el tiempo que $t = i$, lo que se proponía al inicio de la sección. Cuando $\lambda = 1$, tenemos la misma expresión que antes, es decir 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).

6.3. Aprendizaje de aperturas de juego

Los jugadores humanos no se basan solamente en su habilidad de estimar el valor de cada jugada y posiciones, sino que también juegan ciertas posiciones basándose en conocimiento previo sin la necesidad de pensar las consecuencias en la siguiente posición. Esto es el resultado del estudio rutinario de aperturas de juego que han sido resumidas por jugadores profesionales. Dada la capacidad de memorización de las computadoras, el uso de una base de datos de aperturas de juegos es una manera fácil de incrementar el nivel de juego de un jugador artificial. Sin embargo la construcción de dicha base de datos y su mantenimiento es una tarea extremadamente complicada. La idea de utilizar una base de aperturas de juego ha sida utilizada desde los principios de la creación de jugadores artificiales. El uso de dicho conocimiento en forma de jugadas precalculadas, las cuales tienen un resultado bien conocido, evitaría búsquedas en el árbol de juego. Otro problema inherente a esta técnica es el hecho de construir una heurística para la decisión de que jugada utilizar en cada caso particular, problema que esta lejos de ser trivial. Un aspecto interesante es el hecho de no basarse únicamente en aperturas de juego tomadas de libros profesionales, sino también extender la base de aperturas, es decir, automáticamente generar nuevo conocimiento sobre aperturas de juegos.

6.4. Aprendizaje de patrones

Los patrones son maneras de representar conocimiento y en el caso del Go están presentes en casi todos los prototipos. Tradicionalmente los patrones han sido generados manualmente, es decir, tomados de expertos y codificados a mano. Recientemente se han propuesto acercamientos donde 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. Primero comenzaremos por describir claramente en que consiste un patrón y su aplicación especifica en el Go. En [12] se da una descripción muy clara la cual utilizaremos. Un patrón consiste básicamente un conjunto de intersecciones donde cada una puede tener un estado determinado (en el caso del Go: negro, blanco o vacío). En la figura 6.1 se muestra un ejemplo de un patrón.

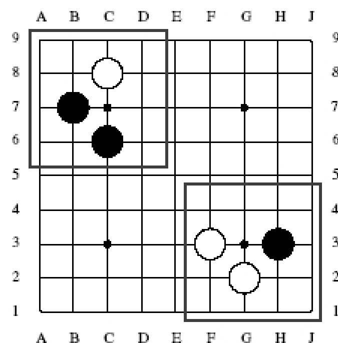


Figura 6.1: Las dos partes recuadradas aplican al mismo patrón.

Un patrón en nuestro caso tiene dos dimensiones y debe ser comparado con una posición dada de un tablero completo para probar su aplicabilidad. Un patrón cualquiera puede tener 8 diferentes instancias (aplicaciones) por medio de la rotación o la simetrización horizontal y vertical, por otro lado, también los colores pueden ser intercambiados. En la figura los dos casos son instancias del mismo patrón. Como es de esperar, es necesario un gran poder de procesamiento para poder comparar cada patrón con cada posible localización del mismo en una posición de tablero. Gracias a técnicas de filtrado, es posible reducir considerablemente el overhead de procesamiento. Una simple técnica consiste en guardar la información de en que localizaciones determinados patrones aplican y en donde ningún patrón aplica. De este modo, en los siguientes movimientos, dicha información permanecerá incambiada en localizaciones del tablero donde no han habido cambios. Además de la información acerca de las intersecciones contenidas en el patrón, este debe contener información del contexto en el cual puede ser aplicado, es decir, restricciones que deben ser satisfechas en las fronteras externas al patrón. Dichas restricciones pueden consistir en cantidad de libertades que debe tener una piedra en el borde del patrón, etc. Luego de que es probado el hecho de que el patrón aplica en una localización particular del tablero, la información contenida en el patrón puede ser utilizada. Esta información puede consistir en movimientos recomendados para diferentes fines tales como mantener un grupo vivo, atacar un grupo, conectar dos grupos, etc. También podría tratarse de información para ser utilizada en la evaluación de una posición. En conclusión, un patrón encapsula conocimiento que puede ser muy útil para un programa de Go. Si un patrón aplica, entonces podría evitar la búsqueda innecesaria en el árbol de juego, el patrón actúa como una especie de jugada precalculada, es decir, si el patrón aplica y movemos en el lugar indicado por el patrón, sabremos cual será el resultado final sin realizar una búsqueda en el árbol. Sin embargo es necesario tener en cuenta optimizaciones para reducir el tiempo de buscar patrones que apliquen a una posición determinada. Por otro lado, puede que en una localización particular más de un patrón pueda ser

aplicado, esto provoca la necesidad de contar con algoritmos que puedan decidir que patrón utilizar. En el caso de generación automática de patrones, existe otra consideración de diseño a tener en cuenta: el momento y circunstancias en la cual generar los patrones. Una opción podría ser aprender los patrones en el momento en que el prototipo juega (online). Otra opción puede ser aprender los patrones a partir de una base de datos fuera de línea (*offline*). Por ultimo el diseño podría utilizar una estrategia mixta que reúna los dos acercamientos anteriores.

6.4.1. Explanation Based Generalization (EBG)

Es un proceso por el cual se deducen reglas sobre un dominio de teorías. La idea básica es encontrar explicaciones de ejemplos particulares a partir del dominio de teorías. Luego generalizando dicha explicación se llega a un patrón o regla general que explica a varios ejemplos similares. En el futuro, el programa utilizara dicho patrón en situaciones donde se cumplan las características necesarias como para aplicar el patrón. Aplicar este método a la resolución de juegos de tablero, resulta en la creación de una base de datos de patrones. En cada movimiento el programa puede intentar aplicar alguno de esos patrones para decidir el siguiente movimiento sin la necesidad de realizar una búsqueda en el árbol de juego. Uno de los problemas que puede presentar esta técnica es el llamado problema de utilidad, básicamente consiste en el hecho de que el sistema puede tender a la creación de una gran cantidad de reglas, muchas de ellas tan especificas que casi en ningún momento serán utilizadas. Obviamente esto no es deseable ya que aumenta innecesariamente el tamaño de la base de datos y el overhead en el tiempo de búsqueda de patrones que se ajusten a una posición dada. Un requisito fundamental del dominio específico donde aplicar una técnica de este estilo, es el hecho de poseer un dominio de teorías correctas. Dado que la idea es realizar deducciones lógicas sobre el dominio de teorías, si alguna de estas teorías fuera errónea, se estaría deduciendo una regla sintácticamente correcta, pero semánticamente incorrecta. En [14], Nilsson describe conceptos importantes sobre esta técnica. Supongamos un conjunto de hechos y teorías T, y una proposición lógica p. En principio puede considerarse la siguiente pregunta: si p se deduce de T, ¿acaso esto se puede considerar como haber aprendido p? En cierta forma, al conocer T, implícitamente ya conocemos p. Entonces, ¿se puede considerar a la deducción como un proceso de aprendizaje? Ciertamente, si, quizás dicha deducción lógica no es tan obvia, sino que fue muy difícil deducirla, en vez de deducir p cada vez que la necesitemos, ¿por que no guardarla y usarla directamente?, Dietterich llama este tipo de aprendizaje como *speed-up* learning. Estrictamente este tipo de aprendizaje no resulta en que el sistema realice decisiones que antes no podía hacer, simplemente hace posible que realice decisiones mas rápidamente. Pero en la práctica este tipo de aprendizaje puede resultar en permitir realizar decisiones que de otra manera no seria posible. Un par de ejemplos son bastante gráficos: Un jugador de ajedrez se dice que aprende ajedrez aun cuando la manera óptima de jugar al ajedrez es inherente a las reglas del juego. En este caso es obvio que solo conociendo

las reglas del juego no significa jugar bien al ajedrez, sino que es necesario un proceso de deducción por el cual ir aprendiendo reglas tácticas y estratégicas. Otro ejemplo, supongamos que se tienen un conjunto de teoremas de geometría y se busca demostrar que la suma de los ángulos de un triángulo rectángulo, es 180 grados. Si realizáramos una demostración que no dependiera del hecho de que el triángulo dado era rectángulo, estaríamos demostrando algo más general, es decir aprendiendo un hecho más general.

Para juegos como el Go, donde se parte de un dominio de teorías bastante fuerte, un método de aprendizaje deductivo puede ser aplicable.

Caso Kojima-Yoshikawa Kojima y Yoshikawa [10, 11] han aplicado EBG específicamente para problemas de captura de piedras en un sistema implementado en Prolog. El objetivo es deducir reglas de fuerza, esto es, reglas en las cuales de ser aplicadas, el oponente aunque juegue óptimamente no puede evitar ser capturado. El sistema comienza con una única regla de fuerza básica (atari) y luego va deduciendo reglas más complejas a partir de ejemplos que se le van suministrando. Al presentarle una posición donde ciertas piedras fueron capturadas (ver figura 6.2), el sistema va deshaciendo los movimientos (interpretándolos como la aplicación de una regla de fuerza previamente aprendida) hasta llegar a una posición donde no es posible aplicar una regla de fuerza conocida. En este punto el sistema crea una nueva regla de fuerza para explicar dicha situación, creando un patrón que generalice las propiedades de la posición.

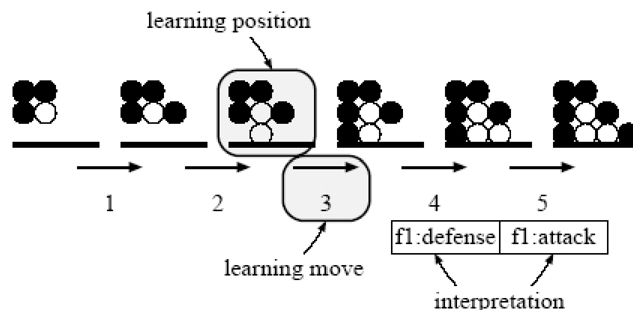


Figura 6.2: Búsqueda en el aprendizaje de un movimiento.

El sistema presentado en [10] consiste en cuatro módulos: Knowledge Base (KB), Decisión Maker, Rule Acquisition Module y Rule Refinement Module. El sistema juega al Go contra alguien, durante un juego, el módulo Decisión Maker busca una regla apropiada en KB a aplicar, si no la encuentra pone una piedra en una posición aleatoria. Luego de terminado el juego, si el sistema pierde, el módulo Rule Acquisition obtiene deduce algunas reglas y las agrega a KB. Cuando el sistema detecta reglas incorrectas, el módulo Rule Refinement, las refina. En el refinamiento, el sistema extrae las reglas incorrectas del KB, las

corrige y las reingresa a la KB.

La KB tiene dos tipos de reglas, reglas básicas y reglas de fuerza. Las reglas básicas son definiciones de conceptos básicos en el Go. Las reglas de fuerza, como se mencionó, son reglas que de ser aplicadas, el jugador que la aplica siempre gana (la contienda local). Un ejemplo de una regla de fuerza es: "si una cadena del oponente esta en atari (tiene una libertad), entonces poner una piedra en su última libertad". En el estado inicial, el sistema solo tiene dicha regla como única regla de fuerza. Luego del proceso de aprendizaje, el sistema va deduciendo nuevas reglas de fuerza. Se propone dos formas de aplicar las reglas de fuerza: aplicar una regla de fuerza para ataque o para defensa. Es decir, cuando un jugador utiliza la regla para en este caso capturar piedras contrarias, es una aplicación para ataque. Cuando el jugador reconoce que el oponente puede aplicar una regla de fuerza para capturar sus piedras en el siguiente turno, el jugador debería mover en la posición que indica dicha regla de fuerza, de modo de escapar del ataque, es decir, una aplicación para defensa. Las decisiones a la hora de elegir un movimiento, realizadas por el módulo Decision Maker, es simple, el siguiente algoritmo resume la idea:

1. Si es posible aplicar regla f1 para ataque entonces aplicarla.
2. Si el oponente puede aplicar la regla f1 para ataque, entonces aplicar la regla f1 como defensa.
3. Si es posible aplicar una regla de fuerza diferente de f1 para ataque, etonces aplicarla.
4. Si el oponente puede aplicar una regla diferente de f1 para ataque, entonces aplicar la misma regla para defensa.
5. Elegir aleatoriamente un lugar libre donde poner una piedra.

El módulo de obtención de reglas básicamente consiste en tres etapas. En la primer etapa busca movimientos para ser aprendidos a parir de una secuencia de movimientos en el juego. El sistema investiga los movimientos desde el ultimo al primero (sentido inverso). Este busca interpretar los movimientos de los jugadores como aplicaciones de reglas de fuerza. En la figura 6.2 el movimiento 5 es interpretado como la aplicación de la regla *f1* (atari, la regla inicial) para ataque por el negro. El movimiento 4 es interpretado como la aplicación de la regla *f1* para defensa por el blanco. El movimiento 3 no puede ser interpretado por ninguna regla de fuerza que el sistema posea actualmente. Por lo tanto una regla que interprete dicho movimiento debe ser aprendida. En la segunda etapa el sistema extrae la parte esencial del tablero para el cual la regla puede ser aplicada en el futuro. Se utiliza una técnica de EBG para extraer dicha información mediante la construcción de un árbol de explicación. En la ultima etapa se generaliza la posición y el movimiento (la regla) cambiando las coordenadas constantes por variables, de esta forma es posible utilizarla en situaciones generales.

6.4.2. Metaprogramación y EBG

El objetivo de la metaprogramación en el caso del Go, es escribir programas (metaprogramas), que escriban otros programas que permitan realizar podas en los árboles de búsqueda, y de esta forma permitir grandes mejoras en el tiempo de ejecución. En el caso de los juegos, las metareglas son utilizadas para crear teoremas que indican que movimientos son interesantes en una posición para poder lograr un objetivo táctico. También son utilizados para crear reglas que encuentren conjuntos completos de movimientos forzados que prevengan que el oponente logre determinado objetivo táctico. Cada vez que el sistema intenta ver el grado de logro de un objetivo, debe realizar dos árboles de prueba AND/OR, uno con el color amigo primero y otro con el color enemigo primero.

Caso Cazenave En [4, 3, 5], Cazenave ha realizado otro acercamiento en el uso de EBG. Se basa en un sistema para metaprogramación basado en EBG llamado Introspect. Este crea reglas tácticas para muchos juegos, en particular el Go, utilizando lógica de predicados para representar las reglas, ej:

```
conectar(S1,S2,I,Color):-
    color_bloque(S1,Color),
    color_bloque(S2,Color),
    libertad(I,S1),
    libertad(I,S2).
```

Las reglas generadas son aplicables a sub-objetivos tácticos del Go, ej: captura de piedras, hacer que piedras sigan vivas, conectar grupos, desconectar grupos, hacer ojos, remover ojos. Estas son utilizadas para guiar la búsqueda en árboles de juego, podar el árbol de búsqueda reduciendo el número de movimientos que son examinados. Es interesante el hecho que el conjunto de reglas de fuerza obtenidas es completo, esto es, si podemos probar que ninguna de las reglas que se tienen son aplicables, entonces puede ser probado que ningún movimiento sirve. De esta forma si una regla concluye que un objetivo fue logrado, entonces la búsqueda puede ser terminada. Las reglas generadas mediante Introspect son luego compiladas en código C y utilizadas en su programa de go, Gogol.

Por otro lado, los sistemas basados en EBG tienden a crear mucho conocimiento, y muchas veces, reglas que no tienen utilidad y enlentecen el sistema. Para evadir este problema (problema de utilidad), Introspect genera reglas que contengan menos de 200 condiciones y que no concluyan en más de 5 movimientos forzados. También pueden crearse metaprogramas que eliminen reglas que nunca son aplicables o que simplifiquen reglas.

El siguiente ejemplo es una regla generada por Introspect para la captura de una cadena de piedras:

```
capturar(C,S,I):-
    color_opuesto(C1,C),
    color_bloque(S,C1),
```

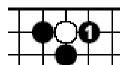
```
libertades(S,2),
minimo_numero_libertades_bloques_adyasentes(S,2),
libertad(I,S),
minimo_numero_libertades_si_mueve(I,C,2),
libertad(I1,2),
I=\=I1,
numero_libertades_bloques_si_mueve(S,[I,C],[I1,C1],1).
```

6.4.3. Inductive Logic Programming (ILP)

El aprendizaje inductivo puede verse como el aprendizaje de la representación de una función, esto es, a partir de una colección de ejemplos de una función f, retornar una función h que aproxima f. Para poder cumplir con dicho objetivo, el sistema de aprendizaje debe buscar sobre un gran espacio de hipótesis para encontrar la hipótesis (generalización) h que mejor se ajusta a los ejemplos de entrada. ILP, también llamado aprendizaje multirelacional, es una subespecialidad del campo de aprendizaje automático la cual utiliza conocimiento previo y ejemplos para inducir reglas en forma de predicados de primer orden, más específicamente, 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, dicho aprendizaje es usualmente llamado *Inductive Logic Programming* (ILP), esto es, inducción de programas lógicos. Los sistemas de ILP representan los ejemplos y las hipótesis utilizando cláusulas Horn, e incluyen conocimiento previo (*background knowledge* en ILP). 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 una definición de 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.

Caso Ramon-Francis-Blockeel Como fue descrito anteriormente, un factor importante que afecta la utilización de técnicas de búsqueda en el Go es el gran factor de ramificación, incluso para problemas locales. En [16] se afirma que los jugadores humanos son muy fuertes en el reconocimientos de patrones frecuentes y puntos vitales del juego. Esto les permite seleccionar las jugadas más prometedoras y de esta forma podar el árbol de búsqueda. En su trabajo, afirman que mucho de estos patrones pueden ser representados con conceptos relacionales. Presentan su aplicación basada en el sistema de ILP TILDE, el cual aprende heurísticas que evalúan los movimientos candidatos en situaciones de tsume-go (vida o muerte). Uno solo puede aprender lo que puede ser expresado,

de esta afirmación se desprende la importancia del lenguaje en el cual representar el conocimiento. La mayoría de los acercamientos en el aprendizaje de patrones describen los patrones como partes del tablero donde las piezas son asignadas en posiciones específicas. Este es un acercamiento proposicional, en la mayoría de los juegos funciona bien, incluso en algunos patrones elementales del Go. Sin embargo, los autores afirman que muchos de los conceptos manejados en el Go son relacionales, y por lo tanto patrones proposicionales no son adecuados para describirlos. Un ejemplo de un patrón proposicional se muestra en la figura 6.3



IF (-3,1)=empty, (-2,0)=black, (-2,1)=empty, (-2, 2)=edge, (-1,-1)=black, (-1,0)=white, (-1,1)=empty, (0,1)=empty, (1,0)=empty THEN black plays on (0,0)

Figura 6.3: Ejemplo de un patrón proposicional.

Las formulas proposicionales se basan en formas del estilo *atributo* = *valor*, en el caso del Go los atributos podrían ser posiciones del tablero y sus valores negro, blanco, vacío, borde, no importa, etc. Otros acercamientos proposicionales han intentado ampliar la expresividad por ejemplo, permitiendo más flexibilidad en la representación. Sin embargo, en el Go, un acercamiento proposicional no es natural, es muy difícil poder representar 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. En [16] se propone la representación relacional de la figura 6.4.

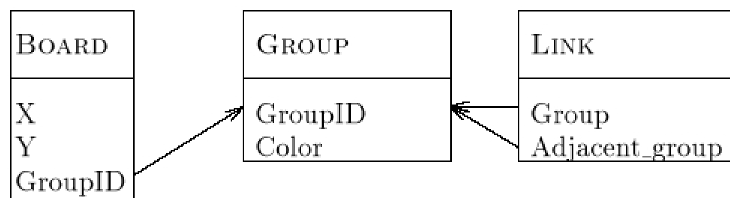


Figura 6.4: Representación relacional [16].

Cada intersección en el tablero pertenece a un grupo indicado en la relación Board. Un grupo puede ser una intersección vacía o una cadena de piedras. Esto es definido en la relación Group, que especifica el color del grupo. También para cada grupo, la lista de los grupos adyacentes es guardada en la relación Link. A partir de dichas relaciones, se puede definir conocimiento previo (background knowledge). Las siguientes son algunas de estas relaciones en el lenguaje

PROLOG.

Predicado	**Significado**
liberty(G,L)	L es una libertad del grupo G
libertycnt(G,N)	N es el número de libertades de G
stonecnt(G,N)	N es el número de piedras en el grupo G
grouponpos(Pos,Disp,G)	G es el grupo en la intersección Pos+Disp
distancetoedge(Pos,Dir,Dist)	Dist es la distancia de Pos al borde en la dirección Dir
$X < Y, X =< Y, X = Y$	Operadores de comparación

En la práctica, algunos de dichos predicados tienen argumentos extras que le permiten manejar la invarianza respecto a la simetría axial, de rotación y de intercambio de colores. El objetivo es que un algoritmo de aprendizaje automático pueda aprender conceptos del Go, por ejemplo conceptos como doble atari y geta.

```
double_atari((X,Y),A,B) :-
     liberty_cnt(A,2), liberty_cnt(B,2),
     liberty(A,L),liberty(B,L), board(X,Y,L).
geta_2((X,Y),Color,A) :-
     group_on_pos((X,Y),(1,1),A),group(A,Color),
     liberty_cnt(A,2),liberty(A,L1),liberty(A,L2),
     group_on_pos((X,Y),(1,0),L1),
     group_on_pos((X,Y),(0,1),L2),
     group_on_pos((X,Y),(1,-1),L3),group(L3,empty),
     group_on_pos((X,Y),(-1,1),L4),group(L4,empty).
```

Figura 6.5: Ejemplos de reglas inducidas [**?**].

Es destacable observar que las definiciones de las reglas son cortas, claras y naturales, esto respalda la teoría de que un lenguaje relacional es más apropiado y por otro lado simplifica el trabajo del programador.

En [16] se presentan resultados experimentales que respaldan la teoría de que una representación más expresiva es superior a las proposicionales. La performance de la técnica presentada es comparable con otros acercamientos de aprendizaje y heurísticas programadas manualmente. Como trabajo futuro se menciona el hecho de dar predicciones de movimientos en todas las secuencias en un problema local tanto como captura y ataque. Por otro lado se podría intentar utilizar la misma técnica para otras situaciones del juego como por ejemplo la apertura del juego y le terminación. De este modo la técnica podría tener una aplicación muy interesante en un jugador artificial real.

6.4.4. Algoritmos ecológicos

Caso Kojima-Yoshikawa En [10] Kojima y Yoshikawa proponen un algoritmo ecológico para la generación de patrones en el Go. Básicamente se puede ver como una simulación de una evolución. El objetivo del algoritmo es obtener reglas útiles, las cuales llaman "individuos", que aplican a datos de entrenamiento, los cuales llaman "comida". Cada regla es una producción de la forma IF (condiciones) THEN (acciones). Cada regla tiene un valor asociado el cual llaman "valor de activación". En el estado inicial del algoritmo no hay reglas. Las reglas que aplican a los datos de entrenamiento reciben comida, por lo que su valor de activación se incrementa. Si ninguna regla aplica a algún dato, una nueva regla es creada que aplica a dicho dato, con una sola condición en la parte IF. Por lo tanto en las etapas iniciales del algoritmo, la cantidad de reglas con solo una condición crece considerablemente. Las reglas que luego de ser alimentadas tienen un valor de activación mayor a un determinado umbral, son separadas en dos reglas, una regla igual a la original y una regla más compleja. Por lo tanto, las reglas complejas son creadas a partir de la separación. En cada etapa, las reglas consumen comida, lo que les permite aumentar su valor de activación. Por otro lado en cada paso también se le descuenta una unidad en el valor de activación. De esta forma las reglas que son muy complejas y obtienen comida muy extraña, a la larga mueren. De este modo es de esperar que las reglas obtengan comida a la misma frecuencia, y de este modo se eliminan reglas poco aplicables. La figura 6.6 muestra el algoritmo presentado en [10].

En particular para el Go, las reglas son descritas en términos relativos, de este modo pueden ser aplicadas por cualquier jugador y en situaciones simétricas (rotaciones de 90 grados, reflexiones). En la figura 6.7 se muestra la forma de estas reglas.

Cada $[x_i, y_i]$ representa una coordenada relativa a la posición de acción (la posición donde la piedra es colocada según la regla), dicha coordenada es siempre [0,0]. Obj_i es alguno de los siguientes objetos:

1. SAME: el mismo color que la piedra donde se mueve.
2. DIFF: el color opuesto a la piedra donde se mueve.
3. EDGE: el borde del tablero.
4. -n, el n-esimo movimiento previo.

La figura 6.8 muestra un ejemplo de una regla, el jugador que debe jugar es el negro, la piedra a jugar siempre es la que es presentada con mayor valor, en este caso "2". La regla se lee como: "si la jugada previa es jugada en el punto [-1,-1] y una piedra del mismo color del que va a jugar (negro) existe en el punto [0,-1] y una piedra de diferente color del que va a jugar (blanco) existe en el punto [-2,-1] y un borde existe en el punto [0,-5], entonces jugar en [0,0].

En el proceso de alimentación, sobre las reglas que aplican a un dato, solo son alimentadas aquellas que no tienen una regla mas especifica. Si las siguientes cinco reglas aplican a un dato, las reglas 1 y 3 no son alimentadas ya que las

```
1   step ← 1
    while step ≤ the number of iterations
2       choose a random game record from the game
        database
3       move ← 1
        while move ≤ the number of moves in the
        game
4           a training datum ← move-th move
5           if no rule matches the datum
                then create a new rule
                else feed matched rules
            for all rules
6               if activation of a fed rule >
                threshold
                    then split the rule
7               activation of a rule ← activation of a rule −
                1
8               if activation of a rule = 0
                    then the rule dies
            end for
9           move ← move + 1
10          step ← step +1
        end while
    end while
```

Figura 6.6: Algoritmo ecológico [10].

IF exist($[x_1, y_1], obj_1$) $\wedge$ ...$\wedge$ exist($[x_n, y_n], obj_n$)
THEN play([0,0]),

Figura 6.7: Modelo de regla del algoritmo ecológico [10].

reglas 2 y 5 son más específicas respectivamente. Por lo tanto solo las reglas 2, 4 y 5 son alimentadas. En este caso cada regla obtiene un tercio de la comida provista por el dato.

1. IF C1 THEN A1
2. IF C1 $\wedge$ C2 THEN A1
3. IF C2 $\wedge$ C3 THEN A1
4. IF C4 $\wedge$ C5 THEN A1
5. IF C2 $\wedge$ C3 $\wedge$ C4 THEN A1

En lo que respecta a la separación y creación de reglas más complejas, cuando una regla llega a tener un valor de activación mayor a un determinado umbral se separan en dos reglas: ella misma (llamada padre) y otra regla más compleja (llamada hija) que se obtiene agregándole una nueva condición a la regla original.

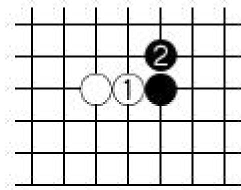


Figura 6.8: Un ejemplo de una regla: IF ([-1,-1],-1) ∧ ([0,-1],SAME) ∧ ([-2,-1],DIFF) ∧ ([0,-5],EDGE) THEN play([0,0]])[**?**].

La nueva condición es elegida aleatoriamente sobre los objetos de la posición que se este analizando (piedras, bordes, jugadas anteriores).

Bibliografía

[1] Bruno Bouzy and Tristan Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[2] Jay Burminister and Janet Wiles. Ai techniques used in computer go.

[3] Tristan Cazenave. Automatic acquisition of tactical Go rules. In H. Matsubara, editor, *Proceedings of the 3rd Game Programming Workshop*, Hakone, Japan, 1996.

[4] Tristan Cazenave. Metaprogramming forced moves. In H. Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 645–649, Brighton, U.K., 1998. Wiley.

[5] Tristan Cazenave. Metaprogramming domain specific metaprograms. *Lecture Notes in Computer Science*, 1616:235–??, 1999.

[6] Gunnar Farnebäck. *Specification of the Go Text Protocol, version 2,draft 2*, October 2002. Draft, http://www.lysator.liu.se/ gunnar/gtp.

[7] Johannes 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.

[8] Michael Gherrity. *A Game-Learning Machine*. PhD thesis, San Diego, CA, 1993.

[9] Nobuhiro Inuzuka, Hayato Fujimoto, Tomofumi Nakano, and Hidenori Itoh. Pruning nodes in the alpha-beta method using inductive logic programming. 1999.

[10] Takuya Kojima and Atsushi Yoshikawa. Knowledge acquisition from game records. 1999.

[11] Takuya Kojima. A model of acquisition and refinement of deductive rules in the game of Go. Master's thesis, 1995.

[12] Jens Lehmann. Computer go. 2004.

[13] Martin Müller. Computer go. *Artif. Intell.*, 134(1-2):145–179, 2002.

[14] Nils J. Nilsson. Introduction to machine learning. An Early Draft of Proposed Textbook, 1996.

[15] Jan Ramon and Hendrik Blockeel. A survey of the application of machine learning to the game of go. 2001.

[16] Jan Ramon, Tom Francis, and Hendrik Blockeel. Learning a Go heuristic with TILDE. 2000.

[17] A. M. Turing. Computing machinery and intelligence. 1950.