Sistemas Informáticos

Curso 2.004 - 2.005

Fractal Go

David Carro Albarrán

Adrián Riesco Rodríguez

Francisco Pedro Rodríguez Martínez

Dirigido por:

José Alberto Verdejo López

Departamento de Sistemas Informáticos y Programación

Facultad de Informática

Universidad Complutense de Madrid

Índice general

Índice de figuras

Resumen

El objetivo de este proyecto es desarrollar una aplicación que, mediante técnicas de inteligencia artificial sea capaz de jugar al Go, ofreciendo las posibilidades de juego humano vs humano, humano vs cpu y cpu vs cpu.

Para ello, diferenciamos entre las distintas fases del juego (apertura, medio y final), guardando en una base de datos la disposición de los tableros correspondientes, y complementándolo con un módulo de aprendizaje para añadir nuevo conocimiento.

Hemos dividido la carga de trabajo que supone jugar al Go entre distintos procesadores y hemos incorporado a nuestra aplicación la capacidad de comunicarse con otras utilizando el Go Text Protocol (GTP), desarrollado dentro del proyecto GNU Go y utilizado para la interconexión de distintas implementaciones de este juego.

Palabras clave: Go, patrones, inteligencia artificial, GTP, búsqueda PN, aprendizaje, distribuido.

Abstract

The object of the project is to develop an application that can play Go using artificial intelligence techniques and offering human vs human, human vs cpu and cpu vs cpu modes.

In order to get it, we distinguish between the different phases of the game (opening, medium and ending), storing patterns in a data base and using it properly.

We have divided the work-load among different processors and our applicaction has the cappability of communicating with other applications by using the GTP (Go Text Protocol), which is developed by the GNU Go project and used to interconnect different Go programs.

Keywords: Go, patterns, artificial intelligence, GTP, PN-search, learning, distributed.

Autorización

Nosotros, los integrantes del grupo desarrollador del proyecto *Fractal Go* de la asignatura de Sistemas Informáticos: David Carro Albarrán, Adrián Riesco Rodríguez y Francisco Pedro Rodríguez Martínez, autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, el trabajo presentado, siempre que seamos mencionados expresamente como los autores.

Fdo.: David Carro Fdo.: Adrián Riesco Fdo.: Francisco P. Rodríguez

Madrid, 30 de junio de 2005

Capítulo 1

Introducción

En esta sección introducimos brevemente el juego del Go, así como el vocabulario básico para entender el resto de la memoria.

1.1. Breve historia del Go

El juego se llama Uei-chi (o Wei-qi) y es muy popular en China, Corea y sobre todo en Japón. Su nombre Go, conocido en occidente, es la pronunciación japonesa de la palabra china "chi" (o qi), nombre original de este juego; en Corea se le conoce por "baduk".

Según la leyenda china, fue el emperador Yao (entre los años 2357 y 2255 a.C.) quien inventó el uei-chi, y lo enseñó a su hijo Chu-tan para entrenarle a gobernar el país. Otras versiones de la misma leyenda difieren ligeramente: el emperador pidió a uno de sus consejeros que inventase un juego para desarrollar la capacidad mental de su hijo, que era algo retrasado.

Algunos historiadores creen que por lo menos en la dinastía Chow (1134-247 a.C.) ya existía este juego, aunque en forma rudimentaria, pues en los libros de Confucio (551-497 a.C.) se hablaba ya de él.

Hasta los primeros años de la época de los Tres Reinos (alrededor de los años 220-226 de la era cristiana) en el "Tratado de las artes" se decía que el "chi" tenía 17 líneas horizontales y 17 verticales. Pero después de unos seiscientos años, al final de la dinastía Tang, en el siglo IX, se decía que el tablero tenía ya 19 líneas. Y desde entonces no ha sufrido ningún cambio.

El "chi" fue siempre un juego de las clases intelectuales y gobernantes de la

antigua China. Nunca ha sido popularizado para el vulgo. Por su parecido y semejanza a la guerra, también fue un juego favorito de los militares. Según un libro escrito en la dinastía Suei, el emperador Liang U-ti (502-549 de la era cristiana) redactó un tratado de "chi" y lo incluyó en la "Gran estrategia", obra maestra del arte de la guerra, estudio obligatorio para todos los mandos militares del Ejército.

El "chi" fue introducido en el Japón hacia el año 735 de nuestra era por un bonzo budista japonés llamado Kibi Dajin, y ha tenido una acogida muy favorable en ese país. Al comienzo estuvo restringido a los ambientes cortesanos, pero poco a poco se divulgó entre los samurais, budistas, shintoístas.

Japón fue el primer país que estableció un sistema de jugadores profesionales, hace 400 años. La famosa Academia japonesa de Go "Nihon Ki-in", una especie de autoridad académica y federativa, fue fundada en 1924. Sus funciones fundamentales consisten en enseñar el arte de Go, supervisar las competiciones públicas y conceder títulos de categoría a profesionales y aficionados. Las categorías se clasifican en nueve grados superiores o "dan" y nueve inferiores. La Academia también concede títulos especiales a los verdaderos campeones, por ejemplo, el Hon in-bo (Campeón Nacional) y el Mei jin (Gran Maestro).

Se estima que es jugado por unos cincuenta millones de personas en el sureste asiático. Normalmente se aprende a jugar en la infancia. Se juega al Go en las escuelas, en los comedores de las empresas, y en lugares especiales llamados "Salones de Go". Se juega por placer, para estimular la mente, o los jugadores profesionales por premios sustanciosos.

Aunque el juego del Go fue descrito por los viajeros europeos que fueron al extremo oriente durante el siglo XVII, no se empezó a jugar en Europa hasta 1880. Aún así el juego se ha ido extendiendo lentamente. En 1958 se jugó el primer campeonato de Europa. Hoy en día se juega al Go en todos los países europeos, si bien el nivel de juego está significativamente por debajo del de los profesionales orientales: los mejores jugadores europeos acuden a Japón a estudiar el juego.

Es posible que fuera introducido en España por Ambrosio Wang An-po, quien publicó un tratado de 181 páginas en castellano en 1970, titulado "El Cercado".

1.2. Las reglas del Go

El Go se juega en un tablero con un número igual de líneas horizontales y verticales. Aunque son habituales las versiones del juego en 9×9, 13×13 o 19×19, el más grande (19×19) es el tablero más común y siempre usado en los torneos.

Uno de las posibilidades que presenta el Go es compensar la diferencia de fuerza entre dos jugadores: el jugador más fuerte le da un cierto número de piedras de ventaja (handicap) al jugador más débil. Las piedras del handicap siempre se colocan en las intersecciones del tablero marcadas con un punto grueso, y se distribuyen en dichos puntos antes de iniciar la partida. El máximo de piedras de ventaja es 9. Cuando una partida se juega con handicap, las piedras de handicap colocadas son negras y las blancas inician la partida.

Al principio del juego el tablero está vacío, excepto si se juega con handicap. Las piedras se colocan en los puntos de intersección de las líneas. Una vez jugadas nunca se mueven, excepto cuando se capturan. Las negras inician la partida poniendo una de sus piedras en cualquier intersección vacía del tablero, el turno pasa a las blancas, que pueden pasar o poner una piedra, y así sucesivamente. No se permite poner una piedra en una intersección del tablero que ya está ocupada. El juego termina cuando los dos jugadores pasan consecutivamente.

Hay diversas formas de puntuar, según el país en el que se juega. Nosotros solo explicamos la manera en la que cuenta nuestra aplicación, correspondiente a la puntuación japonesa. Al acabar la partida, los puntos de un jugador son la suma de las piedras en el tablero, las piedras capturadas al jugador contrario y las intersecciones pertenecientes a su territorio. Gana el jugador que más puntos tenga.

Si un jugador tapa la última libertad (véase sección 1.3) de una de las piedras o de una cadena (sección 1.3) del contrario, retira estas piedras del tablero y guarda la piedra capturada o piedras ("prisioneros") hasta el final del juego.

No está permitido quitar la última libertad de una cadena o piedra propia, a menos que este movimiento provoque la captura de algunas de las piedras del adversario, lo que le daría a la piedra que jugó por lo menos una libertad nuevamente. Por ejemplo, en la figura 1.1, si las negras intentan jugar en la posición marcada por el aspa, en principio resultaría ser un suicidio de la cadena negra `B` y estaría prohibido. Sin embargo, al colocar en esa posición una piedra negra, resulta la eliminación de la cadena blanca `A`, por lo que la jugada es legal, y la cadena negra

`B` termina teniendo más libertades que antes de la jugada.

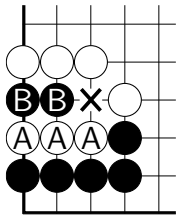

Figura 1.1: Reglas del Go: Suicidio

En Go está prohibido repetir la configuración del tablero, esta regla se llama regla del Ko, y solo es aplicable en ocasiones especiales, semejantes a la figura 1.2. En esta figura mostramos una secuencia de movimientos que queda prohibida por el Ko; en la situación en la izquierda, las blancas juegan en la posición señalada, lo que nos lleva a la situación central. Si entonces las negras jugasen en la situación marcada, volverían a eliminar a la piedra blanca recién puesta, y terminaríamos en la configuración inicial (posición derecha).

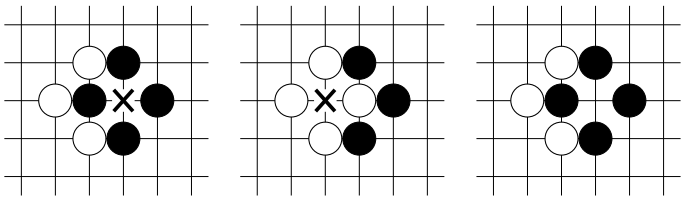

Figura 1.2: Reglas del Go: Ko

1.3. Términos importantes en Go

Hay una serie de términos relacionados con el juego del Go que es necesario conocer para entender adecuadamente esta memoria:

Cadena Dos o más piedras conectadas entre sí en vertical o en horizontal se llaman una cadena; una piedra sola también forma una cadena. En la figura 1.3 las piedras rotuladas con `A` forman una cadena, mientras que la rotulada con `B` está separada de ellas y es una cadena por sí misma.

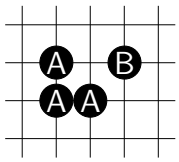

Figura 1.3: Definición de cadena

Libertades Dada una intersección en la que se encuentre situada una piedra, se llaman libertadas de esa piedra al conjunto de las intersecciones contiguas (en vertical y horizontal) que están libres. Dada una cadena, sus libertades son la unión de las libertades de las piedras que forman la cadena. Por ejemplo, en la figura 1.4 la cadena `A` tiene 2 libertades, `B` tiene 3 libertades, `C` tiene 7 libertades y `D` tiene 3 libertades.

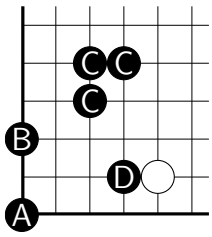

Figura 1.4: Definición de libertades

Cadena muerta Se dice que una cadena está muerta cuando, aunque no se haya eliminado aún del tablero, cualquier combinación de jugadas lleva a su eliminación.

Cadena viva Se dice que una cadena está viva cuando aún no se sabe si está muerta.

Ojos Cuando una cadena consigue una configuración tal que es imposible capturarla, ya que para ocupar al menos dos de sus libertades las piedras de color contrario deberían ponerse en situación de "suicidio" (sección 1.2), se dice que la cadena tiene **ojos**. En la figura 1.5 la cadena negra que aparece tiene ojos, puesto que las intersecciones marcadas con un aspa es imposible que sean ocupadas.

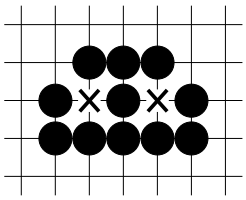

Figura 1.5: Cadena con ojos

Vida y muerte en Go Conocido como Tsume Go, se refiere al problema en Go para reconocer, de forma automática, las cadenas vivas y muertas. Se considera que implementar una aplicación que resuelva este problema es tan complejo como resolver el propio juego [CRS04, Mül02].

Territorio El territorio de un jugador son aquellas intersecciones vacías del tablero las cuales, en pocas jugadas y pese a la intervención del contrario, pueden pasar a tener piedras de ese jugador. Este problema está directamente relacionado con la vida y muerte en Go, puesto que para determinar territorios es necesario decidir qué cadenas están muertas, y por tanto no deben influir en el cálculo de territorios.

Atari Se dice que una cadena está en atari cuando solo tiene una libertad.

Capítulo 2

Estructuras de datos

Para diseñar las estructuras de datos necesarias para implementar eficientemente nuestra aplicación, primero estudiamos nuestros objetivos y nuestros recursos. El objetivo era hacer un juego de Go que trabajase en varios ordenadores simultáneamente, cada uno de ellos trabajando en una zona concreta del tablero (independiente del resto), para poder aumentar la profundidad del árbol de búsqueda y conseguir mejores jugadas. Para ello eran necesarias estructuras de datos que se actualizasen rápidamente, y además un algoritmo de división en zonas igualmente rápido. Es decir, el recurso que queríamos optimizar era el tiempo, por lo que decidimos que podíamos utilizar más memoria, pues cada ordenador debía trabajar solo sobre una zona y un consumo adicional en estructuras de datos no tendría repercusión negativa en el cómputo del árbol de juego.

2.1. El tablero

El tablero es la estructura de datos principal del programa. Nuestro objetivo era implementar unas estructuras de datos que facilitasen en lo posible el cálculo de la división en zonas pero que, a su vez, se actualizasen rápidamente. Para conseguirlo, es necesario utilizar memoria adicional, que usamos para mantener información sobre los conjuntos de piedras que se influyen mutuamente (en lugar de guardar únicamente información individual de cada piedra) y sobre las posiciones vacías en las que una cadena podría estar en el futuro próximo.

2.1.1. Información básica

Hay cierta información sobre la partida que es imprescindible guardar para poder llevar el desarrollo de la misma.

Esta información que debemos guardar es qué ocurre en cada intersección del tablero. Para ello simplemente utilizamos un array bidimensional de enteros, en el que indicamos si la intersección está vacía, con una piedra blanca o con una negra. Si solo contásemos con esta información es posible calcular todas las relaciones entre piedras, pero tendríamos que hacer búsquedas muy costosas, en algunos casos supondría incluso el recorrido de todo el tablero.

2.1.2. Información adicional

Cualquier información que guardemos sobre el tablero que no sea la descrita anteriormente es información adicional, y sirve para facilitar los cálculos y, en nuestro caso, optimizarlos.

La estructura más pequeña considerada por un jugador humano de Go es una cadena. A partir de las cadenas, de sus libertades y la interacción entre distintas cadenas una persona es capaz de distinguir qué movimiento creo que es el mejor en cada momento, y jugar en consecuencia.

Por ello, decidimos tener una clase `Tablero`, que además de tener simplemente como atributo la matriz de posiciones (sección 2.1.1) tuviese también listas de cadenas, siendo cada una de ellas una instancia de la clase `Cadena`.

Cada una de estas cadenas guarda la información sobre su color, las intersecciones en las que se encuentra, las casillas en las que tiene libertades y la zona a la que pertenece. La clase `Cadena` tiene métodos que permiten, de manera eficiente, ver cuántas libertades le quedan a una cadena, buscar cadenas cercanas (aquellas con las que comparte libertades) o ver cuáles son las posibles jugadas que puede hacer la inteligencia artificial. Con solo la implementación básica de cadena, estas operaciones suponen costes en tiempo muy altos, pues supone, para cada piedra encontrada de la cadena, comprobar todas las intersecciones a su alrededor.

Es importante asegurar que el resto de operaciones, como son añadir una ficha o fusionar dos cadenas, siguen siendo eficientes, pues en caso contrario la información adicional que estamos añadiendo no tendría utilidad. En efecto, poner una piedra supone añadir esta piedra a la cadena; fusionar dos cadenas supone borrar una y

hacer referencia a sus datos desde la otra.

Dada una cadena, es fácil ver qué intersecciones están ocupadas por ella, pero el caso contrario, es decir, dada una intersección, ver a qué cadena pertenece, suele ser bastante más costoso (supone buscar cadena por cadena esa intersección hasta encontrarla). Por ello, en el tablero tenemos un array bidimensional en el que guardamos referencias a las cadenas que se encuentran en cada intersección.

También almacenamos como información adicional el número de fichas de cada color que hay sobre el tablero, que es uno de los factores que determinan la puntuación.

Por último, guardamos las estructuras de datos necesarias para hacer la división en zonas localmente en cada jugada (como veremos en la sección 2.2).

2.1.3. Comprobación de restricciones

La operación que más veces se realiza sobre el tablero es, dada una posición, comprobar si es legal poner en ella una piedra de cierto color. Las reglas de colocación de piedras del Go no son complicadas, y una persona que las conozca no tiene problemas para saber si se cumplen. Sin embargo, al formalizarlas, una mala elección de las estructuras de datos puede relentizar mucho su comprobación. Recordemos que la regla básica indica que se puede colocar una piedra si al hacerlo la cadena a la que pertenece continúa teniendo alguna intersección libre alrededor (en caso contrario, se dice que la cadena trata de "suicidarse"). Sin embargo, el "suicidio" se permite si al poner la piedra se elimina alguna cadena de distinto color. Por ello, al poner una piedra hay que saber qué cadenas son adyacentes; como hemos visto, utilizando el array de referencias a las cadenas tenemos inmediatamente esta información. Para las cadenas de diferente color, hay que saber si alguna se queda sin libertades, es decir, comprobar si la lista de libertades de esa cadena tiene una sola posición (si hay solo una, necesariamente es en la que ponemos la piedra). De ser así, la posición es válida; en caso contrario, comprobamos si hay cadenas adyacentes del color de la piedra y si las libertades que tienen, junto a las que añade la nueva piedra, son al menos una. De esta manera, con unas pocas comprobaciones (todas en tiempo constante) sabemos si una posición es legal. Esta eficiencia es muy importante a la hora de desarrollar el algoritmo de búsqueda.

Merece verse por separado la regla del ko, que indica que cuando una piedra de

un jugador sea capturada por el contrario, y al mismo tiempo pueda ser recuperada en el siguiente turno, capturando a su vez la pieza que produjo la captura sin cambiar la formación (lo que daría lugar a una serie de capturas mutuas), el jugador que ha perdido la piedra debe esperar al menos un turno para recuperar la ficha. Esta regla supone en implementaciones sencillas recordar el estado del tablero hace una y dos jugadas. El gasto en memoria, usando esta aproximación, resultaba prohibitivo al desarrollar el árbol de juego y en muchos juegos de Go se considera una jugada poco probable y no se tiene en cuenta a la hora de comprobar las restricciones. En nuestra implementación, solo es necesario guardar dos tuplas diciendo en qué coordenada se colocó la piedra que comió una cierta cadena, y la cadena que fue comida, y solo en el caso en el que la piedra no se fusiona con ninguna cadena del mismo color adyacente y la cadena comida solo tiene una piedra. De esta manera, comprobando en el siguiente turno si el contrario trata de poner la piedra en el lugar que ocupaba la anterior, y esta posición supone comerse la piedra indicada en el par (y solo esa), entonces no está permitida esta jugada. Así, tanto el coste en memoria (guardar en cada turno dos referencias a coordenadas, que sustituyen a las dos anteriores) como en tiempo (comprobar si se coloca la piedra en una posición guardada, y si solo mata a la piedra guardada en la otra posición) es mínimo.

2.1.4. Patrones de diseño

Durante el diseño de las estructuras de datos, vimos que se creaban muchas instancias de la clase `Casilla`, pues eran necesarias para actualizar el tablero, calcular libertades, buscar en el espacio de soluciones, etc. Muchas de estas instancias eran iguales (una casilla solo tiene información sobre su fila y columna) pero sin embargo eran objetos diferentes que consumían memoria. Por ello, decidimos unir dos patrones de diseño [GH04], creamos una factoría abstracta de objetos "singleton". Es decir, todas las clases que necesiten crear en algún momento una casilla, tendrán el mismo objeto de tipo factoría, que será la que reciba las peticiones de nuevas casillas. Si se pide una casilla que ya se había creado en algún momento, devolverá una referencia a la misma, mientras que si se pide una casilla que no se había creado hasta ahora, la creará y devolverá referencias a ella en el futuro.

2.2. Zonas

Es interesante para nuestra aplicación ver que las cadenas del tablero pueden agruparse en zonas. Una zona es una conjunto de intersecciones del tablero tal que, dada una piedra que se encuentre en alguna de sus intersecciones, la cadena completa, todas las cadenas relacionadas con ella por algún patrón (que veremos más adelante) y sus libertades también pertenecen.

Esta división no se realiza de manera fija, es decir, no podemos considerar que el tablero siempre va a tener un número fijo de zonas, dividir las intersecciones del tablero entre ellas y asignar a cada zona unas posiciones concretas. Las zonas son dinámicas, y es posible que cambien con cada jugada. Para decidir la división existen una serie de patrones, que podemos clasificar como:

Patrones cadena Indican cuándo varias piedras forman una cadena. Estos patrones no es necesario calcularlos, puesto que las estructuras de datos las guardan explícitamente. La figura 2.1 muestra todos los patrones de formación de cadenas de piedras blancas; el caso de las cadenas negras es análogo.

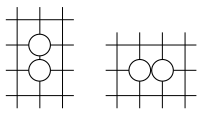

Figura 2.1: Patrones cadena

Patrones de unión Indican cuándo dos cadenas del mismo color están en la misma zona. La figura 2.2 indica todos los patrones de unión entre cadenas blancas, salvo simetrías; el caso de las cadenas negras es análogo.

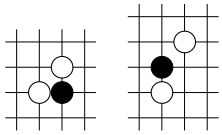

Figura 2.2: Patrones de unión

Patrones de separación Indican cómo una o más cadenas de un mismo color pueden hacer que no se aplique un patrón de unión. La figura 2.3 muestra todos los patrones de separación entre cuatro cadenas, mientras que la figura 2.4 muestra el patrón de separación de cadenas blancas; en el caso de ser las piedras negras el patrón es análogo.

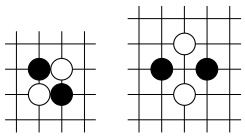

Figura 2.3: Patrones de separación 1

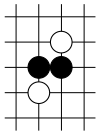

Figura 2.4: Patron de separación 2

Patrones de contacto Indican influencia entre cadenas de distinto color. Un patrón de contacto no provoca, en principio, la unión entre las zonas de las dos cadenas; para que esto ocurra es necesario que una zona "esté rodeada" por cadenas de distinto color con las que tiene contacto. En este caso, la zona central sí se uniría a las zonas en contacto. Las zonas entre las cuales se dan patrones de contacto pero no unión se llaman zonas adyacentes. La figura 2.5 muestra los distintos patrones de contacto.

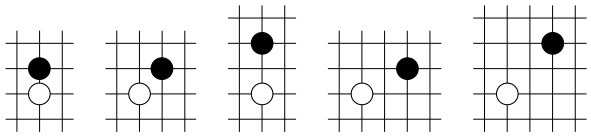

Figura 2.5: Patrones de contacto

Por tanto, para calcular las zonas debemos en primer lugar comprobar si se da un patrón de separación. Si se da, solo podemos afirmar que las cadenas están

separadas en ese punto, puesto que puede que estén unidas en otro lugar del tablero. Si no están separadas, comprobamos que estén realmente unidas. En tercer lugar comprobaríamos los contactos. En otras implementaciones [CRS04], la división en zonas se hace cada vez que se hace una jugada, por lo que se comprueban los patrones en todo el tablero cada vez que el juego debe decidir una jugada. Esta forma de actuar es muy costosa en tiempo, aunque consume menos memoria (solo necesita estructuras de datos simples para indicar qué cadenas pertenecen a cada zona) y es mucho más sencilla de implementar. Sin embargo, en nuestra aproximación decidimos que el tiempo para hacer las divisiones consumiría gran parte del tiempo reservado al cómputo del árbol de juego, por lo que era preferible hacer algoritmos más elaborados y hacer que las zonas se actualizasen localmente al hacer la jugada, es decir, que al poner una piedra solo sea necesario actualizar la zona en la que se pone la piedra, y quizás alguna adyacente.

Para poder actualizar las zonas, necesitamos que cada una de ellas guarde, durante todo el juego, información sobre las zonas en contacto por cada lado, las cadenas que tiene y las posiciones en las que puede jugarse al desarrollar el árbol de juego. Esta forma de actualización hace que surjan muchos problemas que no aparecen en la división estática:

1. Es posible que la colocación de una piedra elimine una cadena. La cadena eliminada podía tener uniones con cadenas que, sin ella, ya no pertenezcan a la misma zona, por lo que es necesario comprobar estas situaciones, y hacerlo de manera eficiente.

2. Es posible que la colocación de una piedra produzca un patrón de separación que separe dos cadenas que antes estuviesen unidas. En este caso no basta con quitar una de las cadenas de la zona, sino que hay que comprobar si las cadenas están unidas por alguna otra parte del tablero, y en el caso de separarlas, comprobar qué cadenas estaban unidas a cada una de las partes.

3. Cuando tenemos que unir una zona con sus contactos, debemos hacer un recorrido por ellos para irlos insertando en la nueva zona. A la vez que recorremos estos, debemos unir sus contactos a los de la nueva zona. Hacer esto simultáneamente puede producir errores de coherencia, ya que estaríamos insertando en la misma estructura de datos que leemos.

Para solucionar los dos primeros problemas, hemos implementado métodos que

dividen una zona en nuevas zonas representadas por una sola cadena, y después unen estas nuevas zonas entre ellas, acotando el coste de las operaciones al número de cadenas de la zona original. En el tercer caso, hemos decidido utilizar memoria adicional para ir acumulando las zonas en contacto que debemos añadir a la nueva zona, y una vez hemos recorrido la lista original de zonas, hacemos referencia al nuevo conjunto de zonas.

La operación que con más frecuencia se puede presentar entre zonas es la unión entre varias, por lo que este es un método que debemos asegurarnos que funciona eficientemente. Al fusionar dos zonas no se crea una nueva, sino que una de ellas desaparece mientras la otra queda como contenedora de ambas. Así, la operación de fusión consiste principalmente en añadir las referencias de una de las zonas a la otra, y actualizar los objetos que pudiesen tener referencias a la zona que desaparece por referencias a la nueva. Para ello es necesario actualizar todas las cadenas y las zonas vecinas. Otra operación frecuente es la división de una zona, esta operación tiene un coste lineal en el número de cadenas de la zona. Como última operación importante tenemos la reestructuración de zonas después de una división. Esta operación en principio es lineal en el número de piedras, sin embargo, en la práctica, tras analizar las primeras cadenas se han formado ya todas las zonas posibles (puesto que provienen de una zona común, es raro que las cadenas resultantes de la división se agrupen en más de tres o cuatro zonas) y la comprobación del resto de piedras es simplemente un recorrido por las cadenas sin realizar cómputo, es decir, pasa a ser lineal en el número de cadenas.

Puesto que hablamos de coste en el número de piedras, cadenas y zonas, es conveniente indicar qué cantidad de cada una de estas estructuras podemos encontrarnos en una zona. En un momento avanzado del juego, lo normal es que haya aproximadamente nueve zonas, por lo que el número de zonas vecinas a otra no puede ser superior a este número; el número de zonas es más alto al principio, ya que hay muchas cadenas de pocas piedras, pero al avanzar la partida no es común que haya más de siete u ocho cadenas de cada color en una misma zona. El número de piedras es muy variable, y puede depender mucho del estilo de juego y de los intereses de los jugadores en una zona concreta, pero no deberían superar las 50 o 60 piedras. Estos costes, que son los que tenemos para el peor de los casos (en el que ocurre una división o una unión), son muy inferiores al coste de calcular estáticamente las zonas, que requieren como mínimo recorrer una vez el

tablero (alrededor de 400 intersecciones). Además debemos tener en cuenta que es necesario realizar este cálculo cada vez que hacemos un movimiento en el árbol de búsqueda, por lo que la ganancia en tiempo es más importante.

2.2.1. Relación entre las zonas y el tablero

Para mejorar el tiempo necesario para calcular relaciones entre las cadenas y las zonas, muchos objetos tienen referencias cruzadas, por ejemplo, una zona sabe qué cadenas tiene, y a su vez una cadena sabe a qué zona pertenece: cada vez que fusionamos o eliminamos una cadena o una zona debemos actualizar convenientemente sus referencias. Para ello, cada una de estas estructuras tiene métodos específicos con este fin.

Es importante ver el ahorro de tiempo de cómputo que supone para el cálculo de zonas tener ya calculadas las relaciones de cadenas en el tablero. Este conocimiento nos permite no solo ahorrarnos uniones de zonas, sino que evita comprobación de patrones entre piedras de la misma cadena (puesto que ya sabemos que pertenecen a la misma zona).

2.2.2. Relación entre las zonas y la inteligencia artificial

Este ahorro en tiempo nos permite hacer divisiones en zonas en los nodos del árbol de búsqueda (sección 3.2), permitiendo una búsqueda más exacta, ya que el desarrollo del árbol puede hacer que ciertas zonas se separen o unan; no tener en cuenta estos cambios podría llevarnos a estudiar situaciones que nunca llegarán a darse y dejar otras que sí sucederán.

La eficiencia en la división en zonas también permite que sean los clientes los que calculen en cada jugada la división en zonas (como veremos en la sección 7.1), evitando que sea el servidor quien calcule las zonas en cada jugada, lo que podría suponer la sobrecarga de la red.

2.3. Árboles PATRICIA

En nuestra implementación, usamos árboles PATRICIA (Practical Algorithm To Retrieve Information Coded In Alphanumeric) [Gon88, Gon91] para guardar los patrones (capítulo 5).

En general, un árbol PATRICIA es un árbol digital (también conocido como trie, por re**trie**val). Cada nodo de estos árboles está asociado a una clave, para llegar hasta él descomponemos la clave en caracteres, y los usamos como índices.

Suponiendo que los caracteres de las claves pertenecen a un alfabeto A (por tanto las claves pertenecen a A^*), cuyo cardinal sea M, las características de un trie [Fre60, Kol05] son:

1. Árbol M-ario.
2. Cada nodo del árbol es un vector de M componentes.
3. Cada componente se corresponde con un carácter del alfabeto.
4. La raíz es un nodo especial que no contiene valores.

En la figura 2.6 tenemos un árbol PATRICIA que representa claves del alfabeto $\{A, B, C\}$. El árbol contiene dos valores; asociado a la cadena "C" tenemos el 5, y a la cadena "BCAC" el 7. Podemos ver en este pequeño ejemplo que para contener solo dos valores son necesarios seis nodos, lo que nos lleva a pensar si esta representación es realmente eficiente, pues muchos de los nodos quedan vacíos, excepto por una de las casillas de sus arrays; más adelante veremos que hemos optimizado estos árboles para solucionar este problema.

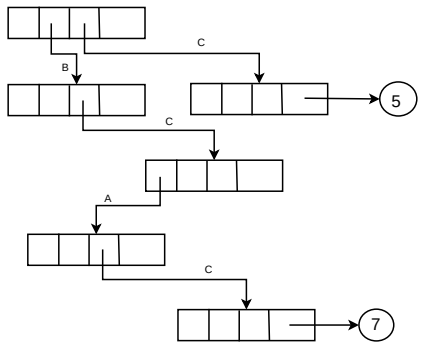

Figura 2.6: Árbol PATRICIA

Por último, explicamos brevemente las operaciones de inserción y búsqueda en árboles PATRICIA.

Inserción Para insertar un valor en un árbol PATRICIA vamos recorriendo el árbol siguiendo el camino indicado por la clave. Cuando el camino indicado

por la clave es un nodo, pasamos a él; cuando es un valor que aún está a `null`, creamos un nuevo nodo y entonces pasamos a él. Al acabar la clave, insertamos el valor indicado en el nodo en el que nos encontramos.

Búsqueda Buscamos en el árbol siguiendo el orden indicado en la clave. Si los nodos se acaban antes de llegar al final de la clave, el valor buscado no está; en caso contrario, si nos encontramos en un nodo al acabar de recorrer la clave, comprobamos si ese nodo tiene un valor asociado y, en tal caso, lo devolvemos; en caso contrario, la clave no está.

2.3.1. Optimizaciones

El problema que presentan estos árboles es que si tenemos pocos valores con claves largas, es muy posible que tengamos largas ramas en las que todos los nodos tienen un solo hijo, y por tanto consumimos memoria para mantener arrays con un solo valor, y además las búsquedas son más lentas.

Para mejorar esta situación, podemos hacer que los nodos con un solo hijo lo tengan directamente como valor en su array, en lugar de tener la rama completa hasta él, aunque para poder reconocerlo es necesario que ahora el nodo guarde también información sobre su clave (lo que no es, en modo alguno, comparable con la memoria necesaria para crear todos los nodos hasta él). Como se ve en la figura 2.7, el árbol que guarda los valores asociados a las claves “C” y “BCAC” es ahora mucho más pequeño (tres nodos frente a los seis que tenía el árbol sin optimizar que guardaba los mismo valores en la figura 2.6). También vemos que ahora los valores no son solo un número, sino que hemos añadido la clave para identificarlos, lo que podrá ser necesario, por ejemplo, en el nodo identificado por “BCAC”, ya que si no lo marcamos, representa el valor de cualquier clave que comience por “bc”.

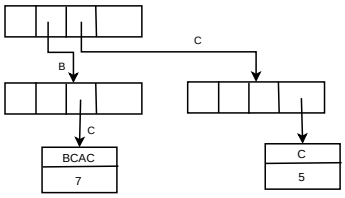

Figura 2.7: Árbol PATRICIA optimizado

Con esta solución conseguimos hacer las búsquedas más eficientes y ahorrar memoria, pero complicamos los algoritmos, ya que ahora es necesario comprobar en cada hijo de un nodo si es un valor o es otro nodo, para no intentar expandir un valor en lugar de un nodo. Las nuevas operaciones de inserción y búsqueda quedan de esta manera:

Inserción Para insertar un valor en esta nueva implementación, seguimos los caracteres de la clave hasta llegar a un valor `null` o al último nodo indicado por la clave, e insertamos (nótese que en el segundo caso insertamos un valor en un nodo, mientras que en el primer caso colocamos un valor como hijo de un nodo). El problema se presenta si al recorrer la clave llegamos a un valor, en lugar de un nodo. Si esto ocurre, creamos un nuevo nodo, que colocamos en lugar del valor encontrado, hacemos una inserción del valor encontrado, comenzando en el nuevo nodo, y después continuamos la inserción. En la figura 2.8 vemos el resultado de la inserción del valor 2 asociado a la clave "BCBC"; cuando llegamos buscando el nodo correspondiente al primer carácter "c" de la clave, nos encontramos con que es un valor, por lo que creamos un nuevo nodo, que ponemos en su lugar, hacemos una inserción de ese valor desde el nodo creado, y después insertamos nuestro valor.

Búsqueda Buscamos en el árbol siguiendo el orden indicado por la clave. Si llegamos a `null`, es porque el valor no está. Si llegamos a un valor o al último nodo indicado por la clave, comprobamos si el valor encontrado corresponde con la clave buscada, en cuyo caso devolvemos ese valor; en caso contrario, el valor no está.

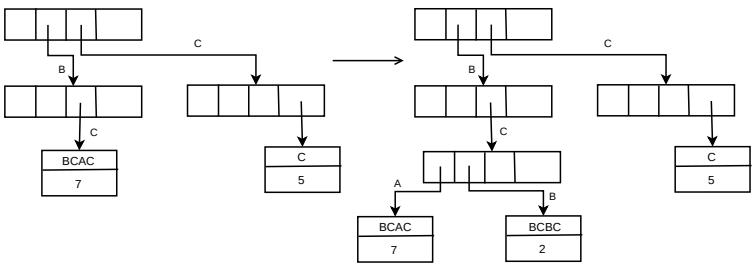

Figura 2.8: Inserción en árboles PATRICIA optimizados

2.3.2. Funciones adicionales

Como veremos en el capítulo 5, hay ciertas intersecciones en los patrones que toman el valor "no importa", es decir, que es válido tener una piedra blanca, negra o una intersección vacía. Por tanto, cuando estemos recorriendo un patrón y en la clave debamos tratar un carácter de este tipo no debemos solo pasar al nodo etiquetado con ese carácter, sino hacer también la búsqueda para los caracteres que representen a las piedras blancas y a las negras. Por tanto, es necesaria una búsqueda que busque en todas las ramas y se quede el máximo (el valor con la clave más larga).

También debido a los patrones, las búsquedas no deben ser exactas, sino que deben buscar un patrón hasta la máxima profundidad posible, y devolver el patrón encontrado, aunque no sea exactamente el deseado.

Estas posibilidades hacen que sea necesario un nuevo tipo de búsqueda: si estamos en un cierto nodo y estamos tratando el índice I, I = blanco o I = negro, entonces buscamos el mayor patrón en la rama I y en la rama "no importa"; una vez obtenidos ambos, nos quedamos con el máximo.

Además, hemos hecho que los nodos seleccionados en esta nueva búsqueda cumplan ciertos requisitos, como veremos en el capítulo 5. Si no tuviésemos esto en cuenta, podríamos devolver un patrón para un color que no juega en este momento, o una intersección fuera del tablero. Para ello, cuando en la búsqueda llegamos a un valor, le hacemos pasar un "test" en el que comprobamos si el color es el adecuado y si la intersección en este caso concreto es válida. Si es válido, entonces lo devolveremos, en caso contrario, consideramos que no hemos encontrado nada.

También debemos tener en cuenta que los patrones son simétricos con respecto al color (sección 6.2), por lo que debemos buscar el mejor patrón que se adapte al tablero (es decir, que pase el "test" mencionado antes), después buscar el mejor patrón simétrico, y devolver el mejor de ambos.

De esta manera es posible hacer la búsqueda para localizar patrones de tamaño máximo que encajen con una configuración dada del tablero.

Capítulo 3

Inteligencia artificial

En este capítulo presentamos cómo hemos dotado de inteligencia artificial a nuestro juego. Actualmente, hay dos tendencias a la hora de implementar este módulo en los juegos de Go.

Función de evaluación compleja Las aplicaciones que eligen este sistema emplean la mayor parte del tiempo de cómputo en calcular la función de evaluación, lo que les impide explorar un espacio de estados grande.

Búsqueda exhaustiva Estas aplicaciones realizan búsquedas en un espacio de estados muy grande, y tratan de decidir la mejor jugada llegando a configuraciones finales.

En nuestro programa hemos decidido estudiar las dos posibilidades por separado; por un lado usamos un algoritmo que toma posiciones al azar, las evalúa de forma compleja y se queda la mejor (algoritmo **Monte Carlo**), y por otro lado exploramos cuanto podemos el espacio de estados y evaluamos solo los estados finales a los que llegamos, usando la **Búsqueda Proof-Number**.

3.1. Algoritmo Monte Carlo

Como primera aproximación a la inteligencia artificial, decidimos usar un algoritmo que eligiese aleatoriamente una jugada en cada una de las zonas, comprobase si era una jugada legal y, en tal caso, le asignase un valor y la devolviese como la jugada seleccionada para esta zona. Una variante más desarrollada de este algoritmo, consistente en elegir aleatoriamente varias posiciones, desarrollar un árbol de

juego para cada una y quedarse con la más prometedora, ha sido implementada [Bou, Bou03a, Bou04] y utilizada para jugar al Go, y ha obtenido buenos resultados en tableros pequeños (9×9), pero no ha funcionado en tableros mayores.

Como en este método de juego la selección en cada zona es muy poco costosa (simplemente se elige al azar una posición por zona), decidimos centrar nuestros esfuerzos en la función de evaluación (sección 3.4), para que la zona en la que finalmente decidamos jugar sea la mejor opción posible.

Nuestra aproximación resultó funcionar también en tableros pequeños (9×9) y con jugadores principiantes, aunque pensamos que no es una solución adecuada para tableros de mayor tamaño, por lo que no quisimos continuar con esta línea de trabajo.

3.2. Búsqueda Proof-Number

Para implementar la inteligencia artificial, decidimos utilizar un árbol de búsqueda [BPP94]. La búsqueda más común es un árbol mini-max [Bou01, CRS04], sin embargo, siguiendo nuestra filosofía de aprovechar la memoria para mejorar los resultados en tiempo, decidimos utilizar un nuevo tipo de búsqeda, y al final optamos por la búsqueda Proof-Number [All94, Mül01b, Mül02] (también conocida como búsqueda PN). Este tipo de búsqueda comprueba un espacio de estados menor que el mini-max con poda alfa-beta, pero tiene como inconveniente que tiene un coste en espacio lineal en el número de nodos, mientras que el mini-max es lineal en la altura del árbol. Como este tipo de búsqueda no suele ser conocido, daremos una explicación del algoritmo en primer lugar, y después pasaremos a explicar nuestro caso en concreto.

3.2.1. Algoritmo PN

La búsqueda PN es un algoritmo de la familia "primero el mejor" [Pla96]. Los algoritmos de esta familia seleccionan, dentro de los nodos no terminales, el más prometedor según un cierto criterio, lo expanden y entonces actualizan, si es necesario, la información del grafo para continuar el proceso. El factor que distingue los distintos algoritmos de esta familia es la selección del "mejor nodo".

El árbol de juego utilizado es un árbol AND/OR [BPP94]. Los árboles de este tipo tienen las siguientes características:

1. Tienen dos tipos de nodos, los nodos OR y los nodos AND.

2. Cada nodo del árbol puede ser evaluado, y su valor puede ser **true**, **false** o **unknown** (no debe confundirse un nodo evaluado a un valor **unknown** con un nodo sin evaluar).

3. Los nodos evaluados a **unknown** pueden expandirse, lo que significa que pasan a tener una lista no vacía de hijos con el nodo expandido como padre de cada uno de estos nuevos nodos.

4. Los nodos expandidos se llaman *nodos internos*. Los nodos sin expandir son nodos *hoja*.

5. Hay tres tipos de nodos hoja: los nodos con valor **true** o **false**, un nodo evaluado a **unknown** sin hijos y un nodo sin evaluar. A los dos últimos tipos de nodo también se les llama *nodos frontera*.

6. Hay dos procedimientos de creación de árboles:

   - La evaluación inmediata, en la cual los nodos se evalúan en el momento de su creación.
   - La evaluación "pospuesta", en la cual la evaluación de los nodos se retrasa hasta que el nodo es elegido.
   - El objetivo de este tipo de árboles es probar un cierto predicado, diferente para cada aplicación, para la raíz. Para ello, puede tratar de probar sus hijos y propagar sus resultados.

7. El valor de un nodo AND interno A se determina de la siguiente manera: si al menos uno de sus hijos tiene el valor **false**, entonces A tiene el valor **false**; en caso contrario, si alguno de sus hijos tiene valor **unknown**, entonces A tiene valor **unknown**; en caso contrario, A tiene valor **true**.

8. El valor de un nodo OR interno O se determina de la siguiente manera: si al menos uno de sus hijos tiene el valor **true**, entonces O tiene el valor **true**; en caso contrario, si alguno de sus hijos tiene valor **unknown**, entonces O tiene valor **unknown**; en caso contrario, O tiene valor **false**.

9. Un árbol está resuelto si su raíz tiene el valor **false** o **true**.

10. Si la raíz de un árbol resuelto tiene valor **true**, entonces el árbol *está probado*, en caso contrario, el árbol *está refutado*.

El funcionamiento de la búsqueda PN se basa en las siguientes definiciones:

1. Para cualquier árbol AND/OR T, un conjunto de nodos frontera S se dice que es un *conjunto de prueba* si todos los nodos de S prueban T.

2. Para cualquier árbol AND/OR T, un conjunto de nodos frontera S se dice que es un *conjunto de refutación* si todos los nodos de S refutan T.

3. Para cualquier árbol AND/OR T, el *proof number* de T se define como la cardinalidad del menor conjunto de prueba de T.

4. Para cualquier árbol AND/OR T, el *disproof number* de T se define como la cardinalidad del menor conjunto de refutación de T.

5. Para cualquier árbol AND/OR T, el nodo más prometedor de T es un nodo frontera de T tal que, si obtiene el valor **true** reduce el proof number de T en 1, mientras que si obtiene el valor **true** reduce el disproof number de T en 1.

6. El proof number y el disproof number de un nodo frontera es 1.

7. El proof number de un nodo evaluado a **true** es 0 y de un nodo evaluado a **false** es ∞.

8. El disproof number de un nodo evaluado a **true** es ∞ y de un nodo evaluado a **false** es 0.

Ahora, podemos considerar que el proof number de un nodo interno AND es la suma de los de sus hijos (puesto que es necesario probarlos todos), mientras que en un nodo OR es el mínimo de los de sus hijos (ya que basta probar uno de ellos). En la figura 3.1 vemos el camino para llegar el nodo más prometedor, que ha consistido en ir eligiendo el menor hijo en los nodos OR (circulares); en los nodo AND (cuadrados) la decisión era más díficil de tomar, ya que es necesario probar todos los hijos, pero en este caso G es el único nodo AND al que llegamos; como solo tiene un hijo sin evaluar, lo elegimos como más prometedor. Sin embargo, es muy posible que nos encontremos con árboles cuyo nodo más prometedor no puede decidirse simplemente con el proof number.

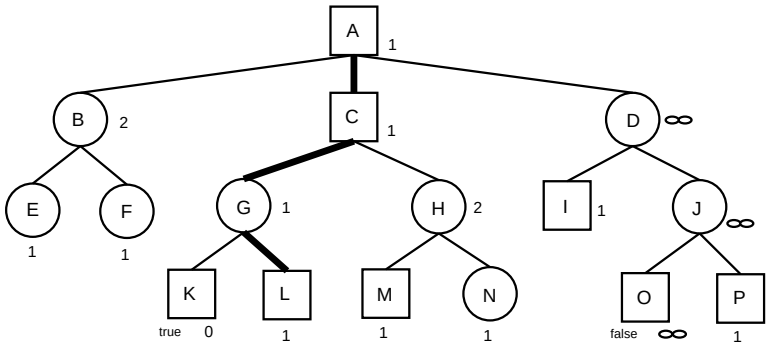

Figura 3.1: Árbol PN: *proof number*

Lo contrario ocurre con el disproof number. En los nodos OR es la suma de los disproof numbers de sus hijos, mientras que en los nodos AND es el mínimo de los disproof numbers de sus hijos. En la figura 3.2 vemos que guiándonos solo por el disproof number tenemos para elegir varios nodos como más prometedores (marcados sus caminos con trazo grueso), y en concreto uno de ellos coincide con el mostrado en la figura 3.1 (marcado el camino con trazo continuo). En este caso vemos claramente que solo con el disproof number no podemos decidir cuál es el nodo más prometedor.

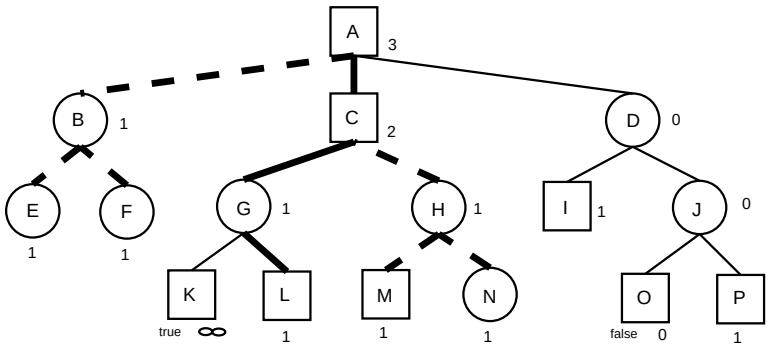

Figura 3.2: Árbol PN: *disproof number*

Utilizando tanto el proof number como el disproof number podemos hacer un recorrido por el árbol que nos lleve a probarlo, eligiendo en los nodos OR el hijo con menor proof number y en los nodos AND el hijo con menor disproof number. En la figura 3.3 se muestre un árbol de búsqueda PN; los nodos representados con círculos son nodos OR, mientras que los representados con cuadrados son nodos

AND. Cada nodo tiene asociada una pareja de enteros: el primero de ellos se refiere a su proof number y el segundo a su disproof number. Con trazo más grueso se marca el camino hasta el nodo más prometedor; en los nodos OR A y E elegimos los hijos con menor proof number, respectivamente B y N; en los nodos AND B y N elegimos los hijos con menor disproof number: E y R. De esta manera llegamos a la conclusión que el nodo más prometedor es R (dado que hay que resolver R y S, elegimos el nodo en orden lexicográfico).

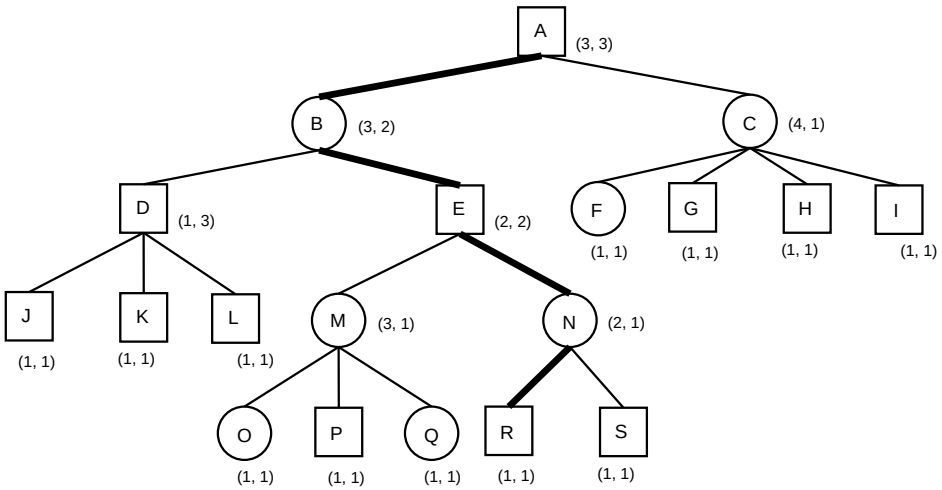

Figura 3.3: Búsqueda del nodo más prometedor

3.2.2. Nuestra implementación

Para poder utilizar la búsqueda PN, decidimos usar las siguientes asignaciones:

1. Representamos los turnos en los que van a jugar las piedras negras como nodos OR, mientras que cuando juegan blancas los representamos con nodos AND.

2. Probar el predicado para las piedras negras consiste en conseguir **true** en la raíz, mientras que probarlo para las blancas es conseguir **false**. Con esto indicamos que probando el árbol completo haríamos ganar a las negras, mientras que refutándolo las blancas podrían como mínimo empatar [All94].

   Por el momento hemos hablado de "probar un predicado". Hemos visto en la sección 3.2 que habitualmente el objetivo que se trata de alcanzar es algo

del estilo "puede ganar"; no así en nuestro caso. Dado que hemos dividido el tablero en zonas (sección 2.2), probar que podemos ganar en una no significa que podamos ganar en el tablero completo, por lo que hemos decidido satisfacer sub-objetivos (sección 3.3). Esta decisión complica el cálculo del árbol de búsqueda, ya que hay que decidir qué objetivo es el más adecuado para cada zona.

Uno de los problemas más serios que se nos han planteado al hacer el árbol de búsqueda ha sido guardar los tableros en cada uno de los nodos. En Go es imposible saber, dada una intersección en la que hemos jugado y una configuración de las piedras, cuál era la configuración anterior, por lo que es necesario guardar el estado del tablero incluso cuando generamos los hijos de un nodo (puesto que al generar un hijo el tablero cambia).

Guardar el tablero no supone solo guardar qué piedras hay en cada intersección, sino guardar todas las zonas asociadas. Para ello hubo que hacer un método que guardase toda la información referente al tablero; este consumo de memoria lo hemos optimizado borrando todos los nodos de los sub-árboles totalmente evaluados.

El algoritmo de búsqueda PN [All94] expande "mientras queden recursos". Nosotros hemos considerado que tenemos recursos mientras no llegue a una cierta profundidad, con lo que limitamos el uso tanto de memoria como de tiempo.

3.3. Objetivos

Como hemos visto en la sección 3.2, el árbol de juego generado por la búsqueda PN trata de hacer cierto un predicado, normalmente algo del estilo *puede_ganar(raíz)*. Sin embargo, este predicado es difícil de decidir en el Go, ya que la victoria depende de muchos factores, y en cada situación concreta son más importantes unos que otros. Más aún, en nuestra implementación hemos dividido el tablero en zonas, y no podemos asegurar que logrando una victoria en una zona concreta se consiga en todo el tablero.

Por ello, decidimos que cada zona debía resolver el objetivo adecuado a la situación actual de sus piedras: al iniciarse la búsqueda PN, decidimos qué objetivo buscamos. Es importante ver que la comprobación del objetivo debe ser

muy eficiente, ya que debe realizarse en cada nodo del árbol de juego, pero sin ser demasiado sencilla, ya que no proporcionaría suficiente información.

Por estas razones, nosotros hemos incluido en nuestra aplicación tres objetivos diferentes para la búsqueda PN:

Ojos Una de las prioridades en Go es dotar a tus cadenas de ojos (sección 1.3), ya que de esta manera las cadenas no pueden ser eliminadas. Cuando ninguna de las cadenas de una cierta zona tiene ojos, tratamos de realizar las jugadas adecuadas para que al menos una tenga.

Cohesión Cuantas menos cadenas de un color hay en una zona, mayor es su cohesión: si en una zona hay varias cadenas, y al menos una de ellas tiene ojos, es más interesante unirlas para crear una sola cadena, que intentar que todas ellas tengan ojos.

Ganar piedras Este objetivo es un objetivo general que utilizamos cuando los demás están alcanzados. Consideramos ganancia de piedras no solo a tener más de nuestro propio color, sino a capturar las contrarias.

Además, en la búsqueda Monte Carlo (sección 3.1) el objetivo que buscamos es ganar, dado que no hay división entre zonas, e intentar alcanzar los objetivos anteriores en una sola jugada sería ciertamente difícil.

3.4. La función de evaluación

Dado que el Go es un juego aún no superado, las aplicaciones comerciales no han publicado sus funciones de evaluación, por lo que decidimos estudiar qué factores era interesante tener en cuenta en la función de evaluación [Bou01, CRS04]. Llegamos a la conclusión que los factores más importantes son:

Piedras ganadas El número de piedras de nuestro color que hemos puesto. Si evaluamos una situación en la que sólo hemos hecho un movimiento, como en el caso del método Monte Carlo (sección 3.1), será solo una piedra (o ninguna, si hemos pasado), pero si estudiamos situaciones más complicadas, puede ser un factor importante.

Piedras eliminadas El número de piedras del color contrario que hemos eliminado. Es importante ver que el número de piedras eliminadas cuenta en la

puntuación final, por lo que no es lo mismo, en el caso que estemos estudiando una situación a la que hemos llegado tras varias jugadas, el número de piedras eliminadas y la diferencia de piedras que tenía el contrario antes de las jugadas o después (por ejemplo, si hacemos cuatro jugadas, en la primera comemos tres piedras, y en el resto ninguna, el número de piedras inicial y final será el mismo, pero nosotros tendremos 3 puntos más). Este hecho hace complicado calcular las piedras eliminadas si calculamos la función de evaluación después de varias jugadas, como en el caso de la búsqueda PN (sección 3.2).

Libertades ganadas Es la cantidad de nuevas libertades que hemos obtenido al hacer las jugadas. A diferencia de las piedras ganadas, que influyen directamente en la puntuación, las libertades no puntúan, pero sirven como indicador de la "vida" de las cadenas.

Libertades destruidas Calcula cuántas libertades ha perdido el jugador contrario.

Territorio ganado Calcula el territorio que se ha ganado con las jugadas. Este aspecto es el más difícil de calcular, puesto que afecta directamente al problema de vida y muerte en Go, pero también es el más importante, por lo que prestamos especial atención a este aspecto (ver sección 4) para la función de evaluación.

Territorio destruido Contabiliza el territorio que el jugador contrario ha perdido a consecuencia de las jugadas realizadas.

Una vez obtenidos estos valores, no basta con hacer su media aritmética, ya que parece que hay ciertos factores más importantes que otros, y que por tanto eran necesarios pesos que ponderasen cada una de las características anteriores.

El problema consiste ahora en asignar los pesos a cada uno de los factores, pues para ello necesitaríamos varios jugadores expertos de Go, que considerasen distintas situaciones y nos diesen un valor. Pero aun contando con la ayuda de expertos, el peso para cada uno de estos factores no parece que se pueda obtener de manera directa, ya que los humanos consideramos toda la situación como un conjunto, y también debemos tener en cuenta la subjetividad con la que cada uno considera una situación. Por ello, decidimos que la función de evaluación se

actualizase automáticamente (sección 6.1) con ejemplos de situaciones ya resueltas de forma óptima.

Además, hemos visto que según la situación la búsqueda puede estar dirigida hacia un objetivo o hacia otro (sección 3.3), por lo que no basta con tener un peso asociado a cada uno de los factores de la función de evaluación. En su lugar, hemos guardado un peso por cada factor y objetivo, para que cada objetivo tenga mayor independencia y se pueda refinar más.

Capítulo 4

Vida y muerte en Go

Como vimos en la sección 1.3, el problema de vida y muerte en Go es decisivo a la hora de crear una aplicación que juegue bien. Más aún, es necesario calcular el territorio controlado por cada jugador al final de la partida para saber su puntuación.

Para solucionar el problema de vida y muerte tenemos que ser capaces de reconocer territorios y cadenas muertas.

4.1. Reconocimiento de territorio

En una partida de Go, un jugador no controla sólo las intersecciones en las que tiene piedras. Es muy posible que controle además otras intersecciones, entendiendo por *controladas* aquellas intersecciones en las cuales el contrario no puede jugar (por ejemplo, porque esas intersecciones son los ojos de una cadena, visto en la sección 1.2), o bien puede jugar, pero esas piedras serán capturadas en pocas jugadas sin oposición posible.

Para reconocer el territorio vamos a aplicar una técnica utilizada inicialmente para reconocimiento de imágenes: la morfología matemática [Bou03b]. Para nuestra aplicación solo nos interesan dos operadores: la erosión y la dilatación.

Dado un conjunto A, definimos la dilatación de A como el conjunto formado por A unido al conjunto de sus vecinos; definimos erosión de A al conjunto formado por A menos los vecinos del complementario de A.

Utilizando una variación de estos dos operadores propuesta por Zobrist [Zob69] podemos calcular la influencia y el territorio de las cadenas en Go. Este modelo

consiste en asignar el valor +64 a las intersecciones del tablero con una piedra negra, −64 a las intersecciones con una piedra blanca, y 0 al resto. Los operadores quedan como sigue:

Dilatación Consiste en sumar al valor absoluto de la intersección el número de intersecciones vacías o del mismo color, suponiendo que los vecinos solo son del mismo color o intersecciones vacías.

Erosión Consiste en restar, al valor absoluto de la intersección, el número de vecinos de distinto color o vacíos. La erosión no debe cambiar el signo de la intersección.

Con esta técnica, basta mirar las intersecciones antes vacías y ahora con un valor para comprobar qué color tiene influencia en ellas. Si el valor es mayor que cero serán las negras; en caso contrario, las blancas.

Mediante dilataciones podemos observar la influencia de las cadenas. En la figura 4.1 podemos ver cómo reconocemos influencia mediante aplicaciones consecutivas del operador dilatación; los cuadrados negros indican influencia de las piedras negras, mientras que los cuadrados blancos indican influencia blanca.

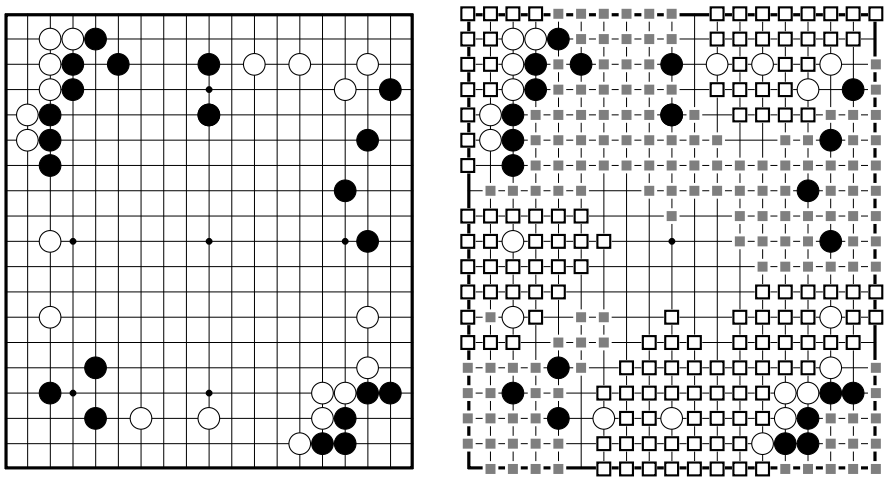

Figura 4.1: Tablero inicial después de 3 dilataciones

Sin embargo, para reconocer territorios es necesario aplicar varias dilataciones y posteriormente varias erosiones. En la figura 4.2 podemos ver cómo reconocemos

territorios dilatando el tablero y posteriormente erosionándolo; los cuadrados negros indican territorio de las piedras negras, mientras que los cuadrados blancos indican territorio blanco.

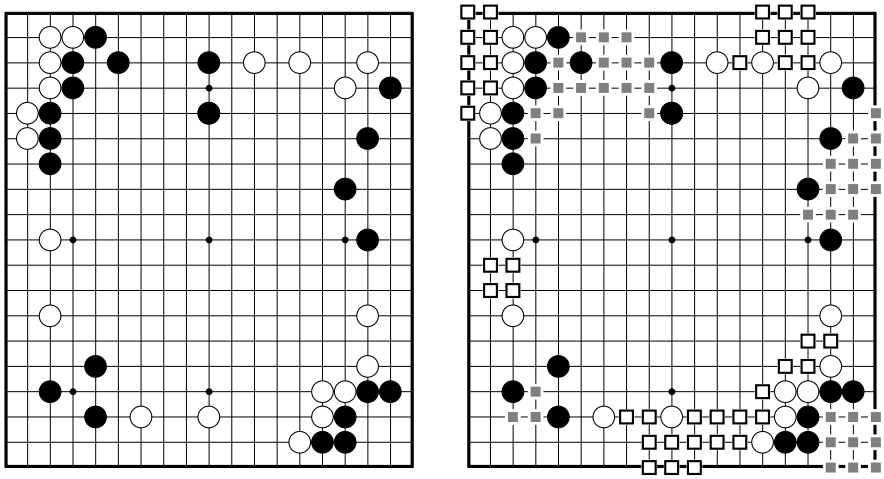

Figura 4.2: Tablero después de 3 dilataciones y 6 erosiones

Empíricamente se ha visto que, llamando e al número de erosiones y d al número de dilataciones, el valor óptimo de erosiones es $e = d(d-1)$ [Bou03b]. Cuanto mayor sea el número de dilataciones, mayor será la precisión del reconocimiento; sin embargo, también será mayor el tiempo necesario para calcularlo.

El cálculo de territorios, como hemos visto, es complejo pero proporciona muy bueno resultados. En la actualidad, que nosotros sepamos, solo el GNUGO [Niu04] tiene esta característica, que le ha llevado a ganar los campeonatos de algoritmos de los últimos años.

Capítulo 5

Patrones

Para mejorar la eficiencia y los resultados de nuestra aplicación decidimos usar patrones [Caz99].

Hay dos tipos de programas que juegan al Go que utilizan patrones [Bou01]: unos se basan en los patrones para decidir casi todas las jugadas, y calculan el resto [Mül01a]; los otros calculan todas las jugadas, pero si encuentran en la base de datos el patrón, lo aplican directamente. El tamaño de las bases de datos en el primer caso es de varios millones de patrones, mientras que el tamaño medio en el otro caso es de alrededor de 3000 patrones [Bou01].

Nosotros decidimos hacer una aplicación del segundo caso, dado que no tenemos los conocimientos de Go suficientes para crear una base de datos lo suficientemente grande como para hacer funcionar adecuadamente un programa basado principalmente en patrón. La sección 6.2 indica cómo insertamos los patrones necesarios para crear una base de datos para nuestra aplicación.

Para representar los patrones codificamos las intersecciones en un patrón con 1 si la piedra es blanca, con 2 si la piedra es negra y con 0 si "no importa" [Mül01a]. Así, si por ejemplo tenemos el patrón 1002, la situación 1122 casa con ese patrón, ya que tiene los mismos valores para las piedras blancas y negras, y otros distintos en las posiciones que "no importan". También debemos pensar que un patrón se puede adaptar a una porción de tablero mayor; por ejemplo, el patrón 1002 debería casar con el tablero 100222, ya que tiene las casillas necesarias para aplicar el patrón.

Para guardar los patrones, en primer lugar consideramos usar una tabla indexada por el patrón, y con la jugada correspondiente como valor asociado. El

problema de esta representación es que dada una sección del tablero para la cual buscamos un patrón, solo lo podemos encontrar si el patrón tiene tanto los mismos valores como si es del mismo tamaño.

Para solucionar estos problemas, usamos un árbol PATRICIA (sección 2.3). El vocabulario del que hablamos en esa sección se concretará en $A = \{0, 1, 2\}$. También es necesario para usarlos definir un orden lineal entre las intersecciones, puesto que el orden en el que recorramos determinará la forma de la clave, que como veremos a continuación es necesario para aprovechar al máximo esta representación.

El orden escogido para el patrón consiste en comenzar con la esquina superior izquierda del patrón, e ir recorriendo el patrón como se indica en la figura 5.1.

El problema que plantea esta decisión es que los patrones deben ser cuadrados. Al ser el tablero cuadrado, cualquier patrón puede representarse como un cuadrado, pero en comparación con una representación en rectángulo supone un gasto en memoria mayor. Sin embargo, tiene como gran ventaja que permite reconocer patrones de distintos tamaños: si tenemos un patrón de longitud l, y calculamos desde el mismo lugar (es decir, empezando en la misma esquina) un patrón mayor, digamos de longitud $l' > l$, entonces los primeros l caracteres de esta nueva clave coincidirán con la clave del patrón menor.

Los árboles PATRICIA solucionan los problemas anteriores:

Patrones de distinto tamaño Para este caso, buscaremos en el árbol PATRICIA la clave correspondiente a la sección de tablero que queremos comprobar: si existe patrón para ese tablero, lo encontrará; en caso contrario, el mayor patrón correspondiente a ese tablero será el patrón que esté a más profundidad en el árbol. Esto lo podemos hacer, como hemos dicho anteriormente,

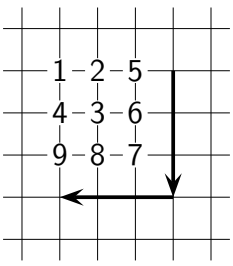

Figura 5.1: Orden lineal para las intersecciones en un patrón

por la forma en la que hemos definido el orden lineal en las intersecciones.

Patrones equivalentes Como hemos visto antes, es posible que dos patrones de la misma longitud encajen, puesto que hay ciertos índices que tienen como valor "no importa". Para este caso hicimos una búsqueda (sección 2.3.2) que buscaba el mayor patrón entre el índice actual y el índice "no importa".

Una vez encontrado el patrón, necesitábamos una representación de la jugada a realizar independiente de cualquier situación concreta. La información necesaria para saber qué jugada hacer en un momento concreto es el color de las piedras que deben jugarla y dónde deben hacerlo. El color es independiente de la jugada, pero no así la casilla, por lo que la representación que guardamos es el desplazamiento a la derecha y hacia abajo desde la esquina superior izquierda (nótese que si queremos ir hacia la izquierda o hacia arriba basta con usar valores negativos). Con esta representación, una vez encontrado un patrón, podemos realizar la jugada.

Los patrones pueden presentar problemas según la situación concreta. Por ejemplo, puede que el patrón que obtengamos nos proponga jugar en una intersección en la que no podemos por distintas razones, como puede ser que nos salgamos del tablero porque el patrón encontrado justo está en el borde del tablero. Si buscásemos los patrones en el árbol y por algún motivo no fuesen válidos, deberíamos buscar otra vez evitando este patrón, lo que supondría un gasto extra de tiempo. Por ello, la búsqueda que vamos a realizar en los árboles (sección 2.3.2) va a ser con ciertas restricciones: la jugada debe ser del color de las piedras que buscamos actualmente, y además debe ser en una intersección válida.

Capítulo 6

Adquisición de conocimiento

En esta sección explicamos las diferentes maneras en las que nuestra aplicación "aprende", es decir, cómo mejoran su eficiencia y sus resultados conforme le vamos proporcionando situaciones resueltas, que posteriormente usará en sus propios cómputos.

6.1. Refinamiento de la función de evaluación

Como hemos visto en la sección 3.4, es complicado dar a priori los valores de los pesos asociados a cada uno de los factores importantes de la función de evaluación. Para ello, nuestra aplicación proporciona la posibilidad de refinar estos pesos mediante situaciones para las que se conoce su mejor jugada.

Más concretamente, nuestro programa necesita que se le proporcione una configuración del tablero antes de realizar la jugada y la jugada que debe realizarse. Además, como hemos visto en la sección 3.3, es posible que a lo largo de la partida intentemos alcanzar diferentes objetivos, por lo que en la sección 3.4 decidimos asociar pesos no solo a factores, sino a factores y objetivos. Por ello, la aplicación necesita también saber qué objetivo vamos a satisfacer con la jugada que realizamos.

Una vez hemos introducido estos datos en la aplicación, calcula el valor de la función de evaluación para todas las posibles jugadas, cambiando cada uno de los pesos ligeramente, y quedándose con el peso que cumple los siguientes requisitos:

1. El valor de la función en la posición proporcionada por el usuario se maximiza.

2. La distancia entre el valor de la función en la intersección indicada y el valor de la función en el resto de posiciones se maximiza.

Es decir, no basta con que al cambiar el peso el valor de la función en ese punto aumente, sino que tiene que aumentar dejando que el valor de la función para el resto de las intersecciones disminuya o, en el peor de los casos, aumente lo mínimo posible.

Este cálculo debe repetirse un número de veces tal que la situación que aprendemos resulte útil; si lo hiciésemos pocas veces, la variación del peso sería tan pequeña que apenas contaría para futuros cómputos, pero si lo hiciésemos demasiadas veces, la función de evaluación solo funcionaría bien en situaciones similares a la aprendida. Después de realizar varias pruebas, vimos que un valor válido para ello eran 5 repeticiones.

6.2. Aprendizaje de patrones

Para obtener nuevos patrones, lo primero que hace nuestra aplicación es calcular las esquinas superior izquierda y la inferior derecha, calculamos qué lado es el mayor y lo tomamos como lado del cuadrado, puesto que los patrones que podemos aprender son cuadrados (como dijimos en el capítulo 5). Recorremos las intersecciones comenzando en la esquina superior izquierda utilizando el orden lineal establecido en el ese capítulo y una vez obtenido, comprobamos si ya estaba en el árbol PATRICIA. Si no está, podemos proseguir aprendiendo el patrón.

Lo siguiente que necesitamos para tener completo el patrón es la intersección en la que jugamos y con qué color. Como hemos visto en el capítulo 5, es necesario abstraer la situación concreta de la jugada, para que pueda servir cuando encontremos este patrón en cualquier otro lugar del tablero. Una vez calculamos estos desplazamientos, los guardamos junto con el color en el árbol PATRICIA.

Los patrones son simétricos con respecto al color, es decir, el mismo patrón, cambiando de color las piedras, es aplicable en la misma posición cambiando también el color de la piedra que colocamos. Por tanto se nos plantean dos posibilidades, añadir en el árbol tanto el patrón como el simétrico, o bien hacer dos búsquedas para cada patrón. Dado que no solo hemos elegido árboles para mejorar las búsquedas (sección 5), sino que además nuestros árboles están optimizados (sección 2.3.1), pensamos que la mejor decisión es realizar dos búsquedas, por lo

que solo es necesaria una inserción.

También son simétricos los patrones si los giramos, por ello, cuando insertamos aprendemos un patrón, también aprendemos todos sus simétricos. Para hacerlo, una vez hemos calculado el patrón y los desplazamientos necesarios, calculamos el giro del patrón y sus nuevos desplazamientos, y los insertamos en el árbol. En la figura 6.1 se muestra todas las posiciones simétricas de un cierto patrón, que serían aprendidas insertando una cualquiera de ellas.

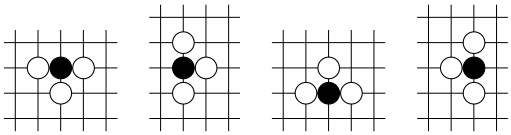

Figura 6.1: Simetrías por giro de los patrones

Por supuesto, todas las jugadas que hagamos en el proceso de aprendizaje de patrones deben ser válidas, por lo que el programa hace las comprobaciones correspondientes.

Capítulo 7

Módulo de red

Para aumentar la capacidad de cómputo de la inteligencia artificial, es decir, para lograr una inteligencia artificial más potente, aprovechando la divisibilidad del juego en zonas, se ha creado una arquitectura cliente-servidor que se encarga de distribuir la carga de trabajo de la expansión de nodos del árbol en varios PC's. Asimismo, como hemos creado una base de datos de patrones y un buscador de los mismos (capítulo 5) utilizamos uno de los clientes para la búsqueda de los patrones, mientras que el resto expanden el árbol de búsqueda por si ningún patrón fuera encontrado. De esta forma conseguimos realizar las dos búsquedas de manera simultánea y distribuida.

7.1. Arquitectura de red

La implementación realizada es una arquitectura de red básica, en la que hay un servidor que controla de manera centralizada todas las comunicaciones, de modo que el servidor es el encargado de transmitir el movimiento que ha elegido el jugador *humano* o la inteligencia artificial *enemiga* a todos los clientes, para que realicen sus búsquedas sobre un tablero de juego actualizado, y de manera similar, se encarga de recoger todos los resultados obtenidos por los clientes, seleccionar la mejor de las jugadas encontradas y comunicar cuál ha sido esta. De manera equivalente, cuando ha recogido la información de todos los clientes y ha seleccionado cuál es la mejor jugada de entre todas las recibidas, ha de informar a todos cuál ha sido su decisión, para que todos los clientes puedan actualizar sus tableros. Además, para evitar un gasto excesivo de CPU por parte de los clientes, como el buscador de

patrones es más rápido que la expansión de nodos del árbol, se encarga de informar al servidor de que un patrón ha sido encontrado y este a su vez transmite dicha información al resto de clientes para que interrumpan sus cómputos. Veremos en las siguientes secciones con más detalle el protocolo completo de comunicaciones así como otros detalles de implementación del servidor (sección 7.2) y del cliente (sección 7.3).

7.2. Servidor

El servidor, eje centralizado de las comunicaciones, se encarga de mantener a todos los clientes actualizados para que su trabajo sea útil, así como de elegir cuál de las jugadas propuestas por los clientes es la más adecuada. Lo primero que se hace cuando empieza a ejecutarse es esperar la conexión del número de clientes que se le indica en una de las constantes del sistema. Una vez ha sucedido esto, cuando ya todas las conexiones están establecidas, pasa a inicializar todos los clientes, asignándoles un identificador y suministrándoles información sobre el comienzo de la partida (comienzan blancas o negras), sobre qué jugador representan (blancas, negras o ambos) y sobre el tamaño del tablero, así como el algoritmo que

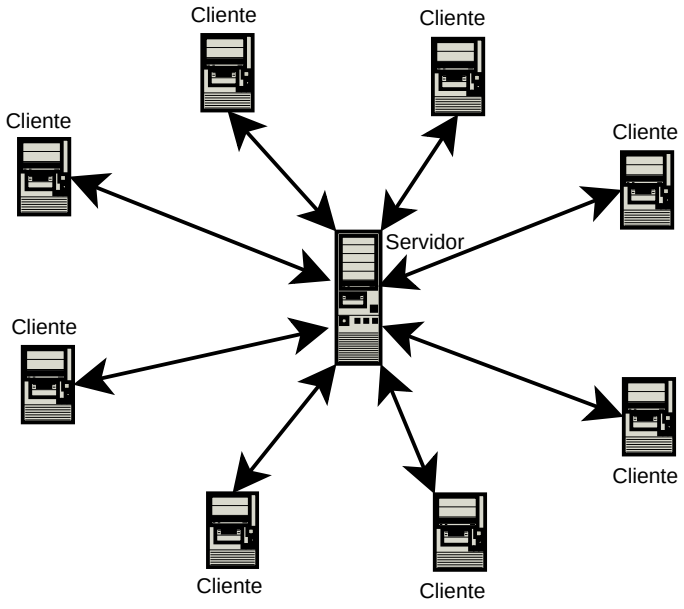

Figura 7.1: Arquitectura de red

deben ejecutar para realizar la búsqueda de la jugada, esto es, la *habilidad* de la inteligencia artificial.

7.3. Cliente

Es el último extremo de la comunicación, el que se encarga de comunicar los métodos correspondientes a la inteligencia artificial con el servidor. Al ejecutar un cliente, lo primero que hace es solicitar la dirección en la que se aloja el servidor, y una vez introducida, se conecta al mismo y recibe los datos para una correcta inicialización (tamaño del tablero, quién comienza, etc.). Una de las partes fundamentales de la inicialización es decidir qué cliente será el que asuma el rol de buscador de patrones. Por defecto, es aconsejable que siempre se ejecute la aplicación completa con un mínimo de dos clientes, para que al menos uno sea el buscador de patrones y haya otro que se encargue de ejecutar la expansión de nodos del árbol de búsqueda por si no hay ningún patrón aplicable. El número de clientes está definido en una de las constantes del sistema, y no tiene un máximo estipulado, aunque, como ha de haber un buscador de patrones y la división del tablero en zonas como máximo suele llegar a las nueve zonas, el número razonable de clientes está entre seis y ocho, así se logra equilibrar el número de clientes que están realizando trabajo *válido*. Si se configura la aplicación para que haya menos clientes, puede ser que haya zonas del tablero sobre las que no se está realizando expansión del árbol. Por otro lado, si optamos por utilizar más de nueve, la mayoría del tiempo habrá varios que estén realizando trabajo *inútil*, puesto que trabajarán con la misma zona que estarán compartiendo la zona de búsqueda con algún otro cliente.

Para garantizar la ejecución de un cliente con el rol de buscador de patrones, siempre se asocia dicho rol al primer cliente que se conecta con el servidor, el resto por tanto se encargarán de expandir el árbol de búsqueda.

7.4. Protocolo de red

Desde el comienzo del proyecto, el protocolo de red ha sufrido muchos cambios, para adaptarse a los progresos que se han ido realizando.

7.4.1. Versión inicial

Al comienzo del proyecto no había ningún tipo de buscador de patrones, con lo que el protocolo sólo se encargaba de comunicar al servidor con los clientes, estos expandían los nodos del árbol de la zona que se les hubiera asignado, contestaban con la mejor jugada al servidor y este los actualizaba para que todos supieran cuál había sido la jugada elegida. En las figuras 7.2 y 7.3 se puede observar el diagrama de flujo completo, tanto del cliente como del servidor, que sirvió de base para la construcción del protocolo final.

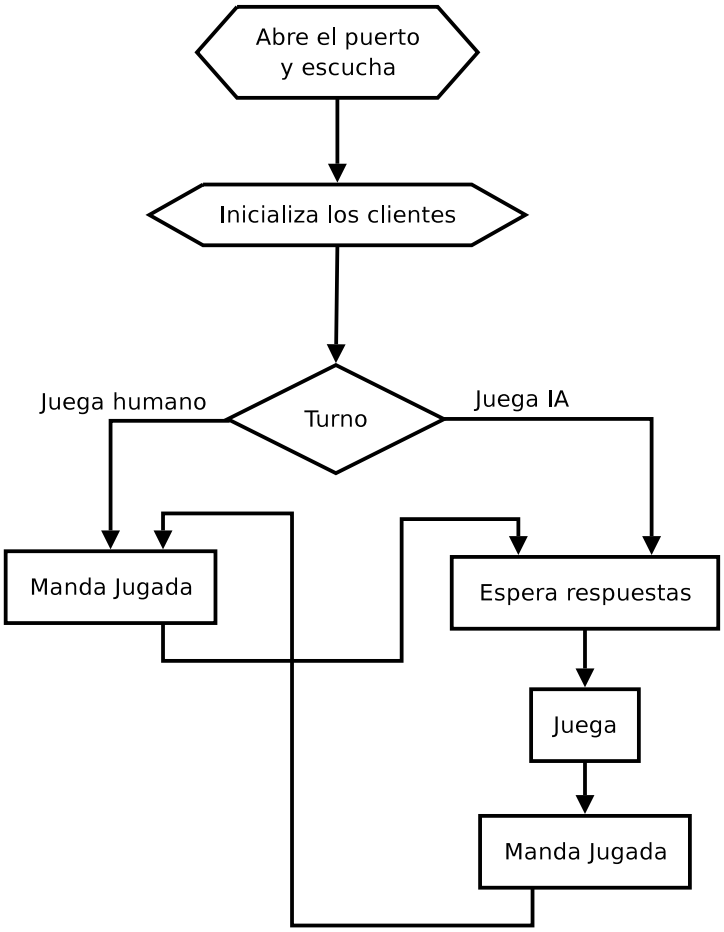

Figura 7.2: Versión inicial del protocolo, servidor

Para garantizar que la comunicación esté siempre sincronizada, es decir, que tanto el cliente como el servidor conozcan siempre el estado de la otra parte de

la comunicación, se optó por implementar un protocolo muy poco libre, pero muy robusto, donde todos los pasos en el intercambio de datos siguen una estructura fija:

Servidor:

1. Abre un puerto y escucha hasta que ha recibido todos los clientes.
2. A cada cliente recibido, le manda las opciones de inicialización.
3. Si juega el humano, manda la jugada a todos los clientes.
4. Si no, recibe todas las respuestas de los clientes, selecciona la mejor de todas las posibles jugadas e informa a todos los clientes de cuál ha sido la jugada elegida.

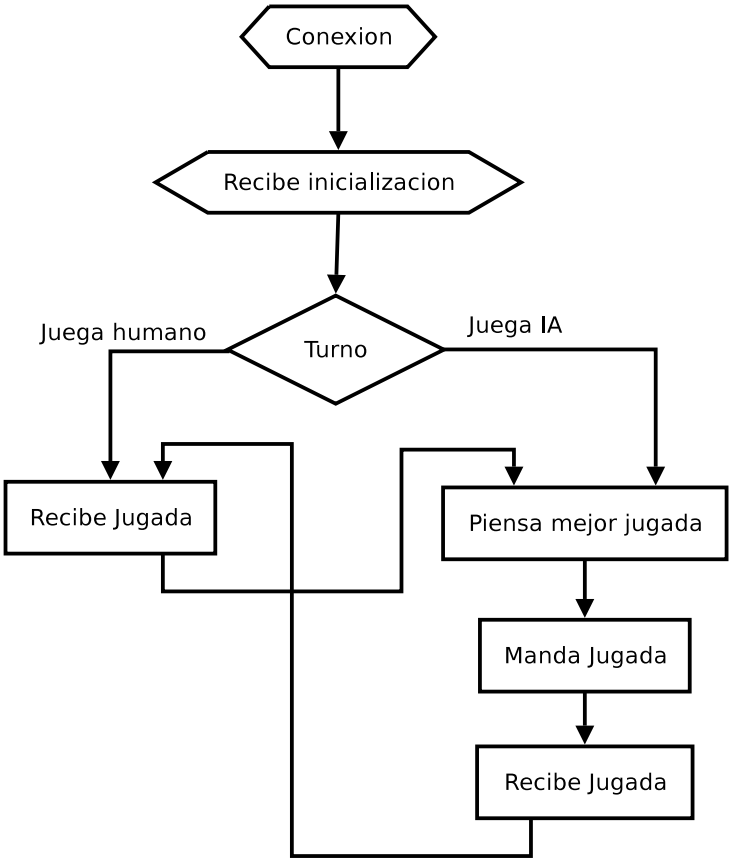

Figura 7.3: Versión inicial del protocolo, cliente

5. Repite el bucle desde el punto 3 hasta el final de la partida.

Cliente:

1. Solicita la dirección del servidor y con ella, se conecta al mismo.
2. Recibe la información de inicialización.
3. Si juega el humano, recibe la jugada que ha elegido.
4. Si no, ejecuta la inteligencia artificial y le manda el resultado al servidor; después, recibe la jugada que ha elegido el servidor para la inteligencia artificial.
5. Repite el bucle desde el punto 3 hasta el final de la partida.

7.4.2. Segunda fase

Con los primeros avances del proyecto, cuando se incluyó la posiblidad de que existiera un buscador de patrones, hubo que actualizar el protocolo, de modo que ahora había que identificar a uno de los clientes de un modo especial para que se encargara de ejecutar el buscador de patrones en lugar de expandir nodos del árbol de búsqueda. Así, si el servidor recibía del cliente asociado a la búsqueda de patrones una jugada que efectivamente fuera un patrón, descartaba el resto de respuestas que recibía por los demás clientes y utilizaba el patrón como la mejor de las posibles jugadas.

7.4.3. Versión final

Tras comprobar que el buscador de patrones era más rápido que la expansión de nodos se pasó a la tercera y última evolución del protocolo, en la que los clientes que expanden el árbol se transforman en un proceso que ejecuta dos hilos, uno de ellos es el que se encarga de la búsqueda mientras que el otro permanece a la escucha por si el servidor tiene algo que comunicarle, y es que, si el buscador de patrones encuentra un patrón aplicable, el servidor informa inmediatamente al resto de clientes de que se ha recibido un patrón y que por tanto, pueden interrumpir el proceso de búsqueda porque la jugada que iban a elegir nunca será seleccionada como la mejor jugada.

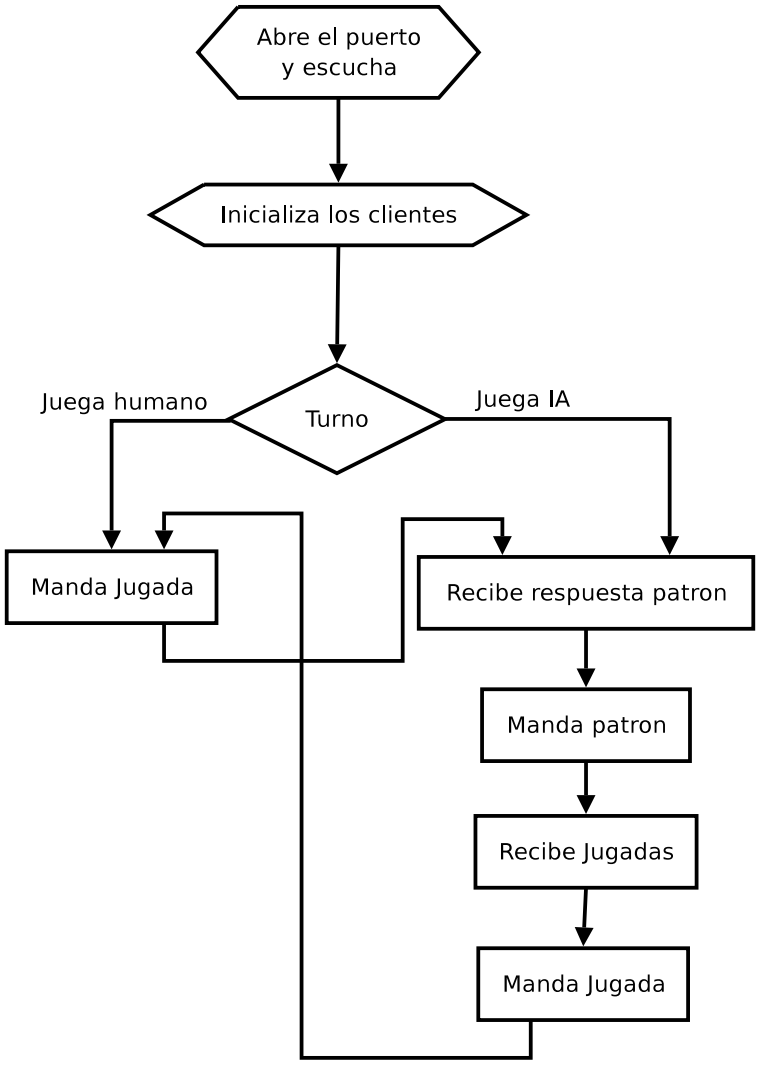

Figura 7.4: Versión final del protocolo, servidor

De este modo, tal y como queda descrito en la figuras 7.4 y 7.5, el servidor, tras la inicialización de los clientes, si en el turno actual juega la inteligencia artificial, recibirá la respuesta que le envíe el cliente encargado de buscar los patrones, y si esta se corresponde con un patrón, informará a todos los clientes mandándoles la información de que un patrón ha sido encontrado. Tras esto, el servidor vacía todos los flujos de entrada (por si algún cliente hubiera contestado antes de que llegara la jugada del patrón), y entonces actualiza con la jugada elegida a todos los clientes.

Los clientes, para no perder la sincronización con el servidor, una vez terminado

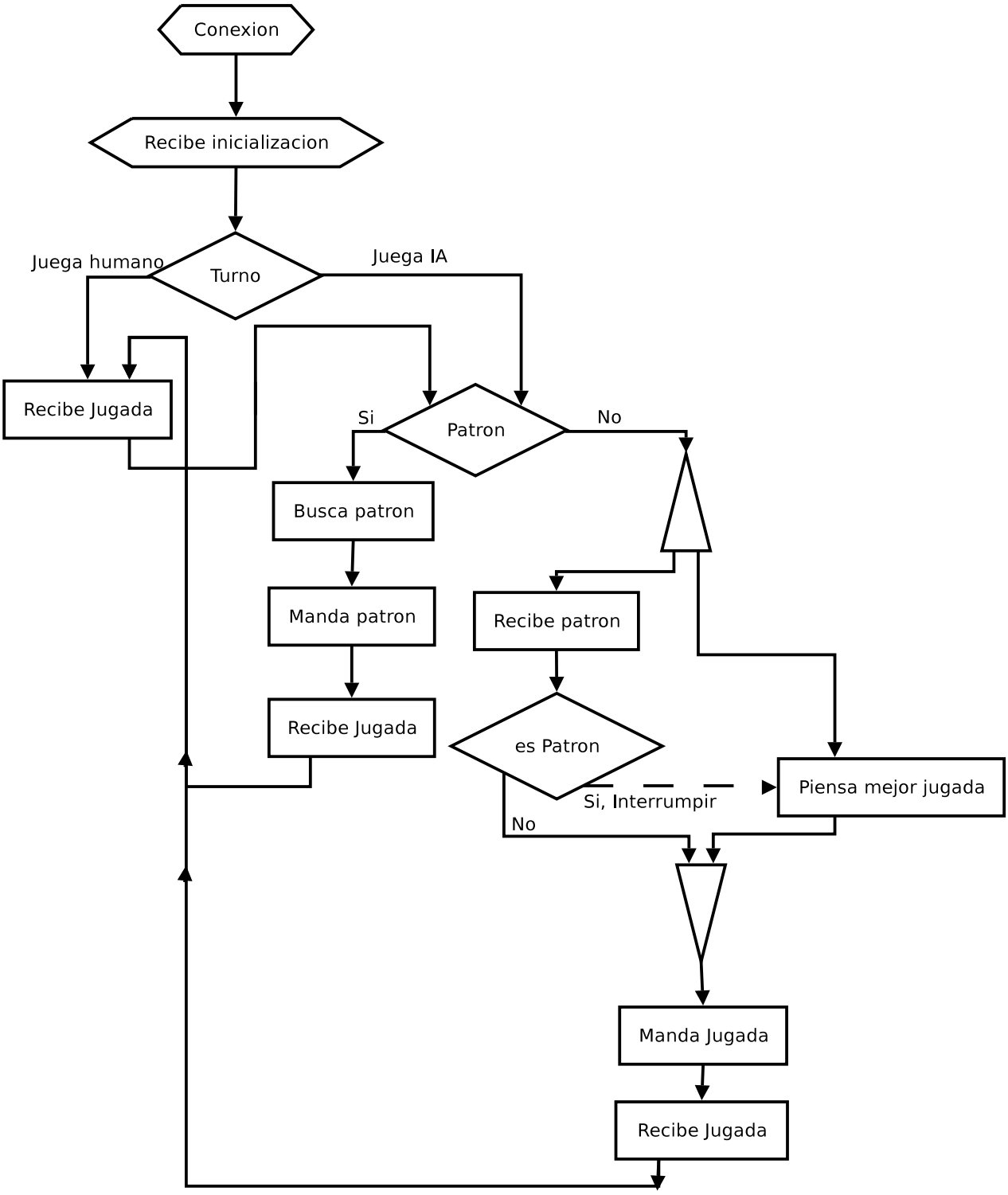

Figura 7.5: Versión final del protocolo, cliente

su proceso de búsqueda, bien porque han sido informados de la existencia de un patrón aplicable, bien porque han encontrado una jugada expandiendo el árbol de búsqueda, envían una jugada al servidor y después reciben la jugada finalmente elegida. De esta manera, el protocolo queda finalmente definido como sigue:

Servidor:

1. Abre un puerto y escucha hasta que ha recibido todos los clientes.
2. A cada cliente recibido, le manda las opciones de inicialización.
3. Si juega el humano, manda la jugada a todos los clientes.
4. Si no, recibe la jugada que manda el buscador de patrones e informa a los clientes de los resultados obtenidos del mismo, es decir, si se ha encontrado o no un patrón.
5. Recibe las jugadas de todos los clientes.
6. Selecciona la mejor y actualiza a los clientes.
7. Repite el bucle desde el punto 3 hasta el final de la partida.

Cliente:

1. Solicita la dirección del servidor y con ella, se conecta al mismo.
2. Recibe la información de inicialización.
3. Si juega el humano, recibe la jugada que ha elegido.
4. Si no:
   - Si es el encargado de buscar patrones:
     a) Busca un patrón aplicable.
     b) Si ha encontrado alguno se lo manda al servidor, si no, le informa de que no hay ningún patrón aplicable.
     c) Recibe la jugada finalmente elegida por el servidor.
   - si no:

a) Crea un hilo de ejecución en el que busca la mejor jugada posible.

b) Recibe la información sobre si se ha encontrado un patrón.

c) Si es así: Interrumpe el hilo de ejecución.

d) Si no: espera a que termine la búsqueda.

5. Manda la jugada elegida.

6. Recibe la mejor jugada que haya seleccionado el servidor.

7. Repite el bucle desde el punto 3 hasta el final de la partida.

7.4.4. Implementación

Para la implementación del protocolo se han utilizado las clases `java.net.ServerSocket` y `java.net.Socket` (conexión entre el servidor y los clientes) y `java.lang.Thread` (para el hilo de la versión final del cliente).

Se optó por la opción de usar `java.net.Socket` y `java.net.ServerSocket` porque la gestión del enlace es muy sencilla. El servidor consta de un `java.net.ServerSocket` que escucha por el puerto asociado a la aplicación (definido junto al resto de constantes del sistema) y los clientes utilizan un `java.net.Socket` para conectarse al servidor e intercambiar la información con él.

Para facilitar la comunicación entre el servidor y los clientes, en lugar de escribir un flujo de bits por el canal abierto, se aprovechan las facilidades que ofrecen los *sockets* de Java (`java.net.Socket` y`java.net.ServerSocket` para enviar objetos completos. Se han creado dos tipos de objetos diferentes, uno específico para la inicialización (donde se almacenan y transmiten todos los detalles sobre el tamaño del tablero, jugador que se asocia a la inteligencia artificial, etc.) y otro para la transmisión de las jugadas durante el resto del ciclo de vida del juego.

Para distinguir los casos de jugadas especiales, como los patrones, o indicar a los clientes que deben terminar su ejecución porque la partida ha terminado se utilizan unas constantes especiales que definen el tipo de la jugada que se manda.

Como una jugada está compuesta por el movimiento que representa y por el valor que se le da en la búsqueda para saber qué jugada es mejor que otra, aprovechamos ese entero, de forma que no produce un incremento del tráfico en la red.

Otro dato a destacar de la implementación es la creación del hilo (`java.lang.Thread`), y sobre todo, su interrupción. Para evitar la posibilidad de que al interrumpir o destruir un hilo se puedan quedar bloqueadas variables usadas para sincronización y llegar así a producir interbloqueos entre varios procesos, el uso de las funciones necesarias para la interrupción de hilos de ejecución está desaprobado (*deprecated*). Sin embargo, dado que no se utiliza ninguna de las posibles causas de riesgo, mantenemos el uso de las funciones, que por otro lado, no tienen ningúna alternativa que no esté desaprobada, con lo que si queremos mantener dicha funcionalidad para evitar un gasto innecesario de CPU, hemos de hacerlo así.

Capítulo 8

GTP (Go Text Protocol)

La intención del GTP es proporcionar una manera fácil y flexible de implementar un protocolo de comunicación entre programas de Go. Su principal propósito es permitir que dos programas puedan jugar el uno contra el otro, aunque también es útil para hacer tests regresivos (un controlador especifica una situación del tablero y le pide a un programa que, por ejemplo, genere un movimiento) y para la comunicación con una GUI preparada para ello o con un servidor de go. En la mayoría de casos se requiere un programa de apoyo externo, pero este puede ser compartido entre todos los programas que soporten el GTP.

En el tablero de Fractal Go, pueden tener lugar partidas entre nuestra aplicación y cualquier otra que implemente este protocolo. Para ello contamos con un controlador que es la GUI (en este caso, la clase `Gtp_Shell`) que se encarga de comunicarse con un motor (el otro programa de Go), de forma que le envía las jugadas de nuestra inteligencia artificial y recoge sus respuestas, tanto su confirmación de que ha recibido la jugada como su propia jugada.

8.1. Bases del protocolo

Todos los mensajes intercambiados en este protocolo son considerados como secuencias de caracteres de 8 bits. Sólo se usan los caracteres en US-ASCII (ANSI X3.4-1986) para los comandos estándar y las respuestas. Otros caracteres pueden usarse en los comentarios y las extensiones privadas.

8.1.1. Caracteres de control

Los caracteres con valor 0-31 y 127 son caracteres de control en ASCII. De éstos, tienen un significado en el protocolo, los siguientes:

`HT`(9) Tabulador horizontal
`CR`(13) Retorno de carro
`LF`(10) Avance de línea

Todos los demás caracteres de control deben ser descartados en la entrada y no usados en la salida.

8.1.2. Espacios en blanco

Los siguientes caracteres ASCII se pueden utilizar para indicar espacio en el protocolo:

`SPACE`(32) Espacio
`HT`(9) Tabulador horizontal

En el resto de la especificación utilizamos "espacio" para denotar un carácter de espacio en blanco. En la entrada esto puede ser un `SPACE` o un `HT`. En la salida solamente un `SPACE` puede ser utilizado.

8.1.3. Nueva línea

Una línea nueva es indicada por un carácter sencillo `LF`. Cualquier ocurrencia de un carácter `CR` debe ser descartada en la entrada. En la salida, tanto `LF` como cualquier combinación de `CR` y `LF` pueden ser usadas. En posteriores descripciones de sintaxis usaremos `\n`.

8.1.4. Estructura de los comandos

Un comando es exactamente de una línea de longitud, con la sintaxis
`[ id ] nombre_comando [ argumentos ]`

Aquí `id` es un número de identificación y `nombre_comando`, una cadena. En el resto de la línea (hasta el primer `\n`) están los argumentos del comando.

8.1.5. Estructura de la respuesta

Si es acertada, el motor devuelve una respuesta que puede tener las siguientes cuatro formas

```
=id respuesta\n\n
=id\n\n
=respuesta\n\n
=\n\n
```

`=` indica éxito, `id` es el número de identificación dado en el comando, y el resultado es un texto con dos `\n` consecutivos.

8.1.6. Mensajes de error

Si no tiene éxito, el motor devuelve una respuesta de la forma

```
?[ id ] mensaje_de_error\n\n
```

Aquí `?` indica fallo, `id` es el número de identificación dado en el comando, y el `mensaje_de_error` da una explicación del fallo, también terminando con dos `\n` consecutivos.

Si el motor recibe un comando desconocido o no implementado, se usa el mensaje de error "unknown command".

8.1.7. Sincronización

No hay requisitos de sincronización entre el controlador y el motor. El controlador puede enviar comandos en cualquier momento, sin importar si ha obtenido las respuestas para los comandos anteriores. El motor puede enviar respuestas siempre que estén preparados. Debe, sin embargo, responder a los comandos en el mismo orden en el cual le llegaron. Se permite que el motor haga pausas mientras que envía una respuesta.

8.1.8. Comentarios

Se pueden incluir comentarios en el flujo de comandos. Todo el texto entre `#` y el siguiente `\n` se considera como comentario y se debe desechar en la entrada.

8.1.9. Líneas vacías

Las líneas vacías y las líneas que tienen solamente un espacio o espacios en blanco enviadas por el controlador deben ser ignoradas por el motor; no debe ser generarse ninguna respuesta. Las líneas vacías y las líneas con solamente un espacio en blanco enviadas por el motor y ocurridas fuera de una respuesta deben ser omitidas por el controlador. Nótese que las líneas de comentario puras aparecerán como líneas vacías después de que se haya desechado el comentario.

8.1.10. Coordenadas del tablero

Las intersecciones del tablero, son codificadas por una letra más un número. En un tablero 19×19, las letras van de la A a la T, excluyendo la I, de izquierda a derecha. Los números van del 1 al 19, de abajo arriba. Así la esquina izquierda inferior es la coordenada A1, la esquina derecha inferior es la T1, la esquina izquierda superior, A19, y la derecha superior, T19. Los tableros más pequeños utilizan el subconjunto obvio de estas coordenadas. Tableros más grandes, hasta 25×25, se manejan extendiendo las letras de la U a la Z según sea necesario. El protocolo no soporta tableros más grandes que este tamaño.

8.2. Detalles del protocolo

8.2.1. Preprocesamiento

Cuando un comando llega a un motor, se espera que realice las cuatro operaciones siguientes antes que hacer cualquier otro análisis sintáctico:

1. Quitar todos las ocurrencias de `CR` y de otros caracteres de control a excepción de `HT` y `LF`.
2. Para cada línea con `#` eliminar todo el texto que lo sigue incluido este carácter.
3. Convertir todas las ocurrencias de `HT` a `SPACE`.

4. Desechar cualquier línea vacía o con solo espacios en blanco.

Cuando una respuesta llega a un controlador, se espera que haga solamente los pasos 1 y 3.

8.2.2. Entidades sintácticas

A continuación enumeramos las distintas entidades que se usan en los comandos.

Entidades simples

int Entero sin signo en el intervalo $0 \leq x \leq 2^{31} - 1$.

float Número en coma flotante representado por un real IEEE 754 de 32 bits.

string Secuencia de caracteres, excepto espacios en blanco. Las cadenas respetan las mayúsculas.

vertex Coordenada del tablero consistente en una letra y un número como hemos dicho anteriormente o en la cadena "pass". Las coordenadas no respetan las mayúsculas. Ejemplos: "C15", "h1".

color Un color es una de las cadenas "white" o "w" para las piedras blancas o "black" o "b" para las negras. Los colores no respetan las mayúsculas.

move Un movimiento es la combinación de un color y un vertex, separado por un espacio. Los movimientos no respetan las mayúsculas. Ejemplos: "white h10", "B F5", "w pass".

boolean Expresión booleana "true" o "false".

Entidades compuestas

Collection Un {x y} es una x seguida de una y separada por un espacio, donde x e y son cualquier combinación de entidades simples. Esta construcción puede ser generalizada a cualquier número fijo de entidades.

List Un x* es una lista separada por espacios de entidades del tipo x, donde x puede ser cualquiera de las entidades especificadas hasta ahora. La lista puede tener un número arbitrario de elementos y continúa hasta que se encuentra un `LF`.

Alternativas Un x|y es una x o una y.

Lista multilínea Un x& es una lista de entidades del tipo x, separadas por `LF`, donde x puede ser cualesquiera de las entidades especificadas hasta ahora. La lista multilínea puede tener un número de líneas arbitrario y continúa hasta que se encuentran dos `LF`s consecutivos.

8.2.3. Comandos

Los comandos básicos son:

- `protocol_version`

  argumentos: ninguno
  efectos ninguno
  salida: `int numero_version` - la versión del protocolo GTP
  fallos: nunca
  comentarios: para esta especificación del protocolo, la 2.

- `name`

  argumentos: ninguno
  efectos: ninguno
  salida: `string* name` - Nombre del motor del juego
  fallos: nunca
  comentarios: Por ejemplo, “GNU Go”, “Aya”, “Many Faces of Go”. El nombre no incluye ninguna información sobre la versión, la cual es proporcionada por el comando siguiente.

- `version`

  argumentos: ninguno
  efectos: ninguno
  salida: `string* version` - Versión del motor

fallos: nunca
comentarios: Por ejemplo "3.1.33", "10.5". Los motores sin número de versión deben devolver una cadena vacía.

- `known_command`

  argumentos: `string nombre_comando` - Nombre del comando
  efectos: ninguno
  salida: `boolean known` - "true" si el comando es reconocido por el motor, "false" en otro caso.
  fallos: nunca
  comentarios: El protocolo no hace ninguna distinción entre los comandos desconocidos y los conocidos pero no implementados.

- `list_commands`

  argumentos: ninguno
  efectos: ninguno
  salida: `string& commands` - Lista of comandos, uno por fila
  fallos: nunca
  comentarios: Incluye todos los comandos conocidos, incluidos los básicos y las extensiones de cada motor.

- `quit`

  argumentos: ninguno
  efectos: La sesión finaliza y la conexión se cierra
  salida: ninguna
  fallos: nunca
  comentarios: La respuesta completa de este comando debe ser enviada antes de que el motor cierre la conexión. El controlador debe recibir la respuesta antes de que la conexión se cierre en su lado.

- `boardsize`

  argumentos: `int tamaño` - Nuevo tamaño del tablero.
  efectos: cambia el tamaño del tablero. La configuración del tablero, el número de piedras capturadas y el registro de movimientos se hacen arbitrarios.
  salida: ninguna

fallos: error de sintaxis. Si el motor no puede manejar el nuevo tamaño, falla con el mensaje de error "unacceptable size".
comentarios: En la versión 1 del GTP este comando también hacía el trabajo de `clear_board`. Esto puede ocurrir o no en las implementaciones de la versión 2 del GTP. Así el controlador debe llamar a `clear_board` explícitamente. Incluso si el nuevo tamaño del tablero es igual que el viejo, la configuración del tablero se hará arbitraria.

- `clear_board`

  argumentos: ninguno
  efectos: Se limpia el tablero, el número de piedras capturadas se pone a cero en ambos colores y se borra el historial de movimientos.
  salida: ninguna
  fallos: nunca.

- `komi`

  argumentos: `float nuevo_komi` - El nuevo valor de komi.
  efectos: Se cambia el komi.
  salida: ninguna
  fallos: error de sintaxis
  comentarios: El komi es una puntuación extra que se le da al jugador de las piedras blancas por la desventaja de empezar en segundo lugar. El motor debe aceptar el komi aunque el número sea ridículo.

- `play`

  argumentos: `move movimiento` - Color y coordenada del movimiento
  efectos: Una piedra del color en cuestión se juega en la coordenada pedida. Se actualiza el número de piedras capturadas si es necesario y el movimiento es añadido al historial de movimientos.
  salida: ninguna
  fallos: error de sintaxis, movimiento ilegal.
  comentarios: Movimientos consecutivos del mismo color no son considerados ilegales desde el punto de vista del protocolo.

- `genmove`

argumentos: `color color` - Color para el cual se genera un movimiento
efectos: Una piedra del color requerido es jugada donde el motor elija. Se actualiza el número de piedras capturadas si es necesario y el movimiento es añadido al historial de movimientos
salida: `vertex|string vertex` - Coordenada del movimiento o cadena "resign".
fallos: nunca
comentarios: Nótese que "pass" es un vertex válido y es devuelto si el motor quiere pasar. Usa "resign" si quiere abandonar la partida. Se permite que el controlador use este comando sea quien sea el que haya jugado en el último lugar.

Otros comandos

A parte de los básicos arriba mencionados, existen otros comandos como:

`fixed_handicap`
`place_free_handicap`
`set_free_handicap`
`loadsgf`
`reg_genmove`
`undo`

De estos solo usamos en Fractal Go el comando `set_free_handicap`, puesto que nuestra colocación de las piedras de handicap difiere de la del protocolo. Con este comando, le enviamos al otro programa las coordenadas de estas piedras.

- `set_free_handicap`

argumentos: `vertex* coordenadas` - Una lista de coordenadas donde deben ser colocadas las piedras de handicap en el tablero.
efectos: Las piedras de handicap son situadas en las coordenadas requeridas
salida: ninguna
fallos: error de sintaxis, tablero no vacío, lista de coordenadas incorrecta
comentarios: Este comando es solamente válido si el tablero está vacío. La

lista debe tener por lo menos dos elementos y no más que el número de intersecciones del tablero menos una. El motor debe aceptar la colocación de las piedras de handicap. Las piedras de handicap no se incluyen en el historial de movimientos. Las coordenadas no deben estar repetidas o incluir "pass".

Más información sobre este protocolo en [Far02].

8.3. Implementación

El módulo GTP incluye el interfaz `IGtp`, la clase que lo implementa, `Gtp`, y las clases `Gtp_Shell` (interfaz gráfica de usuario de este módulo), `Traductor`, `Color` y `StringUtils`. `Gtp_Shell` actúa de controlador, gestionando los turnos de la IA de Fractal Go y la de la otra máquina. Todo ello con el soporte de `Gtp` que se encarga de la sintaxis de los comandos, de su envío, de la recepción de las respuestas y de la gestión de los errores, para lo que cuenta con la ayuda de `StringUtils` y `Color`.

`Traductor` es una clase auxiliar que convierte las coordenadas que utilizamos en nuestra aplicación a las coordenadas que establece el protocolo.

Las clases `Gtp`, `StringUtils` y `Color` están basadas parcialmente en las clases del mismo nombre de la aplicación GoGui (http://sourceforge.net/projects/gogui/), que cuenta con licencia GNU GPL. `StringUtils` y `Color` son clases auxiliares de `Gtp`, de la cual hemos eliminado todos los métodos que tuvieran que ver con el análisis sintáctico de las respuestas de comandos que nosotros no enviamos. Para el análisis de los que sí lo hacemos, utilizamos nuestros propios métodos adaptados a nuestro diseño y que se encuentran en la clase `Traductor`. Además hemos incorporado métodos para el envío del comando `set_free_handicap`.

A continuación describimos el funcionamiento de este módulo.

Obtenemos la ruta del programa con el que queremos enfrentar a Fractal Go a través del menú de opciones de nuestra aplicación.

Lo ejecutamos en modo gtp y le enviamos el comando `version` para obtener la versión del protocolo que implementa para así poder tratarlo en consecuencia. Después limpiamos su tablero con el comando `clear_board` que lleva como parámetro el tamaño que se ha establecido previamente en el menú y, luego le pasamos este tamaño a través de `boardsize`.

Si se ha indicado que hay piedras de handicap ahora sería cuando se lo indicaríamos a la otra máquina con el comando `set_free_handicap` seguido de la lista de coordenadas de las piedras. Conviene advertir que máquinas como "Aya" no soportan este comando puesto que los comandos de handicap son opcionales como ya hemos dicho anteriormente.

Si no hay handicap, se pasa directamente aquí: comienzan a jugar las negras, si juega nuestra IA, enviamos a la otra máquina la jugada mediante el comando `play` seguido del movimiento escrito tal y como especifica el protocolo, es decir, si por ejemplo jugara en la coordenada A13, sería "`play black A13`". Después verificamos que lo ha recibido correctamente recogiendo su respuesta que, como dijimos en el apartado de la sintaxis, debe ser "`=\n`" puesto que el comando no tiene parámetro de salida.

Si, por el contrario, tiene que empezar a jugar la otra máquina, le enviamos el comando `genmove` seguido del color, en este caso, negro, es decir "`genmove black`" y posteriormente, recogemos la respuesta que será de la forma "`=movimiento\n`".

Después de esto, la partida se va desarrollando de la misma forma, jugando cada uno en su turno. Así, en el primer caso, la siguiente jugada sería de las blancas con las que juega la otra máquina y, por tanto, habría que pedírsela mediante el comando `genmove`. En el segundo caso, sería Fractal Go quien jugaría con las blancas y le tendría que enviar a la otra máquina su jugada con el comando `play`.

Capítulo 9

Interfaz Gráfica de Usuario

Nuestra idea era acompañar al juego de una interfaz amigable y atractiva y desde la que se pudiera controlar múltiples parámetros de la partida. En cuanto al aspecto, está ambientada en el *manga*.

9.1. Características generales

Nuestro juego puede verse en tres idiomas y tiene dos modos. Al ejecutar la aplicación aparece un primer menú con las siguientes opciones: (figura 9.1)

Idioma Español, inglés y japonés. Animamos al lector a que descubra el misterio del cuarto idioma seleccionable.

Modo Podemos elegir entre normal y aprendizaje. Al marcar el modo normal, accederemos a un segundo menú, el menú de opciones de partida. Si seleccionamos modo aprendizaje, aparecerá un tablero de tamaño 19×19 en el que podemos enseñar a Fractal Go la jugada adecuada a partir de la colocación de las piedras que el jugador le indique. Todo esto lo comentamos detalladamente más adelante.

Figura 9.1: Menú inicial

9.2. Modo normal

9.2.1. Menú de opciones

Fractal Go tiene un amplio menú de opciones (figura 9.2) en el que se pueden ajustar los siguientes parámetros:

Tipo de partida

Podemos elegir una partida entre dos jugadores humanos, entre un jugador humano y una máquina (la nuestra) o entre dos máquinas, una de ellas Fractal Go y la otra puede ser Fractal Go u otra aplicación que implemente el protocolo GTP (Go Text Protocol) en su primera versión o en la segunda, que era la última realizada en el momento de la finalización de este proyecto. Sobre el GTP ya hablamos en el capítulo 8.

Tamaño del tablero

Las partidas se pueden jugar en el clásico tablero de tamaño 19×19 o bien en los tamaños 9×9 o 13×13.

Figura 9.2: Menú de opciones

Handicap

Número de piedras de handicap que un jugador (el que juega con blancas) le da a otro de ventaja (negras). Se puede elegir un número entre 2 y 9.

Nombre de los jugadores

Cada jugador puede elegir con qué nombre quiere aparecer durante la partida. Si el jugador es una máquina, automáticamente se le llamará como su archivo ejecutable sin extensiones.

Nivel de Fractal Go

Hay dos niveles de dificultad: 0 y 1. En el nivel 0, Fractal Go calcula sus jugadas mediante el algoritmo Monte Carlo (ver sección 3.1) y en el nivel 1, mediante una búsqueda Proof-Number (ver sección 3.2).

Inteligencia artificial

Aquí es donde, si hemos seleccionado anteriormente una partida humano vs. máquina o máquina vs. máquina, indicamos quién/es es/son las máquinas, pudiendo escoger una aplicación distinta a Fractal Go, si es el caso.

9.2.2. Personajes

Para darle un toque divertido y de competitividad a la aplicación, incluimos una pequeña presentación antes de la partida, donde se le asigna a cada jugador un personaje manga, desafiando cada uno de ellos a su rival.

Figura 9.3: Pantalla de presentación de personajes

9.2.3. Tablero de juego

Tal y como podemos apreciar en la figura 9.4, la interfaz de la partida en el modo normal consta del tablero propiamente dicho, un panel con las estadísticas de cada jugador y con las opciones para pasar o rendirse y una barra de menús en la parte superior. El tablero será del tamaño establecido en el menú de opciones y alrededor de él se muestran las columnas, que van de la A a la T, excluyendo la I, y las filas de 1 a 19, todo ello para el tablero 19×19. Menores serán para los otros tamaños.

Si hay algún jugador humano, este puede poner su piedra, simplemente pulsando con el ratón en la intersección que desee y sea posible según las reglas del Go. Esto está implementado para que no sea necesario pulsar en el píxel exacto de la imagen, sino que bastará con pulsar en alguno de los cercanos.

Así mismo también puede dejar de poner piedra en su turno pulsando el botón "Paso" o abandonar la partida, haciendo lo propio en el botón "Rendirse".

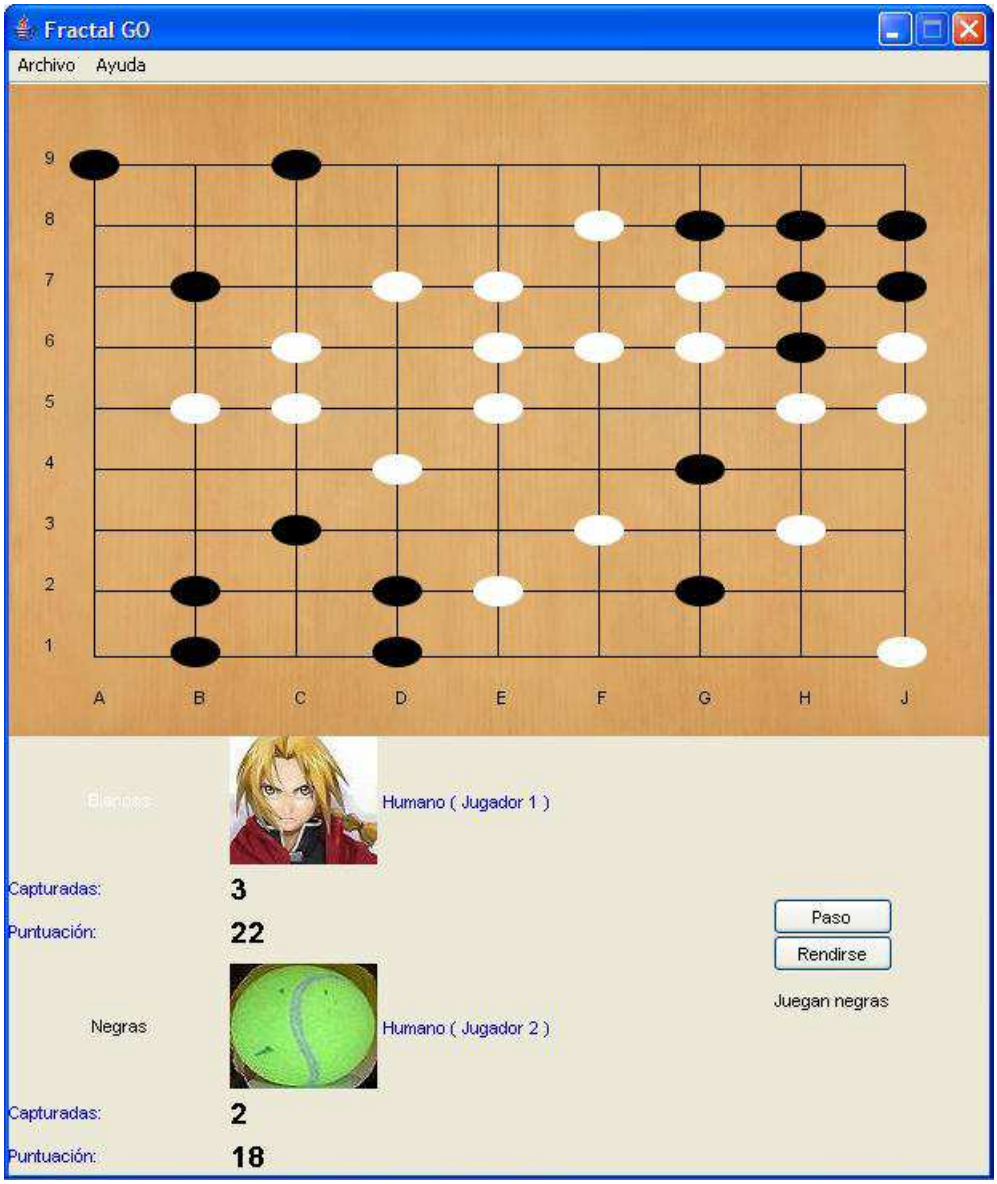

Figura 9.4: Tablero de juego en modo normal

Si algún jugador es máquina, se mostrará en cada turno, su jugada en el tablero, dibujándose en la intersección la piedra del color correspondiente.

Debajo del tablero se muestra la información de cada jugador que consiste en su nombre, si es humano o máquina, la foto de su personaje, el número de fichas capturadas y su puntuación. Además en la esquina inferior derecha se muestra el turno, es decir, a quien le toca jugar en cada momento.

La barra de menús de la parte superior consta de:

- Menú "Archivo", con las opciones de "Reiniciar partida", "Volver a menú principal" y "Salir".

- Menú "Ayuda", con las opciones de "Las Reglas de Go", donde podemos encontrar una guía rápida de cómo jugar al Go y "Acerca de...", donde se encuentran la información de la versión de Fractal Go, quiénes forman parte del proyecto y agradecimientos.

La partida terminará cuando los dos jugadores pasen o uno de ellos se rinda. Si pasan, se mostrará quién es el ganador y las puntuaciones finales de los dos jugadores, que incluye los puntos correspondientes que se les da por su territorio.

Figura 9.5: Final de partida

Además después del mensaje de final de partida (figura 9.5), se mostrará en el tablero este territorio con piedras más pequeñas para los lugares en los que no existiera la propia piedra.

9.3. Modo aprendizaje

A través de este modo podemos enseñar a Fractal Go a resolver situaciones que se pueden dar en una partida. Como podemos ver en la figura 9.6, la interfaz de este modo consta del tablero, en este caso siempre con el tamaño clásico 19×19, una barra de menús situada en la parte superior y a su derecha un panel con varios botones para la ejecución de diversas acciones.

La barra de menús de la parte superior consta de:

- Menú "Forma de Aprendizaje", en el que podemos elegir aprendizaje por patrones o aprendizaje por pesos, y en estos a su vez se puede escoger entre varios objetivos: "Ganar", "Conseguir ojos", "Cohesión", "Puntos" y "Conseguir ojo".

- Menú "Ayuda", con las mismas características que el del modo normal.

La mecánica de este modo consiste en hacer clic en las intersecciones deseadas poniendo las piedras de cada color que se crean convenientes, pulsar el botón "Aprender situación" para que aprenda la colocación de las piedras que se ha hecho y, a continuación, volver a hacer clic (se puede cambiar el color de la piedra) en la intersección en la que la inteligencia artificial de la aplicación debería poner la piedra en una partida. De esta manera, todo queda aprendido.

Acompañan al botón anterior, el botón "Cambiar color", para poner las piedras de un color o de otro; "Limpiar Tablero", para borrar todas las piedras colocadas; "Empezar partida", para regresar al menú de selección de idioma y de modo; y, finalmente, el botón "Salir", para abandonar la aplicación.

La información del color de la piedra siguiente que vamos a colocar se encuentra como en la interfaz del modo normal, en la esquina inferior derecha.

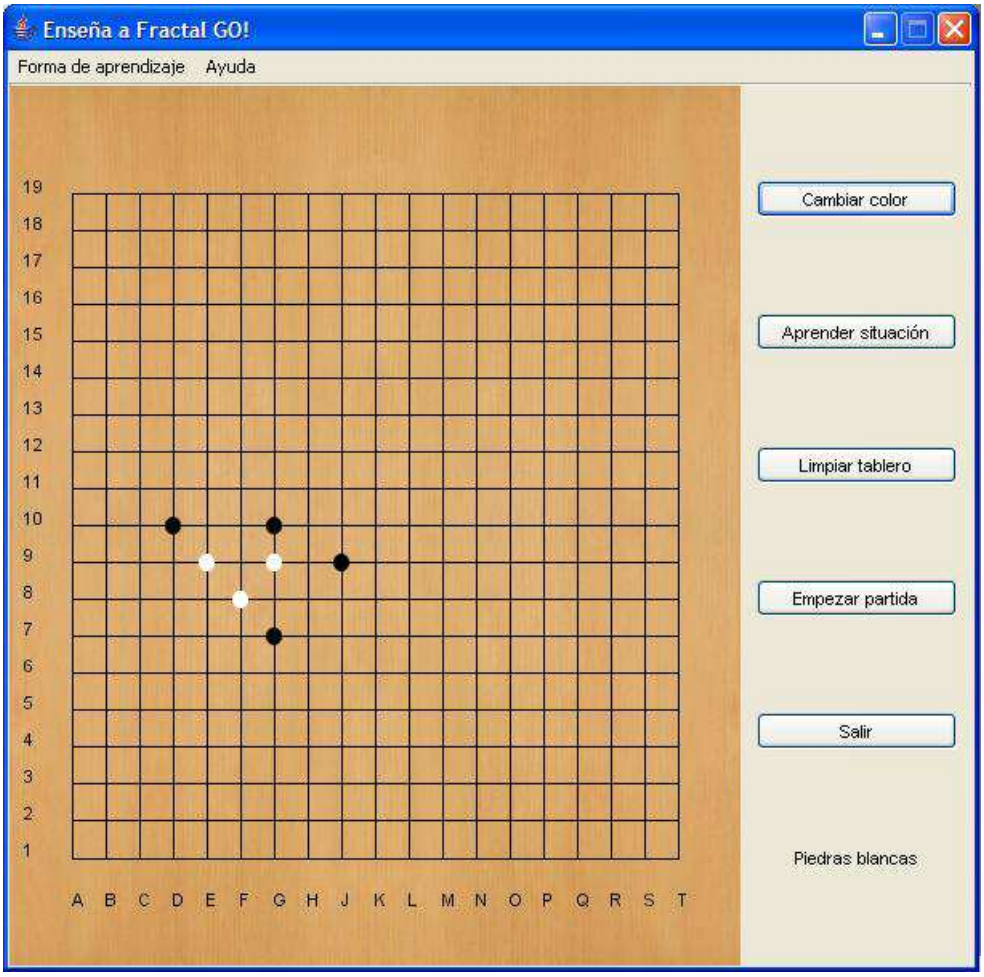

Figura 9.6: Tablero de juego en modo aprendizaje

9.4. Implementación de la interfaz gráfica de usuario

Este módulo contiene las siguientes clases: `Main`, `GUI` y `GUIAprendizaje`. La clase `Gtp_Shell` perteneciente al módulo gtp, es también en realidad una interfaz, la interfaz de las partidas en modo GTP, y mientras que en la sección 8.3 explicábamos su función en cuanto a dar los relevos a las dos máquinas, aquí tendremos en cuenta su implementación en cuanto interfaz.

La clase principal de este módulo es la clase `Main`, y se encarga de la visualización del menú de idioma y modo y del menú de opciones, así como de la creación del objeto de la interfaz del modo adecuado.

Las clases `GUI`, `GUIAprendizaje` y `Gtp_Shell` utilizan el patrón fachada, a través del interfaz `IJuego`, es decir, este interfaz es una vía de acceso unificada hacia el resto de interfaces de nuestra aplicación.

La clase `GUI` usa para representar el tablero la clase `Graphics` de `Java`, gracias a la cual, dibujamos las líneas, las coordenadas y las fichas, así como la imagen de fondo del tablero.

En la implementación de estas clases el principal problema que tuvimos fue dibujar las líneas de tal forma que se mantuviera el tamaño de las celdas del tablero, ya que los píxels son números enteros y al dividir el espacio disponible entre el número de intersecciones, en muchas ocasiones, la aproximación dejaba demasiado espacio para la última fila y/o para la última columna.

Además en las partidas GTP, las jugadas se sucedían sin que diera tiempo a que el tablero se refrescara y por tanto los movimientos no eran visibles. La solución fue mostrar un diálogo que informara de cada movimiento de los dos jugadores, deteniendo así el juego hasta que se presionara “Aceptar”.

Capítulo 10

Conclusiones

Entre los objetivos conseguidos cabe destacar que se ha creado un sistema completo de aprendizaje, un sistema distribuido de inteligencia artifial y la implementación de un protocolo de comunicación estándar para que *Fractal Go* pueda participar en partidas contra otros programas que también implementen dicho protocolo (capítulo 8).

10.1. Aprendizaje

Dado que la base principal del proyecto es la inteligencia artificial, uno de los objetivos que se pretendían conseguir era desarrollar un sistema de aprendizaje, puesto que sin aprendizaje no hay inteligencia.
Además, como el sistema utiliza dos métodos de buscar la mejor jugada, búsqueda de patrones (capítulo 5) y expansión de nodos en un árbol de búsqueda (capítulo 3), el sistema puede aprender patrones nuevos y asociar valores a determinadas jugadas para que se puedan utilizar luego en la expansión del árbol. De este modo, el sistema consta de una buena base para poder llegar a ser algún día (con el debido entrenamiento) un gran *jugador* de Go.

Otra de las virtudes de *Fractal Go* es que no da de lado a ninguna de las dos opciones de búsqueda. Habitualmente los programas de Go trabajan mucho una de las dos ramas, o la búsqueda de patrones, o la expansión de un árbol de búsqueda, con lo que consiguen especializarse en un método pero dando de lado al otro, y es aquí donde *Fractal Go* puede marcar la diferencia, puesto que al poder aprender

para las dos estrategias de búsqueda, aprovecha los recursos de ambas.

10.2. Distribución

El hecho de que en lugar de un procesador pueda haber desde dos hasta alrededor de siete trabajando para buscar la mejor jugada provoca de manera inmediata la sensación de que se pueden conseguir muchas más cosas en el mismo tiempo. La divisibilidad en zonas de un juego de Go ha contribuido para que se pueda plantear una solución de este tipo, que por ejecutarse en varias máquinas completas, en lugar de un multiprocesador nos permite ahorrar costes, además siempre se mantiene el nivel de trabajo de los clientes al mínimo posible (se interrumpe la expansión de nodos si se encuentra un patrón, ...) para que no haga falta un grupo completo de computadoras dedicadas al desarrollo de la partida, si no que permite que haya *colaboradores* que sigan desarrollando su trabajo mientras su máquina contribuye a, por ejemplo, ganar un torneo de Go disputado a través de internet.

10.3. Go Text Protocol

Las posiblidades que brinda la implementación del GTP (capítulo 8) dotan al sistema de una gran capacidad de introducción en el mundo real de los juegos de Go por computadora, puesto que es el estándar que se utiliza en todos los torneos en los que participan este tipo de programas. Además, en *Fractal Go* se ha diseñado también una cómoda y divertida interfaz grafica (capítulo 9) para que no solo se puedan ver partidas entre computadoras, si no también para poder disputar partidas dos jugadores entre sí o jugar contra *Fractal Go* de una manera sencilla.

Capítulo 11

Trabajo futuro

Para trabajos posteriores que deseen continuar con el proyecto de *Fractal Go*, quedan ciertos campos todavía abiertos:

- Completar todas aquellas partes opcionales del GTP (Go Text Protocol, capítulo 8) que no ha dado tiempo a implementar. Este es un apartado interesante, puesto que no hay aún ningún proyecto juego de Go que implemente todas las partes opcionales del protocolo, si no que la mayoría implementan solo aquellas partes mínimas necesarias para el desarrollo de una partida.

- Adaptar toda la arquitectura de red, para que todos los intercambios de información se realicen aprovechando el GTP, de modo que no sea necesaria una arquitectura totalmente independiente, si no que directamente la información de las jugadas de los otros juegos, se puedan pasar directamente a los clientes, sin hacer la traducción ahora necesaria.

- Cambiar todo el sistema de inteligencia artificial y arquitectura de red, por ejemplo una red neuronal o un algoritmo evolutivo, o mejorar nuestra inteligencia artificial añadiendo más objetivos.

- Conseguir que un experto utilice todo el sistema de aprendizaje para dotar a *Fractal Go* de una capacidad que todavía no posee, para que pueda llegar a ser un duro rival en los campeonatos que se realizan anualmente para enfrentar a este tipo de programas.

Bibliografía

[All94] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. Tesis Doctoral, University of Limburg, Maastrich, 1994.

[Bou] B. Bouzy. Associating domain-dependent knowledge and Monte Carlo. Informe técnico.

[Bou01] B. Bouzy. *Computer Go: an AI Oriented Survey*, 2001.

[Bou03a] B. Bouzy. Developements on Monte Carlo Go. Informe técnico, Université Paris 5, UFR de mathematiques et d'informatique, 2003.

[Bou03b] B. Bouzy. Mathematical morphology applied to computer Go. *IJPRAI*, 2003.

[Bou04] B. Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9×9 Go. Informe técnico, Université Paris 5, UFR de mathématiques et d'informatique, 2004.

[BPP94] A. de Bruin, W. Pijls y A. Plaat. Solution Trees as a Basis for Game Tree Search. Informe técnico, Erasmus University, 1994.

[Caz99] T. Cazenave. *Generation of Patterns with External Conditions for the Game of Go*, 1999.

[CRS04] J. Chavero, J. P. Ramírez y E. Sánchez. Proyecto de juego de Go, 2004.

[Far02] G. Farnebäck. Specification of the Go text protocol, version 2, draft 2. Informe técnico, 2002.

[Fre60] E. Fredkin. *Trie Memory*. ACM Press, 1960.

[GH04] E. Gamma y R. Helm. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 2004.

[Gon88] G. H. Gonnet. Efficient Searching of Text and Pictures. Informe técnico, Center for the New Oxford English Dictiionary, 1988.

[Gon91] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison Wesley, 1991.

[Kol05] M. J. Kollingbaum. *Norm-Governed Practical Reasoning Agents*. Tesis Doctoral, University of Aberdeen, 2005.

[Mül01a] M. Müller. Pattern Matching in Explorer. Informe técnico, ETH Zurich, 2001.

[Mül01b] M. Müller. Proof-Set Search. Informe técnico, University of Alberta, 2001.

[Mül02] M. Müller. DF-PN in Go: An Application to the one-eye problem. Informe técnico, University of Alberta, 2002.

[Niu04] X. Niu. *Recognizing safe territories and stones in computer Go*. Proyecto Fin de Carrera, Department of Computing Science, University of Alberta, 2004.

[Pla96] A. Plaat. Best-First and Depth-First MiniMax Search in Practice. Informe técnico, Erasmus University, 1996.

[Zob69] A. Zobrist. A model of visual organization for the game of Go. En *AFIPS Spring Joint Computer Conference*. 1969.